Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Отчёт по второму заданию

Вариант №16: «Выполнимость КНФ»

Задание выполнил студент 427 группы
Кольцов Михаил Андреевич

Москва, 2014

Содержание

1 Постановка задачи

Требуется написать программу на языке Scheme, которая по заданной *конъюнктивной нормальной форме* (далее — КНФ) определяет, является ли она выполнимой. При этом в решении должен использоваться *генетический алгоритм* (описан в главе 3).

Формально:

Литера: переменная x_i или отрицание переменной $\overline{x_i}$;

Дизъюнкция n литер $L_1, L_2, \ldots, L_n$: $L_1 \vee L_2 \vee \ldots \vee L_n$ (здесь $\vee$ – опрератор логического ИЛИ);

КНФ n дизъюнкций $A_1, A_2, \ldots, A_n$: $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ (здесь $\wedge$ – опрератор логического И). Будем обозначать $Var(X)$ множество всех литер КНФ X;

КНФ X выполнима тогда и только тогда, когда $\exists v = (b_1, b_2, \ldots, b_{|Var(X)|}), b_i \in \{0, 1\}$, такое, что если в X заменить $\forall x_i \in Var(X)$ на b_i, то полученное выражение будет истинно.

В программу КНФ задаётся в виде списка списков. Все списки внутри большого списка рассматриваются как дизъюнкции, объединённые конъюнкцией. В каждом списке-дизъюнкции элементами являются символы (например, $alpha$ или x) либо списки из двух элементов (not символ), обозначающие отрицания.

Например, выражение

$$(\overline{x_1}) \wedge (x_{10} \vee \overline{x_{11}} \vee x_{101})$$

будет представлено в виде списка (((not x1)) (x10 (not x11) x101)).

2 Генерация тестов

Чтобы проверить правильность функционирования программы, был подготовлен генератор тестов, который создаёт (в зависимости от параметров) тестовую КНФ, а также заведомо правильный ответ для неё, а в случае выполнимости — ещё и одно из возможных решений.

2.1 Поддерживаемые параметры

1. Количество дизъюнкций в КНФ (минимальное/максимальное);
2. Количество литер в дизъюнкциях (минимальное/максимальное);
3. Количество различных литер (минимальное/максимальное);
4. Количество решений (минимальное/максимальное);
5. Параметр *амлпификации.*

2.2 Амплификация

Для того, чтобы вместе с тестами создавать возможные решения, использовался переборный алгоритм: для каждого $v = (b_1, b_2, \ldots, b_{|Var(X)|})$ проверяем истинность выражения. В случае истинности добавляем v во множество ответов.

Поскольку переборный алгоритм не справляется с решением задачи при большом числе переменных, мною был придуман способ увеличения тестового примера с сохранением знания обо всех решениях, который я назвал **амплификация** (от англ. *amplify* – «усиливать»). Суть способа в следующем. Пусть у нас имеется КНФ

$$X = A_1 \wedge A_2 \wedge \ldots \wedge A_n.$$

Составим новую КНФ:

$$X_{amp} = X_1 \wedge X_2 \wedge \ldots \wedge X_{amp\_factor}$$

$$X_j = A_{1j} \wedge A_{2j} \wedge \ldots \wedge A_{nj},$$

где A_{ij} — дизнюнкция A_i, в которой каждя литера t переименована в t_j.

Заметим, что если известен ответ

$$v = (b_1, b_2, \ldots, b_{|Var(X)|})$$

для X, то ответ для X_{amp} представляет из себя

$$v_{amp} = (b_{11}, b_{12}, \ldots, b_{1|Var(X)|}, b_{21}, b_{22}, \ldots, b_{2|Var(X)|}, \ldots, b_{amp\_factor1}, \ldots, b_{amp\_factor|Var(X)|}),$$

где $b_{ij} = b_j$.

Также зафиксируем соотношения между параметрами в исходном и амплифицированном выражениях:

- У X есть num решений $\rightarrow$ у X_{amp} есть num^{amp_factor} решений.
- В X присутствует l различных литер $\rightarrow$ в X_{amp} присутствует $l * amp\_factor$ различных литер.

2.3 Тестовые данные

Таким образом, если сгенерировать «маленький» тест и приписать его самому к себе с переименованием переменных amp_factor раз, то получится тест со сколь угодно (в зависимости от величины amp_factor) большим количество переменных, решений, дизъюнкций и литер.

Параметры генерации тестовых случаев выбирались из следующих соображений:

- $\frac{s^{amp\_factor}}{2^{m*amp\_factor}} <= 10^{-9}$, где s – параметр *количества решений* , m – параметр *количества различных переменных*. Иными словами, решением должен являться не более чем каждый миллиардный случайно выбранный булев вектор.
- Параметры *длины дизъюнкции* и *количества дизъюнкций* выбирались так, чтобы генерация теста не занимала много времени. От этих параметров зависит количество решений (чем больше дизъюнкций, тем больше шанс сгенерировать неразрешимое противоречие $x_i \wedge \overline{x_i}$; а чем больше длина дизъюнкции, тем выше шанс получить подвыражение вида $x_i \vee \overline{x_i}$, которое автоматически делает истинным данную дизъюнкцию). Оптимальными я считаю значения 5 <= длина дизъюнкции <= 7 и 7 <= количество дизъюнкций <= 12.
- *Параметр амплификации* выбирался таким, чтобы итоговое количество переменных не превышало 100 (в целях более быстрого прогона тестов).

Решению подавались на вход как амплифицированные, так и неамплифицированные тесты.

2.4 Основные моменты реализации

В программе за генерацию тестов отвечает функция **generate-tests**. Значения параметров установлены в виде констант. Основные функции: *gen-brace* (генерирует дизъюнкцию), *gen-expr* (генерирует КНФ), *amp-vars* и *amp-expr* (амплифицируют КНФ и решение), а также *gen-substitutions* (участвует в переборном решении).

Результат генерации возвращается функций *output-loop* в виде списка тестов, где каждый тест — тройка <КНФ, есть решение?, одно из решений (или пустой список, если решения нет)>.

