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

Отчет по второму заданию

Вариант 10. Разрезание циклов вершинами

Выполнил:
Беззубиков Александр Александрович
группа 428

Москва, 2015

Оглавление

1. Постановка задачи

На вход программе подается описание невзвешенного ориентированного графа G = (V, E) в виде списка ребер. Вершины графа $v_i \in V$ представлены атомами, а описание каждого ребра имеет вид (FROM TO). Затем на вход программе подается число K. Нужно определить, существует ли в графе G такое подмножество вершин W (W ⊆ V) мощности не большей K, что каждый цикл в графе G содержит хотя бы одну вершину из множества W. Если такого подмножества вершин не существует, напечатать #f, а если такое подмножество вершин существует, напечатать #t, а затем список вершин, входящих в это подмножество. Из всех допустимых подмножеств выбрать подмножество минимальной мощности.

Пример входных данных:

((a b)(b c)(c d)(d a))
1

Пример печати результата:

#t
1
(a)

2. Генерация тестов

2.1 Описание тестовых случаев

В процессе тестирования использовались следующие тестовые наборы:

- Полный граф с числом вершин, определяемым пользователем
  - K = |V|-1, при этом ответ #t
  - K = |V|-2,при этом ответ #f
- Ациклический граф большого размера (|V| > 200)
- Граф, представляющий собой цикл и не имеющий никаких циклов, кроме самого себя, и никаких ребер и вершин, не входящих в этот цикл (|V| > 200)
- Графы, генерируемые случайным образом (|V| > 25)

Алгоритм построения полного графа тривиален – генерируются вершины, а затем каждая пара вершин (a, b) соединяется двумя ребрами: (a b) и (b a).

Общий размер одного тестового набора равен 104: полный граф с двумя значениями K, большие циклический и ациклический граф, 100 случайных графов.

2.2 Генерация циклического и ациклического графов

Сначала генерируется случайное число вершин N больше заданного заранее порога. Затем из каждой i-й вершины (0 <= i < N) строится ребро в вершину с индексом (i+1)mod N. Число K выбирается случайным образом на отрезке [1; N].

Процесс генерации ациклического графа отличается от построения циклического графа лишь тем, что для пары вершин с индексами (N-1, 0) строится не ребро (N-1 0), а ребро (0 N-1), тем самым гарантирую отсутствие цикла.

2.3 Результат генератора тестов

После генерации каждого графа для него вычисляется ответ методом полного перебора. При этом перебор множеств вершин происходит по возрастанию мощности этих множеств и останавливается при нахождении первого множества, удовлетворяющего условию, поэтому минимальность результирующего множества гарантирована.

По завершению генерации всех графов, каждый из них вместе со своими результатами записывается в следующем формате в файл:

(<входные данные в формате, описанном в главе 1>, <result, number>),

где result - #t или #f, number – мощность множества, вычисленного при генерации ответа. Само множество, найденное перебором, в файл не записывается, т.к. в общем случае у задачи может быть несколько равносильных ответов равной мощности. Для проверки самого множества существует отдельная функция.

2.4 Организация автоматизированного тестирования

Программе на вход подается имя файла, содержащего тесты. Для каждого теста программа вычисляет результат R = (result, number, A) с помощью генетического алгоритма и сравнивает его с результатом R_pred, записанном в файле с тестами. Если значения result в R и R_pred различаются, или процедура проверки множества A вернула #f (а это значит, что множество A не удовлетворяет условию задачи, проверку того, что |A| <= K, тоже выполняет эта процедура), то тест считается проваленным. В случае если ответ корректен, но number в R и R_pred различаются, то печатается предупреждение, что ответ не минимален.

По завершению тестирования выдается вердикт: если хотя бы один тест был провален, то печатается 'Fail!’ и список номеров проваленных тестов. Иначе печатается ‘Success!’ и список номеров тестов с ответами, не являющимися минимальными.

3. Генетический алгоритм

3.1 Предварительный поиск циклов

Поиск циклов в графе осуществляется классическим алгоритмом поиска в глубину и является самой трудной с точки зрения реализации в рамках функциональной парадигмы частью программы, т.к. сам алгоритм описан с точки зрения процедурной парадигмы.

Список циклов в графе вычисляется один раз перед запуском генетического алгоритма в силу своей сложности и временных затрат.

3.2 Хромосома

Хромосома представляет собой список вершин графа без повторений.

3.3 Оценочная функция

Значение оценочной функции на хромосоме X равно

$$fit(X) = \frac{good(X)}{|cycles|},$$

где good(X) равно числу таких циклов в графе, которые содержат хотя бы одну вершину из множества X, а |cycles| - общее число циклов в графе. Если fit(X)=1, то X является ответом (т.к. при создании начальной популяции гарантируется то, что размер каждой особи <= K, а операция скрещивания (см. далее) не может увеличить размер особи.

Также в целях увеличения числа минимальных множеств-результатов были испробованы следующие способы предобработки множества X:

1. Удаление вершин, не лежащих ни в одном цикле
2. Удаление вершин, чьи соответствующие множества циклов полностью вложены в множество циклов другой вершины из множества X

На основании результатов экспериментов способ 1 стал применяться уже к результату генетического алгоритма, а способ 2 применяется только к начальной популяции.

3.4 Создание начальной популяции

Начальная популяция представляет собой множество хромосом мощности POPUL_LEN=50 (подобрано на основе экспериментов), которое генерируется случайным образом. При этом размер каждой особи меньше или равен K.

