МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени М.В.ЛОМОНОСОВА
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ
КАФЕДРА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

Отчёт по второму заданию
Вариант 4Г:
Раскраска рёбер графа

выполнил:
студент 427 группы
Манджиев Айта Викторович

Москва 2017

Содержание

1. Постановка задачи

В рамках задания необходимо написать программу на языке программирования Scheme.

На вход программы подаётся список рёбер неориентированного графа G = (V, E). Вершины $v_i \in V$ обозначаются атомами, рёбра представляются списками вида (NODE1 NODE2). Затем на вход программы подаётся целое число K ($1 \leq K$).

Можно ли раскрасить ребра графа G не более чем в K цветов, то есть существует ли функция $c : E \rightarrow \{1. . .K\}$ такая, что если $(u, v), (u, w), (x, u) \in E$, то $c((u, v)) \neq c((u, w))$ и $c((u, v)) \neq c((x, u))$ при $v \neq w$ и $v \neq x$? Если раскраски рёбер графа в K цветов не существует, напечатайте #f. Если раскраска рёбер графа существует, то сначала напечатайте #t, затем количество цветов, в которые раскрашен граф, а затем список рёбер с указанием их цветов.

Элементами этого списка являются списки из двух элементов следующего вида ((NODE1 NODE2) COLOR). Из всех возможных вариантов раскраски выберите тот, который требует минимального количества цветов.
В процессе нахождения решения должен быть предусмотрен сбор данных о ходе процесса приближения текущего решения задачи к оптимальному.

Также требуется создать тесты - средство автоматизированного тестирования и средства визуализации процесса поиска решения и итогового ответа.

Пример входных данных:
((a b) (b c) (c d) (a d))
2
Пример печати результата:
#t
2
((a b) 1)((b c) 2)((c d) 1)((a d) 2))

2. Генерация тестов

Генератор тестов написан на Scheme и находится в файле generate_tests.rkt. При запуске данной программы сгенерируются два файла:

1) input_file.txt - файл с входными данными для генетического алгоритма
2) result_file.txt - файл с решениями.

В качестве тестов генерируются графы нескольких видов: цепочка, цикл, полный граф, двудольный граф. Данные виды графов были выбраны в связи с тем, что мы заранее знаем количество цветов необходимых для их раскраски, следовательно, мы можем сгенерировать сколь угодно большое количество тестов с графами разных размеров. В каждой из функций генерации определенного вида графов (print-chains, print-cycle...) в качестве параметра (например k) передается количество графов, которое необходимо сгенерировать (от минимального размера графа до k). Далее генерируются по два теста (с возможной и невозможной раскраской) для каждого графа.

Для автоматизации запуска тестов был создан скрипт new_run_tests.sh на языке bash. Данный скрипт последовательно считывает входные данные из input_file.txt для генетического алгоритма, запускает основную программу genetic.rkt и подает ей на вход считанные данные. После этого результат отработанной программы сравнивает с эталонным ответом из файла result_file.txt. После каждого теста в stdout отписывается либо #t (тест успешно пройден), либо #f(тест провален).

3. Генетический алгоритм

Программа, решающая поставленную задачу с помощью генетического алгоритма находится в файле genetic.rkt. В программе не используются никакие мутаторы структур данных.

В качестве хромосомы используется список цветов рёбер. Начальная популяция создаётся случайно, то есть цвет каждого ребра каждого представителя популяции выбирается случайно и независимо. Но диапазон цветов изначально строго задан (мы будем перебирать этот диапазон линейным и бинарным поисками).

В качестве оценочной функции используется количество конфликтов в раскраске – пар смежных рёбер, имеющих одинаковый цвет. Считается каждая пара, то есть N смежных рёбер одного цвета породят N*(N-1) конфликтов.

Целью является минимизация оценочной функции. Если она равна 0, то раскраска графа корректна, и она сразу используется в качестве ответа.

Каждый представитель нового поколения – либо гибрид 2 особей из фиксированной доли лучших, либо не изменённый представитель из фиксированной доли лучших. Конкретный вариант выбирается случайно.

После создания нового поколения, каждый представитель популяции может с некоторой вероятностью мутировать, то есть каждое его ребро может изменить свой цвет на случайный с некоторой вероятностью. Несмотря на это, небольшая часть лучших представителей популяции гарантированно переходит в следующее поколение без мутаций. Мутация происходит относительно случайно выбранной вершины. Все ребра входящие или выходящие из этой вершины берутся от одного родителя, а все остальные от другого.