3 Генетический алгоритм

Общая схема генетического алгоритма в программе состоит из *итераций*, на каждой из которых происходит следующее:

1. Определённый процент особей из популяции претерпевает *мутацию*;
2. Определённый процент особей из популяции *скрещивается* друг с другом;
3. Выбирается самый лучший (согласно *оценочной функции*) представитель популяции. Если функция на нём принимает максимально возможное значение, то алгоритм завершается и выдаёт в качестве ответа этого представителя;
4. В противном случае, часть особей погибает в процессе естественного *отбора*, а алгоритм переходит к следующей итерации.

Перед первой итерацией создаётся *начальная популяция* случайным образом.

3.1 Описание особи

Каждая особь в популяции представляет из себя булев вектор $v_i = (b_1, b_2, \ldots, b_{|Var(X)|})$, где X – входная КНФ. Одна хромосома – одна из компонент этого вектора.

3.2 Описание оценочной функции

Пусть в тестовой КНФ X содержится n дизъюнкций $A_1, A_2, \ldots, A_n$. Обозначим за $A_j(v_i)$ выражение, полученное из A_j заменой всех переменных на соответствующие им значения из v_i. Тогда функция $score$, оценивающая особь v_i из популяции, задаётся следующим образом:

$$score(v_i) = \frac{|\{A_j | A_j(v_i) = true, j \in \overline{1..n}\}|}{n}.$$

3.3 Описание мутации и скрещивания

При мутации, особь v_i инвертирует несколько своих случайно выбранных переменных.

При скрещивании двух особей v_i и v_j происходит следующее. Сначала в качестве результата $child$ берётся v_i. Затем $\forall k \in \overline{1..n}$ происходит замена k-го элемента $child$ на k-ый элемент v_j. Если при такой замене оценочная функция становится больше, то в $child$ записывается k-й элемент из v_j. В противном случае k-е значение остаётся неизменным.

3.4 Описание способа отбора

Поддерживается инвариант: на начало каждой итерации в популяции должно быть заданное (зависящего от входной КНФ) количество особей. Для обеспечения этого инварианта в конце каждой итерации происходит отбрасывание самых худших (согласно оценочной функции) особей.

3.5 Критерий останова

Если на шаге 3 генетического алгоритма обнаруживается ответ, то алгоритм немедленно прекращает работу. В противном случае, по прошествии заданного числа итераций (зависящего от входной КНФ) алгоритм завершает работу с результатом «решения нет».

3.6 Основные моменты реализации

Главная функция генетического алгоритма – **genetics-solver**. Внутри в виде переменных записаны параметры работы алгоритма (количество итераций, размер популяции, процент скрещиваемых на каждой итерации особей, процент мутирующих на каждой итерации особей, количество изменяемых хромосом в процессе мутации). Основные функции: *breed-one-vs-one* (скрещивание двух особей), *gen-mutate* (мутация особи), *make-initial-population* (создание начальной популяции), *genetics-iteration-loop* (цикл итераций генетического алгоритма).

4 Визуализация

4.1 Приближение текущего решения задачи к оптимальному

На каждой итерации генетического алгоритма снимаются два показания: наибольшее и наименьшее значение оценочной функции в текущей популяции. Эта информация собирается и визуализируется в виде графика (см. рис. 1), который выводится после окончания работы алгоритма. Красной ломаной обозначается наибольшее значение оценочной функции, зелёной – наименьшее. Этот график позволяет оценить, насколько быстро решение подбирается к оптимальному.

Реализация этой части программы расположена в функции **check-test** в виде вывода графиков.

4.2 Найденное решение

Чтобы визуализировать найденное решение, я строю двудольный граф. В первой доле находятся дизъюнкции, во второй – переменные. Ребро проведено из каждой дизъюнкции в каждую переменную, которая в ней встречается. Если переменная имеет при подстановке в дизъюнкцию значение $True$, то ребро окрашено зелёным цветом, в противном случае – синим.

После окончания работы алгоритма (в случае, если было найдено решение) появляется картинка (см. рис. 2), на которой изображается вышеописанный граф. Каждой вершине приписывается символьное описание (переменным – название и значение, дизъюнкциям – математическая запись). Благодаря цвету стрелок можно понять, насколько уникальными являются найденные значения переменных. Размер картинки выбирается в зависимости от входной КНФ, так что различные дизъюнкции визуально не «наезжают» друг на друга.

Реализация этой части программы расположена в функции **draw-solution**. В ней происходит инициализация картинки, построение вершин графа, их соединение и раскраска рёбер. Используется много вспомогательных функций.

