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

Отчёт по второму заданию

Вариант 17. Задача сельского почтальона

Асирян Александр Камоевич
Группа 428

Москва, 2015

Оглавление

Глава 1

Постановка задачи

На вход программе подаётся описание взвешенного неориентированного графа $G = (V, E)$ в виде списка описаний рёбер. Каждое описание ребра $e \in E$ имеет вид ($NODE1$ $NODE2$ $WEIGHT$), где $NODE1$, $NODE2$ – атомы, $WEIGHT$ – целое положительное число. Затем на вход программе подаётся целое положительное число K и список рёбер L. Существует ли в графе гамильтонов G цикл, то есть путь, начинающийся и заканчивающийся в одной и той же вершине и проходящий через каждую вершину из графа G только один раз, стоимости не более чем K, включающий в себя все рёбра из списка L (и, возможно, другие)? Стоимость пути – это суммарная стоимость рёбер, составляющих путь. Если искомый цикл не существует, напечатайте #f. Если искомый цикл существует, напечатайте #t, далее стоимость цикла, а затем найденный цикл в виде списка. Первым и последним элементом списка должен быть один и тот же атом. Если существует несколько возможных решений, найдите решение с минимальной стоимостью.

Решение задачи разделено на три этапа:

1. На первом этапе разрабатываются тесты, на которых будет проверяться программа.

2. На втором этапе разрабатывается версия программы, которая находит решение, но не выполняет визуализацию.

3. На третьем этапе разрабатывается полная версия программы, выполняющая и решение задачи, визуализацию хода решения, визуализацию найденного ответа.

Глава 2

Генерация тестов

На данном этапе рассматривались только несколько видов графов, каждый из которых обладает своими свойствами. Вес для ребер генерировался случайно на отрезке $[0, 9]$. Количество сгенерированных контрольных ребер равнялось $\max(\lfloor |V| \ / \ 2 \rfloor \ + \ 1, \ 1)$.

2.1 Граф-цикл

Граф-цикл представляет собой последовательность вида $((v_1 \ v_2 \ weight_{1,2}) \ (v_2 \ v_3 \ weight_{2,3}) \ ... \ (v_{N-1} \ v_N \ weight_{N-1,N}) \ (v_N \ v_1 \ weight_{N,1}))$. В качестве контрольного веса бралась величина, равная $|V| \times MAX\_WEIGHT \ / \ 2 \ + \ (random \ (|V| \times MAX\_WEIGHT))$, где MAX_WEIGHT – число, на единицу превышающее максимальный вес ребра, то есть 10. В качестве списка контрольных ребер был взят пустой список. Это объясняется тем, что граф-цикл сам по себе является гамильтоновым циклом, и добавление проверочных ребер, которые генерируется случайным образом, с большой вероятностью приведут к отрицательному ответу. Ответ находится очень легко – цикл из любой вершины и сумма всех весов. Единственным критерием существования ответа является сверка с контрольным весом.

2.2 Граф-звезда

Граф-цикл представляет собой последовательность вида $((v_1 \ v_2 \ weight_{1,2}) \ (v_1 \ v_3 \ weight_{1,3}) \ ... \ (v_1 \ v_N \ weight_{1,N}))$. Контрольный вес был принят за $(random \ (|V| \times MAX\_WEIGHT))$. Контрольные ребра генерировались. Вес и ребра никак не влияют на ответ, т.к. он заведомо отрицательный из-за структуры данного вида графов.

2.3 Полный граф

Полный граф – это граф вида $((v_1\ v_2\ weight_{1,2})\ ...\ (v_1\ v_N\ weight_{1,N})\ (v_2\ v_N\ weight_{2,N})\ ...\ (v_{N-1}\ v_N\ weight_{N-1,N}))$. Контрольный вес такой же, как и в случае графа-цикла. Контрольные ребра генерировались. Ответ находится переборным алгоритмом. В полном графе заведомо есть гамильтонов цикл, влияние на ответ оказывают лишь контрольные вес и ребра.

2.4 Случайный граф

Случайный граф – это генерация различных ребер. Изначально их количество равняется $(|V| - 1) \times |V| / 2$, т.е. количеству ребер в полном графе. Но т.к. ребра генерируются случайно, и в список не добавляются ребра с одинаковыми вершинами и те, которые уже есть в списке, то итоговый граф будет не полным с очень большой вероятностью. Все входные параметры такие же, как и в случае полного графа. Ответ также находится переборным алгоритмом.

Всего было сгенерировано 150 различных тестовых наборов:

- 25 графов-циклов с количеством вершин от 3 до 27
- 15 графов-звезд с количеством вершин от 4 до 18
- 50 полных графов: по 5 на каждый граф с количеством вершин от 1 до 10
- 60 случайных графов: по 5 на каждый граф с количеством вершин от 1 до 12

Программа генерации тестов была написана на языке *Scheme*. Так же были написаны два скрипта на языке *Python* для автоматизирования генерации тестов и проверки работы генетического алгоритма. Первый скрипт создает файл, в котором отражается время генерации тестовых наборов для каждого вида графа. Второй скрипт в свою очередь создает файл в виде *html*-страницы с информацией о прохождении тестов (рис. 2.1): результат, количество пройденных и непройденных тестов, процент пройденных тестов, общее время работы, среднее количество итераций генетического алгоритма, количество итераций для каждого теста.

Tests	Passed	Failed	Results	Time (sec)	Avg. iter.
150	149	1	99.00 %	671	855

Test	Result	Iteration
cycle-test-v10	PASSED	1381
cycle-test-v11	PASSED	4411
cycle-test-v12	PASSED	5272
cycle-test-v13	PASSED	1817
cycle-test-v14	PASSED	504
cycle-test-v15	PASSED	4831
cycle-test-v16	PASSED	3384
cycle-test-v17	PASSED	505
cycle-test-v18	PASSED	1908
cycle-test-v19	PASSED	771
cycle-test-v20	PASSED	2773
cycle-test-v21	PASSED	1683
cycle-test-v22	PASSED	543
cycle-test-v23	PASSED	3294
cycle-test-v24	PASSED	1678
cycle-test-v25	PASSED	2155
cycle-test-v26	PASSED	6126
cycle-test-v27	FAILED	523
cycle-test-v3	PASSED	2
cycle-test-v4	PASSED	12
cycle-test-v5	PASSED	941
cycle-test-v6	PASSED	5148
cycle-test-v7	PASSED	147
cycle-test-v8	PASSED	4849
cycle-test-v9	PASSED	1132
full-test-v1-1	PASSED	2
full-test-v1-2	PASSED	2
full-test-v1-3	PASSED	2
full-test-v1-4	PASSED	2
full-test-v1-5	PASSED	2
full-test-v10-1	PASSED	549
full-test-v10-2	PASSED	105
full-test-v10-3	PASSED	531
full-test-v10-4	PASSED	97
full-test-v10-5	PASSED	344
full-test-v2-1	PASSED	2
full-test-v2-2	PASSED	2
full-test-v2-3	PASSED	2
full-test-v2-4	PASSED	2
full-test-v2-5	PASSED	2
full-test-v3-1	PASSED	2
full-test-v3-2	PASSED	2
full-test-v3-3	PASSED	2
full-test-v3-4	PASSED	2
full-test-v3-5	PASSED	2
full-test-v4-1	PASSED	8
full-test-v4-2	PASSED	14
full-test-v4-3	PASSED	17
full-test-v4-4	PASSED	4
full-test-v4-5	PASSED	2
full-test-v5-1	PASSED	92
full-test-v5-2	PASSED	21
full-test-v5-3	PASSED	447
full-test-v5-4	PASSED	277
full-test-v5-5	PASSED	11
full-test-v6-1	PASSED	258
full-test-v6-2	PASSED	54
full-test-v6-3	PASSED	334
full-test-v6-4	PASSED	610
full-test-v6-5	PASSED	142
full-test-v7-1	PASSED	53
full-test-v7-2	PASSED	197

Рис. 2.1: *html*-страница с результатами работы алгоритма

Глава 3

Генетический алгоритм

Алгоритм в большинстве своем основывается на исследовании, произведенном в статье [1]. Каждая особь представляет из себя некий гамильтонов цикл. Например для графа с 6-ю вершинами особью может быть последовательность $L = \{v1\ v4\ v5\ v6\ v3\ v2\}$, то есть геном является вершина графа. Начальная популяция представляет собой некоторое число сгенерированных случайным образом особей.

Далее происходит скрещивание популяции. Для каждой особи из популяции выбирается случайным образом еще одна особь. Потом они скрещиваются в 2 этапа:

1. Из второго родителя берется срез длиной, равной половине его длины, начиная с любой позиции, где этот срез возможен. Данный срез присваивается потомку в те же позиции, что и у родителя.

