Московский Государственный Университет имени М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Системного Программирования

Отчёт по второму заданию

Вариант 15 «Доминирующее множество ребер графа»

Автор:
гр. 427 (КФ)
Овчинников Дмитрий Константинович

Москва, 2015

Содержание

1 Постановка задачи

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 K). Существует ли в графе G доминирующее подмножество рёбер, то есть такое подмножество рёбер $F \subseteq E$, что |F| K и каждое ребро e не из этого подмножества рёбер ($e \subseteq E - F$) имеет общую вершину по крайней мере с одним ребром из подмножества F?

Если доминирующего подмножества рёбер не существует, напечатайте f. Если доминирующее подмножество рёбер существует, напечатайте t, затем количество рёбер в доминирующем подмножестве, а затем список рёбер, входящих в доминирующее подмножество. Из всех возможных вариантов доминирующего подмножества выберите вариант с минимальным количеством рёбер. В процессе нахождения решения должна быть предусмотрена визуализация процесса приближения текущего решения задачи к оптимальному. Предусмотрите возможность визуализации результата.

Пример входных данных:

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

1

Пример печати результата:

t

1

((a d))

2 Генерация тестов

Поскольку не существует теории, которая каким-либо образом описывала верхние и нижние границы размера минимального доминирующего набора ребер для всех графов (в том числе и не связных), генерировать тесты придется только основываясь на известные факты о графах и их доминирующих множеств. Поэтому не гарантируется максимальное покрытие множества графов в тестовых наборах, которые будут генерироваться.

Итак, будем строить только те графы, у которых заранее известна размерность минимального покрывающего множества ребер, чтобы мы могли проверить правильно ли работает генетический алгоритм. В тестах будет генерироваться 3 вида графов, рассмотрим каждый из них по подробнее.

Полный граф. Полный граф - это граф, у которого между каждыми двумя вершинами есть ребра. Нетрудно заметить, что у полного графа с N вершинами из каждой вершины выходит по N-1 ребер, а значит, что все они будут смежными. Теперь рассмотрим две вершины из полного графа. Если мы предположим, что ребро, соединяющее эти две вершины, находится в покрывающем множестве, то получается, что одно ребро покрывает 2*(N-1) ребер в графе с N вершинами. Отсюда следует, что минимальное, покрывающее множество ребер будет иметь размер N/2, причем с отсечением дробной части (truncate в Scheme). Генерация полного графа реализована в виде функции, принимающей на вход максимально возможное количество вершин в графе. Функция случайным образом генерирует число вершин, верхней границей которого является тот самый параметр - максимальное число вершин.

Граф-многогранник. Далее будем строить графы, похожие на многогранники, т.е. первая вершина связана ребром со второй, вторая с третьей и так далее(последняя с первой). Очевидно, что каждое ребро у таких графов покрывает еще 2 соседних графа. Т.е. чтобы найти минимальное доминирующее множество, нужно брать каждое 3-ье ребро в графе. Таким образом, у графа с N вершинами будет N/3 покрывающих ребра, причем с округлением в большую сторону (ceiling в Scheme). Генерация такого вида графов реализована, опять же, в виде функции, которая принимает на вход максимальное число ребер графа. Реальное число ребер графа генерируется случайно (аналогично предыдущему случаю).

Последний вид графов, который производится генератором - граф-звездочка. Это граф

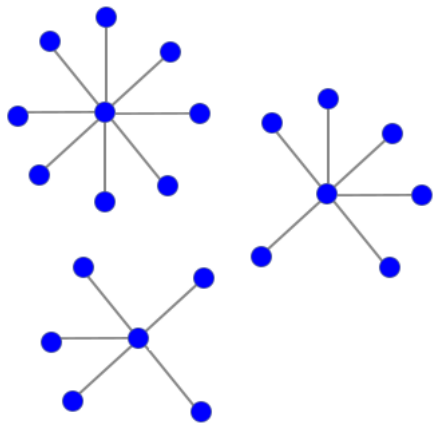

Рис. 1: Графы-"звездочки"

(вообще говоря несвязный), у которого есть несколько вершин взятых за центр звездочки, остальные же соединены с "центрами". Рис. Графы-"звездочки" Если учитывать, что все ребра в одной звезде смежны, то достаточно взять по одному ребру у каждой звезды и это будет искомым множеством. Следовательно, число их - столько же, сколько и несвязных звездочек в задаче. Реализация выглядит аналогично с предыдущими, это сделано специально, чтобы иметь возможность случайно выбирать, каким будет граф, который нужно сгенерировать.

Как было уже упомянуто выше, при генерации тестов, выбирается случайный граф, а также случайное число K (см. Постановка задачи). Генератор тестов записывает графы и их числа K в один файл, а ответы на них - в другой, так, чтобы можно было запустить генетический алгоритм на файле с вопросами, а потом сверить их с правильными ответами. Сам генератор описан в файле test_ generator.ss (см. Приложение. Код программы). Для того, чтобы сверять ответы предусмотрен отдельный скрипт, который также написан на Scheme (см. test_ comparer.ss в Приложение. Код программы). Работает он довольно-таки прямолинейно и предсказуемо - сверяет ответы алгоритма с правильными, если ответ и число - верны, то проверяет, является ли множество ребер действительно покрывающем и только в этом случае считает, что алгоритм справился с задачей. В качестве результата работы скрипта на экране выводится количество правильных ответов, процент правильных ответов среди всех, а также тесты, в которых

алгоритм дал неправильные результаты.

Важное замечание по работе test_ comparer'a: если алгоритм сработал лучше, чем было в ответах, т.е. нашел более минимальное покрывающее множество, чем в правильном ответе теста, то скрипт, засчитывает его как верный. Такая реализация верна, поскольку в случае, когда генерируются графы-звездочки, не всегда удается правильно посчитать количество несвязных звезд, поскольку они генерируются случайно (имеется ввиду ребра к звездам появляются случайно).