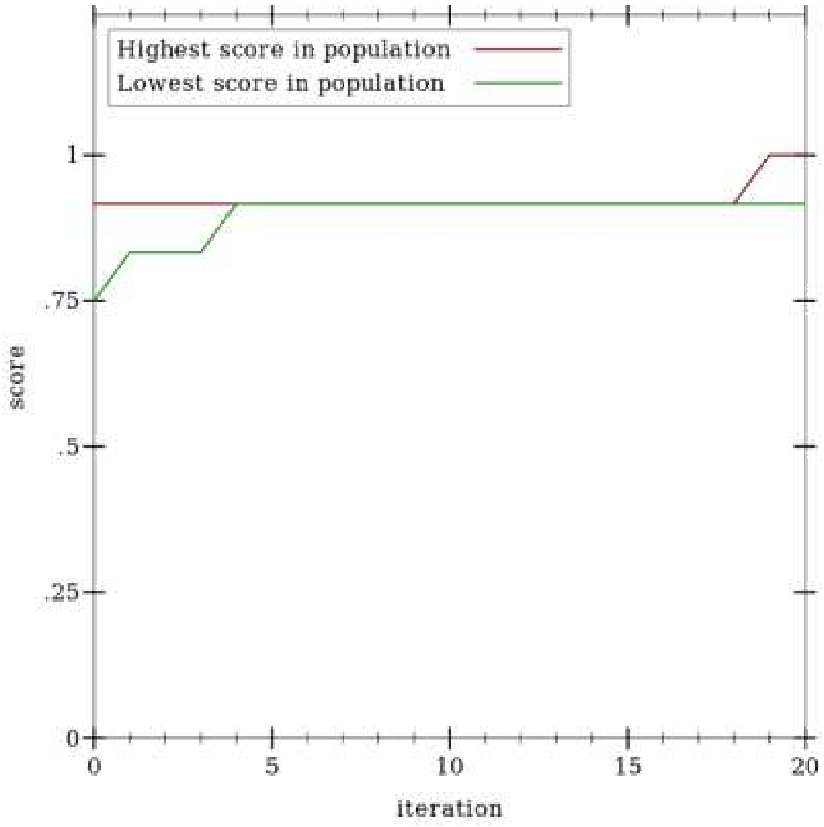

Рис. 1: Пример визуализации процесса приближения текущего решения к оптимальному

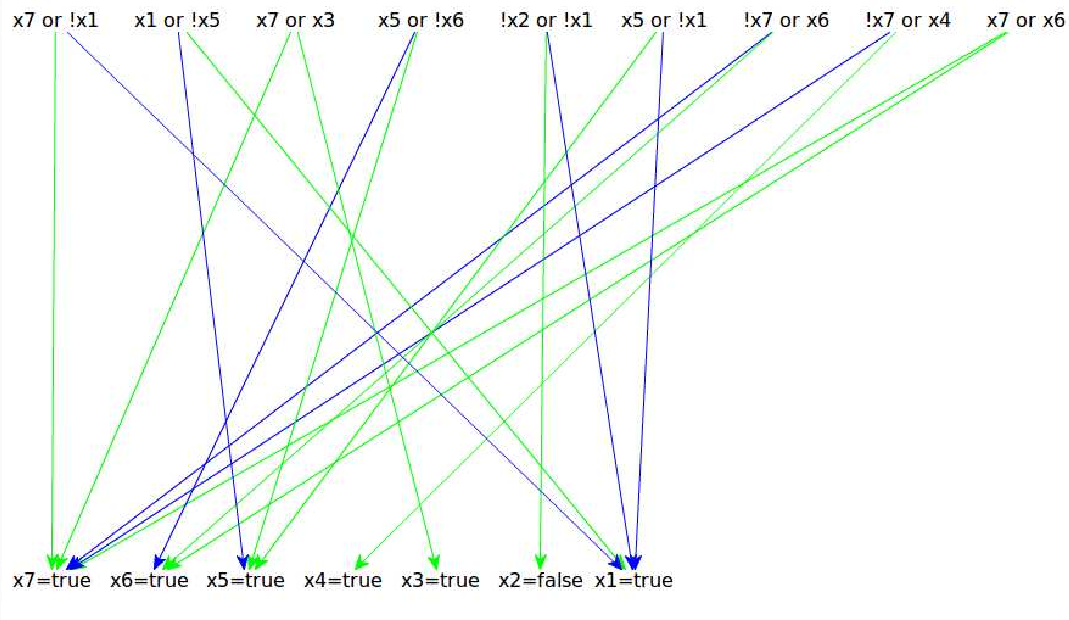

Рис. 2: Пример визуализации решения

5 Результаты

5.1 Неамплифицированные тесты

В данном тестовом случае решение работало верно более чем в 90% запусков. По времени запуск одного теста занимает меньше одной минуты. Причиной единичных неудач я вижу «невезучесть» рандомизации: видимо, иногда случайные мутации и скрещивания происходили исключительно непродуктивно. Впрочем, это случалось не так уж часто.

5.2 Амплифицированные тесты

Для амплифицированных тестов успешность запусков программы составляет около 50%. Тесты, в которых правильным ответом является «решения нет», программа проходит безошибочно. Там, где решение есть, часто происходит «застревание» прогресса со значением функции около 0.9. Я думаю, что реализовал алгоритм слишком неоптимально, и поэтому не хватает итераций для нахождения решения. Время работы программы на одном тесте: 10-20 минут.

Заключение

Написанная программа занимает 643 строки, в большом количестве случаев работает правильно и рисует графы. В процессе написания я часто обращался к документации языка, что позволило мне закрепить знания, полученные на лекциях; а также к статьям по теме решения задачи выполнимости КНФ, что позволило мне понять, что ценность реализованной мною программы для решения этой задачи, к сожалению, практически нулевая (современные алгоритмы могут решать её для миллионов переменных). В общем и целом, знакомство с языком Scheme я считаю успешным.