2. Оставшиеся гены заполняются из генов первого родителя: родитель просматривается справа налево, и в случае отсутствия гена у потомка, ген добавляется потомку слева направо.

Например, есть особь $L_1 = \{v1\ v2\ v3\ v4\ v5\ v6\}$. Допустим, в пару ей была выбрана особь $L_2 = \{v4\ v1\ v3\ v5\ v2\ v6\}$. Делается срез длины 3, начиная со 2-й позиции и помещается в потомка: $L_{1,2} = \{*\ v1\ v3\ v5\ *\ *\}$. Теперь берутся гены первого родителя: ген $v6$ есть в родителе и отсутствует в потомке, значит надо добавить – $L_{1,2} = \{v6\ v1\ v3\ v5\ *\ *\}$. Таким образом на выходе получим $L_{1,2} = \{v6\ v1\ v3\ v5\ v2\ v4\}$.

В результате скрещивания популяция увеличивается в 2 раза (2 родителя – 1 потомок). Но дальше происходит селекция: на следующий этап алгоритма проходит только определенная доля лучших (по значению фитнес-функции) особей из родительской и дочерней популяций. Причем сумма процентов для обоих видов особей равняется 100. Так как обе популяции равны, то общая популяция сократится в два раза и сравняется по количеству с начальной.

Худшая половина отобранных особей подвергается мутации с некоторой долей вероятности. Мутация представляет собой перестановку двух случайных генов местами. Например имеется особь $L_3 = \{v6\ v2\ v1\ v5\ v4\ v3\}$ и в качестве мутации первый ген меняется местом с четвертым, получается особь $L_3^M = \{v5\ v2\ v1\ v6\ v4\ v3\}$. Размер популяции не меняется.

В качестве целевой функции считается вес пути, причем если ребро отсутствует в графе или контрольное ребро отсутствует в пути, к значению функции добавляется штраф (заменяется на вес ребра в первом случае), равный длине пути, умноженной на увеличенный на единицу максимальный вес ребра. Штраф добавляется за каждое отсутствие, он однозначно отделит особи, которые не могут быть путями в графе, от потенциальных ответов. В конце, если значение функции превышает штраф, то ей прибавляется контрольный вес плюс один, чтобы особь не могла быть выбрана в качестве ответа. Например, дан граф $G = \{(v1\ v2\ 4)\ (v1\ v3\ 5)\}$, максимальный вес ребра – 9, контрольных ребер нет, контрольный вес – 50, вес особи $L = \{v2\ v3\ v1\}$ равняется $W_L = 3 \times 10 + 5 + 4 + 50 + 1 = 90$.

В качестве критериев остановки работы алгоритма является максимальное количество итераций и отличие среднего веса новой популяции от старой меньше, чем на некотрое δ.

В программе были использованы следующие параметры:

- Максимальный вес ребра – 9
- Размер популяции – 30
- Вероятность мутации особи – 40%
- Доля родителей в селекции – 60%
- Максимальное количество итераций – 10000
- Отличие весов популяций $\delta = 0.00001$

Глава 4

Визуализация

На данном этапе были использованы библиотеки The Racket Graphical Interface Toolkit, Plot, The Racket Drawing Toolkit. Пользователю предлагается выбрать вид графа, количество вершин в нем и все параметры для работы генетического алгоритма. Кроме того можно: убрать показ весов ребер, т.к. при их большом количестве трудно понять какой вес к какому ребру относится; задать генерацию контрольных ребер для работы алгоритма; установить скорость показа результатов (рис. 4.1). Все входные данные проверяются на корректность.

При нажатии на кнопку "Запустить алгоритм" генерируется граф, для него решается задача о поиске гамильтонова цикла минимального веса, меньше контрольного веса, включающего контрольные ребра. Алгоритм показывается по итерациям, выводя информацию о лучшей и худшей особях, а также среднем весом популяции (рис. 4.2). Также показывается статистика об изменении среднего веса популяции на каждой итерации (рис. 4.3).