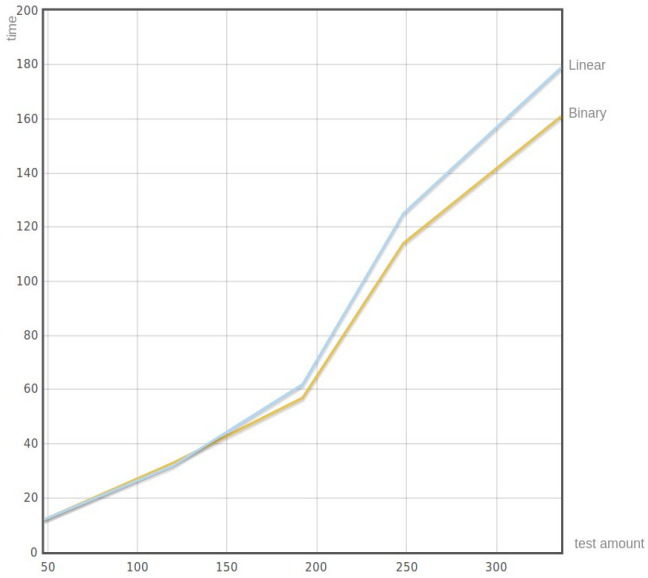
Эволюция продолжается до тех пор, пока не будет получена правильная раскраска, или пока с момента последнего улучшения оценочной функции лучшего представителя популяции не пройдёт определённое количество поколений, в этом случае считается, что раскраски с требуемым количеством цветов не существует.

Поиск минимального количества цветов реализован двумя методами: линейный поиск и бинарный поиск. Изначально минимально возможным количеством цветов считается наибольшая степень вершины графа, а максимально возможным – минимум из указанного во входных данных ограничения на количество цветов и количества вершин графа, округлённого вниз до ближайшего нечётного числа (так как полный граф с 2N вершинами может быть раскрашен в 2N-1 цветов).

Входные данные решение получает со стандартного потока ввода, ответ выводит на стандартный поток вывода.

4. Результаты

На следующем графике представлена скорость работы алгоритма на тестовых данных разного размера при линейном и бинарном поисках. По оси OX — количество тестов, по оси OY — количество секунд, потраченных на обработку данных тестов.



Из графика видно, что при увеличении количества тестов бинарный алгоритм поиска наилучшего количества цветов выигрывает по производительности.

На используемом наборе тестов доля ошибок достаточно мала. Ошибки неизбежны, поскольку в процессе решения используется случайность. Подбор различных параметров (количество поколений до останова, процент лучших особей и т.д.) позволяет увеличить точность за счёт увеличения времени работы, или наоборот, уменьшить время работы за счёт снижения точности.

При выбранных мной параметрах количество ошибок составляет не более 5%.

Далее представлен график прогресса построения решения генетическим алгоритмом для успешно пройденного теста:

граф:

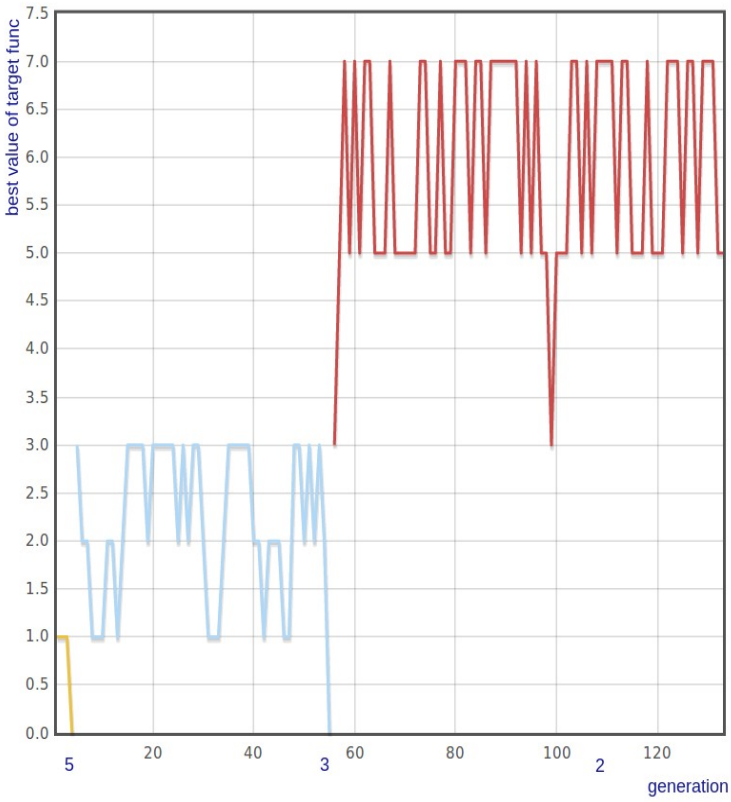
((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (8 9) (9 10) (10 11) (11 12) (12 13) (13 14) (14 0))