3 Генетический алгоритм

Генетический алгоритм был позаимствован из биологии и заключается он в том, что особи эволюционируют, умирают, на их место становятся новые особи и так далее. В итоге получается идеальная особь, которая больше не эволюционирует - она и является результатом эволюции. Суть генетического алгоритма заключается в том, чтобы найти наиболее подходящую особь (какая именно особь является таковой будет оговорено позже) за конечное число шагов эволюции (число поколений).
В генетическом алгоритме законно присутствуют элементы:

1. Хромосомы - набор данных, дающих информацию об особи.
   В данной задаче в качестве хромосомы взято одно ребро из графа.

2. Особи - набор хромосом. Количество хромосом зависит от задачи, которую пытается решить генетический алгоритм.
   В качестве особи взято потенциальное покрывающее множество графа, размер задается K, которое фигурирует в постановке задачи.(Постановка задачи)

3. Популяция - некоторое количество особей - материал для эволюции

4. Фитнес-функция - функция, численно описывающая, насколько "идеальна"особь, которая подается к ней на вход (иногда требуется, чтобы функция была метрической, но в общем случае - необязательно).
   Фитнес-функция в этой задаче показывает насколько "покрывает"данная особь граф, причем результат выдается в виде того, сколько еще осталось до полного покрытия графа (удобнее для внутренних вычислений).

5. Кроссинговер - скрещивание двух особей, чтобы получить потомство.
   Скрещивание двух особей происходит следующим образом: выбираются две хромосомы из каждого особя и из них случайным образом(случайно с учетом веса) выбирается одна, которая будет в особе-потомке.

6. Мутация - показатель мутации потомства относительно предков.
   Мутация в данной задаче реализована внутри кроссинговера, т.е. на самом деле выбираются не 2, а 3 хромосомы, одна из них - случайная из графа.

Сам же генетический алгоритм можно описать следующими шагами:

1. Создать начальную (случайную) популяцию.
2. Применить к каждой особи фитнес-функцию, если существует "идеальная"особь, то остановить алгоритм.
3. На основе результатов фитнес-функции выбрать 2*(РАЗМЕР-ПОПУЛЯЦИИ) лучших особей для скрещивания и применить кроссинговер к ним.
4. Применить мутацию к полученному потомству
5. Если количество поколений недостаточно, повторить пункт 2, иначе - остановить алгоритм.

В данной реализации критерием останова является превышение количества поколений, либо появление "идеальной"особи. В начале работы алгоритма расчитываются веса каждого ребра - насколько ребро покрывает граф. Это необходимо для того, чтобы выбирать (случайно с учетом веса) наиболее "тяжелые"ребра, в случае равенства всех весов вероятность каждого ребра равна и веса не играют никакой роли. Скрещивание происходит вместе с мутацией: Берутся две особи(читать "два набора хромосом") и между каждой парой хромосом выбирается наилучшая с учетом веса, а также берется случайное ребро из графа и тоже участвует в отборе для каждой пары.

4 Визуализация

Для выполнения данного пункта были использованы библиотеки Racket Drawing Toolkit, Racket Graphical Interface, plot. Визуализация выполнена в отдельном файле и представляет собой несколько функций, таких как draw-edges, draw-vertices, draw-graph, draw-graphics.
Вся логическая часть по прорисовке графов также находится в данном файле. Логика прорисовки в общих чертах можно описать следующими шагами:

1. Расположить на плоскости все вершины графа.
   Из списка ребер извлекаются все вершины, которые там встречаются. В ходе реализации было решено располагать вершины на окружности, центр которой находится в центре холста. Радиус окружности зависит от размеров холста. В последней реализации размер холста имел размер 800*600, координаты центра - (400, 300), а радиус - 280.
   Расположение вершин выполняется примитивным образом: $2\pi/N$ - угол между двумя вершинами; с использованием матрицы поворота и известным углом вершины располагаются на окружности.
   В последующем список вершин хранит еще и координаты.

2. Нарисовать ребра.
   Имея уже список вершин с координатами, не составляет особого труда нарисовать связи между ними. Для красоты было решено использовать сплайны(draw-spline в Racket Drawing Toolkit) вместо обычных прямых, где в качестве третьей точки взята 0,3 доля от перпендикуляра к прямой между двумя вершинами. В итоге получилось что-то вроде паутины, которая более наглядная, чем просто прямые.

3. Нарисовать покрывающие ребра.
   Так как предполагается визуализировать сам процесс поиска покрывающего множества ребер, ребра из этого множества окрашиваются в другой цвет. Прорисовка осуществляется аналогично простым ребрам.

4. Нарисовать сами вершины.
   Вершины полагается рисовать в самую последнюю очередь, для того чтобы они

