Московский Государственный Университет имени М. В. Ломоносова
Факультет Вычислительной математики и кибернетики
Кафедра Системного программирования

Отчет по второму заданию.
Вариант №8. Клика.

Выполнил
студент 428 группы
Морозов Иван Павлович

Москва
2015

Содержание

1. Постановка задачи

На вход программе подаётся описание неориентированного невзвешенного графа $G = (V, E)$ в виде списка описаний рёбер. Вершины $v_i \in V$ обозначаются атомами, рёбра представляются списками вида $(v_k\ v_l)$. Затем на вход программе подаётся целое число K $(1 \leq K \leq |V|)$.

Необходимо определить, существует ли в графе G клика мощности, не меньшей K? Кликой называется полный подграф (граф, каждая пара различных вершин которого является смежной) заданного графа. Мощность клики – это количество вершин в ней. Если такой клики не существует, ответом должно быть *#f*. Если клика существует, следует напечатать *#t*, затем мощность клики, затем список вершин, входящих в клику. Из всех возможных решений задачи, удовлетворяющих условию, нужно выбрать решение с максимальной мощностью клики.

В процессе нахождения решения должна быть предусмотрена визуализация процесса приближения текущего решения задачи к оптимальному, например, в виде графика минимума и максимума функции отбора в текущей популяции, и визуализация результата.

Предлагается решить поставленную задачу, используя алгоритмы «генетического программирования». Для этого необходимо определить, что является «хромосомой», «популяцией», как выполняется «скрещивание», «мутация» и «отбор лучших особей».

Разработанный алгоритм нужно реализовать на языке программирования scheme (в реализации DrRacket [7]). В процессе реализации необходимо подобрать параметры «генетического алгоритма» таким образом, чтобы добиться наилучшей скорости сходимости и точности.

2. Генерация тестов

Для оценки качества реализованного алгоритма необходимо подобрать (сгенерировать) набор задач-тестов с заранее известными ответами. Построение подходящего набора тестов – достаточно сложная задача. Одна из проблем заключается в том, что нет простой оценки сложности задачи для генетического алгоритма, выраженной через свойства графа. Экспериментальные исследования показали, что плотность, размер, порядок, относительный размер клики, количество клик графа не являются подходящими мерами сложности [4].

Для тестирования реализации алгоритма используются готовые задачи из двух наборов. Эти наборы состоят из достаточно большого количества графов разного размера (от нескольких тысяч, до нескольких сотен тысяч ребер), которые сгенерированы специально для тестирования подобных алгоритмов на основе реальных задач и считаются сложным для поиска клики. Многие исследователи используют их для тестирования своих алгоритмов, поэтому, в перспективе, можно сравнить эффективность нашего алгоритма с другими.

Первый набор задач сформирован организаторами и участниками международной конференции DIMACS (International algorithm implementation challenge on maximum clique, graph coloring, and satisfiability organized by the Center for Discrete Mathematics and Theoretical Computer Science [3]).

Графы вместе с известным размерами максимальной клики доступны по адресу http://iridia.ulb.ac.be/~fmascia/maximum_clique/. Там же для большинства семейств графов приводятся программы-генераторы с подробным описанием алгоритмов построения. Набор состоит из 39 графов, включая случайные графы (*Cx.y* и *DSJCx.y* порядка x и плотности $0.y$), графы Штейнера (англ. Steiner Triple graphs): *MANN_a27* - 378 вершин и 70 551 ребро, графы Брокингтона (англ. Brockington graphs): *brockx_2* и *brockx_4* порядка x, графы Хемминга (англ. Hamming graphs): *hamming8-4* порядка 256, графы Санчис (англ. Laura Sanchis graphs): *sanrx_y, sanx_y_z*, *genx_p0.9_z* порядка x и плотности y, графы P-hat (*p_hatx-z* порядка x) и C-fat (*c-fatx-z* порядка x).

Второй набор задач – подмножество графов библиотеки BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems (Maximum Clique, Maximum Independent Set, Minimum Vertex Cover and Vertex Coloring). В этих задачах глобально-оптимальное решение «скрыто» среди множества локально-оптимальных решений, что, как известно, вызывает проблемы у генетических алгоритмов.