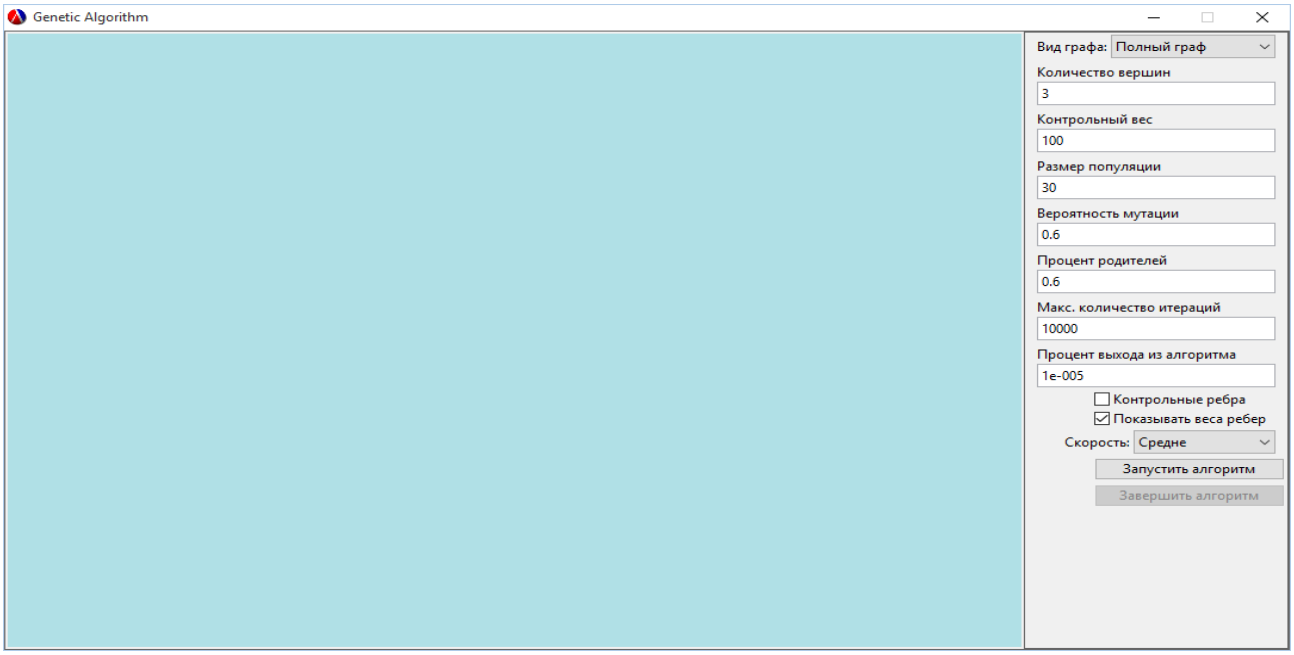

Рис. 4.1: Пример начального состояния окна

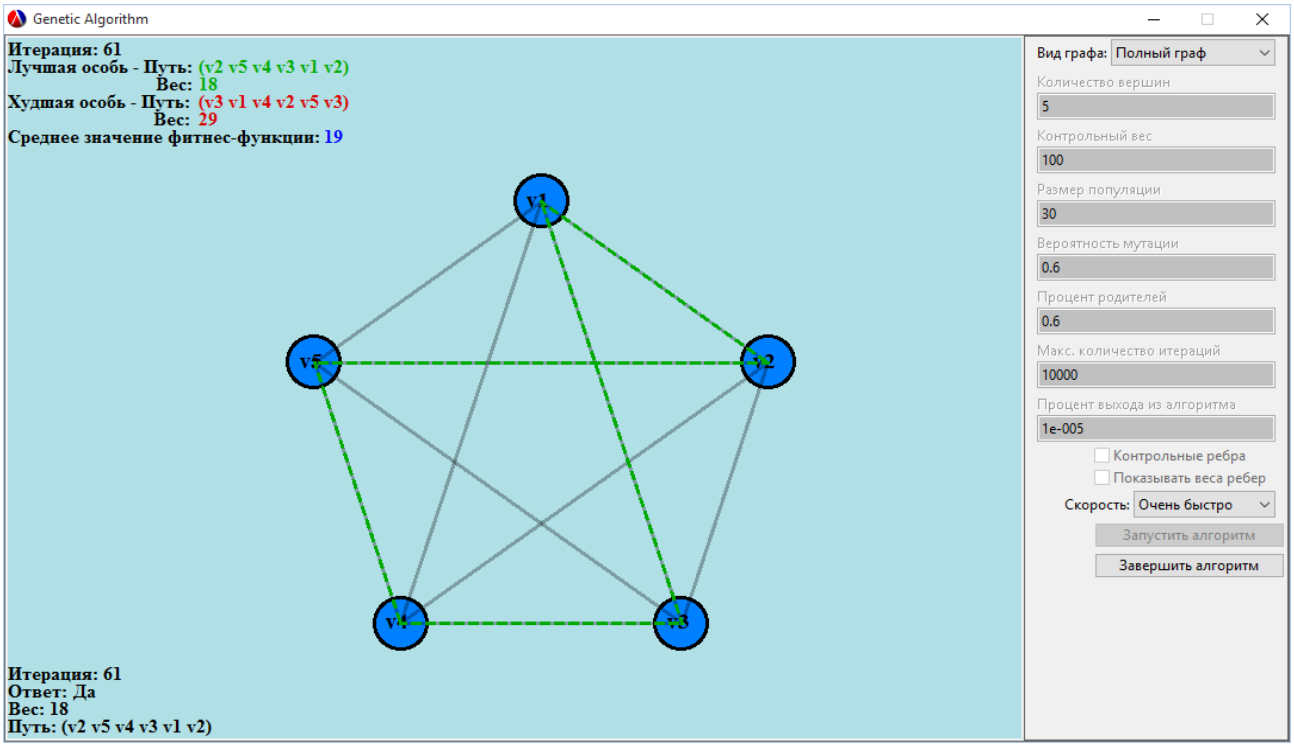

Рис. 4.2: Пример работы интерфейса. Зеленым выделена лучшая особь на данной итерации

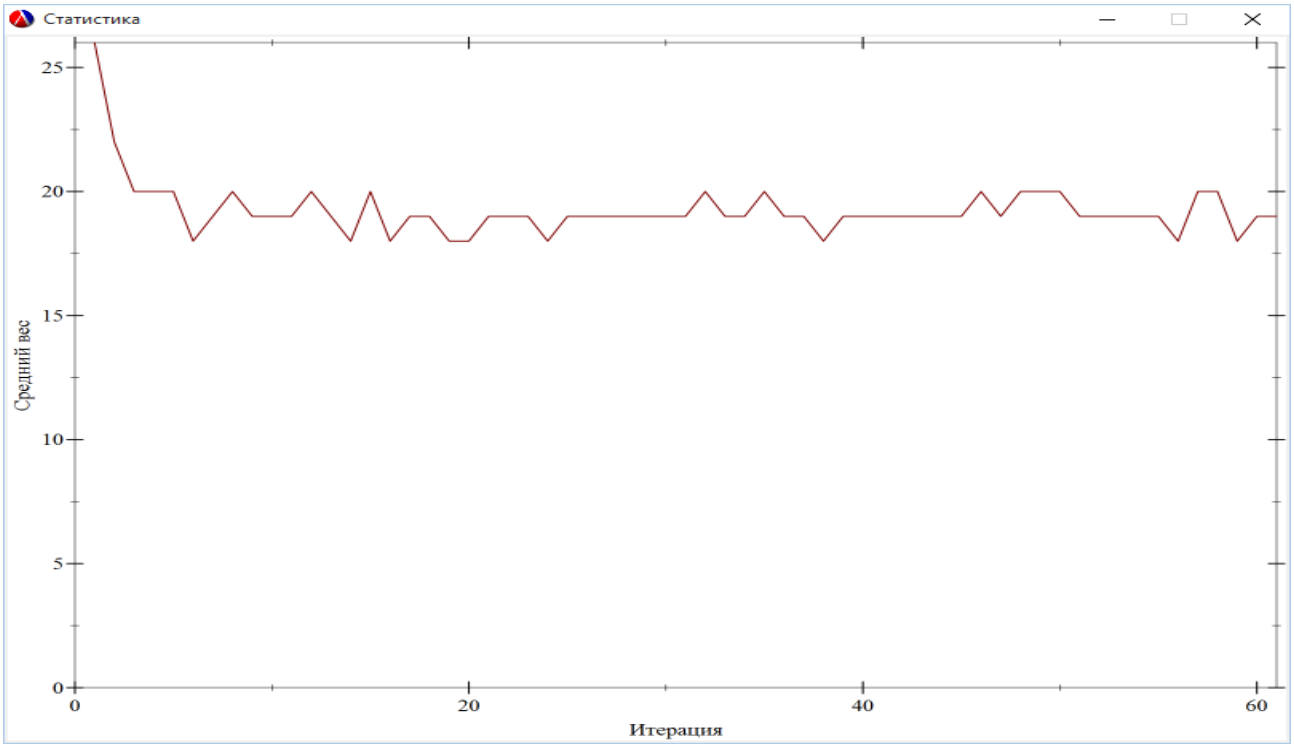

Рис. 4.3: Статистика

Глава 5

Результаты

Результаты алгоритма на описанных выше входных параметрах оказались следующими:

- Средний процент прохождения тестов $\approx 96\%$
- Среднее количество итераций алгоритма ≈ 700, а случаи выхода из алгоритма по максимальному количеству итераций единичны
- Среднее время работы алгоритма ≈ 9 минут на всех тестах, дольше всего алгоритм работает на графах-циклах (≈ 30 секунд при количестве вершин ≥ 20). Но тем не менее общее время гораздо меньше, чем решение задачи переборным алгоритмом (≈ 9 минут только на полных графах)

Данная реализация генетического алгоритма не находила верный ответ в основном для графов-циклов (были единичные случаи для полных и случайных графов, когда алгоритм находил решение, но оно не было лучшим, то есть в графе существовал гамильтонов цикл с меньшим весом). Это объясняется тем, что при большом количестве вершин очень мала вероятность генерации случайным образом графа-цикла. В дальнейшем при скрещивании и мутации может случиться так, что все графы в популяции как не были циклами, так и остались, то есть средний все популяции не изменился, тогда срабатывает остановка алгоритма. Вероятность этого растет с увеличением числа вершин в графе. Данная проблема решается, например, увеличением числа особой до 50, но тогда возрастает время работы алгоритма.

Заключение

Реализованная версия генетического алгоритма решает поставленную задачу с высокой точностью. Также алгоритм выигрывает у решения перебором по времени. Это достигается за счет значительного сокращения рассматриваемых потенциальных ответов. Увеличение числа вершин в графе несильно влияет на время работы генетического алгоритма, в то время как у переборного оно растет по экспоненте. Так как задача сельского почтальона (как и похожая задача коммивояжера) принадлежит классу NP, то можно считать, что генетический алгоритм успешно с ней справился.