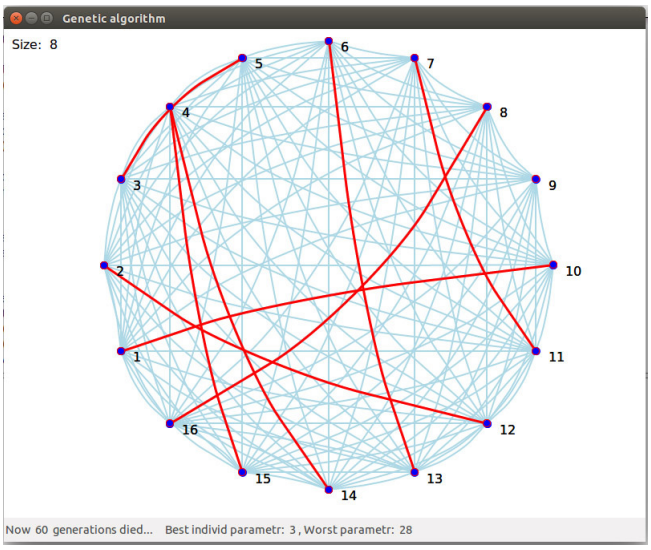

Рис. 2: Визуализация поиска покрывающего множества

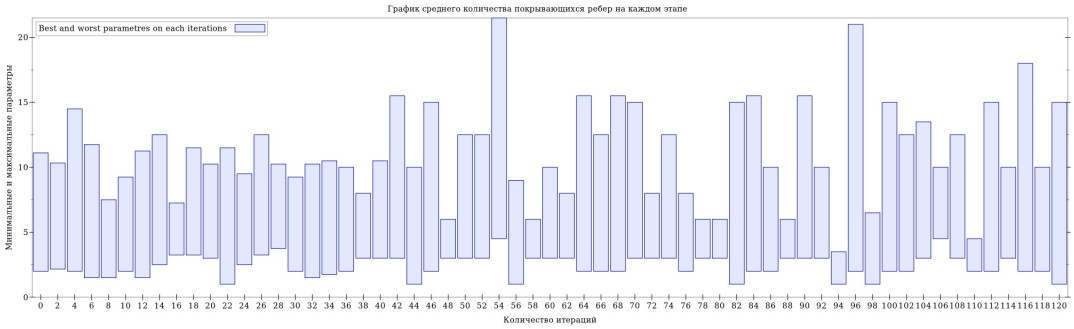

Рис. 3: Статистика лучших и худших параметров для каждой итераций

не могли быть зарисованы ребрами. Вершина представляет собой круг с заливкой определенного цвета.

Список ребер и вершин представляет собой глобальные переменные. Это было сделано для того, чтобы все функции прорисовки имели к ним доступ.

Также предусмотрена вывода статистики на выбор - средние параматры на каждой итерации или время выполнения в зависимости от количества ребер.

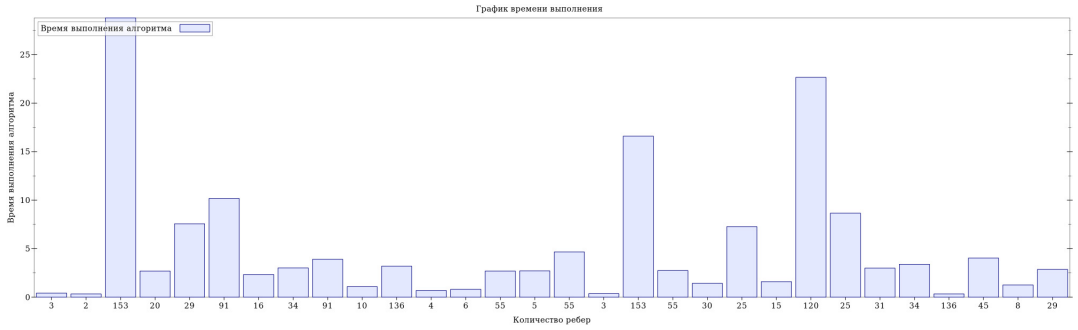

Рис. 4: Время выполнения для каждого теста

5 Результаты

Параметры, которые можно изменять для получения оптимального результата в плане времени и выполнения и качества выполнения это:

1. Максимальное количество поколений
2. Количество предок у каждой пары особей
3. Количество особей в популяции

При запуске на различных тестах, сгенерированных генератором (Генерация тестов), были получены следующие параметры:
Максимальное количество поколений - 120
Количество предок у каждой пары особей - 2
Количество особей в популяции - K (Постановка задачи)
Попробуем исследовать конкретный набор тестов при таком задании параметров:
с помощью генератора сгенерируем 30 тестовых наборов.
Тесты выполненны верно - 30 из 30. Как видно из графика, время выполнения сильно зависит от количества ребер, но иногда получаются случаи, когда первая популяция выбрана более удачно и время выполнения сокращается в разы (например, два графа со 153 ребрами, один - $\tilde{2}9$ сек., другой -$\tilde{1}6$ сек.).

Заключение

В заключении хотелось бы сказать, что генетический алгоритм очень хорошо подходит для такого типа задач и работает гораздо лучше, чем перебор и даже некоторые другие алгоритмы. Генетический алгоритм реализован удачно. Слабое место в конкретной реализации - фитнес-функция, т.к. для того, чтобы посчитать насколько особь нам подходит, требуется проход по всему графу, это занимает большую часть времени выполнения алгоритма. Во всем остальном были произведены максимальные улучшения (например, вместо линейного поиска - бинарный и т.п.).

Приложение. Код программы

————————gen.ss————————-

lang scheme

(require "drawing.ss")

|

Генетический алгоритм для построения минимального покрывающего множества ребер для графов.

Автор: Дмитрий Овчинников (2015 г)

В качестве "особей"взяты потенциальные минимальные доминирующие множества.

В качестве "хоромосомы"берется одно ребро.

Скрещивание хромосом см. функцию crossingover.

Время работы над полным графом с 40 ребрами 5-7 сек.

Параметры алгоритма, которые нужно модифицировать для достижения оптимального (качество/время) результата:

1) Количество итераций (поколений)

- параметр к genetic-driver-loop пол названием population-count.

(вызывается в функции genetic)

2) Количество потомков у каждых двух родителей

- параметр к merge-many - "size вызывается в crossingover.

3) Количество особей в популяции параметр "individs-count"в функции genetic

|

|

Функция edge-is-covered?.

Вход: lst - мно-во ребер, edge - ребро.