Приложение 1 Код программы

```
#lang scheme/base

(require racket/list)
(require racket/bool)
(require plot)
(require
   (lib "mred.ss" "mred")
   (lib "graph.ss" "mrlib")
   (lib "class.ss")
   )

(define ns (make-base-namespace)) ; used for (eval ...) calls

(define MAX_ITERATIONS 20) ; must be a multiple of 5 for plotting
   purposes

(define (pick-random lst)
  ; Returns random element from 'lst'.
  (list-ref lst (random (length lst))))

(define (vector-assoc val vec)
  ; Same as 'assoc' function in Scheme, but for vectors.
  (letrec (
           [len (vector-length vec)]
           [helper (lambda (pos)
                     (cond
                       [(= pos len) #f]
                       [(and (pair? (vector-ref vec pos)) (equal?
                          (car (vector-ref vec pos)) val))
                          (vector-ref vec pos)]
                       [else (helper (+ pos 1))]))])
    (helper 0)))

(define (first-n-elems lst n ret)
  ; Returns first 'n' elements in 'lst', in reversed order.
  ; Example usage:
  ;    (first-n-elems '(1 2 3 4) 3 '()) => '(3 2 1)
  (if (= n 0)
      ret
      (first-n-elems (cdr lst) (- n 1) (cons (car lst) ret)))
)

(define (prob n1 n2)
```

```
  (< (random n2) n1))

(define (member? x lst)
  ; Checks if 'x' is member of 'lst'.
  (cond
    ((null? lst) #f)
    ((equal? x (car lst)) #t)
    (else (member? x (cdr lst)))
  )
)

(define (find-pair vars var)
  ; Given a list 'vars' of pairs (x1, y1) (x2, y2) ... (xn, yn)
     returns an yi such as xi = 'var'.
  (if (equal? (caar vars) var)
      (cadar vars)
      (find-pair (cdr vars) var)
  )
)


(define (sub-var vars expr)
  ; Sub-var replaces every occurence of variable in 'expr' with
     its value (according to 'vars').

  ; Example usage:
  ;    (sub-var '((x1 #f) (x10 #t) (x11 #f) (x101 #t)) '(((not
     x1)) (x10 (not x11) x101))) => (((not #f)) (#t (not #f) #t))
  (cond
         ((null? expr) '())
         ((list? expr) (cons (sub-var vars (car expr)) (sub-var
            vars (cdr expr))))
         ((member? expr (map (lambda (x) (car x)) vars))
            (find-pair vars expr))
         (else expr)
   )
)

(define (make-and-eval-cnf expr)
  ; Makes a valid scheme boolean expression, that stands for
     conjunction normal form of 'expr'.

  ; Example usage:
  ;    (make-and-eval-cnf '(((not #f)) (#t (not #f) #t))) => !just
     for clarity! (and (or (not #f)) (or #t (not #f) #t)) => #t
```

```
  (eval (cons 'and (map (lambda (x) (cons 'or x)) expr)) ns)
)

(define (make-and-sum-cnf expr)
  ; Counts the number of disjunctions in cnf 'expr', which
     evaluate to #t.

  ; Example usage:
  ;    (make-and-sum-cnf (((not #f)) (#t (not #f) #t))) => 2
  ;    (make-and-sum-cnf `((#f) (#t))) => 1
  (length (filter (lambda (x) x) (eval (cons 'list (map (lambda
     (x) (cons 'or x)) expr)) ns)))
)

(define (score man test)
  ; Score equals to number of disjunctions that equal to #t when
     using 'man' as variables in expression 'test'.
  ; Notice that if a variable is not in 'man', score will be equal
     to 'test''s length (need to be careful when calling this
     function).
  (if (null? man)
      100500
      (make-and-sum-cnf (sub-var man test)))
)

(define (generate-tests)
    ; Generates a list of testcases for SAT problem.
    ; Each testcase is a triple: <expression, is-solvable?, one
       answer>.


    (define MIN_BRACE_SIZE 2) ; minimal length of disjunction in
       generated expression
    (define MAX_BRACE_SIZE 5) ; maximal length of disjunction in
       generated expression

    (define MIN_VARIABLES 7) ; minimal number of variables in
       generated expression
    (define MAX_VARIABLES 9) ; maximal number of variables in
       generated expression

    (define MIN_LENGTH 7) ; minimal length of generated expression
       (number of conjunctions)
    (define MAX_LENGTH 12) ; maximal length of generated
       expression (number of conjunctions)
```

```
(define MAX_SOLUTIONS 22) ; maximal number of solutions
   (satisfactory boolean vectors) for expression
(define MIN_SOLUTIONS 1) ; minimal number of solutions
   (satisfactory boolean vectors) for expression
(define AMP_FACTOR 9) ; number of times that expression is
   appended to itself, when doing amplification
(define LAST_AMPLIFIED 0) ; number of testcases that must be
   amplified

(define N_TESTCASES 5) ; number of generated testcases



(define (gen-expr num_var num_brace result)
  ; Generates a valid input (CNF) for our problem. Variables
     are named "x_1" .. "x_num_var" (maybe not all are
     actually presented),
  ; CNF consists of 'num_brace' conjunctions.

  ; Example usage:
  ;    (gen-expr 6 3 '()) => (("x6" (not "x2") (not "x6") "x5"
     "x4" (not "x1") "x1" (not "x3")) ((not "x1")) ("x5" "x3"
     "x2" (not "x1") (not "x1")))
  (if (= num_brace 0)
      result
      (gen-expr num_var (- num_brace 1) (cons (gen-brace
         num_var (max MIN_BRACE_SIZE (random (+ 1
         MAX_BRACE_SIZE))) '()) result))
   )
)

(define (gen-brace num_var size result)
  ; Generates one brace (i.e one conjunction of CNF) with
     'num_var' variables and size = 'size'.

  ; Example usage:
  ;    (gen-brace 5 3 '()) => ((not "x2") "x4" (not "x1"))
  (if (= size 0)
      result
      (let* ((cur_var (make-variable-name (max 1 (random (+ 1
         num_var)))))
           (cur_brace (if (prob 1 2) cur_var (list 'not
              cur_var)))
            )
```

```
        (gen-brace num_var (- size 1) (cons cur_brace result))
      )
    )
)

(define (make-variable-name n)
  ; Takes the number 'n', returns string "xn".
  (string->symbol (string-append "x" (number->string n)))
)

(define (amp-vars vars rate)
  ; Produces list of renamed variables of 'vars', each
     variable "x" turns into "x'rate'".

  ; Example usage:
  ;    (amp-vars '(("x1" #t) ("x2" #f)) 4) => '(("x14" #t)
     ("x24" #f))
  (map (lambda (x) (list (string->symbol (string-append
     (symbol->string (car x)) (number->string rate))) (cadr
     x))) vars)
)

(define (amp-expr expr rate)
  ; Appends expression 'expr' to itself 'rate' times.
  (cond
         ((null? expr) '())