Приложение. Код программы

Листинг 5.1: Генерация тестов

```
#lang scheme/base
(define (gen-vertex-name n)
  (string->symbol (string-append "v" (number->string n))))

(define (weight)
  (random 10))

(define folder "tests/")

(define (inc x)
  (+ x 1))

(define (push lst elem)
  (if (not (member elem lst)) (cons elem lst) lst))

(define (all-vertex graph)
  (define (helper graph result)
    (if (null? graph) result
        (helper (cdr graph) (push (push result (caar graph)) (cadar graph)))))
  (helper graph '()))

(define (vertex-neighbors vertex graph)
  (define (helper edge)
    (cond ((and (equal? (car edge) vertex) (not (equal? (cadr edge) vertex)))
       (cons (cadr edge) (caddr edge)))
          ((and (equal? (cadr edge) vertex) (not (equal? (car edge) vertex)))
             (cons (car edge) (caddr edge)))
          (else '())))
  (if (not (member vertex (all-vertex graph))) '()
      (filter (lambda (x) (not (null? x))) (map helper graph))))

(define (solver graph control-weight edge-list)
  (define (find-hamiltonian-cycle start visited neighbors weight control-edges)
    (if (null? neighbors) '()
        (if (and (assoc start neighbors) (eq? (length (all-vertex graph))
           (length visited)))
            (let ((answer (list (reverse (cons start visited)) (+ weight (cdr
               (assoc start neighbors))))))
              (if (and (null? (remove (list start (car visited))
```

```
                                        (remove (list (car visited) start)
                                            control-edges))) (not (> (cadr answer)
                                            control-weight))) answer '()))
                (if (member (caar neighbors) visited)
                    (find-hamiltonian-cycle start visited (cdr neighbors) weight
                        control-edges)
                    (let ((ans1 (find-hamiltonian-cycle start
                                                        (cons (caar neighbors)
                                                            visited)
                                                        (vertex-neighbors (caar
                                                            neighbors) graph)
                                                        (+ weight (cdar neighbors))
                                                        (remove (list (caar
                                                            neighbors) (car visited))
                                                                (remove (list (car
                                                                    visited) (caar
                                                                    neighbors))
                                                                    control-edges))))
                          (ans2 (find-hamiltonian-cycle start visited (cdr
                              neighbors) weight control-edges)))
                      (cond ((null? ans1) ans2)
                            ((null? ans2) ans1)
                            ((< (cadr ans1) (cadr ans2)) ans1)
                            (else ans2)))))))

  (if (or (null? graph) (< control-weight 0)) #f
      (let ((v-list (all-vertex graph)))
        (if (null? (cdr v-list)) (list #t 0 v-list)
            (let ((answer (find-hamiltonian-cycle (car v-list) (list (car
                v-list)) (vertex-neighbors (car v-list) graph) 0 edge-list)))
              (if (not (null? answer))
                  (list #t (cadr answer) (car answer)) #f))))))

(define (gen-test-edges num-of-vertex)
  (define (helper iters result)
    (if (= iters 0) result
        (let* ((n1 (inc (random num-of-vertex))) (n2 (inc (random
            num-of-vertex))) (new (list (gen-vertex-name n1) (gen-vertex-name
            n2))))
          (if (or (= n1 n2) (member new result) (member (reverse new) result))
              (helper iters result)
              (helper (- iters 1) (cons new result))))))
  (helper (random (max (+ (quotient num-of-vertex 2) 1) 1)) '()))

(define (gen-cycle n)
  (define (helper n result)
    (if (= n 1) result
        (helper (- n 1) (cons (list (gen-vertex-name (- n 1)) (gen-vertex-name
            n) (weight)) result))))
  (if (> n 2)
      (helper n (list (list (gen-vertex-name n) (gen-vertex-name 1) (weight))))
          '()))

(define (gen-star n)
  (define (helper n result)
    (if (= n 1) result
```

```
        (helper (- n 1) (cons (list (gen-vertex-name 1) (gen-vertex-name n)
            (weight)) result))))
  (if (> n 3)
      (helper n '()) '()))

(define (gen-full n)
  (define (helper n result)
    (if (= n 1) result
        (helper (- n 1) (foldr cons result (map (lambda (x) (list
            (gen-vertex-name (inc x)) (gen-vertex-name n) (weight))) (build-list
            (- n 1) values))))))
  (cond ((= n 1) (list (list (gen-vertex-name 1) (gen-vertex-name 1) (weight))))
        ((> n 0) (helper n '()))))

(define (gen-random v-count)
  (define (push elem list)
    (if (and (not (equal? (car elem) (cadr elem)))
             (not (assoc (cadr elem) (vertex-neighbors (car elem) list)))
             (not (assoc (car elem) (vertex-neighbors (cadr elem) list))))
        (cons elem list)
        list))

  (define (add-single graph)
    (define all (all-vertex graph))
    (foldr cons graph
           (map (lambda (x) (list (gen-vertex-name x) (gen-vertex-name x)
               (weight)))
             (filter (lambda (x) (not (member (gen-vertex-name x) all)))
                (build-list v-count inc)))))

  (define (helper e-c v-c result)
    (if (= e-c 0)
        (if (= (length (all-vertex result)) v-c) result (add-single result))
        (helper (- e-c 1) v-c (push (list (gen-vertex-name (inc (random v-c)))
            (gen-vertex-name (inc (random v-c))) (weight)) result))))
  (if (> v-count 0)
      (helper (/ (* (- v-count 1) v-count) 2) v-count '()) '()))


(define (gen-cycle-tests count)
  (define num-of-vertex (+ count 2))
  (define (summ-of-cycle cycle)
    (foldl + 0 (map caddr cycle)))

  (let ((cycle (gen-cycle num-of-vertex)) (w (+ (* num-of-vertex 5) (random (*
     num-of-vertex 10)))))
    (call-with-output-file (string-append folder "cycle-test-v" (number->string
       num-of-vertex) ".in")
      (lambda (out)
        (begin
          (writeln cycle out)
          (writeln w out)
          (writeln '() out)))
      #:exists 'truncate)
    (call-with-output-file (string-append folder "cycle-test-v" (number->string
       num-of-vertex) ".out")
      (lambda (out)
```

```
        (if (< w (summ-of-cycle cycle))
            (write #f out)
            (begin
              (writeln #t out)
              (writeln (summ-of-cycle cycle) out)
              (write (cons  (gen-vertex-name 1) (map cadr cycle)) out))))
      #:exists 'truncate))
  (if (> count 1)
      (gen-cycle-tests (- count 1)) '()))

(define (gen-star-tests count)
  (define num-of-vertex (+ count 3))
  (call-with-output-file (string-append folder "star-test-v" (number->string
     num-of-vertex) ".in")
    (lambda (out)
      (begin
        (writeln (gen-star num-of-vertex) out)
        (writeln (random (* num-of-vertex 10)) out)
        (write (gen-test-edges num-of-vertex) out)))
    #:exists 'truncate)
  (call-with-output-file (string-append folder "star-test-v" (number->string
     num-of-vertex) ".out")
    (lambda (out)
      (write #f out))
    #:exists 'truncate)
  (if (> count 1)
      (gen-star-tests (- count 1)) '()))

(define (gen-full-tests count)
  (for ([i (build-list 5 inc)])
    (let ((full (gen-full count)) (weight (+ (* count 5) (random (* count
       10)))) (edges (gen-test-edges count)))
      (call-with-output-file (string-append folder "full-test-v"
         (number->string count) "-" (number->string i) ".in")
        (lambda (out)
          (begin
            (writeln full out)
            (writeln weight out)
            (writeln edges out)))
        #:exists 'truncate)
      (call-with-output-file (string-append folder "full-test-v"
         (number->string count) "-" (number->string i) ".out")
        (lambda (out)
          (let ((answer (solver full weight edges)))
            (if answer
                (begin
                  (writeln (car answer) out)
                  (writeln (cadr answer) out)
                  (write (caddr answer) out))
                (write #f out))))
        #:exists 'truncate)
      ))
  (if (> count 1)
      (gen-full-tests (- count 1)) '()))

(define (gen-random-tests count)
  (for ([i (build-list 5 inc)])
```

```
    (let ((rnd (gen-random count)) (weight (+ (* count 5) (random (* count
       10)))) (edges (gen-test-edges count)))
      (call-with-output-file (string-append folder "random-test-v"
         (number->string count) "-" (number->string i) ".in")
        (lambda (out)
          (begin
            (writeln rnd out)
            (writeln weight out)
            (writeln edges out)))
        #:exists 'truncate)
      (call-with-output-file (string-append folder "random-test-v"
         (number->string count) "-" (number->string i) ".out")
        (lambda (out)
          (let ((answer (solver rnd weight edges)))
            (if answer
                (begin
                  (writeln (car answer) out)
                  (writeln (cadr answer) out)
                  (write (caddr answer) out))
                (write #f out))))
        #:exists 'truncate)
      ))
  (if (> count 1)
      (gen-random-tests (- count 1)) '()))

(define (gen-tests)
  (call-with-output-file "testsLog.txt"
    (lambda (out)
      (define alltime (current-seconds))
      (define timeC (current-seconds))
      (gen-cycle-tests 25) ;25
      (display "CYCLE-TESTS-GENERATED, TIME ELAPSED (sec): " out)
      (displayln (- (current-seconds) timeC)out)
      (define timeS (current-seconds))
      (gen-star-tests 15)  ;25 + 15 = 40
      (display "STAR-TESTS-GENERATED, TIME ELAPSED (sec): " out)
      (displayln (- (current-seconds) timeS)out)
      (define timeF (current-seconds))
      (gen-full-tests 10)   ;40 + 5 * 10 = 90
      (display "FULL-TESTS-GENERATED, TIME ELAPSED (sec): " out)
      (displayln (- (current-seconds) timeF)out)
      (define timeR (current-seconds))
      (gen-random-tests 12) ;90 + 5 * 12 = 150
      (display "RANDOM-TESTS-GENERATED, TIME ELAPSED (sec): " out)
      (displayln (- (current-seconds) timeR)out)
      (display "ALL-TESTS-GENERATED, TIME ELAPSED (sec): " out)
      (display (- (current-seconds) alltime)out))
    #:exists 'truncate))

(gen-tests)
```