Выход: t or f, в зависимости от того, покрывается ли ребро множеством.

(Т.е. является ли смежным с каким-нибудь ребром или само входит в множество.)

|

(define (edge-is-covered? lst edge)

(if (null? lst)

f

(not (null? (filter
(lambda (x)
(or (equal? (car edge) (car x))
(equal? (cadr edge) (cadr x))
(equal? (car edge) (cadr x))
(equal? (cadr edge) (car x))))
lst)))))

|
Функция pick-random.
Выдает случайный элемент из списка lst.
|
(define (pick-random lst)
(list-ref lst (random (length lst))))

|
Функция calculate-edges-weights.
Вход: lst - мно-во ребер, для которого нужно посчитать веса,
full-lst - все, множество ребер.
result - накопитель результата.
Выход: Список размером с lst, в котором хранятся "веса"для каждого ребра.
("вес- количество смежных с этим ребром ребер)
|
(define (calculate-edges-weights lst full-lst result)
(if (null? lst)
(reverse result)
(begin (let ((calc (car (calculate-cover (list (car lst)) full-lst (length full-lst) 0))))
(if (= calc 0)
(if (null? result)
(calculate-edges-weights (cdr lst) full-lst
(cons 1000000.0 result))
(calculate-edges-weights (cdr lst) full-lst

(cons (+ 1000000.0 (car result)) result)))
(if (null? result)
(calculate-edges-weights (cdr lst) full-lst
(cons (/ 1 calc) result))
(calculate-edges-weights (cdr lst) full-lst
(cons (+ (/ 1 calc) (car result)) result))))))))

|
Функция get-edge-weight.
Параметры:
prev - вес предыдущего элемента (для первого 0)
elem - элемент, для которого нужно найти вес
lst - список элементов
weights - соответствующий список весов.
Результат: Возвращает вес определенного элемента, если он имеется в lst.
(примечание: размер lst должен совпадать с weights.)
|
(define (get-edge-weight prev elem lst weights)
(if (null? (cdr lst))
(- (car weights) prev)
(if (and
(equal? (car elem) (car lst))
(equal? (cadr elem) (cadr lst)))
(- (car weights) prev)
(get-edge-weight (car weights) elem (cdr lst) (cdr weights)))))

|
calculate-cover.
Параметры:
individ - особь, для которой нужно посчитать, насколько она покрывает граф,
lst - граф (список ребер),
lst-size - размер графа.

result - накопитель результата.
Результат: Число - насколько особь покрывает граф.
(Прим.: Результатом является не число смежных ребер,
а количество всех ребер "минус"количество смежных с этим множеством ребер.)ы
|
(define (calculate-cover individ lst lst-size result)
(cond ((null? lst)
(list (- lst-size result) individ))
((edge-is-covered? individ (car lst))
(calculate-cover individ (cdr lst) lst-size (+ result 1)))
(else
(calculate-cover individ (cdr lst) lst-size result))))

|
make-one-individ.
lst - множество ребер графа,
chromo-size - количество хромосом в одной особи,
result - накопитель результата.
Результат: Создается особь, с количеством хромосом, равным chromo-size
|
(define (make-one-individ lst chromo-size result)
(if (= chromo-size 0)
result
(make-one-individ lst (- chromo-size 1) (cons (pick-random lst) result))))

|
generate-k-individs.
Параметры - те же самые, что и у функции выше + k - количество особей.
Результат: Генерируется случайная популяция из k особей с количеством хромосом - chromo-size
|
(define (generate-k-individs lst chromo-size k result)

(if (or
(= k 0)
(= chromo-size 0))
result
(generate-k-individs lst chromo-size (- k 1) (cons (make-one-individ lst chromo-size '()) result))))

(define PARAMS '())

|
GENETIC. - Основная функция программы, реализует генетический алгоритм для данной задачи.
Параметры:
individs-count - количество особей в популяции,
chromo-count - количество хромосом у особи.
Результат: Ответ на поставленный в задаче вопрос -
"Имеется ли в данном графе lst, минимальное доминирующее множество ребер размером
не больше k (= chromo-count)?"
Если t ("ДА"), то каков размер и само множество (результат в одном списке вида (t k (множество)))
|
(define (genetic lst weights individs-count chromo-count result)
(define cur-time (current-inexact-milliseconds))
(define (genetic-driver-loop population lst weights population-count)
(let ((possible-answers (filter
(lambda (x) (= 0 (car x)))
population))
(count (- 120 population-count))
(best (car (find-min population)))
(worst (car (find-max population))))
(set! PARAMS (cons (list count best worst) PARAMS))

```
(if (> (- (current-inexact-milliseconds) cur-time) 300)
(begin
(send (get-msg) set-label (string-join (list "Now "(number->string count) "generations died...
"
"Best individ parametr: "
(number->string best)
Worst parametr: "
(number->string worst))))
(set! cur-time (current-inexact-milliseconds)))
null)
(cond ((not (null? possible-answers))
(list t (length (cadr (car possible-answers))) (cadr (car possible-answers)) ))
((or
(= population-count 0)
(null? population))
(list f))
(else
(genetic-driver-loop
(next-generation population lst weights)
lst
weights
(- population-count 1))))))
(let ((first-population (map
(lambda (x) (calculate-cover x lst (length lst) 0))
(generate-k-individs lst chromo-count individs-count '()))))
(let ((answer (genetic-driver-loop first-population lst weights 120)))
(set-EDGES (lst->lst-string lst))
(set-VERTICES (get-vertices (get-EDGES)))
(set-VERTICES (get-coordinates 280 (/ (* 2 pi) (length (get-VERTICES))) 400 290 (get-
VERTICES) '()))
(send (get-canvas) refresh-now [lambda (dc) (draw-graph (get-DC))])
(if (not (car answer))
```

```
(begin
(draw-graphics PARAMS (string-join (list (number->string (length (get-VERTICES)))
(number->string (length (get-EDGES))))))
(set! PARAMS '())
(car result))
(begin
(draw-edges (get-DC) (lst->lst-string (caddr answer)) "red"3)
(draw-vertices (get-DC) (get-VERTICES))
(send (get-DC) draw-text (string-join (list "Size: "(number->string (cadr answer)))) 10 10)
(genetic lst weights individs-count (- chromo-count 1) (cons answer result)))))))
```

|

next-generation.