Тестовые графы вместе с известными размерами максимальной клики доступны по адресу http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm. Там же можно найти программы-генераторы и подробное описание структуры этих графов. В набор тестов включено 15 графов различного размера (от 450 до 760 вершин, от 83 198 до 246 801 ребер).

Для запуска тестов в автоматическом режиме реализована простая система тестирования. Она состоит из множества вышеупомянутых тестов-графов, каждый из которых хранится в отдельном файле в виде списка ребер, файла конфигурации, в

котором перечислены все файлы с тестами и программы на языке программирования scheme.

Программа загружает файл конфигурации и последовательно подает перечисленные в нем задачи на вход основному алгоритму. Каждый запуск алгоритма оформлен в виде теста RackUnit [7], который позволяет выводить результаты тестирования как в текстовом, так и в режиме графического интерфейса. Тест считается пройденным успешно, если найдена клика размера не меньше, чем требовалось. Программа тестирования проверяет корректность полученного решения (найденный граф является полным подграфом заданного) и замеряет время его поиска.

Ниже приведен пример вывода программы тестирования (в текстовом режиме). Для каждого теста печатается время поиска решения (в миллисекундах); данные теста (название, порядок и размер графа, требуемая мощность клики); количество шагов генетического алгоритма, потребовавшихся для поиск решения; найденный ответ (корректность и мощность клики). После выполнения всех тестов печатается общая статистика.

```
cpu time: 392358 real time: 392343 gc time: 12950
test: DSJC500.5 (500 vertices, 62623 edges) best known: 13
steps: 171
found: 13 is-clique: #t

cpu time: 16 real time: 9 gc time: 0
test: Simple test (5 vertices, 10 edges) best known: 5
steps: 1
found: 5 is-clique: #t

2 success(es) 0 failure(s) 0 error(s) 2 test(s) run
```

3. Генетический алгоритм

3.1. Общая схема алгоритма

Общая схема генетического алгоритма приведена на рисунке ниже.

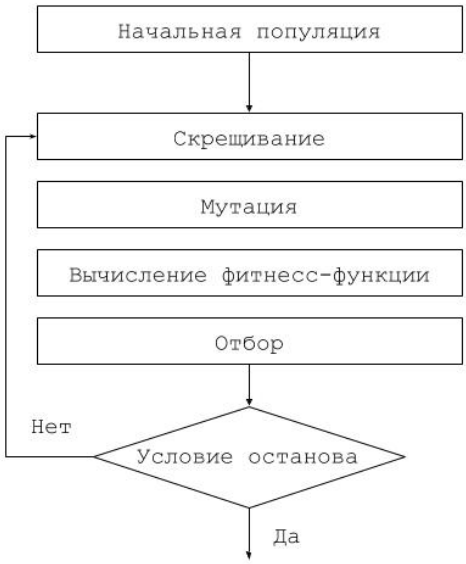

Рисунок 1. Общая схема генетического алгоритма.

На первом этапе генерируется начальная популяция. Далее, пока не выполнено условие останова (например, найдено подходящее решение или исчерпано время), последовательно выполняются шаги генетического алгоритма. Каждый шаг состоит из следующих действий: скрещивание, мутация, вычисление функции приспособленности (фитнесс-функции), отбор.

Отдельная хромосома представляет подграф S_G графа $G = (V, E)$ в виде битового массива x длины $N = |V|$. $x_i = 1$ ($x_i = 0$), если вершина с номеров i входит (не входит) в S_G.

Начальная популяция генерируется случайно.

Функция приспособленности $f: chromosome \to [0, N]$ хромосомы x определяется как $f(x) = |x|$ (количество вершин хромосомы x), если x – клика, иначе $f(x) = 0$.

Функция скрещивания получает на вход две родительские хромосомы и возвращает новую хромосому. Для построения новой хромосомы используется равномерный кроссовер: для каждого гена генерируется случайное число $p \in [0,1]$;

если $p \geq 0.5$, ген берется из первого родителя, иначе – из второго. Такое скрещивание выполняет слияние двух графов.