3.5 Скрещивание

Применяется т.н. одноточечное скрещивание, т.е. результатом скрещивание родителей $X = \{x_1,\ldots,x_m\}$ и $Y=\{y_1, \ldots, y_n\}$ будут $Z_1=\{x_1,\ldots,x_c,y_{c+1},\ldots,y_n\}$ и

$Z_2=\{y_1,\ldots,y_c,x_{c+1},\ldots,x_m\}$, где точка с выбирается случайно на промежутке [1,min(m,n)). Пары выбираются случайно, но не пересекаются. Практически это реализуется путем использования *shuffle* перед началом скрещивания и взятием по 2 элемента из головы списка, передавая остаток дальше.

Как следует из сказанного выше, максимальная длина особей-потомков не может превышать длину родителей. Это дает возможность после создания начальной популяции нигде больше не проверять условие |X| <= K.

3.6 Селекция

После скрещивания наше популяция имеет размер 2*POPUL_LEN. Селекция заключается в сортировке особей по убыванию значения оценочной функции и взятии в качестве популяции для следующей итерации верхнюю половину этого отсортированного списка.

Таким образом, на каждой итерации размер начальной популяции равен POPUL_LEN.

3.7 Мутация

Мутации подвергаются все особи кроме лучшей четверти. Сам процесс мутации заключается в случайной замене компонент особи (с вероятностью 1/10) на случайные же вершины графа.

3.8 Критерий останова

Генетический алгоритм завершает свою работу только в двух случаях:

- найдена особь со значением fit(X)=1
- достигнуто максимальное число итераций (задается при начальной настройке алгоритма, по умолчанию равно 30)

В первом случае ответом являются особи из множества особей со значением оценочной функции, равным 1, минимальные по длине; во втором ответом является #f, т.к. иначе алгоритм бы завершился по первому условию.

3.9 Детали реализации

Всю основную работу выполняет функция *step-gen*, вызываемая из своей обертки *genetics-solve*. У *step-gen* в качестве параметров есть текущая популяция в виде списка вершин, множество всех циклов в графе в виде списка списков вершин, номер текущей итерации и вспомогательные данные для визуализации. Если в популяции нет особей со значением оценочной функции, равным 1, то производится скрещивание (функция ***reprod***, которая присоединяет к текущей популяции новых особей-детей), селекция (с помощью ***top*** *lst num*, которая берет *num* лучших по значению оценочной

функции элементов списка *lst*), мутация(***mutate***) и переход на новую итерацию, пока не выполнится критерий останова.

4. Визуализация

4.1 Визуализация решения

В силу специфики задачи, нет смысла визуализировать граф полностью. Так как важно лишь присутствие либо отсутствие вершин из некоторого множества циклов, то разумно визуализировать именно циклы, что и сделано в данном задании.

Каждый цикл из списка всех циклов графа визуализируется следующим образом:

- каждая вершина цикла лежит на окружности, радиус которой прямо пропорционален числу вершин; длины дуг между каждой парой соседних вершин одинаковы
- каждые две соседние вершины соединяются прямой

При этом если вершина входит в визуализируемое множество, то она закрашивается красным цветом во всех циклах, иначе – черным.

Кроме того, выводится результат алгоритма в виде простого списка вершин.

4.2 Визуализация процесса решения задачи

Процесс решения задачи визуализируется по итерациям, т.е. для каждой итерации алгоритма сохраняется визуализация лучшей особи этой итерации. Эта особь визуализируется точно так же, как и итоговое решение задачи.

4.3 Визуализация итоговой статистики

Кроме визуализации непосредственно решения предусмотрен сбор статистики по итерациям и итоговая визуализация собранных данных в виде графика.

На каждой итерации генетического алгоритма в специальные списки-параметры функции ***step-gen*** добавляются максимум, минимум и, соответственно, среднее значение оценочной функции популяции на данной итерации.

На графике по оси Ox отложены номера итераций, по Oy – значения оценочной функции.

4.4 Графический интерфейс пользователя

Визуализация выполняется в отдельном окне. Для выбора визуализируемых данных предусмотрен выпадающий список, в котором можно выбрать, что отрисовывать: итерацию с каким-либо номером или итоговый график. График и циклы поддерживают масштабирование и прокрутку.