         ((list? expr) (cons (amp-expr (car expr) rate)
            (amp-expr (cdr expr) rate)))
         ((and (symbol? expr) (not (eq? 'not expr)))
            (string->symbol (string-append (symbol->string
            expr) (number->string rate))))
         (else expr)
   )
)

(define (amplify testcase rate result)
  ; 'testcase' = <expr, solvable?, solution>.
  ; Amplify returns <expr_amplified, solvable?,
     solution_amplfied>, as described in all_variants function
     comments.
  (if (= 0 rate)
      result
      (amplify testcase (- rate 1)
               (list (append (amp-expr (car testcase) rate)
                  (car result)) (cadr result) (append
```

```
                    (amp-vars (caddr testcase) rate) (caddr
                    result))))
    )
)

(define (gen-substitutions n so_far)
  ; Generates all possible boolean vectors of length n, where
     each position correspond to some variable name.

  ; Example usage:
  ; (gen-substitutions 4 '()) => ((("x1" #t) ("x2" #t) ("x3"
     #t) ("x4" #t))
  ;                               (("x1" #f) ("x2" #t) ("x3"
     #t) ("x4" #t))
  ;                               (("x1" #t) ("x2" #f) ("x3"
     #t) ("x4" #t))
  ;                               (("x1" #f) ("x2" #f) ("x3"
     #t) ("x4" #t))
  ;                               (("x1" #t) ("x2" #t) ("x3"
     #f) ("x4" #t))
  ;                               (("x1" #f) ("x2" #t) ("x3"
     #f) ("x4" #t))
  ;                               (("x1" #t) ("x2" #f) ("x3"
     #f) ("x4" #t))
  ;                               (("x1" #f) ("x2" #f) ("x3"
     #f) ("x4" #t))
  ;                               (("x1" #t) ("x2" #t) ("x3"
     #t) ("x4" #f))
  ;                               (("x1" #f) ("x2" #t) ("x3"
     #t) ("x4" #f))
  ;                               (("x1" #t) ("x2" #f) ("x3"
     #t) ("x4" #f))
  ;                               (("x1" #f) ("x2" #f) ("x3"
     #t) ("x4" #f))
  ;                               (("x1" #t) ("x2" #t) ("x3"
     #f) ("x4" #f))
  ;                               (("x1" #f) ("x2" #t) ("x3"
     #f) ("x4" #f))
  ;                               (("x1" #t) ("x2" #f) ("x3"
     #f) ("x4" #f))
  ;                               (("x1" #f) ("x2" #f) ("x3"
     #f) ("x4" #f)))
  (if (= n 0)
      (list so_far)
```

```
      (append (gen-substitutions (- n 1) (cons (list
         (make-variable-name n) #t) so_far))
              (gen-substitutions (- n 1) (cons (list
                 (make-variable-name n) #f) so_far)))
   )
)
(let ((all_variants (gen-substitutions MAX_VARIABLES '())))
; Output-loop is generating random tests, checks if they are
; a) unsolvable;
; b) solvable using <= MAX_SOLUTIONS boolean vectors.
; Test expressions, that match a) or b) are added into tests.
; Returns num_test of triples <testcase, solvable?, solution>.

; Some tests are "amplified", by that I mean, that the
   resulting test has arbitrary length,
; but we still know the answer (or at lest one of them),
   despite that brute-force can not achieve it.

; The process goes as follows. Let's look at the example:
; X = (("x1") ((not "x2")) ((not "x3") "x4")) - this CNF is
   satisfied with vectors (1, 0, 0, 1), (1, 0, 1, 1) and (1,
   0, 0, 0), three solutions overall.
; Let's add X to itself, but with x1..4 renamed to x5..8:
; X2 = (("x1") ((not "x2")) ((not "x3") "x4") ("x5") ((not
   "x6")) ((not "x7") "x8")). To satisfy X2, we need to
   satisfy both X and renamed X.
; They both have 3 solutions, so the overall number of
   solutions for X2 is 3 * 3 = 9 (for example, one of them is
   (1, 0, 0, 1, 1, 0, 1, 1)).
; But notice that for X, we had 2 ^ 4 = 16 possibilities of
   boolean
; vectors. Now, for X2, this number is 2 ^ (4 * 2) = 2 ^ 8 =
   256.

; Generally, if we have some expression E with V variables and
   it has R solutions, then denote
; E_i = E with variable x_j (j = 1..V) renamed to x_(i * V +
   j) (my code does slightly different renaming), i = 0..K,
; and then take E_amplified = (E_K) (E_(K - 1)) ... (E_0).
; E_amplified has V * K variables, so the search space for
   satisfactory vectors has size = 2 ^ (V * K), and the number
   of satisfactory vectors is
; precisely R ^ K (since only variations of initial R
   solutions are satisfactory).
```

```
; If we take, for example, K = 100, V = 5, R = 20, then the
   portion of valid answers in the search space is
; (20 ^ 100) / (2 ^ (100 * 5)), which is approximately 10 ^
   (-21). That means, that even by picking 10 ^ 9 random
   vectors every second,
; the brute-force program will take 3000 years to find the
   answer.

; If there is no answer for E, then E_amplified also does not
   have one.

; Only few last cases are amplified, so that we can test
   program on "small" tests and see that it works fine, and
   then proceed with "large" ones.

; Example usage (without amplification):
; (output-loop 15 '()) => ((("x4") ("x1" (not "x2") "x4")
   ("x2" "x1") ("x1" "x4" "x1")) #t (("x1" #t) ("x2" #t) ("x3"
   #t) ("x4" #t) ("x5" #t)))
;                           ((("x2" "x2" "x1") ("x2" "x2")
   ((not "x1")) ((not "x1") "x2")) #t (("x1" #f) ("x2" #t)
   ("x3" #t) ("x4" #t) ("x5" #t)))
;                           ((("x1") ((not "x1")) ("x3" (not
   "x1") (not "x1"))) #f ())
;                           ((("x1" "x1" "x1") ((not "x1")
   (not "x1")) ("x1" (not "x1") (not "x1")) ((not "x1"))) #f
   ())
;                           ((((not "x1"))) #t (("x1" #f)
   ("x2" #t) ("x3" #t) ("x4" #t) ("x5" #t)))
;                           ((((not "x1") "x1") ((not "x1"))
   ("x1")) #f ())
;                           ((((not "x1") "x2" (not "x1"))
   ("x3" (not "x1"))) #t (("x1" #t) ("x2" #t) ("x3" #t) ("x4"
   #t) ("x5" #t)))
;                           ((((not "x1") (not "x1"))) #t
   (("x1" #f) ("x2" #t) ("x3" #t) ("x4" #t) ("x5" #t)))
;                           ((((not "x5") "x1" "x4") ((not
   "x3")) ("x4")) #t (("x1" #t) ("x2" #t) ("x3" #f) ("x4" #t)
   ("x5" #t)))
;                           ((((not "x3")) ((not "x4") (not
   "x2")) ((not "x1")) ("x1")) #f ())
;                           ((((not "x1") (not "x1")) ((not
   "x1") "x1") ("x1" (not "x1") (not "x1"))) #t (("x1" #f)
   ("x2" #t) ("x3" #t) ("x4" #t) ("x5" #t)))