Функция мутации получает на вход хромосому, инвертирует случайно выбранный ген и возвращает измененную хромосому.

Отбор выживших хромосом и хромосом для скрещивания выполняется с помощью рулеточного колеса (выбор пропорционально приспособленности). Если f_i – приспособленность i-ой особи, то вероятность ее выбора $p_i = \frac{f_i}{\sum_j f_j}$.

Вычисления останавливаются, если выполнено максимальное количество шагов или лучшая хромосома не улучшается на протяжении некоторого количества шагов.

Классический генетический алгоритм при решении задачи поиска максимальной клики графа показывает результаты слабее, чем другие методы локального поиска [1, 2]. Отчасти это объясняется тем, что выбор подходящих фитнесс-функции и семантически осмысленных операторов скрещивания и мутации весьма не прост. Естественный и наиболее распространенный подход для улучшения результатов – применение учитывающих специфику задачи эвристик, которые становятся частью оператора мутации. Иногда даже наивные эвристики могут значительно улучшить результаты. Одна из таких эвристик предложена E. Marchiori [4, 5]. Она состоит из трех шагов: Relax (добавление нескольких вершин к существующему подграфу), Repair (усечение подграфа до клики), Extend (расширение клики).

3.2. Детали реализации

Один из ключевых вопросов реализации – выбор программного представления хромосомы. От него зависит реализация операторов мутации, скрещивания, функции приспособленности и скорость вычислений. Хромосома может представляться списком, вектором, множеством или другой структурой данных. У каждого из этих вариантов есть преимущества и недостатки. Поскольку язык программирования scheme (в реализации DrRacket) поддерживает длинные целые числа и побитовые операции над ними (`bitwise-ior`, `bitwise-xor`, `bitwise-and`, `bitwise-not`, `bitwise-bit-set?`, `arithmetic-shift` и другие [7]), можно представить отдельную хромосому целым числом. Такой подход позволяет эффективно использовать память. Все операции над хромосомами сводятся к нескольким последовательным побитовым умножениям, сложениям и сдвигам, записываются кратко и выполняются быстро. Популяция – список отдельных хромосом.

По условию задачи, граф подается на вход в виде списка ребер. Но работать с таким представлением не удобно, поэтому оно преобразуется в матрицу смежности. Матрица смежности представляет собой вектор из целых чисел (по аналогии с хромосомой).

Параметры алгоритма – размер начальной популяции (100) и вероятность мутации (0.1). Выбранный оператор мутации улучшает решение, но не справляется с

проблемой сходимости к локальному оптимуму. Размер популяции не меняется со временем. Количество генерируемых с помощью скрещиваний потомков равно размеру начальной популяции (100), они мутируют с заданной вероятностью. Выжившие (100) отбираются из списка-объединения потомков и родителей. Лучшая особь сохраняется.

Некоторые функции можно немного оптимизировать. Например, в функции отбора суммарное значение приспособленности вычисляется несколько раз на каждом шаге эволюции, хотя достаточно всего одного. Подобные оптимизации могут незначительно увеличить скорость вычислений, но сильно усложняют код.

4. Визуализация

Визуализация выполняется при помощи библиотек Plot, Racket Drawing Toolkit и Racket Graphical Interface Toolkit [7]. Все данные, необходимые для визуализации конечного результата и процесса приближения решения к оптимальному (входной граф, клика, значение функции приспособленности всех хромосом каждого поколения) собираются на этапе решения задачи и сохраняются в файл. В программе визуализации можно открыть подобный файл (меню `File->Load`). Готовую картинку можно сохранить на диск (меню `File->Save`). Таким образом, поиск решения отделен от визуализации. Для того, чтобы интерфейс не «подвисал», построение изображения выполняется в фоновом потоке. Информация о ходе построения отображается в строке состояния.