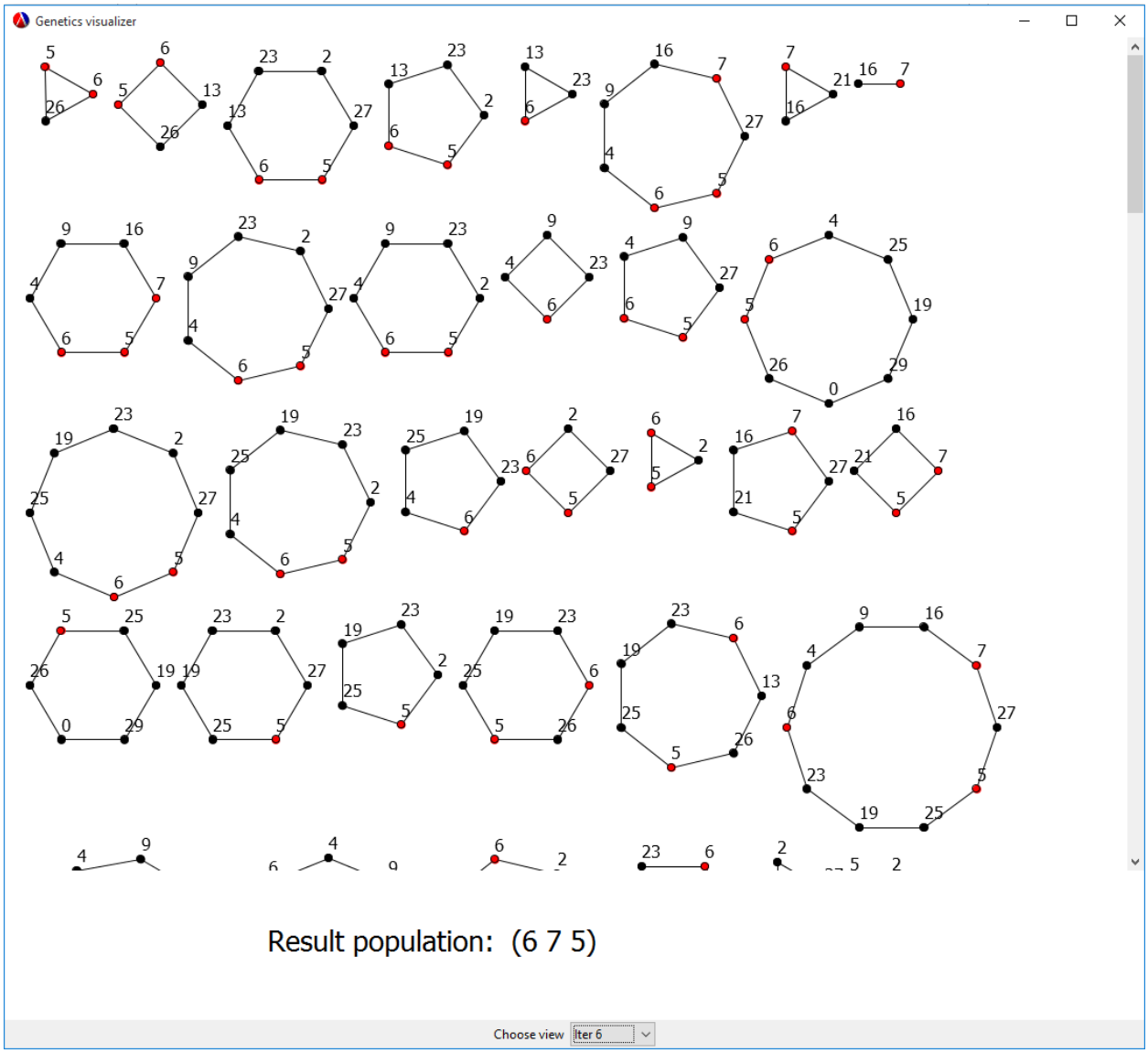

Рисунок 1. Пример визуализации - итоговая популяция

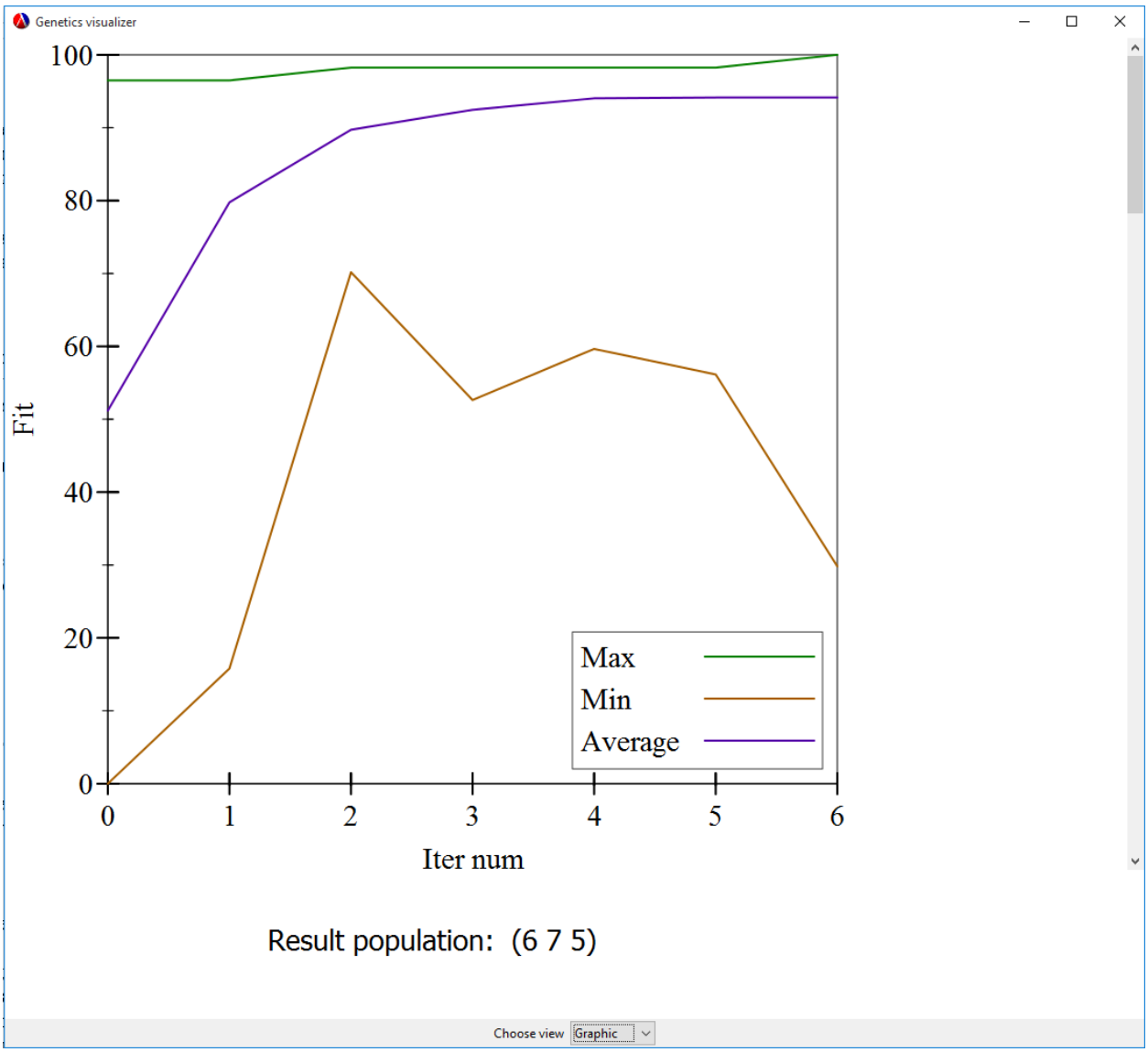

Рисунок 2. Пример визуализации – итоговая статистика

4.5 Детали реализации

Для создания графического интерфейса использовался пакет **racket/gui**, конкретно использовались элементы **choice**, **canvas**, **frame**.

Графики и циклы рисовались в **bitmap** из **racket/draw**, при этом в функции **step-gen** в качестве параметра есть список bitmap’ов, каждый элемент которого соответствует одной итерации генетического алгоритма. Функция **step-gen** возвращает список, одним из элементов которого как раз является список bitmap’ов. В процедуре визуализации при выборе итерации из выпадающего списка в **canvas** просто посылается **draw-bitmap** с нужным элементом этого списка в качестве параметра.

Для рисования графика использовался пакет **plot**, а именно **lines** и **points**.

5. Результаты

Представленный алгоритм был протестирован на наборах, описанных в главе 1 (причем на нескольких наборах случайных графов, чтобы повысить точность проверки). По результатам тестов можно сделать следующие выводы:

- Алгоритм работает корректно, т.е. всегда находит решение, если оно существует, множество-ответ действительно является решением
- Алгоритм не всегда находит минимальное решение: в среднем на 2 теста из 104 алгоритм выдает ответ, не являющийся минимальным по длине. Стоит отметить, что эти случаи относятся только к графам, генерируемым случайным образом. Приемы, описанные в разделе 3.1 не помогают полностью избавиться от этой проблемы, т.к. они не могут исправить начальную популяцию. Также наблюдаются наборы, на которых алгоритм работает абсолютно точно

Заключение

В данном отчете был описан алгоритм и особенности устройства программы решения задачи разрезания циклов вершинами с помощью генетического алгоритма на языке Scheme.

По результатам проделанной работы хочется отметить некоторые моменты:

- Генетический алгоритм хорошо подходит к данной задаче – производительность (особенно на больших графах) заметно выше алгоритма полного перебора, точность тоже достигается приемлемая, хоть и не всегда абсолютная с точки зрения минимальности