максимальное количество цветов:

10

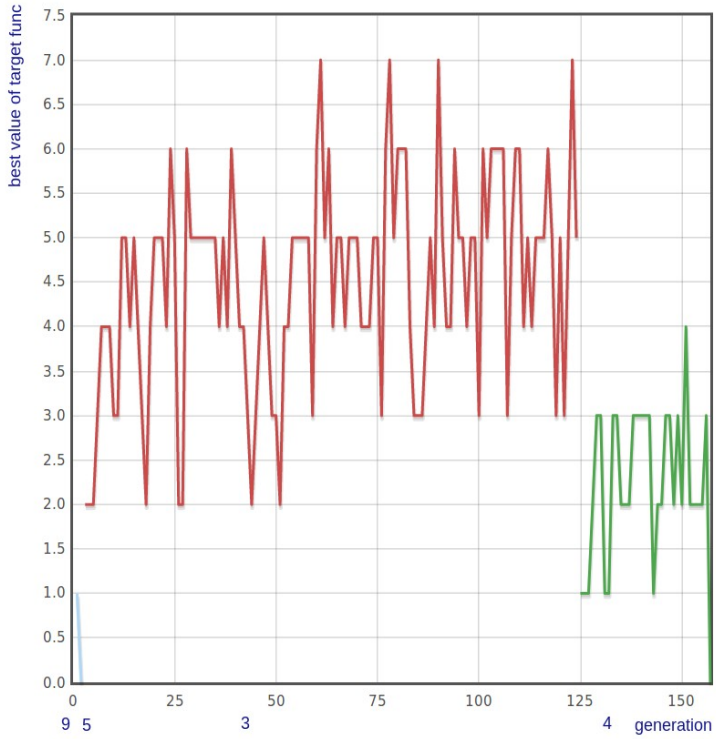
Из графика можно увидеть, как алгоритм изначально строит покраску для 5, 3, 2

цветов (рассматриваем случай бинарного поиска). Для 5 цветов ему это удается, затем удается и для 3 цветов, но для 2 цветов мы исперпываем максимальное количество поколений, поэтому в результате получаем правильный ответ — 3 цвета и раскраску.



Из графика видно, что чем меньше цветов берется для раскраски, тем большее количество конфликтов возникает при генерации первого поколения.

Рассмотрим также пример теста, на котором генетический алгоритм вычислил неоптимальный ответ:



граф:

((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (8 9) (9 10) (10 11) (11 12) (12 13) (13 14) (14 15) (15 16) (16 0))

максимальное количество цветов:

17

На данном тесте последовательно были предприняты построить раскраску из 9, 5, 3, 4 цветов. При 9 цветах раскраска была построена на первом же шаге при рандомной генерации, поэтому на графике это точка (0;0). Правильная раскраска для 5 цветов была достигнута за 1 шаг. Затем нам не удалось построить раскраску для 3 цветов, так как был исперчан лимит поколений. В результате мы получаем неправильный ответ в виде раскраски из 4 цветов, когда правильный ответ — 3. Легко догадаться, что при увеличении количества поколений данный тест будет пройден, что только подтверждает зависимость правильных ответов от подобранных параметров.

Заключение

На основе полученных результатов мы можем прийти к выводу, что раскраска графов с помощью генетического алгоритма возможна, но с поправкой на то, что некоторый процент ошибки будет сохраняться в любом случае. Но данный процент можно минимизировать с помощью правильного подбора параметров.