Параметры:

population - текущая популяция.

lst - граф,

wieghts - веса ребер графа.

Результат: Генерация нового поколения.

|

```
(define (next-generation population lst weights)
(let ((calc-popul (map
(lambda (x) (cons (/ 1 (car x)) (cdr x)))
population)))
(map (lambda (x y)
(crossingover (car (cdr x)) (car (cdr y)) lst weights))
(get-parents calc-popul (length calc-popul) '() lst)
(get-parents calc-popul (length calc-popul) '() lst))))
```

|

get-parents.

Результат:

Выборка родителей для следующего поколения (используется pick-random-by-weight)
|
(define (get-parents population size result lst)
(if (= size 0)
result
(let ((pick (calculate-cover (pick-random-by-weight population) lst (length lst) 0)))
(get-parents population
(- size 1)
(cons (cons (/ 1 (car pick)) (cdr pick)) result)
lst))))
|
crossingover.
Скрещивание происходит одновременно с мутацией,
т.е. новое ребро может браться как от родителей, так и просто с графа,
это выбирается случайно, с учетом весов этих ребер.
|
(define (crossingover ind1 ind2 lst weights)
(define (merge ind1 ind2 lst weights)
(map (lambda (x y)
(let ((rnd-elem (pick-random-by-weight2 lst weights)))
(let ((possible-lst (list
(list (get-edge-weight 0 x lst weights) x)
(list (get-edge-weight 0 y lst weights) y)
(list (get-edge-weight 0 rnd-elem lst weights) rnd-elem))))
(pick-random-by-weight possible-lst)))) ind1 ind2))

(define (merge-many ind1 ind2 lst weights size result)
(if (= size 0)
result
(merge-many ind1 ind2 lst weights (- size 1) (cons (merge ind1 ind2 lst weights) result))))
(find-min (map
(lambda (x) (calculate-cover x lst (length lst) 0))

(merge-many ind1 ind2 lst weights 2 '()))))

|
find-min.
Параметры: lst - список вида ((number1 elem), (number2 elem), (number3 elem) ...)
Результат: Возрвращается элемент с минимальным number*
|
(define (find-min lst)
(if (not (null? lst))
(let ((y (car lst)))
(if (null? (cdr lst))
y
(let ((m (find-min (cdr lst))))
(if (< (car m) (car y))
m
y))))
(list 0 '())))

(define (find-max lst)
(if (not (null? lst))
(let ((y (car lst)))
(if (null? (cdr lst))
y
(let ((m (find-min (cdr lst))))
(if (> (car m) (car y))
m
y))))
(list 0 '())))