4.1. Визуализация решения

Для изображения найденного решения используется силовой алгоритм размещения вершин графа [6]. Он основывается на простом физическом наблюдении: всякая физическая система стремится к минимуму своей энергии, в котором минимум дефектов и наибольшая симметрия. В вершины графа помещаются заряды, которые отталкиваются, будучи одноимёнными, а ребра заменяются на пружины одинаковой равновесной длины и жесткости; система случайным образом размещается на плоскости, приводится в движение и дальше отпускается, эволюционируя по законам физики, пока не придёт в равновесие.

Ниже на рисунке приведен пример визуализации результата: изображен исходный граф, вершины и ребра найденной клики выделены красным цветом.

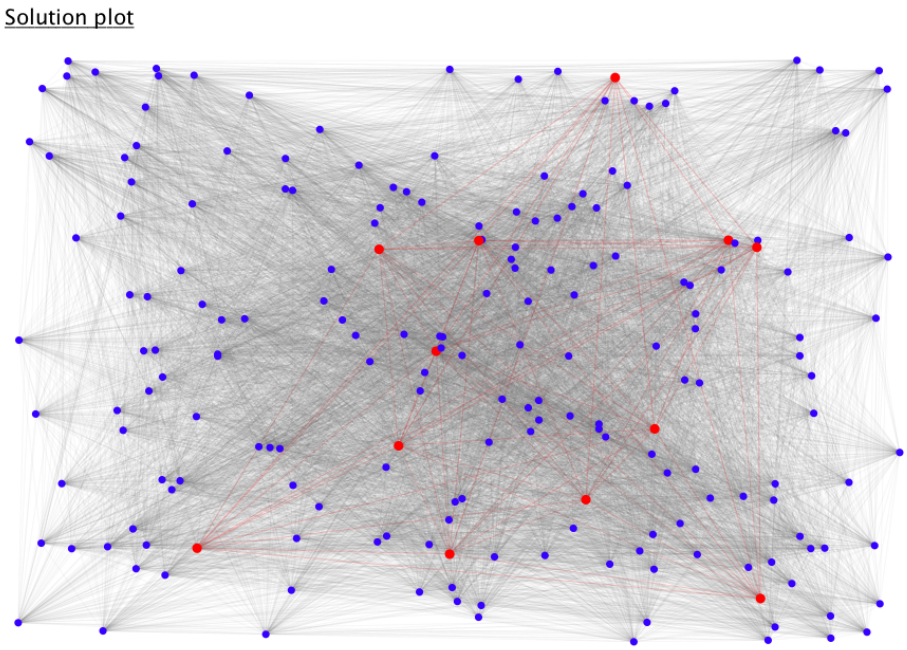

Рисунок 2. Визуализация решения.

4.2. Визуализация процесса приближения решения к оптимальному

Отображение процесса приближения текущего решения задачи к оптимальному состоит из двух рисунков. На первом изображен плоский график минимума, максимума и среднего значение функции отбора на каждом шаге генетического алгоритма. На втором изображен объемный график функции отбора всех хромосом и ее среднего значения на каждом шаге генетического алгоритма. Оба графика строятся при помощи библиотеки Plot [7]. Для навигации по графикам можно использовать меню (`Miscellaneous`).

Ниже на двух рисунках приведены примеры графиков первого и второго типов соответственно.