Приложение. Код программы

Код программы по раскраске графа с помощью генетического алгоритма:

```
#lang scheme/base
(require scheme/list)
(require scheme/math)

(define population_size 20)                                          ;Размер популяции
(define max_generations 60)                                          ;Максимальное количество
поколений
(define cross_probability .3)                                        ;Вероятность скрещивания
(define crossing_part (exact-round (* .8 population_size)))          ;Какая часть популяции участвует
в скрещивании
(define mutation_probability .7)                                     ;Вероятность мутировании
хромосомы
(define mutate_colour_prob .05)                                      ;Вероятность мутации одного
цвета хромосомы
(define selection_part (exact-round (* .4 population_size)))         ;Какая часть популяции участвует
в переходе в след. поколение
(define unmutable_part (exact-round (* .1 population_size)))         ;Какая часть популяции
гарантированно не изменяется

;получить список всех вершин
(define (get-all-vertices i ht)
    (cond
        ((< i 0)  ht)
        (else
            (let ((edge (vector-ref graph i)))
                (get-all-vertices (- i 1) (hash-set* ht (car edge) '() (cadr edge) '()))
            )
        )
    )
)

;генерация рандомной хромосомы
(define (random-chromosome colours)
    (build-vector
        edges-amount
        (lambda (i) (+ (random colours) 1))
    )
)
;Сгенерировать начальную популяцию
(define (generate-initial-population colours)
    (build-vector
        population_size
        (lambda (i) (random-chromosome colours))
    )
)

;Количество конфликтов в хромосоме
(define (get-amount-conflicts chromosome)
    (define (loop i ht answer)
        (if (< i 0)
            answer
            (let*
                (
                    (key1 (cons (car (vector-ref graph i)) (vector-ref chromosome i)))
                    (val1 (+ 1 (hash-ref ht key1 -1)))
                    (key2 (cons (cadr (vector-ref graph i)) (vector-ref chromosome i)))
                    (val2 (+ 1 (hash-ref ht key2 -1)))
                )
                (loop
                    (- i 1)
                    (hash-set* ht key1 val1 key2 val2)
                    (+ answer val1 val2)
                )
            )
        )
    )
    (loop
        (- (vector-length chromosome) 1)
        (make-immutable-hash)
        0
```

```
    )
)

; получить вектор, состоящий из количества конфликтов для каждой хромосомы популяции.
(define (get-conflicts-all-chromosomes population)
    (build-vector
        (vector-length population)
        (lambda (i) (get-amount-conflicts (vector-ref population i)))
    )
)

; скрещивание двух хросом (выбирается точка скрещивания и первая часть берется и второй хромосомы, а
вторая из первой)
(define (cross-chromosome a b)
    (let ((cross_point (random edges-amount)))
        (build-vector
            edges-amount
            (lambda (i)
                (if (< i cross_point)
                    (vector-ref b i)
                    (vector-ref a i)
                )
            )
        )
    )
)

; Мутировать каждый цвет хромосомы с заданной вероятностью
(define (mutate-chromosome colours chromosome)
    (build-vector
        edges-amount
        (lambda (i)
            (cond
                ((< (random) mutate_colour_prob)
                (+ 1 (random colours)))
                (else (vector-ref chromosome i)))
        )
    )
)

;Поиск наименьшего количества конфликтов
(define (find-best scores)
    (define (loop i j)
        (if (<= i 0)
            j
            (loop
                (- i 1)
                (if (< (vector-ref scores i) (vector-ref scores j))
                    i
                    j
                )
            )
        )

    )
    (loop (- (vector-length scores) 1) 0)
)


;генерация следующего поколения
(define (next-generation colours population old_best remaining_generations)
    (let*
        (
            (scores (get-conflicts-all-chromosomes population))
            (best (find-best scores))
            (bestscore (vector-ref scores best))
        )
        (cond
            (
                (<= remaining_generations 0)                                        ;Если
закончился лимит поколений - возвращаем результат
                (if (= 0 bestscore)
                    (vector-ref population best)
                    '()
                )
            )
            (
```

```
                (> bestscore 0)                                                 ;Если еще не
получили раскраску - продолжаем
                (let*
                    (
                        (
                            next
                            (build-vector                                       ;Генерация
следующего поколения
                                population_size
                                (lambda (i)
                                    (let
                                        (
                                            (
                                                chromosome
                                                (cond
                                                    ((< i unmutable_part)       ;Не меняем
определунную часть хромосом
                                                        (vector-ref population i)
                                                    )
                                                    ((< (random) cross_probability)     ;Получение
новой особи путем скрещивания старых
                                                        (cross-chromosome
                                                            (vector-ref population (random
crossing_part))
                                                            (vector-ref population (random
crossing_part))
                                                        )
                                                    )
                                                    (else                       ;Если не
скрещиванием, то выбираем рандомно