|Функция sum-weight.
Параметры: список, где в каждом вложенном списке первый элемент - вес; накопитель результата.

```
Результат: сумма весов всего списка|
(define (sum-weight lst result)
(if (null? lst)
result
(sum-weight (cdr lst) (+ (car (car lst)) result))))

|Функция pick-random-by-weight.
Параметры: сумма весов всего списка, сам список
Результат: случайный(с учетом веса) элемент списка|
(define (pick-random-by-weight lst)
(define (get-weights lst result)
(if (null? lst)
(reverse result)
(if (null? result)
(get-weights (cdr lst) (cons (car (car lst)) result))
(get-weights (cdr lst) (cons (+ (car (car lst)) (car result)) result)))))
(define (get-elems lst result)
(if (null? lst)
(reverse result)
(get-elems (cdr lst) (cons (cadr (car lst)) result))))
(let ((elems (get-elems lst '()))
(weights (get-weights lst '())))
(let ((elem (pick-random-by-weight2 elems weights)))
elem)))

|
Модифицированная функция выбора случайного элемента с учетом веса
(для случаев, когда веса находятся в отдельном списке)
|
(define (pick-random-by-weight2 lst weights)
(define (bin-search number lst weights a b)
(let ((mid (+ a (truncate (/ (- b a) 2)))))
```

```
(cond ((or
(= a b)
(> a b))
(list-ref lst b))
((> (list-ref weights mid) number)
(bin-search number lst weights a mid))
(else (bin-search number lst weights (+ mid 1) b)))))
(let ((sum (list-ref weights (- (length weights) 1))))
(let ((rnd (/ (random (truncate (* sum 100000))) 100000.0)))
(bin-search rnd lst weights 0 (- (length lst) 1)))))

|
Функция для работы алгоритма через консоль.
|
(define (ask-graph)
(print '(Please enter graph))
(newline)
(let ((graph (read))
(k (read)))
(let ((result (genetic graph k k (list f))))
(print result))))

;(ask-graph)
(define times '())
|
Функция для работы алгоритма через файлы (удобнее работать с тестами)
|
(require racket/file)
(define (with-files)
(define (call-genetic question)
(genetic (car question)
(calculate-edges-weights (car question) (car question) '())
```

```
;(ceiling (* (length (car question)) 0.8))
40
(cadr question)
(list f)))
(define (progress out lst size)
(if (null? lst)
(display "all tests have done!")
(begin
(let ((time (current-inexact-milliseconds)))
(display "working on ")
(display (- size (length (cdr lst))))
(display "of ")
(display size)
(display (call-genetic (car lst)) out)
(display ""out)
(display "Time:")
(display (/ (truncate (- (current-inexact-milliseconds) time)) 1000))
(display "s. Graph-size: ")
(display (length (car (car lst))))
(newline)
(set! times (cons (vector (length (car (car lst))) (/ (truncate (- (current-inexact-milliseconds)
time)) 1000)) times))
(progress out (cdr lst) size)))))

(let ((cases (file->list "test.txt"))
(out (open-output-file "genetic-algorithm-result.txt":exists 'truncate)))
(show-frame)
(progress out cases (length cases))
(draw-graphics-time times "time")
(close-output-port out)))

(with-files)
```

```
lang scheme
(require racket/gui/base)
(require racket/math)
(require racket/draw)
(require plot)

(define no-pen (new pen
(define blue-brush (new brush



|
Инициализация переменных для создания окнаы
|


(define frame (new frame
    label "Genetic algorithm

    width 800

    height 640

    x 300

    y 100

    alignment '(left center)
))
(define canvas1 (new canvas
    min-width 800

    min-height 600

    paint-callback
(lambda (canvas dc)
(set! DC dc)
(draw-graph dc))
```

```
))
(define (get-canvas)
canvas1)
(define subframe (new horizontal-panel
    parent frame

    alignment '(left center)
))
|(define btn (new button
    label "Выключить обновление графа

    callback (lambda (button event)
(if (not stop-refresh)
(begin
(set! stop-refresh t)
(send canvas1 resume-flush)
(send button set-label "Выключить обновление графа"))
(begin
(set! stop-refresh f)
(send canvas1 suspend-flush)
(send button set-label "Включить обновление графа"))))
))
(define (get-btn) btn) |
(define msg (new message
    label "genetic algorithm

    auto-resize t
))
(define (get-msg)
msg)


;
; SHOW!
;
(define (show-frame)
```

```
(send frame show t))
;
;
;

;//////////////////////
; GLOBAL NAMES
; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

|
VERTICES - переменная, хранит все вершины графа и его координаты.
|
(define VERTICES (list "onetwothreefour"
"fivesixseveneight"))
(define (set-VERTICES value)
(set! VERTICES value))
(define (get-VERTICES) VERTICES)

|
EDGES - переменная, список всех ребер графа.
|
(define EDGES (list (list "onesix") (list "twothree") (list "onefive") (list "twoseven")(list
"seveneight")))
(define (set-EDGES value)
(set! EDGES value))
(define (get-EDGES) EDGES)



|
DC - переменная - Drawing-context, хранит контекст для прорисовки графа на холсте.
|
(define DC null)
```

(define (get-DC)
DC)

|
stop-refresh - переменная, (boolean), - показатель обновления экрана.
|
(define stop-refresh t)
; ////////////////////

|
Функция draw-graph.
Рисует сначала ребра, потом вершины.
|
(define (draw-graph dc)
(send dc set-smoothing 'aligned)
(draw-edges dc EDGES "LightBlue"2)
(draw-vertices dc VERTICES))

|
get-vertices.
Функция достает из списка ребер все имеющиеся в нем вершины.ы
|
(define (get-vertices lst)
(define (in-list? elem lst)
(= 0 (length (filter (lambda (x) (equal? elem x))
lst))))
(define (search-loop lst result)
(if (null? lst)
result
(cond ((in-list? (car (car lst)) result)
(if (in-list? (cadr (car lst)) result)

(search-loop (cdr lst) (cons (cadr (car lst)) (cons (car (car lst)) result)))
(search-loop (cdr lst) (cons (car (car lst)) result))))
((in-list? (cadr (car lst)) result)
(search-loop (cdr lst) (cons (cadr (car lst)) result)))
(else (search-loop (cdr lst) result)))))
(search-loop lst '()))

|
get-one-coordinate.
Функция достает координаты для определенной вершины.
Параметры:
elem - вершина,
coords - список координат в виде (x y "string$_n ame$")
|
$(define(get - one - coordinateelemcoords)$
$(if(null?coords)$
$(list00"notfound")$
$(if(equal?elem(caddr(carcoords)))$
$(carcoords)$
$(get - one - coordinateelem(cdrcoords)))))$

|
$draw - edges$
.
:
$dc - drawingcontext$
$edges - list$
$color - string$
$size - integer$
|

$(define(draw - edgesdcedgescolorsize)$
$(senddcset - pencolorsize'solid)$
$(define(get - third - pointelem1elem2)$
$(define(make - projectionx1y1x2y2)$
$(list(-x1x2)(-y2y1)))$
$(let((x1(+5(carelem1)))$
$(y1(+5(cadrelem1)))$
$(x2(+5(carelem2)))$
$(y2(+5(cadrelem2))))$
$(let((proj(make - projectionx1y1x2y2)))$
$(list(+(/(+x1x2)2)(/(carproj)8))$
$(+(/(+y1y2)2)(/(cadrproj)8))))))$
$(define(loopdcedges)$
$(if(null?edges)$
$null$
$(begin$
$(let((elem1(get - one - coordinate(car(caredges))VERTICES))$
$(elem2(get - one - coordinate(cadr(caredges))VERTICES)))$
$(let((third(get - third - pointelem1elem2)))$
$(senddcdraw - spline(+5(carelem1))(+5(cadrelem1))$
$(carthird)(cadrthird)$
$(+5(carelem2))(+5(cadrelem2))))$
$(loopdc(cdredges))))))$
$(loopdcedges))$