```

```
    ;                               ((((not "x1"))) #t (("x1" #f)
       ("x2" #t) ("x3" #t) ("x4" #t) ("x5" #t)))
    ;                               ((((not "x1") "x1") ("x1" "x1"
       "x1") ("x1" "x1" "x1") ("x1" "x1" "x1")) #t (("x1" #t)
       ("x2" #t) ("x3" #t) ("x4" #t) ("x5" #t)))
    ;                               ((((not "x1")) ((not "x4"))) #t
       (("x1" #f) ("x2" #t) ("x3" #t) ("x4" #f) ("x5" #t)))
    ;                               ((("x1") ("x1") ((not "x1") (not
       "x1")) ("x1" (not "x1") "x1")) #f ()))

    (define (output-loop num_test tests)
      ; Main function, produces 'num_test' number of testcases,
         with respect to parameters.
      (if (= 0 num_test)
          tests
          (let* ((num_vars (max MIN_VARIABLES (random (+ 1
             MAX_VARIABLES))))
                 (len (max MIN_LENGTH (random (+ 1 MAX_LENGTH))))
                 (expr (gen-expr num_vars len '()))
                 (answers (filter (lambda (x) (make-and-eval-cnf
                    (sub-var x expr))) all_variants))
                 (testcase (list expr (not (null? answers)) (if
                    (null? answers) '() (car answers))))
                 )
            (if (and (>= (length answers) MIN_SOLUTIONS) (<=
               (length answers) MAX_SOLUTIONS))
                (if (>= num_test LAST_AMPLIFIED)
                    (output-loop (- num_test 1) (cons testcase
                       tests))
                    (output-loop (- num_test 1) (cons (amplify
                       testcase AMP_FACTOR testcase) tests))
                )
                (output-loop num_test tests)
            )
          )
      )
    )
    (output-loop N_TESTCASES '())
    )
)


(define (genetics-solver test)
  ; Genetics-solver produces an answer for testcase 'test'.
  ; Solution is working according to genetic algorithm:
```

```
; there is a "population" of "individuals", each of them is a
   list of variables
; (("x1" #t) ("x2" #f) ("x3" #t)) - example of "individual".
; Process starts with random population. Then, iteratively, it
   produces next generation populations,
; in which the "score" of individuals get higher.

; Score of individual is defined as number of conjunctions in
   CNF, that turn out to #f on this individual.
; Example usage:
;    (genetics-solver '(("x1") ("x2"))) => (("x1" #t) ("x2" #t))
;    (genetics-solver '(("x1") ((not "x1")))) => #f

(define (extract-variables expr)
  ; Example usage:
  ;   (extract-variables '(("x4") ("x1" (not "x2") "x4") ("x2"
     "x1") ("x1" "x4" "x1"))) => ("x4" "x1" "x2")
  (filter (lambda (s) (and (symbol? s) (not (eq? 'not s))))
     (remove-duplicates (flatten expr)))
)

; working-variables - variables, that 'test' contain.
; For example, if test = (("x4") ("x1" (not "x2") "x4") ("x2"
   "x1") ("x1" "x4" "x1"))), then
; working-variables = ("x4" "x1" "x2")
(define working-variables (extract-variables test))

(define number-of-variables (length working-variables)) ; number
   of variables in test
(define N_ITER MAX_ITERATIONS) ; maximal number of generations
   in genetic algorithm
(define N_POPULATION 5) ; number of individuals in every
   generation
(define PERCENT_BREEDS 20) ; percentage of breeding individuals
(define PERCENT_MUTATIONS 40) ; percentage of mutating
   individuals
(define N_MUTATED (max 3 (quotient number-of-variables 10))) ;
   number of inverting variables when mutating

(define (random-solution vars)
  ; Example usage:
  ;   (random-solution ("x4" "x1" "x2")) => (("x4" #t) ("x1" #t)
     ("x2" #f))
  (map (lambda (x) (list x (prob 1 2))) vars)
)
```

```
(define (gen-mutate x rem-mutations)
  ; Returns mutated individual 'x': random 'rem-mutations'
     variables are inverted.
  (if (or (null? x) (= rem-mutations 0))
      x
      (let* ((sh (shuffle x))
             (head (car sh)))
      (gen-mutate (cons (list (car head) (not (cadr head))) (cdr
         sh)) (- rem-mutations 1))))
)

(define (breed-one-vs-one one two)
  ; Crossingover of two individuals.
  ; It goes as follows: take all features of individual 'one'.
     Then repeatedly try to substitute
  ; some variable value to value of 'two'. If the result score
     is better, then do substitution.

  ; Example usage (when test expression is '(("x1") ("x2")):
  ;   (breed-one-vs-one '(("x1" #t) ("x2" #f)) '(("x1" #f")
     ("x2" #t))) => '(("x1" #t) ("x2" #t))

  (define cmp (lambda (x y) (symbol<? (car x) (car y)))) ; we
     need to sort 'one' and 'two' to properly substitute
  (define f (sort one cmp))
  (define m (sort two cmp))
  (define len (length f))
  (define (helper n result)
    (if (= n len)
        result
        (let* ((tail (list-tail f (+ n 1)))
               (head (list-ref m n))
               (candidate1 (append result (cons head tail)))
               (candidate2 (append result (list-tail f n)))
               )
          (if (> (scorer candidate1) (scorer candidate2))
              (helper (+ n 1) (cons head result))
              (helper (+ n 1) (cons (list-ref f n) result))
          )
        )
    )
  )
  (helper 0 '())
)
```

```
(define (breed-one-vs-all one all result)
  ; Produces a list of individuals, that appear after
     crossingovering 'one' vs every individual of 'all'.
  (if (null? all)
      result
      (breed-one-vs-all one (cdr all) (cons (breed-one-vs-one
         one (car all)) result)))
)

(define (breed-all-vs-all men result)
  ; Produces a list of individuals, that appear after
     crossingovering every individual of 'men' against every
     other individual.
  (if (< (length men) 2)
      result
      (breed-all-vs-all (cdr men) (append result
         (breed-one-vs-all (car men) (cdr men) '()))))
)

(define (make-initial-population n)
  ; Returns initial popultion for genetic algorithm. It
     contatins randomly generated individuals.
  (if (= n 0)
      '()
      (cons (random-solution working-variables)
         (make-initial-population (- n 1)))
  )
 )

(define (cached-scorer test)
  ; Wrapper around 'score' function to specific expression
     'test'. Stores 'CACHE_SIZE' last calls (query and answer).