Листинг 5.2: Генетический алгоритм

```
#lang scheme/base
(define pop-size 30)
(define mutate-prob 0.6)
(define par-percent 0.6)
(define max-iter 10000)
```

```
(define exit-percent 0.00001)

(define (genetic-algorithm graph
                           control-weight
                           control-edges
                           pop-size
                           mutate-prob
                           par-percent
                           max-iter
                           exit-percent)
  (define daughter-percent (- 1 par-percent))

  (define (push lst elem)
    (if (not (member elem lst)) (cons elem lst) lst))

  (define (all-vertex graph)
    (define (helper graph result)
      (if (null? graph) result
          (helper (cdr graph) (push (push result (caar graph)) (cadar graph)))))
    (helper graph '()))

  (define (pick-random lst)
    (list-ref lst (random (length lst))))

  (define (slice lst start count)
    (define (get-n-items lst num)
      (if (and (> num 0) (not (null? lst)))
          (cons (car lst) (get-n-items (cdr lst) (- num 1))) '()))
    (if (> start 0)
        (slice (cdr lst) (- start 1) count)
        (get-n-items lst count)))

  (define (path-weight path)
    (define penalty (* (length path) 10))
    (define (find-weight edge)
      (define (helper x)
        (if (or (and (equal? (car x) (car edge)) (equal? (cadr x) (cdr edge)))
                (and (equal? (car x) (cdr edge)) (equal? (cadr x) (car edge))))
            (if (equal? (car edge) (cdr edge)) 0 (caddr x)) '()))
      (filter (lambda (x) (not (null? x))) (map helper graph)))

    (define (helper path result)
      (if (null? (cdr path)) result
          (let ((w (find-weight (cons (car path) (cadr path)))))
            (if (null? w)
                (helper (cdr path) (+ penalty result))
                (helper (cdr path) (+ (car w) result))))))
    (helper (cons (car path) (reverse path)) 0))

  (define (fitness-function individ)
    (define penalty (* (length individ) 10))
    (define (check-edges path)
      (define (find-next y)
        (cadr (member y path)))
      (define (find-control x)
        (ormap equal? (map find-next x) (reverse x)))
```

```
    (* (length (filter (lambda (z) (not z)) (map find-control
       control-edges))) penalty))
  (let ((res (+ (path-weight individ) (check-edges (cons (car individ)
     (reverse individ))))))
    (if (> res penalty)
        (+ res control-weight 1)
        res)))

(define (fitness-pop population)
  (define (results x)
    (cons x (fitness-function x)))
  (sort (map results population) > #:key (lambda (x) (cdr x))))

(define (selection par-pop daughter-pop)
  (let ((par-threshold (- (length par-pop)(round (* (length par-pop)
     par-percent))))
        (daughter-threshold (- (length daughter-pop) (round (* (length
           daughter-pop) daughter-percent)))))
    (append (slice (map car (fitness-pop par-pop)) par-threshold (- (length
       par-pop) par-threshold))
            (slice (map car (fitness-pop daughter-pop)) daughter-threshold (-
               (length daughter-pop) daughter-threshold)))))

(define (gen-first-pop n graph)
  (define (fact n)
    (if (= n 0) 1
        (* n (fact (- n 1)))))

  (define (helper vertices n result)
    (define (gen-random-list vertices result)
      (if (null? vertices) result
          (let ((rnd (pick-random vertices)))
            (gen-random-list (remove rnd vertices) (cons rnd result)))))
    (if (= n 0) result
        (let ((current (gen-random-list vertices '())))
          (if (member current result)
              (helper vertices n result)
              (helper vertices (- n 1) (cons  current result))))))
  (let ((vertices (all-vertex graph)))
    (if (> n (fact (length vertices)))
        (helper vertices (length vertices) '())
        (helper vertices n '()))))

(define (cross-pop population)
  (define (cross par1 par2)
    (define (fill lst1 lst2 to from tmp)
      (cond ((or (null? lst2) (= (+ (length lst1) (length tmp)) (length
         par1))) (append tmp lst1))
            ((member (car lst2) lst1) (fill lst1 (cdr lst2) to from tmp))
            ((> to 0) (fill lst1 (cdr lst2) (- to 1) from (cons (car lst2)
               tmp)))
            ((> from 0) (fill (append lst1 (list (car lst2))) (cdr lst2) to
               (- from 1) tmp))))
    (let* ((n (quotient (length par1) 2)) (g (random (- (length par1) n -1)))
       (child (slice par2 g n)))
      (fill child (reverse par1) g (- (length par1) g n) '())))
```

```
    (define (helper pop result)
      (if (null? pop) result
          (helper (cdr pop) (cons (cross (car pop) (pick-random population))
             result))))
    (helper population '()))

  (define (mutate-pop population)
    (define (mutate individ)
      (define (swap ind1 ind2 lst)
        (append (slice lst 0 ind1) (list (list-ref lst ind2)) (slice lst (+
           ind1 1) (- ind2 ind1 1)) (list (list-ref lst ind1)) (slice lst (+
           ind2 1) (- (length lst) ind2 -1))))
      (define (gen-other-index index)
        (let ((new (random (length individ))))
          (if (= index new) (gen-other-index index) new)))
      (let* ((index1 (random (length individ))) (index2 (gen-other-index
         index1)))
        (if (> index2 index1)
            (swap index1 index2 individ)
            (swap index2 index1 individ))))

    (define (helper population result)
      (if (null? population) result
          (if (> (random) mutate-prob)
              (helper (cdr population) (cons (mutate (car population)) result))
              (helper (cdr population) (cons (car population) result)))))
    (let* ((sorted-pop (map car (fitness-pop population))) (half (quotient
       (length sorted-pop) 2)))
      (append (helper (slice sorted-pop 0 half) '()) (slice sorted-pop half (-
         (length sorted-pop) half)))))

  (define (population-weight population)
    (/ (foldl + 0 (map fitness-function population)) (length population) 1.0))

  (define (evolution population iter total-sum)
    (if (or (= iter max-iter) (= total-sum -1))
        (let ((answer (car (reverse (fitness-pop population)))))
          (if (> (cdr answer) control-weight) (list iter #f)
              (if (null? (cdar answer))
                  (list iter #t (cdr answer) (car answer))
                  (list iter #t (cdr answer) (cons (car (car answer)) (reverse
                     (car answer)))))))
        (let* ((new-pop (mutate-pop (selection population (cross-pop
           population))))
               (new-weight (population-weight new-pop)))
          (if (not (> (abs (- total-sum new-weight)) (* exit-percent
             total-sum)))
              (evolution new-pop (+ iter 1) -1)
              (evolution new-pop (+ iter 1) new-weight)))))
  (let ((first (gen-first-pop pop-size graph)))
    (evolution first 1 (population-weight first))))


(define (cmd-parser)
  (define (parse file)
    (call-with-input-file file
      (lambda (in)
```

```
        (list (read in) (read in) (read in)))))

  (define (display-list lst)
    (if (null? lst) '()
        (begin (displayln (car lst)) (display-list (cdr lst)))))

  (let ((cmd (vector->list (current-command-line-arguments))))
    (cond ((null? cmd) (display "EMPTY"))
          ((not (null? (cdr cmd))) (display "EXPECTED 1 ARGUMENT"))
          (else (let ((args (parse (car cmd))))
                  (let ((answer (genetic-algorithm (car args) (cadr args)
                     (caddr args) pop-size mutate-prob par-percent max-iter
                     exit-percent)))
                    (display-list answer)))))))

(cmd-parser)
```