- Удобство функциональной парадигмы в данной конкретной задаче сомнительно, особенно это касается алгоритма поиска в глубину, написания графического интерфейса пользователя, да и общей структуры программы. Разбиение на отдельные модули не сильно спасло положение, т.к. код все равно имеет слабую читабельность (лично для меня). Предполагаю, что написание кода для решения той же задачи на C/C++ (и на других императивных языках, про Python и не говорю) потребовало бы минимум в 2 раза меньше времени и строк кода
- Общее число строк кода – 634(!), но здесь, наверное, стоит учесть индивидуальные особенности мышления

Приложение

Check-solution.rkt

```
#lang scheme/base
(require racket/list)
(require "solve.rkt")
(require "graph.rkt")
(require "common.rkt")
(require "genetics.rkt")

(define (answer-correct? answer graph)
  (let ((cycles (find-cycles graph)))
    (if (andmap(lambda(cycle)
                 (if (intersects? cycle answer) answer #f))
               cycles)
        #t
        #f)))

(define (read-single-test in)
  (let* ((test (list (read in) (read in))))
    (begin
      (if (eof-object? (car test))
          '()
          (append (read-single-test in) (list test))))))

(define (read-tests)
  (call-with-input-file "tests.txt" (lambda(in)(read-single-test in))))

(define (solve graph max)
  (genetics-solve graph max #f))
  ;(start-bruteforce graph max))

(define (run-test tests acc succ total failed)
  (if (null? tests)
      (begin
        (printf "Tests passed: ~v of ~v\n" succ (sub1 total))
```

```
        (printf "Failed: ~v\n" failed)
        (if (equal? #t acc) (printf "Success!\n") (printf "Fail!\n"))
        )
      (let* ((result (flatten (solve (caaar tests) (cdaar tests))))(eq (or (and
(equal? #f (car result)) (equal? #f (caadar tests))) (and (equal? (car result)
(caadar tests)) (equal? #t (answer-correct? result (caaar tests)))))))
        (begin
          (printf "Running test #~v\n" total)
          (if (equal? eq #t)
              (begin
                (printf "Passed\n")
                (printf "\n")
                (run-test (cdr tests) (and acc eq) (add1 succ) (add1 total) failed))
              (begin
                (printf "Failed\n\n")
                (run-test (cdr tests) (and acc eq) succ (add1 total) (cons total
failed))))
          ))
      ))

(define (check-solution)
  (run-test (read-tests) #t 0 1 '()))

(check-solution)
```