  ; Cache is updated in round-robin manner.
  ; Returns a function that can be called on any individual.

  ; Example usage:
  ;   (let ((scorer (cached-scorer test))) (scorer '(("x1" #t)
     ("x2" #f))))
  (define CACHE_SIZE 10)
  (define cache (make-vector CACHE_SIZE #f))
  (define pos 0)
  (define (answer lst)
    (let ((result-cached (vector-assoc lst cache)))
      (if result-cached
```

```
          (cdr result-cached)
          (let ((answer (score lst test)))
            (vector-set! cache pos (cons lst answer))
            (set! pos (if (= pos (- CACHE_SIZE 1)) 0 (+ pos 1)))
            answer
          )
      )
    )
  )
  (lambda (x) (answer x))
)

(define scorer (cached-scorer test)) ; function, that are used
   for scoring
(define population (make-initial-population N_POPULATION)) ;
   initial random population
(define test-len (length test)) ; number of conjunctions in
   expression

(define (normalize-metric v) (/ (exact->inexact v) (length
   test))) ; normalizes score to [0, 1]

(define (genetics-iteration-loop iter-list population)
  ; Main function, loops genetic algorithm for 'iter'
     iterations, with current population 'population'. Returns
     pair: <answer, number of iterations>.
  ; If 'iter' = 0, returns #f (i.e. no solution). Otherwise,
     does several steps:
  ; 1. Mutate some individuals and add them into population
  ; 2. Breed some individuals between each other and add them
     into population
  ; 3. Sort individuals according to 'score'
  ; 4. Produce 'next-generation' as N_POPULATION best
     individuals from current population.
  ; 5. Check if the best individual in 'next-generation' is the
     answer. If yes, return it.
  ; Writes histogram of 'score' function on 'population'.
  (if (= (length iter-list) N_ITER)
      (cons #f iter-list)
      (let* ((mutated-pop (append population (map (lambda (x)
         (if (prob PERCENT_MUTATIONS 100) (gen-mutate x
         N_MUTATED) x)) population))) ; step 1
            (breed-candidates (filter (lambda (x) (prob
               PERCENT_BREEDS 100)) mutated-pop)) ; substep of
               step 2
```

```
                (breeded-pop (append mutated-pop (breed-all-vs-all
                   breed-candidates '()))) ; step 2
                (sorted-pop (sort breeded-pop (lambda (x y) (>
                   (scorer x) (scorer y))))) ; step 3
                (next-generation (reverse (first-n-elems sorted-pop
                   N_POPULATION '()))) ; step 4
                (alpha-man (car next-generation)) ; step 5
                (worst-man (car (reverse next-generation)))
                (new-iter-list (cons (cons (normalize-metric
                   (scorer alpha-man)) (normalize-metric (scorer
                   worst-man))) iter-list))
              )
            (if (= test-len (scorer alpha-man))
                (cons alpha-man new-iter-list)
                (genetics-iteration-loop new-iter-list
                   next-generation)
            )
          )
      )
    )

    (genetics-iteration-loop '() population)
)

(define (check-test test)
  ; Runs 'genetic-solver' on testcase 'test', which is a triple:
     <expression, is-solvable?, one possible answer (if any)>.
  ; Prints testcase contents and the result of running the
     algorithm.
  (let* ((expr (car test))
         (is-possible (cadr test))
         (answer (genetics-solver expr))
         (iter-list (reverse (cdr answer)))
         (my-answer (car answer)))
    (cond ((and (not is-possible) my-answer)
           (printf "Test␣failed:␣found␣solution␣when␣there␣is␣
              none\nexpression:␣~a\nexpected␣answer:␣~a\n\n" expr
              (caddr test)))
          ((and is-possible (or (not my-answer)
                  (< (score my-answer expr) (length expr))))
           (printf "Test␣failed:␣did␣not␣find␣the␣solution,␣when␣
              there␣is␣some\nexpression:␣~a\nexpected␣answer:␣
              ~a\n\n" expr (caddr test)))
          (else
```

```
            (begin (printf "Test passed!\nexpression: ~a\nis
               possible to solve: ~a\nmy answer: ~a\n\n" expr
               is-possible my-answer)
                   (if is-possible
                       (draw-solution expr my-answer)
                       (printf "Solution cannot be visualised,
                          because there is none.\n"))
            ))
    )
    ; After each test, plot two 2D-functions f1(x) = highest
       normalized score on iteration x,
    ; f2(x) = lowest normalized score on iteration x. Theese
       functions are based on a list, that is returned
    ; by genetics-solver, which holds pairs ((high1, low1),
       (high2, low2), ..., (highn, lown)).
    (define (pfunc getter) (lambda (x)
       ; Returns a function, that linearly interpolates
          'iter-list' to non-integer points.
       (let* ((len (- (length iter-list) 1))
              (high (min len (inexact->exact (ceiling x))))
              (low (min len (inexact->exact (floor x))))
              (alpha (- x low))
              (x0 (getter (list-ref iter-list low)))
              (x1 (getter (list-ref iter-list high))))
         (if (= high low)
             x0
             (+ (* alpha x1) (* (- 1 alpha) x0))
         )
       )
    ))
    (printf "~a\n" (plot (list (function (pfunc car) 0
       MAX_ITERATIONS #:label "Highest score in population"
       #:color 1)
                 (function (pfunc cdr) 0 MAX_ITERATIONS #:label
                    "Lowest score in population" #:color 2))
                          #:y-min 0 #:y-max 1.24 #:x-label
                             "iteration" #:y-label "score"))
    iter-list
  )
)

(define (draw-solution test ans)
  ; Draws graph, thus visualizing the solution.
  ; Example taken from
     http://lists.racket-lang.org/users/archive/2007-September/020710.html,
```

```
    then adapted
  ; for my needs.

  (define (clause-to-string clause)
    ; Produces string representation of CNF-clause.
    (define (helper x)
      (if (pair? x)
          (string-append "!" (symbol->string (cadr x)))
          (symbol->string x)
      )
    )
    (if (= (length clause) 1)
        (helper (car clause))