Листинг 5.3: Генетический алгоритм

```
#lang scheme/base
(require racket/gui)
(require plot)

;genetic-algorithm
(define (genetic-algorithm graph
                           control-weight
                           control-edges
                           pop-size
                           mutate-prob
                           par-percent
                           max-iter
                           exit-percent)
  (define daughter-percent (- 1 par-percent))

  (define (push lst elem)
    (if (not (member elem lst)) (cons elem lst) lst))

  (define (all-vertex graph)
    (define (helper graph result)
      (if (null? graph) result
          (helper (cdr graph) (push (push result (caar graph)) (cadar graph)))))
    (helper graph '()))

  (define (pick-random lst)
    (list-ref lst (random (length lst))))

  (define (slice lst start count)
    (define (get-n-items lst num)
      (if (and (> num 0) (not (null? lst)))
          (cons (car lst) (get-n-items (cdr lst) (- num 1))) '()))
    (if (> start 0)
        (slice (cdr lst) (- start 1) count)
        (get-n-items lst count)))

  (define (path-weight path)
    (define penalty (* (length path) 10))
    (define (find-weight edge)
      (define (helper x)
```

```
      (if (or (and (equal? (car x) (car edge)) (equal? (cadr x) (cdr edge)))
              (and (equal? (car x) (cdr edge)) (equal? (cadr x) (car edge))))
          (if (equal? (car edge) (cdr edge)) 0 (caddr x)) '()))
    (filter (lambda (x) (not (null? x))) (map helper graph)))

  (define (helper path result)
    (if (null? (cdr path)) result
        (let ((w (find-weight (cons (car path) (cadr path)))))
          (if (null? w)
              (helper (cdr path) (+ penalty result))
              (helper (cdr path) (+ (car w) result))))))
  (helper (cons (car path) (reverse path)) 0))

(define (fitness-function individ)
  (define penalty (* (length individ) 10))
  (define (check-edges path)
    (define (find-next y)
      (cadr (member y path)))
    (define (find-control x)
      (ormap equal? (map find-next x) (reverse x)))
    (* (length (filter (lambda (z) (not z)) (map find-control
       control-edges))) penalty))
  (let ((res (+ (path-weight individ) (check-edges (cons (car individ)
     (reverse individ))))))
    (if (> res penalty)
        (+ res control-weight 1)
        res)))

(define (fitness-pop population)
  (define (results x)
    (cons x (fitness-function x)))
  (sort (map results population) > #:key (lambda (x) (cdr x))))

(define (selection par-pop daughter-pop)
  (let ((par-threshold (- (length par-pop)(round (* (length par-pop)
     par-percent))))
        (daughter-threshold (- (length daughter-pop) (round (* (length
           daughter-pop) daughter-percent)))))
    (append (slice (map car (fitness-pop par-pop)) par-threshold (- (length
       par-pop) par-threshold))
            (slice (map car (fitness-pop daughter-pop)) daughter-threshold (-
               (length daughter-pop) daughter-threshold)))))

(define (gen-first-pop n graph)
  (define (fact n)
    (if (= n 0) 1
        (* n (fact (- n 1)))))

  (define (helper vertices n result)
    (define (gen-random-list vertices result)
      (if (null? vertices) result
          (let ((rnd (pick-random vertices)))
            (gen-random-list (remove rnd vertices) (cons rnd result)))))
    (if (= n 0) result
        (let ((current (gen-random-list vertices '())))
          (if (member current result)
              (helper vertices n result)
```

```
            (helper vertices (- n 1) (cons  current result))))))
  (let ((vertices (all-vertex graph)))
    (if (> n (fact (length vertices)))
        (helper vertices (length vertices) '())
        (helper vertices n '()))))

(define (cross-pop population)
  (define (cross par1 par2)
    (define (fill lst1 lst2 to from tmp)
      (cond ((or (null? lst2) (= (+ (length lst1) (length tmp)) (length
          par1))) (append tmp lst1))
            ((member (car lst2) lst1) (fill lst1 (cdr lst2) to from tmp))
            ((> to 0) (fill lst1 (cdr lst2) (- to 1) from (cons (car lst2)
               tmp)))
            ((> from 0) (fill (append lst1 (list (car lst2))) (cdr lst2) to
               (- from 1) tmp))))

    (let* ((n (quotient (length par1) 2)) (g (random (- (length par1) n -1)))
       (child (slice par2 g n)))
      (fill child (reverse par1) g (- (length par1) g n) '())))

  (define (helper pop result)
    (if (null? pop) result
        (helper (cdr pop) (cons (cross (car pop) (pick-random population))
           result))))
  (helper population '()))

(define (mutate-pop population)
  (define (mutate individ)
    (define (swap ind1 ind2 lst)
      (append (slice lst 0 ind1) (list (list-ref lst ind2)) (slice lst (+
         ind1 1) (- ind2 ind1 1)) (list (list-ref lst ind1)) (slice lst (+
         ind2 1) (- (length lst) ind2 -1))))
    (define (gen-other-index index)
      (let ((new (random (length individ))))
        (if (= index new) (gen-other-index index) new)))
    (let* ((index1 (random (length individ))) (index2 (gen-other-index
       index1)))
      (if (> index2 index1)
          (swap index1 index2 individ)
          (swap index2 index1 individ))))

  (define (helper population result)
    (if (null? population) result
        (if (> (random) mutate-prob)
            (helper (cdr population) (cons (mutate (car population)) result))
            (helper (cdr population) (cons (car population) result)))))
  (let* ((sorted-pop (map car (fitness-pop population))) (half (quotient
     (length sorted-pop) 2)))
    (append (helper (slice sorted-pop 0 half) '()) (slice sorted-pop half (-
       (length sorted-pop) half)))))

(define (population-weight population)
  (/ (foldl + 0 (map fitness-function population)) (length population) 1.0))

(define (info population)
  (let ((sorted-pop (fitness-pop population)))
```

```
      (list (car sorted-pop) (car (reverse sorted-pop)) (quotient (foldl + 0
         (map cdr sorted-pop)) (length sorted-pop)))))

  (define (evolution population iter total-sum results)
    (if (or (= iter max-iter) (= total-sum -1))
        (let ((answer (car (reverse (fitness-pop population)))))
          (if (> (cdr answer) control-weight) (list results (list iter #f)
             graph)
              (if (null? (cdar answer))
                  (list results (list iter #t (cdr answer) (car answer)) graph)
                  (list results (list iter #t (cdr answer) (cons (car (car
                     answer)) (reverse (car answer)))) graph))))
        (let* ((new-pop (mutate-pop (selection population (cross-pop
           population))))
               (new-weight (population-weight new-pop)))
          (if (not (> (abs (- total-sum new-weight)) (* exit-percent
             total-sum)))
              (evolution new-pop (+ iter 1) -1 (append results (list (info
                 new-pop))))
              (evolution new-pop (+ iter 1) new-weight (append results (list
                 (info new-pop)))))))))
  (let ((first (gen-first-pop pop-size graph)))
    (evolution first 1 (population-weight first) (list (info first)))))

;gen-graphs
(define (gen-vertex-name n)
  (string->symbol (string-append "v" (number->string n))))

(define (weight)
  (random 10))

(define (inc x)
  (+ x 1))

(define (push lst elem)
  (if (not (member elem lst)) (cons elem lst) lst))

(define (all-vertex graph)
  (define (helper graph result)
    (if (null? graph) (reverse result)
        (helper (cdr graph) (push (push result (caar graph)) (cadar graph)))))
  (helper graph '()))

(define (vertex-neighbors vertex graph)
  (define (helper edge)
    (cond ((and (equal? (car edge) vertex) (not (equal? (cadr edge) vertex)))
       (cons (cadr edge) (caddr edge)))
          ((and (equal? (cadr edge) vertex) (not (equal? (car edge) vertex)))
             (cons (car edge) (caddr edge)))
          (else '())))
  (if (not (member vertex (all-vertex graph))) '()
      (filter (lambda (x) (not (null? x))) (map helper graph))))

(define (gen-test-edges num-of-vertex)
  (define (helper iters result)
    (if (= iters 0) result
```

```
        (let* ((n1 (inc (random num-of-vertex))) (n2 (inc (random
            num-of-vertex))) (new (list (gen-vertex-name n1) (gen-vertex-name
            n2))))
          (if (or (= n1 n2) (member new result) (member (reverse new) result))
              (helper iters result)
              (helper (- iters 1) (cons new result))))))
  (helper (random (max (+ (quotient num-of-vertex 2) 1) 1)) '()))

(define (gen-cycle n)
  (define (helper n result)
    (if (= n 1) result
        (helper (- n 1) (cons (list (gen-vertex-name (- n 1)) (gen-vertex-name
            n) (weight)) result))))
  (if (> n 2)
      (helper n (list (list (gen-vertex-name n) (gen-vertex-name 1) (weight))))
        '()))

(define (gen-star n)
  (define (helper n result)
    (if (= n 1) result
        (helper (- n 1) (cons (list (gen-vertex-name 1) (gen-vertex-name n)
            (weight)) result))))
  (if (> n 3)
      (helper n '()) '()))

(define (gen-full n)
  (define (helper n result)
    (if (= n 1) result
        (helper (- n 1) (foldr cons result (map (lambda (x) (list
            (gen-vertex-name (inc x)) (gen-vertex-name n) (weight))) (build-list
            (- n 1) values))))))
  (cond ((= n 1) (list (list (gen-vertex-name 1) (gen-vertex-name 1) (weight))))