Common.rkt

```
#lang scheme/base

(provide comb)
(provide gen-random)
(provide pick-random)
(provide intersects?)
(provide full-in?)
(provide average)
```

```
(define (prob n1 n2)
  (if (= 0 n1)
      0
      (< (random n1) n2)))

(define (comb lst k)
  (cond
    ((zero? k) '(()))
    ((null? lst) '())
    (else (append
           (map (lambda (x) (cons (car lst) x)) (comb (cdr lst) (sub1 k)))
           (comb (cdr lst) k)))))

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

(define (intersects? lst1 lst2)
  (foldl (lambda (arg result) (or result (not (equal? #f (member arg lst2))))) #f
lst1))

(define (full-in? lst1 lst2)
  (and (<= (length lst1) (length lst2)) (if (andmap (lambda (arg) (member arg lst2))
lst1) #t #f)))

; same as random, but cannot return 0
(define (gen-random num)
  (let ((res (random num)))
    (if (= 0 res) (add1 res) res)))

(define (average lst)
  (if (null? lst)
      0
      (/ (foldl + 0 lst)
         (length lst))))
```

Genetics.rkt

```
#lang scheme/base


(require racket/list)

(require racket/draw)

(require racket/gui)

(require plot)


(require "common.rkt")

(require "graph.rkt")

(require "solve.rkt")

(require "visual.rkt")


(provide genetics-solve)


(define (start-popul k verts popul-len)

  (define (create-popul lst len vert)

    (if (= len 0)

        lst

        (if (member vert lst)

            (create-popul lst len (pick-random verts))

            (create-popul (cons vert lst) (- len 1) (pick-random verts)))

        ))

  (if (= popul-len 0)

      '()

      (cons (create-popul '() (gen-random (+ k 1)) (pick-random verts)) (start-popul
k verts (- popul-len 1)))))


(define (gen-out-verts verts cycles)

  (define (norm-step lst res int-lst)

    (define (check-vert arg)

      (foldl (lambda (arg1 result) (or result (and (not (equal? (car arg) (car
arg1)))(not (member (car arg1) res)) (full-in? (cdr arg) (cdr arg1))))) #f int-lst))

    (if (null? lst)

        res

        (norm-step (cdr lst)
```

```
                    (if (check-vert (car lst))
                        (cons (caar lst) res)
                        res)
                    int-lst)


        ))
  (let ((int-lst (map (lambda (arg) (cons arg (filter (lambda (arg1) (member arg arg1)) cycles))) verts)))
    (norm-step int-lst '() int-lst)))


(define (normalize verts cycles)
  (let ((out (gen-out-verts verts cycles))(in-cycles (remove-duplicates (flatten cycles))))
    (filter (lambda(arg)(not (member arg out))) verts)))


(define MAX_ITER 30)


(define (genetics-solve graph k enable-visual)
  (define (start-gen cycles)


    (define (top lst num)
      (define (top-step cur len)
        (if (= 0 len)
            '()
            (cons (car cur) (top-step (cdr cur) (- len 1)))))
      (top-step (sort lst > #:key fit) (floor num)))


    (define (bottom lst num)
      (define (bottom-step cur len)
        (if (= 0 len)
            '()
            (cons (car cur) (bottom-step (cdr cur) (- len 1)))))
      (bottom-step (sort lst < #:key fit) (floor num)))



    (define (fit1 lst)
```

```
      (define (fit-vert x)
        (filter (lambda (arg) (member x arg)) cycles))
      (if (null? cycles) 0
          (/ (length (remove-duplicates (foldl (lambda (arg res) (append res (fit-
vert arg))) '() lst))) (length cycles))))


    (define (fit2 lst)
      (- 1 (/ (length (filter (lambda (vert) (ormap (lambda(cycle) (member vert
cycle)) cycles)) lst))
              (length lst))))


    (define (fit3 lst)
      (let ((out (gen-out-verts lst cycles)))
        (/ (length out) (length cycles))))


    (define (fit lst)
      (* (fit1 lst) 100))


    (define (step-gen popul cycles iter dcs maxs mins avs)
      (define (mutate popul verts)
        (define (mutate-person lst)
          (if (member lst (bottom popul (/ (length popul) 4)))
              (remove-duplicates (map (lambda (arg) (if (= 0 (random 10)) (pick-
random verts) arg)) lst))
              lst))
        (define (mutate-step lst num)
          (if (< num 1)
              lst
              (cons (mutate-person (car lst)) (mutate-step (cdr lst) (- num 1)))
              ))
        (mutate-step (shuffle popul) (length popul))
        )


      (define (cr-ovr x y)
        (let* ((c (gen-random (min (length x) (length y)))))
          (list (remove-duplicates (append (drop-right x c) (take-right y c)))
```

```
                (remove-duplicates (append (drop-right y c) (take-right x c))))
          )
        )


      (define (reprod popul)
        (let ((popul (shuffle (if (odd? (length popul)) (cons (pick-random popul)
popul) popul))))
          (if (null? popul)
              '()
              (append (cr-ovr (car popul) (cadr popul)) (reprod (cddr popul))))))


      (if
       (null? cycles)
       (list (list #f))
       (let ((res (filter (lambda (arg) (= 100 (fit arg))) popul)))
         (if (and (null? res) (< 0 iter))
             (let* (
                    (new-popul (append popul (reprod popul)))
                    (surv (top (mutate (top new-popul (length popul)) (get-verts
graph)) (length popul)))
                    (best (fit (car surv)))
                    (worst (fit (list-ref surv (sub1 (length surv)))))
                    (mean (average (map fit surv)))
                    )
               (if (equal? #t enable-visual)
                   (let* ((new-dc (new bitmap-dc% (bitmap (make-bitmap FIELD_WIDTH
FIELD_HEIGHT)))) (draw (draw-graph new-dc (argmin length (filter (lambda(arg)(= (fit
arg) best)) surv)) cycles)) (iter-num (add1 (- MAX_ITER iter))))
                     (step-gen surv cycles (- iter 1) (cons new-dc dcs) (cons (list
iter-num best) maxs) (cons (list iter-num worst) mins) (cons (list iter-num mean)
avs)))
                   (step-gen surv cycles (- iter 1) dcs maxs mins avs))
               )
             (if (equal? #t enable-visual)
                 (let* (
                        ;(new-dc (new bitmap-dc% (bitmap (make-bitmap FIELD_WIDTH
FIELD_HEIGHT))))
                        ;(draw (draw-graph new-dc (argmin length res) cycles))
```

```