        (string-append (helper (car clause)) " or "
           (clause-to-string (cdr clause))))
  )

  (define max-clause-length (* 3 (apply max (map (lambda (x)
     (string-length (clause-to-string x))) test))))
  (define number-of-clauses (length test))


  (define the-frame
    (new frame% ;
         (label "Solution visualisation")
         (width (* max-clause-length (+ 10 number-of-clauses)))
         (height 700)))

  ; Canvas initialization:
  (define the-editor-canvas (instantiate editor-canvas%
     (the-frame)))
  (define draw-lines-pasteboard% (class (graph-pasteboard-mixin
     pasteboard%)
;
                                       (super-new)
                                       (define/public (add-issue issue)
                                         (let* ((issue-snip
                                            (make-object
my-graph-snip% issue))
                                                )
                                           (send this insert
                                              issue-snip)
                                           issue-snip))))
```

```
  (define the-graph-pasteboard (instantiate draw-lines-pasteboard%
     ()))
  (send the-editor-canvas set-editor the-graph-pasteboard)

  (define my-graph-snip% (graph-snip-mixin string-snip%)) ;; use
string-snip%
  (send the-frame show #t)

  (define-values (width height) (send the-editor-canvas
     get-client-size))
  ; end canvas initialization.

  (define (connect parent child color)
    ; Function makes an edge parent->child, which is colored in
       color='color'.
    (define dark-pen (send the-pen-list find-or-create-pen color 1
       'solid))
    (define dark-brush (send the-brush-list find-or-create-brush
       color
'solid))
    (define light-pen (send the-pen-list find-or-create-pen color 1
'solid))
    (define light-brush (send the-brush-list find-or-create-brush
       color 'solid))
    (add-links parent child
               dark-pen light-pen
               dark-brush light-brush ))

  (define (b->s bo)
    ;; boolean->string conversion
    (if bo
        "true"
        "false")
  )

  ; List of vertices of a graph, that correspond to variables.
  (define variable-vertex-list (map (lambda (x) (send
     the-graph-pasteboard add-issue (string-append (symbol->string
     (car x)) "=" (b->s (cadr x))))) ans))

  ; List of vertices of a graph, that correspond to clauses.
  (define clause-vertex-list (map (lambda (x) (send
     the-graph-pasteboard add-issue (clause-to-string x))) test))
```

```
(define variable-dx (/ (- width 10) (length
   variable-vertex-list)))
(define clause-dx (* 2 max-clause-length))

(define (place-vertices lst idx height)
  ; Function places vertices from 'lst' on the given
     height='height'.
  (if (null? lst)
      '()
      (begin
        (send the-graph-pasteboard move-to (car lst) (+ 10 (*
           variable-dx idx)) height)
        (place-vertices (cdr lst) (+ 1 idx) height)
      )
  )
)

(define (find-index lst v)
  ; Finds index i such that lst[i] = v.
  (define (helper cur idx)
    (if (equal? (car cur) v)
        idx
        (helper (cdr cur) (+ 1 idx)))
  )
  (helper lst 0)
)

(define (clause-vars clause)
  ; Returns list of variables presented in clause.
  (if (null? clause)
      '()
      (cons (if (pair? (car clause)) (cadar clause) (car
         clause)) (clause-vars (cdr clause)))
  )
)

(define (connect-clause clause vertex green)
  ; Connects clause 'clause' with vertex 'vertex'.
  (if (null? clause)
      '()
      (let* ((head (car clause))
             (varname (if (pair? head) (cadr head) head))
             (is-neg (pair? head))
             (varvalue (cadr (assoc varname ans)))
```

```
                (varpos (find-index ans (list varname varvalue)))
                (varvertex (list-ref variable-vertex-list varpos))
                (next-green (if (xor is-neg varvalue) (cons
                   varvertex green) green))
            )
          (if (xor is-neg varvalue)
              (connect vertex varvertex "green")
              (if (or (member? varvertex green) (member? varname
                 (clause-vars (cdr clause))))
                  '()
                  (connect vertex varvertex "blue"))
              )
          (connect-clause (cdr clause) vertex next-green)
          ))
  )

  (define (loop-connect idx)
    ; Connects every clause with corresponding vertices.
    (if (= idx (length test))
        '()
        (let ((current (list-ref test idx)))
          (connect-clause current (list-ref clause-vertex-list
             idx) '())
          (loop-connect (+ 1 idx))
          ))
  )

  (loop-connect 0)

  (place-vertices variable-vertex-list 0 500)
  (place-vertices clause-vertex-list 0 40)

)

(define (test-runner)
  ; Runs 'check-test' on generated testcases.
  (define iterations (map check-test (generate-tests)))
  (let (
        (histogram (map (lambda (x) (list x (count (lambda (y) (=
           (length y) x)) iterations))) (range 1 MAX_ITERATIONS)))
      )
    ; Output a histogram for number of iterations till finishing.
    (plot (discrete-histogram histogram) #:x-label "iterations"
      #:y-label "count" #:title "Number␣of␣iterations␣to␣find␣
      answer")
```

```
  )
)

#include "mpi.h"
#include <unistd.h>
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <ctime>

using std::pair;
using std::make_pair;

const int ROWS = 4;
const int COLS = 4;

pair<int, int> number_to_coordinates(int number)
{
    // returns pair (row, column) in 0-indexation
    return make_pair(number / COLS, number % COLS);
}

int coordinates_to_number(pair<int, int> coord)
{
    return coord.first * COLS + coord.second;
}

bool is_valid_coordinates(int x, int y)
{
    return !(x < 0 or x >= ROWS or y < 0 or y >= COLS);
}

void send_to_neighbour(int *buf, int x, int y, int dx, int dy, int idx, int
{
    MPI_Request empty_req;
    int next_x = x + dx;
    int next_y = y + dy;

    if(!is_valid_coordinates(next_x, next_y))
        return;

    int reciever = coordinates_to_number(make_pair(next_x, next_y));
```

```
        MPI_Isend(&buf[idx], 1, MPI_INT, reciever, tag, MPI_COMM_WORLD, &empty_
}
```