        ((> n 0) (helper n '()))))

(define (gen-random v-count)
  (define (push elem list)
    (if (and (not (equal? (car elem) (cadr elem)))
             (not (assoc (cadr elem) (vertex-neighbors (car elem) list)))
             (not (assoc (car elem) (vertex-neighbors (cadr elem) list))))
        (cons elem list)
        list))

  (define (add-single graph)
    (define all (all-vertex graph))
    (foldr cons graph
           (map (lambda (x) (list (gen-vertex-name x) (gen-vertex-name x)
               (weight)))
             (filter (lambda (x) (not (member (gen-vertex-name x) all)))
                (build-list v-count inc)))))

  (define (helper e-c v-c result)
    (if (= e-c 0)
        (if (= (length (all-vertex result)) v-c) result (add-single result))
        (helper (- e-c 1) v-c (push (list (gen-vertex-name (inc (random v-c)))
            (gen-vertex-name (inc (random v-c))) (weight)) result))))
  (if (> v-count 0)
      (helper (/ (* (- v-count 1) v-count) 2) v-count '()) '()))
```

```
;gui
(define (gui)
  (define red (make-object color% 221 0 0))
  (define green (make-object color% 0 170 0))
  (define white (make-object color% 255 255 255))
  (define grey (make-object color% 192 192 192))
  (define blue (make-object color% 0 128 255))
  (define pic-height 600)
  (define pic-width 850)
  (define sphere 45)
  (define pop-size 30)
  (define mutate-prob 0.6)
  (define par-percent 0.6)
  (define max-iter 10000)
  (define exit-percent 0.00001)

  (define (check-pop-size)
    (define (helper value)
      (let ((val (string->number value)))
        (cond ((or (not val) (not (integer? val))) "целое число от 1 до 50")
              ((or (< val 1) (> val 50)) "от 1 до 50")
              (else '()))))
    (let ((check (helper (send i-pop-size get-value))))
      (if (not (null? check))
          (begin
            (send i-pop-size set-field-background red)
            (send i-pop-size set-value check)
            #f)
          (begin (send i-pop-size set-field-background green) #t))))

  (define (check-mutate-prob)
    (define (helper value)
      (let ((val (string->number value)))
        (cond ((or (not val) (not (real? val))) "вещ. число от 0 до 1")
              ((or (< val 0) (> val 1)) "от 0 до 1")
              (else '()))))
    (let ((check (helper (send i-mutate-prob get-value))))
      (if (not (null? check))
          (begin
            (send i-mutate-prob set-field-background red)
            (send i-mutate-prob set-value check)
            #f)
          (begin (send i-mutate-prob set-field-background green) #t))))

  (define (check-par-percent)
    (define (helper value)
      (let ((val (string->number value)))
        (cond ((or (not val) (not (real? val))) "вещ. число от 0 до 1")
              ((or (< val 0) (> val 1)) "от 0 до 1")
              (else '()))))
    (let ((check (helper (send i-par-percent get-value))))
      (if (not (null? check))
          (begin
            (send i-par-percent set-field-background red)
            (send i-par-percent set-value check)
            #f)
```

```
        (begin (send i-par-percent set-field-background green) #t))))

(define (check-max-iter)
  (define (helper value)
    (let ((val (string->number value)))
      (cond ((or (not val) (not (integer? val))) "целое число от 1 до 10000")
            ((or (< val 1) (> val 10000)) "от 1 до 10000")
            (else '()))))
  (let ((check (helper (send i-max-iter get-value))))
    (if (not (null? check))
        (begin
          (send i-max-iter set-field-background red)
          (send i-max-iter set-value check)
          #f)
        (begin (send i-max-iter set-field-background green) #t))))

(define (check-exit-percent)
  (define (helper value)
    (let ((val (string->number value)))
      (cond ((or (not val) (not (real? val))) "вещ. число от 0 до 1")
            ((or (< val 0) (> val 1)) "от 0 до 1")
            (else '()))))
  (let ((check (helper (send i-exit-percent get-value))))
    (if (not (null? check))
        (begin
          (send i-exit-percent set-field-background red)
          (send i-exit-percent set-value check)
          #f)
        (begin (send i-exit-percent set-field-background green) #t))))

(define (check-vertices-num)
  (define (helper type value)
    (let ((val (string->number value)))
      (cond ((or (not val) (not (integer? val))) "целое число")
            ((equal? type "Полный граф")
             (if (or (< val 1) (> val 10)) "от 1 до 10" '()))
            ((equal? type "Граф-звезда")
             (if (or (< val 4) (> val 25)) "от 4 до 25" '()))
            ((equal? type "Граф-цикл")
             (if (or (< val 3) (> val 25)) "от 3 до 25" '()))
            ((equal? type "Случайный граф")
             (if (or (< val 1) (> val 12)) "от 1 до 12" '())))))
  (let ((check (helper (send graph-type get-string-selection) (send
     vertices-num get-value))))
    (if (not (null? check))
        (begin
          (send vertices-num set-field-background red)
          (send vertices-num set-value check)
          #f)
        (begin (send vertices-num set-field-background green) #t))))

(define (check-control-weight)
  (define (helper value)
    (let ((val (string->number value)))
      (cond ((or (not val) (not (integer? val))) "целое число от 1 до 1000")
            ((or (< val 1) (> val 1000)) "от 1 до 1000")
            (else '()))))
```

```
  (let ((check (helper (send control-weight get-value))))
    (if (not (null? check))
        (begin
          (send control-weight set-field-background red)
          (send control-weight set-value check)
          #f)
        (begin (send control-weight set-field-background green) #t))))

(define (check-info)
  (member #f (list (check-vertices-num)
                   (check-control-weight)
                   (check-pop-size)
                   (check-mutate-prob)
                   (check-par-percent)
                   (check-max-iter)
                   (check-exit-percent))))

(define (gen-graph num)
  (let ((type (send graph-type get-string-selection)))
    (cond ((equal? type "Полный граф") (gen-full num))
          ((equal? type "Граф-звезда") (gen-star num))
          ((equal? type "Граф-цикл") (gen-cycle num))
          ((equal? type "Случайный граф") (gen-random num)))))

(define (need-edges v-count)
  (if (send gen-edges get-value)
      (gen-test-edges v-count) '()))

(define (get-coords vertices alpha coef coords radius type)
  (define (helper lst alpha coef)
    (if (null? lst) '()
        (cons (cons (car lst) (cons (+ (car coords) (* radius (sin
           (degrees->radians alpha)))) (- (cdr coords) (* radius (cos
           (degrees->radians alpha))))))
              (helper (cdr lst) (+ alpha coef) coef))))
  (if type
      (helper vertices alpha coef)
      (cons (cons (car vertices) (cons (car coords) (cdr coords))) (helper
         (cdr vertices) alpha coef))))

(define (visualize type result graph v-coords coords radius control-edges spd)
  (define (draw-answer ans)
    (send dc draw-text (string-append "Итерация: " (number->string (car
       ans))) 0 535)
    (send dc draw-text "Ответ: " 0 550)
    (if (cadr ans)
        (begin
          (send dc draw-text "Да" 55 550)
          (send dc draw-text (string-append "Вес: " (number->string (caddr
             ans))) 0 565)
          (send dc draw-text (string-append "Путь: " (~a (cadddr ans))) 0
             580))
        (send dc draw-text "Нет" 55 550)))

  (define (make-full-path path)
    (cons (car path) (reverse path)))
```

```
(define (get-stats st)
  (define (helper stats iter results)
    (if (null? stats) (reverse results)
        (helper (cdr stats) (+ iter 1) (cons (list iter (caddar stats))
           results))))
  (helper st 1 '()))

(define (draw stats iter)
  (define (draw-graph)
    (define (draw-edges graph)
      (if (null? graph) '()
          (if (not (equal? (caar graph) (cadar graph)))
              (begin
                (let* ((c1 (cdr (assoc (caar graph) v-coords))) (c2 (cdr
                   (assoc (cadar graph) v-coords))) (rnd (quotient (round
                   (abs (- (cdr c2) (cdr c1)))) 15)))
                  (send dc set-pen "black" 3 'hilite)
                  (send dc draw-line (+ (car c1) (/ sphere 2)) (+ (cdr c1)
                     (/ sphere 2)) (+ (car c2) (/ sphere 2)) (+ (cdr c2) (/
                     sphere 2)))
                  (if (send show-weight get-value)
                      (send dc draw-text (number->string (caddar graph)) (-
                         (/ (+ (car c1) (car c2) sphere) 2) rnd) (- (/ (+
                         (cdr c1) (cdr c2) sphere) 2) rnd)) '())
                  (send dc set-pen "black" 3 'solid))
                (draw-edges (cdr graph))) '())))

    (define (draw-path path)
      (if (or (null? (cdr path)) (equal? (car path) (cadr path))) '()
          (begin
            (let ((c1 (cdr (assoc (car path) v-coords))) (c2 (cdr (assoc
               (cadr path) v-coords))))
              (send dc set-pen green 3 'short-dash)
              (send dc draw-line (+ (car c1) (/ sphere 2)) (+ (cdr c1) (/
                 sphere 2)) (+ (car c2) (/ sphere 2)) (+ (cdr c2) (/ sphere
                 2)))
              (send dc set-pen "black" 3 'solid))
            (draw-path (cdr path)))))

    (define (draw-vertices v-coords)
      (if (null? v-coords) '()
          (begin