                        (res (if (null? res) (list (list #f)) res))
                        (max-p (if (= iter MAX_ITER) (list (list 0 100)) '()))
                        (min-p (if (= iter MAX_ITER) (list (list 0 (fit (list-ref
(top popul (length popul)) (sub1 (length popul)))))) '()))
                        (av-p (if (= iter MAX_ITER) (list (list 0 (average (map fit
popul)))) '()))
                        (graphic
                         (parameterize
                             (
                              (plot-x-far-ticks no-ticks)
                              (plot-y-far-ticks no-ticks)
                              (plot-x-ticks (linear-ticks #:divisors '(1) #:number
(add1 (- MAX_ITER iter))))
                              )
                           (new bitmap-dc% (bitmap
                                            (plot-bitmap (list (axes)
                                                               (lines maxs #:color 2
#:label "Max")
                                                               (lines mins #:color 4
#:label "Min")
                                                               (lines avs #:color 6
#:label "Average")
                                                               (points max-p #:color
2)
                                                               (points min-p #:color
4)
                                                               (points av-p #:color
6)
                                                               )
                                                         #:x-label "Iter num"
                                                         #:y-label "Fit"
                                                         #:x-min 0
                                                         #:y-min 0 #:y-max 100
                                                         #:legend-anchor 'bottom-
right
                                                         )
                                            )))))
                    (cons res (cons (cons graphic dcs) (add1 (- MAX_ITER iter)))
                          ))
```

```
                (if (null? res) (list (list #f)) res))
            )
        ))
      )
    (let*
        (
         (popul (start-popul k (get-verts graph) 50))
         (sorted-popul (top popul (length popul)))
         (best (fit (car sorted-popul)))
         (dc0 (if enable-visual (new bitmap-dc% (bitmap (make-bitmap FIELD_WIDTH
FIELD_HEIGHT))) '()))
         (draw (if enable-visual (draw-graph dc0 (argmin length (filter
(lambda(arg)(= (fit arg) best)) sorted-popul)) cycles) '()))
         (tmp-res
          (step-gen
           (map (lambda (x) (normalize x cycles)) popul)
           cycles
           MAX_ITER
           (list dc0)
           (list (list 0 best))
           (list (list 0 (fit (list-ref sorted-popul (sub1 (length sorted-popul))))))
           (list (list 0 (average (map fit popul)))))))
      tmp-res))

  (define target (make-bitmap FIELD_WIDTH FIELD_HEIGHT))
  (define dc (new bitmap-dc% (bitmap (make-bitmap FIELD_WIDTH FIELD_HEIGHT))))

  (define frame (new frame%
                     (label "Genetics visualizer")
                     (width (* (/ 11 10) FIELD_WIDTH))
                     (height (/ FIELD_HEIGHT 4))))

  (let* (
         (cycles (find-cycles graph))
         (tmp-res (start-gen cycles)))
    (if (and (equal? (caar tmp-res) #f) (null? cycles))
```

```
        #f
        (if (equal? #t enable-visual)
            (let* ((res (argmin length (car tmp-res))) (dcs (cadr tmp-res)) (iters
(map (lambda(iter)(string-append "Iter " (~a iter))) (range (cddr tmp-res)))))
              (define bmp-canvas
                (new canvas% (parent frame)
                     (style (list 'vscroll 'no-autoclear))
                     (min-height (* FIELD_WIDTH (/ 3 4)))
                     (paint-callback
                      (lambda(c dc)
                        (if (= 0 (- (cddr tmp-res)(send my-choice get-selection)))
                            (send dc set-scale 2 2)
                            (send dc set-scale 1 1)
                            )
                        (send dc draw-bitmap (send (list-ref dcs (- (cddr tmp-
res)(send my-choice get-selection))) get-bitmap) 0 0))
                      )))


              (new canvas% (parent frame)
                   (min-height 100)
                   (min-width 250)
                   ;(horiz-margin (/ FIELD_WIDTH 4))
                   (paint-callback
                    (lambda (c dc)
                      (send dc set-font (send the-font-list find-or-create-font 20
'default 'normal 'normal))
                      (send dc draw-text (string-append "Result population:  " (if
(equal? #f (car res)) "No answer" (~a res))) (/ FIELD_WIDTH 4) 50))
                    ))
              (define my-choice
                (new choice% (parent frame)
                     (label "Choose view  ")
                     (choices (append iters (list "Graphic")))
                     (selection (length iters))
                     (callback
                      (lambda (tp e)
```

```
                        (send (send bmp-canvas get-dc) clear)
                        (if (= 0 (- (cddr tmp-res)(send my-choice get-selection)))
                            (send (send bmp-canvas get-dc) set-scale 2 2)
                            (send (send bmp-canvas get-dc) set-scale 1 1)
                            )
                        (send (send bmp-canvas get-dc) draw-bitmap (send (list-ref
dcs (- (cddr tmp-res)(send tp get-selection))) get-bitmap) 0 0)))))
              (send bmp-canvas init-auto-scrollbars #f FIELD_HEIGHT 0 0)
              (send (send bmp-canvas get-dc) draw-bitmap (send (list-ref dcs 0) get-
bitmap) 0 0)
              ;(send (send bmp-canvas get-dc) set-bitmap (send (list-ref dcs 0) get-
bitmap))
              (send frame show #t)
              (if (equal? #f (car res)) #f
                  (list (not (equal? (car res) #f)) (length res) res)))
            (let ((res (argmin length tmp-res)))
              (list (not (equal? (car res) #f)) (length res) res)))))
  )
```