|
$get - coordinates.$
$.()$
:
$rad - (integer), -,$
$angle - (real).$

$cen-x, cen-y-(integer, integer)-,$
$lst-(list)-.$
$result(list)-.$
|
$(define(get-coordinatesradanglecen-xcen-ylstresult)$
$(if(not(null?lst))$
$(if(null?result)$
$(let((res(list(-cen-xrad)cen-y(carlst))))$
$(get-coordinatesradanglecen-xcen-y(cdrlst)(consresresult)))$
$(let((cur-angle(*(lengthresult)angle))$
$(prev-x(car(list-refresult(-(lengthresult)1))))$
$(prev-y(cadr(list-refresult(-(lengthresult)1)))))$
$(let((res(list(+cen-x(-(*(-prev-xcen-x)(coscur-angle))(*(-prev-ycen-y)(sincur-$
$angle))))$
$(+cen-y(+(*(-prev-xcen-x)(sincur-angle))(*(-prev-ycen-y)(coscur-angle))))$
$(carlst))))$
$(get-coordinatesradanglecen-xcen-y(cdrlst)(consresresult)))))$
$result))$

$(define(re-calculate-coordinatesverticesresult)$
$(define(find-all-neighbourselem)$
$(filter(lambda(x)(not(null?x)))(map(lambda(x)$
$(if(equal?(caddrelem)(carx))$
$(let((coord(get-one-coordinate(cadrx)VERTICES)))$
$(list$
$(-(carcoord)(carelem))$
$(-(cadrcoord)(cadrelem))))$
$(if(equal?(caddrelem)(cadrx))$
$(let((coord(get-one-coordinate(carx)VERTICES)))$
$(list$
$(-(carcoord)(carelem))$

$(-(cadrcoord)(cadrelem))))$
$null)))$
$EDGES)))$
$(define(calc-sumlstsizesum1sum2)$
$(if(null?lst)$
$(list(/sum1size)(/sum2size))$
$(calc-sum(cdrlst)size(+sum1(car(carlst)))(+sum2(cadr(carlst))))))$
$(if(null?vertices)$
$(set!VERTICESresult)$
$(let((neighbours(find-all-neighbours(carvertices))))$
$(let((sum(calc-sumneighbours(lengthneighbours)00)))$
$(re-calculate-coordinates(cdrvertices)$
$(cons(list$
$(+(/(carsum)3)(car(carvertices)))$
$(+(/(cadrsum)3)(cadr(carvertices)))$
$(caddr(carvertices)))result))))))$

$|$
$draw-vetices.$
.
:
$dc-(dc$
$coords-(list)-.$
$|$
$(define(draw-verticesdccoords)$
$(define(draw-one-vertexdclst)$
$(if(null?lst)$
$'()$
$(begin$
$(let((vertex(carlst)))$
$(senddcdraw-ellipse(carvertex)(cadrvertex)1010)$
$(senddcdraw-text(caddrvertex)(+20(carvertex))(+3(cadrvertex)))$

$(draw - one - vertexdc(cdrlst))))))$
$(senddcset - pen"red"1'solid)$
$(senddcset - brush"blue1solid)$
$(draw - one - vertexdccoords))$

|
$lst- > lst - string.$
.
()
:
$lst - (list) - .$
|
$(define(lst- > lst - stringlst)$
$(map(lambda(x)$
$(list(cond((string?(carx))$
$(carx))$
$((number?(carx))$
$(number- > string(carx)))$
$((symbol?(carx))$
$(symbol- > string(carx)))$
$(else"uknownformat"))$
$(cond((string?(cadrx))$
$(cadrx))$
$((number?(cadrx))$
$(number- > string(cadrx)))$
$((symbol?(cadrx))$
$(symbol- > string(cadrx)))$
$(else"uknownformat"))))lst))$

|
$draw - graphics.$

.

:

$lst - (list) - , .$

$file - name - (string) - , .$

|

$(define(draw - graphicslstfile - name)$

$(define(summlstlst - sizeres - sum1res - sum2)$

$(if(null?lst)$

$(ivl(/res - sum1lst - size)(/res - sum2lst - size))$

$(summ(cdrlst)lst - size(+(cadr(carlst))res - sum1)(+(caddr(carlst))res - sum2))))$

$(define(loopcountlstresult)$

$(let((lst1(filter(lambda(x)(or(= count(carx))(= (+count1)(carx))))lst)))$

$(if(= (lengthlst1)0)$

$(reverseresult)$

$(loop(+count2)lst(cons(vectorcount(summlst1(lengthlst1)00))result)))))$

$(plot - width2000)$

$(plot - height600)$

$(plot - file(discrete - histogram$

$(loop0lst'())$

$: label"Bestandworstparametresoneachiterations"$

$: x - min0$

$)$

$: x - label""$

$: y - label""$

$: title""$

$file - name'png))$

$(define(draw - graphics - timelstfile - name)$

$(plot - width2000)$

$(plot - height600)$

$(plot - file(discrete - histogram$

lst
$: label""$
$: x - min0$
$)$
$: x - label""$
$: y - label""$
$: title""$
$file - name'png))$

$(set!VERTICES(get - verticesEDGES))$
$(set!VERTICES(get{-}coordinates280(/(*2pi)(lengthVERTICES))400290VERTICES'()))$

$(provideset - VERTICES)$
$(provideset - EDGES)$
$(provideget - VERTICES)$
$(provideget - EDGES)$
$;(provideget - btn)$
$(providestop - refresh)$
$(provideget - canvas)$
$(provideDC)$
$(provideget - DC)$
$(provideshow - frame)$
$(providedraw - vertices)$
$(provideget - coordinates)$
$(provideget - vertices)$
$(providedraw - edges)$
$(providedraw - graph)$
$(providedraw - graphics)$
$(providedraw - graphics - time)$
$(provideget - msg)$
$(providelst - > lst - string)$