            (send dc draw-ellipse (cadar v-coords) (cddar v-coords) sphere
               sphere)
            (send dc set-font (make-object font% 14 'roman 'normal 'bold))
            (send dc draw-text (symbol->string (caar v-coords)) (+ (cadar
               v-coords) 10) (- (cddar v-coords) -10))
            (draw-vertices (cdr v-coords)))))
    (send dc set-brush blue 'xor)
    (send dc set-pen "black" 3 'solid)
    (draw-vertices v-coords)
    (draw-edges graph)
    (draw-path (make-full-path (caadar stats))))
  (if (null? stats) '()
      (begin
        (send dc clear)
        (send dc set-font (make-object font% 12 'roman 'normal 'bold))
```

```
            (draw-answer (cadr result))
            (if (not (null? control-edges))
                (send dc draw-text (string-append "Контрольные рёбра: " (~a
                   control-edges)) 0 90) '())
            (send dc draw-text (string-append "Итерация: " (number->string
               iter)) 0 0)
            (send dc draw-text "Лучшая особь - Путь: " 0 15)
            (send dc set-text-foreground green)
            (send dc draw-text (~a (make-full-path (caadar stats))) 160 15)
            (send dc set-text-foreground "black")
            (send dc draw-text "Вес: " 124 30)
            (send dc set-text-foreground green)
            (send dc draw-text (number->string (cdadar stats)) 160 30)
            (send dc set-text-foreground "black")
            (send dc draw-text "Худшая особь - Путь: " 0 45)
            (send dc set-text-foreground red)
            (send dc draw-text (~a (make-full-path (caaar stats))) 160 45)
            (send dc set-text-foreground "black")
            (send dc draw-text " Вес: " 118 60)
            (send dc set-text-foreground red)
            (send dc draw-text (number->string (cdaar stats)) 160 60)
            (send dc set-text-foreground "black")
            (send dc draw-text "Среднее значение фитнес-функции: " 0 75)
            (send dc set-text-foreground "blue")
            (send dc draw-text (number->string (caddar stats)) 265 75)
            (send dc set-text-foreground "black")
            (draw-graph)
            (send picture set-label target)
            (if (not (null? (cdr stats)))
                (sleep spd) '())
            (draw (cdr stats) (+ iter 1)))))
  (draw (car result) 1)
  (define targ (plot-bitmap (lines (get-stats (car result))) #:x-min 0
     #:y-min 0 #:x-label "Итерация" #:y-label "Средний вес" #:width 800
     #:height 600))
  (send pl set-label targ)
  (send ploting show #t)
  (send frame show #t))

(define (start-genetics)
  (map (lambda (x) (send x enable #f)
         (send x set-field-background grey)) (cdr inputs))
  (send gen-edges enable #f)
  (send show-weight enable #f)
  (send start enable #f)
  (send end enable #t)
  (send dc clear)
  (send dc set-font (make-object font% 18 'default))
  (send dc draw-text "Загрузка результатов..." 0 0)
  (send picture set-label target)
  (let* ((values (map (lambda (x) (string->number (send x get-value))) (cdr
     inputs)))
         (graph (gen-graph (car values))) (edges (need-edges (car values)))
         (all (all-vertex graph))
         (coords (cons (/ pic-width 2) (/ (+ pic-height 100 -65) 2)))
         (radius (/ (- pic-height 200) 2))
```

```
        (type (not (equal? (send graph-type get-string-selection)
           "Граф-звезда"))))
    (visualize type
               (genetic-algorithm graph (cadr values) edges (caddr values)
                  (cadddr values) (car (cddddr values)) (cadr (cddddr
                  values)) (caddr (cddddr values)))
               graph
               (get-coords all 0 (/ 360 (length all)) coords radius type)
               coords radius edges
               (- 2 (send speed get-selection)))))

(define (end-genetics)
  (send ploting show #f)
  (map (lambda (x) (send x enable #t)
         (send x set-field-background white)) (cdr inputs))
  (send ploting show #f)
  (send gen-edges enable #t)
  (send show-weight enable #t)
  (send dc clear)
  (send picture set-label target)
  (send start enable #t)
  (send end enable #f))

(define frame (new frame% [label "Genetic Algorithm"] [width 1024] [height
   600] [alignment (list 'right 'top)] [style (list 'no-resize-border)]))
(define main-panel (new horizontal-panel% [parent frame] [style (list
   'border)] [alignment (list 'right 'top)] [stretchable-height #f]
   [stretchable-width #t]))
(define target (make-object bitmap% pic-width pic-height))
(define dc (new bitmap-dc% [bitmap target]))
(send dc set-background (make-object color% 176 224 230))
(send dc clear)
(define picture (new message% [label target] [parent main-panel] [auto-resize
   #f] [stretchable-height #t] [stretchable-width #t]))
(define info-panel (new vertical-panel% [parent main-panel] [style (list
   'border)] [alignment (list 'right 'top)] [stretchable-height #t]
   [stretchable-width #f]))
(define graph-type (new choice%
                        [label "Вид графа: "] [parent info-panel] [choices
                           (list  "Полный граф" "Граф-звезда" "Граф-цикл"
                           "Случайный граф")] [horiz-margin 10] [min-width
                           200] [stretchable-width #f]))
(define vertices-num (new text-field%
                          [label "Количество вершин"] [parent info-panel]
                             [init-value "3"] [style (list 'single
                             'vertical-label)] [horiz-margin 10] [min-width
                             200] [stretchable-width #f]))
(define control-weight (new text-field%
                            [label "Контрольный вес"] [parent info-panel]
                               [init-value "100"] [style (list 'single
                               'vertical-label)] [horiz-margin 10] [min-width
                               200] [stretchable-width #f]))
(define i-pop-size (new text-field%
                        [label "Размер популяции"] [parent info-panel]
                           [init-value (number->string pop-size)] [style
                           (list 'single 'vertical-label)] [horiz-margin 10]
                           [min-width 200] [stretchable-width #f]))
```

```
  (define i-mutate-prob (new text-field%
                             [label "Вероятность мутации"] [parent info-panel]
                                [init-value (number->string mutate-prob)]
                                [style (list 'single 'vertical-label)]
                                [horiz-margin 10] [min-width 200]
                                [stretchable-width #f]))
  (define i-par-percent (new text-field%
                             [label "Процент родителей"] [parent info-panel]
                                [init-value (number->string par-percent)]
                                [style (list 'single 'vertical-label)]
                                [horiz-margin 10] [min-width 200]
                                [stretchable-width #f]))
  (define i-max-iter (new text-field%
                          [label "Макс. количество итераций"] [parent
                             info-panel] [init-value (number->string max-iter)]
                             [style (list 'single 'vertical-label)]
                             [horiz-margin 10] [min-width 200]
                             [stretchable-width #f]))
  (define i-exit-percent (new text-field%
                              [label "Процент выхода из алгоритма"] [parent
                                 info-panel] [init-value (number->string
                                 exit-percent)] [style (list 'single
                                 'vertical-label)] [horiz-margin 10] [min-width
                                 200] [stretchable-width #f]))
  (define gen-edges (new check-box%
                         [label "Контрольные ребра"] [parent info-panel]
                            [min-width 160]))
  (define show-weight (new check-box%
                           [label "Показывать веса ребер"] [parent info-panel]
                              [value 1] [min-width 160]))
  (define speed (new choice%
                     [label "Скорость: "] [parent info-panel] [choices (list
                        "Медленно" "Средне" "Очень быстро")] [selection 1]
                        [horiz-margin 10] [min-width 140] [stretchable-width
                        #f]))
  (define inputs (list graph-type vertices-num control-weight i-pop-size
     i-mutate-prob i-par-percent i-max-iter i-exit-percent))
  (define start (new button% [parent info-panel] [label "Запустить алгоритм"]
     [min-width 160] [stretchable-width #f]
                     [callback (lambda (button event) (if (not (check-info))
                        (start-genetics) '()))]))
  (define end (new button% [parent info-panel] [label "Завершить алгоритм"]
     [enabled #f] [min-width 160] [stretchable-width #f]
                   [callback (lambda (button event) (end-genetics))]))
  (define ploting (new frame% [label "Статистика"] [style (list
     'no-resize-border)] [x 0] [y 0]))
  (define pl (new message% [label target] [parent ploting] [auto-resize #t]))
  (send frame show #t))
(gui)
```

Литература

[1] Малыхина М.П., Частикова В.А., Власов К.А. ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ РАБОТЫ МОДИФИЦИРОВАННОГО ГЕНЕТИЧЕСКОГО АЛГОРИТМА В ЗАДАЧАХ КОМБИНАТОРИКИ // Современные проблемы науки и образования. — 2013. — № 3 — С. 2—3.