Gen-test.rkt

```
#lang scheme/base
(require racket/list)
(require "graph.rkt")
(require "solve.rkt")

(define GRAPH_CYCLE_LENGTH 200)
(define RANDOM_GRAPHS_NUM 100)
(define MAX_COMPLETE_GRAPH 6)

;generates graphs for testing
(define (gen-test-graphs)
  (append
   (list
    (cons (gen-cyclic-graph GRAPH_CYCLE_LENGTH) GRAPH_CYCLE_LENGTH)
    (cons (gen-acyclic-graph GRAPH_CYCLE_LENGTH) GRAPH_CYCLE_LENGTH)
    )
   (gen-graphs RANDOM_GRAPHS_NUM)
```

```
   ))


(define (gen-test)
  (let ((file (open-output-file "tests.txt" #:mode 'text #:exists 'replace)) (graphs
(gen-test-graphs)))
    (begin
      (for-each (lambda (graph)(begin
                                 (writeln graph file)
                                 (let ((result (start-bruteforce (car graph) (cdr
graph))))
                                   (if (equal? result #f)
                                       (writeln (list result) file)
                                       (writeln (flatten result) file)
                                       ))
                                 ))
                graphs)
      (let ((complete-graph (gen-complete-graph MAX_COMPLETE_GRAPH)))
        (begin
          (writeln (cons complete-graph MAX_COMPLETE_GRAPH) file)
          (writeln (gen-complete-graph-answer complete-graph MAX_COMPLETE_GRAPH)
file)
          (writeln (cons complete-graph (- MAX_COMPLETE_GRAPH 3)) file)
          (writeln (gen-complete-graph-answer complete-graph (- MAX_COMPLETE_GRAPH
3)) file)
          ))
      (close-output-port file)
      )))


(gen-test)
```

Graph.rkt

```
#lang scheme/base
(require racket/list)
(require "common.rkt")


(provide gen-graphs)
(provide find-cycles)
```

```
(provide count-verts)

(provide get-verts)

(provide gen-edge)

(provide gen-complete-graph)

(provide gen-graph)

(provide gen-cyclic-graph)

(provide gen-acyclic-graph)

(provide gen-complete-graph-answer)

(provide make-edges)


(define MAX_EDGES 30)


; is necessary for converting cycles as list of vertices into a list of edges during visualising

(define (make-edges lst)

  (define (make-edges-step lst prev first)

    (cond ((null? prev) (cons (cons (car lst) (cadr lst)) (make-edges-step (cddr lst) (cadr lst) (car lst))))

          ((null? lst) (list (cons prev first)))

          (else (cons (cons prev (car lst)) (make-edges-step (cdr lst) (car lst) first)))))

  (make-edges-step lst '() '()))


(define (get-verts graph)

  (remove-duplicates (flatten graph)))


(define (count-verts graph)

  (length (get-verts graph)))


; creates an edge from IN to OUT

(define (gen-edge in out)

  (if (= in out) (cons in (+ (gen-random 5) out)) (cons in out)))


; graph edges count depends on NUM

(define (gen-graph num)

  (define (add-edge cur)
```

```
    (if (< cur 0) '()
        (cons (gen-edge (random num) (random num)) (add-edge (sub1 cur)))))
  (let ((result (add-edge (+ num (random num)))))
    (cons (remove-duplicates result) (gen-random (count-verts result)))))

; generates complete graph with NUM vertices
(define (gen-complete-graph num)
  (define (add-edge iter)
    (if (< iter num)
        (append (foldl (lambda (arg result) (if (= arg iter) result (cons (cons iter arg) result))) '() (range num)) (add-edge (add1 iter)))
        '()))
  (add-edge 0))

(define (gen-complete-graph-answer graph K)
  (let ((vert-num (count-verts graph)))
    (if (< K (sub1 vert-num))
        (list #f)
        (append (list #t (sub1 K)) (flatten (append (range (- vert-num 2)) (sub1 vert-num))))
          )
    ))

(define (gen-cyclic-graph max)
  (define (add-edge cur)
    (cond ((= cur (sub1 max)) (list (gen-edge (sub1 max) 0)))
          (else (cons (gen-edge cur (add1 cur)) (add-edge (add1 cur))))
          ))
  (remove-duplicates (cons (gen-edge (floor (/ max 2)) 0) (add-edge 0))))

(define (gen-acyclic-graph max)
  (define (add-edge cur)
    (cond ((= cur (sub1 max)) (list (gen-edge 0 (sub1 max))))
          (else (cons (gen-edge cur (add1 cur)) (add-edge (add1 cur))))
          ))
  (remove-duplicates (add-edge 0)))
```

```
; generates MAX random graphs
(define (gen-graphs max)
  (define (iter i)
    (if (< i max)
        (cons (gen-graph (gen-random MAX_EDGES)) (iter (add1 i)))
        '()))
  (iter 0))


; checks whether LST2 is a cyclic permutation of LST1
(define (cyclic? lst1 lst2)
  (let* ((tail (member (car lst1) lst2))(pos (- (length lst2) (if (equal? #f tail) 0
(length tail)))))
    (equal?
     lst1
     (append (drop lst2 pos) (take lst2 pos))
     )))


; starts depth-first search for GRAPH
(define (dfs graph)
  (define (dfs-step edge vis cur-cycle)
    (begin
      (if (member (cdr edge) vis)
          (list (cons (car edge) (member (cdr edge)(reverse cur-cycle))))
          (foldl
           (lambda (arg result) (append (dfs-step arg (cons (car edge) vis) (cons
(car edge) cur-cycle)) result))
           '()
           (filter (lambda (arg) (= (cdr edge) (car arg))) graph)
           )
          )))


  (define (start-dfs edge)
    (begin
      (let ((cycle (dfs-step edge '() '())))
        cycle
```

```
        )))

  (remove-duplicates (foldl
                      (lambda (vert result)
                        (begin
                          (append (foldl
                                   (lambda (edge result1) (append (start-dfs edge)
result1))
                                   '()
                                   (filter (lambda (edge) (= (car edge) vert))
graph)) result)

                          ))
                      '()
                      (range (add1 (count-verts graph)))) cyclic?))

; returns all cycles existing in GRAPH
(define (find-cycles graph)
  (dfs graph))
```