$(providere - calculate - coordinates)$

$langscheme/base$

|
$generate - test -$
$: number - .$
$: .$
:
$test.txt -$
$test - results.txt - , test.txt$
, ,
.
, .
|

$(define(generate - testsnumber)$
$(define(pick - randomlst)$
$(list - reflst(random(lengthlst))))$
$(define(writing - to - fileout1out2lst)$
$(if(null?lst)$
$(print'(generatingsuccesfullydone))$
$(begin$
$(display(car(carlst))out1)$
$(display"")out1)$
$(display(cadr(carlst))out2)$
$(display"")out2)$
$(writing - to - fileout1out2(cdrlst)))))$
$(define(loopnresult)$
$(if(< n1)$

```
result
(loop(-n1)(cons((pick-random(listcreate-starscreate-polygoncreate-full-graph))20)result))))
(let((out1(open - output - file"test.txt" : exists'truncate))
(out2(open - output - file"test - results.txt" : exists'truncate)))
(writing - to - fileout1out2(loopnumber'()))
(close - output - portout1)
(close - output - portout2)))

(define(create - polygonmax)
(define(polygon - driver - loopstartcurrentendresult)
(if(or(= currentend)(> currentend))
(cons(listcurrentstart)result)
(polygon - driver - loopstart(+current1)end(cons(listcurrent(+current1))result))))
(let((n(+(random(-max2))2)))
(let((answer(ceiling(/n3)))
(k(random(+n1))))
(if(or(< answerk)(= answerk))
(list(list(polygon - driver - loop11n'())k)(listtanswer))
(list(list(polygon - driver - loop11n'())k)f)))))


(define(create - starsmax)
(define(stars - driver - loopcurrentnmaxresult)
(if(or(= currentmax)(> currentmax))
result
(stars - driver - loop(+current1)nmax(cons(list(+(random(-n1))1)current)result))))
(define(re - calculate - starscurrentlstnresult)
(define(is - used - vertexlstvresult)
(if(null?lst)
result
(is - used - vertex(cdrlst)v
(orresult
```

$(equal?v(car(carlst)))$
$(equal?v(cadr(carlst)))))))$
$(if(= currentn)$
$result$
$(if(is-used-vertexlstcurrentf)$
$(re-calculate-stars(+current1)lstn(+result1))$
$(re-calculate-stars(+current1)lstnresult))))$
$(let((stars(+(random(-max3))4))$
$(k(randommax)))$
$(let((lst(stars-driver-loop(+stars1)stars(*max2)'()))))$
$(let((answer(re-calculate-stars1lststars0)))$
$(if(not(< kanswer))$
$(list$
$(listlstk)$
$(listtanswer))$
$(list$
$(listlstk)$
$f))))))$

$(define(create-full-graphmax)$
$(define(loop-supportcurrentstartfinishresult)$
$(if(> startfinish)$
$result$
$(loop-supportcurrent(+start1)finish(cons(listcurrentstart)result))))$
$(define(main-loopcurrentsizeresult)$
$(if(> currentsize)$
$result$
$(main-loop(+current1)size(appendresult(loop-supportcurrent(+current1)size'()))))))$
$(let((k(randommax))$
$(size(+(random(-max3))3)))$
$(let((res(main-loop1size'()))))$
$(if(< k(truncate(/size2)))$

$(list(listresk)f)$
$(list(listresk)(listt(truncate(/size2)))))))$

$langscheme/base$

$(requireracket/file)$

|
$compare$
$: test - results.txt$
$genetic - algorithm - result.txt$
|

$(define(compare)$
$(define(edge - is - covered?lstedge)$
$(if(null?lst)$
f
$(not(null?(filter$
$(lambda(x)$
$(or(equal?(caredge)(carx))$
$(equal?(cadredge)(cadrx))$
$(equal?(caredge)(cadrx))$
$(equal?(cadredge)(carx))))$
$lst)))))$
$(define(covered?covergraph)$
$(not(null?(filter(lambda(x)(edge - is - covered?coverx))graph))))$
$(define(statisticslst)$
$(define(calculate - procentlstcount)$
$(if(null?lst)$
$count$
$(if(not(list?(carlst)))$
$(calculate - procent(cdrlst)(+count1))$

$(calculate - procent(cdrlst)count))))$
$(define(print - lstlstcount)$
$(cond((null?lst)$
$(display"...”))$
$((list?(carlst))$
$(begin$
$(displaycount)$
$(display".”)$
$(display(carlst))$
$(display"”)$
$(print - lst(cdrlst)(+count1))))$
$(else(print - lst(cdrlst)(+count1)))))$
$(newline)$
$(let((proc(calculate - procentlst0)))$
$(printproc)$
$(display"/”)$
$(print(lengthlst))$
$(display)$
$(display"trueanswers”)$
$(newline)$
$(print(round(*(/proc(lengthlst))100.0)))$
$(display"$
$(print - lstlst1)))$
$(let((tests(file- > list"test.txt”))$
$(true - answers(file- > list"test - results.txt”))$
$(answers(file- > list"genetic - algorithm - result.txt”)))$
$(statistics(map(lambda(testtrue - answeranswer)$
$(if(list?true - answer)$
$(if(not(list?answer))$
$(listf"got"answer"expected"true - answer)$
$(if(and$
$(not(= (cadrtrue - answer)(cadranswer)))$

$(not(> (cadrtrue - answer)(cadranswer))))$
$(listf"got"answer"expected <> "true - answer)$
$(if(covered?(caddranswer)(cartest))$
t
$(listf"notcovered"))))$
$(if(list?answer)$
$(listf"got"answer"expected"true - answer)$
$t)))$
$tests$
$true - answers$
$answers))))$
$(compare)$