                                                        (vector-ref population (random
selection__part))
                                                    )
                                                )
                                            )
                                        )                                       ;Мутируем
                                        (if (and (>= i unmutable_part) (< (random)
mutation_probability))
                                            (mutate-chromosome colours chromosome)
                                            chromosome
                                        )
                                    )
                                )
                            )
                        )
                        (if (< bestscore old_best)
                            (next-generation colours next bestscore max_generations)        ;Если
результат улучшился, то обновляем счетчик поколений
                            (next-generation colours next old_best (- remaining_generations 1))
                        )
                    )
                )
                (else
                    (vector-ref population best)
                )
            )
        )
)

;Запуск генетического алгоритма
(define (genetic-solve colours)
    (next-generation
        colours
        (generate-initial-population colours)
        +inf.0
        max_generations
    )
)

;Размечаем полный граф в colours-amount цветов
(define (tag-complete-graph colours-amount)
    (build-vector
        edges-amount
        (lambda (i)
            (let*
                (
```

```
                    (edge (vector-ref graph i))
                    (a (car edge))
                    (b (cadr edge))
                )
                (cond
                    ((>= a colours-amount) (+ b 1))
                    ((>= b colours-amount) (+ a 1))
                    (else
                        (if (even? (+ a b))
                            (+ (quotient (+ a b) 2) 1)
                            (+ (remainder (quotient (+ (+ a b) colours-amount) 2) colours-amount) 1)
                        )
                    )
                )
            )
        )
    )
)
;Пытаемся найти лучшее решение, путем бинарного поиска, каждый раз пытаясь решить задачу
генетическим алгоритмом.
(define (find-better-solution min_colours max-colours upper_answer)
    (if (>= (+ 1 min_colours) max-colours)
        (cons #t upper_answer)
        (let*
            (
                (mid-colours (quotient (+ min_colours max-colours) 2))
                (answer (genetic-solve mid-colours))
            )
            (if (null? answer)
                (find-better-solution mid-colours max-colours upper_answer)
                (find-better-solution min_colours mid-colours (cons mid-colours answer))
            )
        )
    )
)

(define (find-better-solution-linear amnt-colours prev_answer)
    (let*
        (
            (answer (genetic-solve amnt-colours))
        )
        (if (null? answer)
            (cons #t prev_answer)
            (find-better-solution-linear (- amnt-colours 1) (cons amnt-colours answer))
        )
    )
)

;Решаем задачу в зависимости от того, можем ли мы раскрасить граф, как полный или нет.
(define (solve-task)
    (let ((colours-amount-for-complete-graph (- vertices-amount (modulo (+ vertices-amount 1) 2))))
        (if (>= max-colours colours-amount-for-complete-graph)
            ;(find-better-solution
            ;    (- max-vertices-power 1)
            ;    colours-amount-for-complete-graph
            ;    (cons colours-amount-for-complete-graph (tag-complete-graph colours-amount-for-
complete-graph))
            ;)
            (
                find-better-solution-linear
                max-colours
                (cons colours-amount-for-complete-graph (tag-complete-graph colours-amount-for-
complete-graph))
            )
            ;Изначально пытаемся решить задачу с максимальным количеством цветов
            (let ((answer (genetic-solve max-colours)))
                (if (null? answer)
                    '(#f)
                    ;(find-better-solution (- max-vertices-power 1) max-colours (cons max-colours
answer))
                    (find-better-solution-linear (- max-colours 1) (cons max-colours answer))
                )
            )
        )
    )
)
```

```
(define (chromosome-to-answer chromosome)
  (build-list edges-amount (lambda (i) (list (vector-ref graph i) (vector-ref chromosome i)))))


(define graph (list->vector (read)))

;Количество ребер
(define edges-amount (vector-length graph))

(define max-colours (read))

;Проверка крайних случаев
(cond
    ( (< max-colours 0)
        (printf "#f~n")
        (exit 0)
    )
    ( (= edges-amount 0)
        (printf "~a~n~a~n~a" #t 0 '() )
        (exit 0)
    )
    ( (= max-colours 0)
        (printf "#f~n")
        (exit 0)
    )
)

;Переменная, хранящая список всех вершин, отсортированных по возрастанию
(define
    vertices
    (list->vector
        (sort
            (hash-keys (get-all-vertices (- edges-amount 1) (make-immutable-hash)))
            <
        )
    )
)

;Количество вершин
(define vertices-amount (vector-length vertices))

(define (max-vertex-power i ht mp)
    (cond
        ((< i 0) mp)
        (else
            (let*
                (
                    (edge (vector-ref graph i) )
                    (first-vertex (+ 1 (hash-ref ht (car edge) 0)) )
                    (second-vertex (+ 1 (hash-ref ht (cadr edge) 0)) )
                )
                (max-vertex-power
                    (- i 1)
                    (hash-set* ht (car edge) first-vertex (cadr edge) second-vertex)
                    (max mp first-vertex second-vertex)
                )
            )
        )
    )
)

;Максимальная степень вершин
(define max-vertices-power
    (max-vertex-power (- edges-amount 1) (make-immutable-hash) 0)
)

(let ((answer (solve-task)))
  ;(printf "2222~a2222" answer)
  (write (car answer))(newline)
  (cond ((car answer)
         (write (cadr answer))(newline)
         (write (chromosome-to-answer (cddr answer)))(newline))))
```