Solve.rkt

```
#lang scheme/base

(require racket/list)
(require "graph.rkt")
(require "common.rkt")

(provide start-bruteforce)

(define (start-bruteforce graph max)
  (bruteforce graph (find-cycles graph) 1 max))

(define (bruteforce graph cycles depth max)
  (begin
    (if (or (< max depth) (null? cycles))
        #f
        (let ((result (ormap(lambda(verts)
```

```
                              (andmap(lambda(cycle)
                                       (if (intersects? cycle verts) verts #f))
                                     cycles))
                            (comb (get-verts graph) depth))))
          (if (equal? result
                      #f
                      )
              (bruteforce graph cycles (add1 depth) max)
              (list #t (length result) (sort result <)))))))
```

Visual.rkt

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

(require "graph.rkt")

(define FIELD_WIDTH 1000)
(define FIELD_HEIGHT 4000)
(define POINT_RADIUS 4)
(define POINT_DIAM (* 2 POINT_RADIUS))

(provide FIELD_WIDTH)
(provide FIELD_HEIGHT)
(provide draw-graph)

(define (ellipse-point x y)
  (cons (- x POINT_RADIUS) (- y POINT_RADIUS)))

(define (draw-graph dc popul cycles)

  (define (draw-vert vert x y)
    ;(printf "Vert ~v: (~v ~v)\n" vert x y)
    (let* ((draw-place (ellipse-point x y)))
      (if (member vert popul)
          (send dc set-brush "red" 'solid)
```

```
          (send dc set-brush "black" 'solid))
      (send dc draw-text (~a vert) x (- y (+ 20 POINT_RADIUS)))
      (send dc draw-ellipse (car draw-place) (cdr draw-place) POINT_DIAM POINT_DIAM)
      (list vert x y)))


  (define (draw-cycle cycle center diff radius)
    (define (draw-cycle-vert lst angle)
      (if (null? lst)
          '()
          (append (list (list (car lst) (+ (car center) (* (cos angle) radius)) (+
(cdr center) (* radius (sin angle))))) (draw-cycle-vert (cdr lst) (+ angle diff)))))
    (draw-cycle-vert (get-verts cycle) 0))


  (define (eq-vert? v1 v2)
    (equal? v1 (car v2)))


  (define (draw-edge edge verts)
    (let ((vert1 (car (member (car edge) verts eq-vert?)))(vert2 (car (member (cdr
edge) verts eq-vert?))))
      (send dc draw-line (cadr vert1) (caddr vert1) (cadr vert2) (caddr vert2)))
    )


  (define (border-offset x)
    (+ x (* 3 POINT_DIAM)))


  (define (draw-cycles cycles)
    (define (draw-cycles-step lst rad x_ y_)
      (if (null? lst) '()
      (let* ((cur-cycle (car lst))
             (radius (* 10 (count-verts cur-cycle)))
             (next-new (<= FIELD_WIDTH (border-offset (+ x_ radius radius))))
             (vert-num (count-verts cur-cycle))
             (x (if next-new (border-offset 0) x_))
             (y (if next-new (border-offset (+ y_ (max radius rad) (max radius rad)))
y_))
             (vert-places (draw-cycle cur-cycle (cons (+ x radius) (+ y radius)) (/
(* 2 pi) vert-num) (* 10 vert-num))))
```

```
        (for-each
         (lambda(edge)(draw-edge edge vert-places))
         (make-edges cur-cycle))
        (append
         vert-places
         (draw-cycles-step (cdr lst) (max rad radius) (border-offset (+ x radius
radius)) y)))))
    (draw-cycles-step cycles 0 (border-offset 0) (border-offset 0)))
  (send dc set-smoothing 'aligned)
  (let ((verts (draw-cycles cycles)))
    (for-each (lambda(vert) (draw-vert (car vert) (cadr vert) (caddr vert))) verts))
  )
```