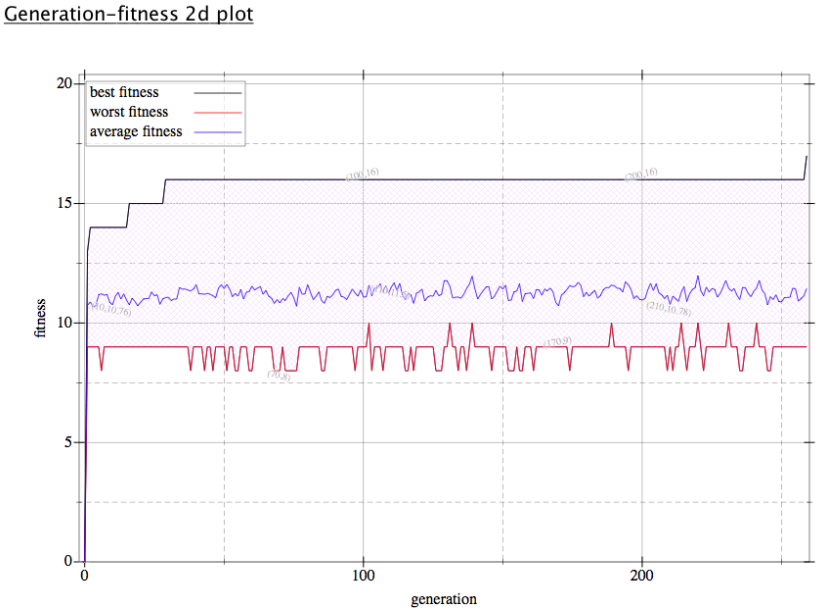

Рисунок 3. Плоский график функции приспособленности.

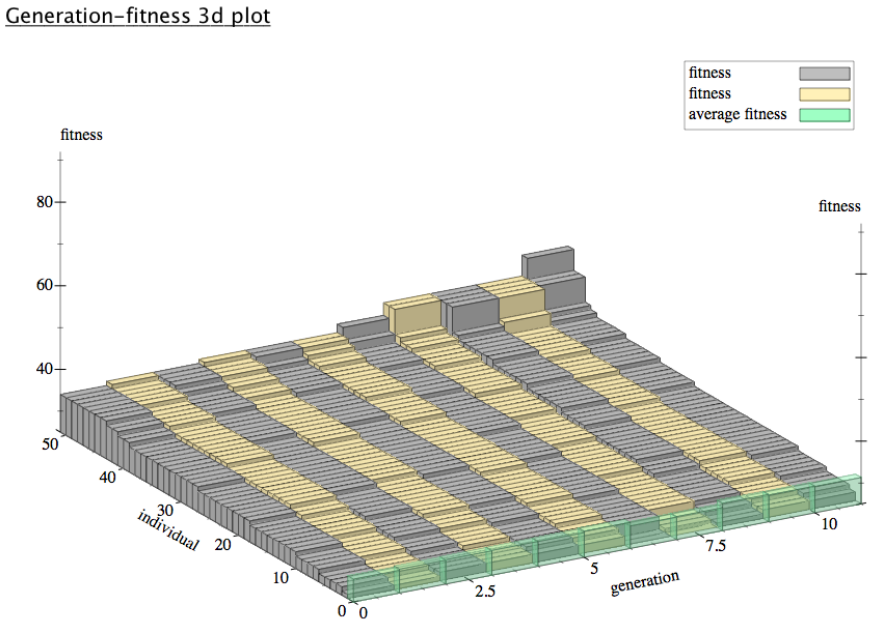

Рисунок 4. Объемный график функции приспособленности.

5. Результаты

В среднем, на вычисления уходит несколько минут, так как тестовые графы достаточно большие и плотные. Некоторые задачи решаются быстро (несколько секунд).

Для ряда тестовых графов удалось найти наилучшее известное решение. Таблица 1 с результатами приведена ниже.

Название теста	Порядок графа	Размер графа	Ответ	Время в миллисекундах	Количество шагов
brock200_2	200	9 875	12	11 881	31
brock200_4	200	13 088	15	6 552	15
p_hat300_1	300	10 932	8	10 547	13
p_hat300_2	300	21 927	25	84 553	91
san200_0.7_1	200	13 929	30	14 204	29
san200_0.9_1	200	17 909	70	236 407	276
san400_0.5_1	400	39 900	13	92 634	61
san400_0.7_1	400	55 859	40	202 895	117
sanr400_0.5	400	39 983	13	367 068	246
sanr400_0.5	400	39 983	13	367 068	246
sanr400_0.7	400	55 868	21	478 364	294
sanr200_0.7	200	13 867	18	117 859	269
sanr200_0.9	200	17 862	42	166 002	252
c-fat200-1	200	1 533	12	2 036	6
c-fat200-2	200	3 234	24	5 065	12
c-fat200-5	200	8 472	58	5 325	11
c-fat500-1	500	4 458	14	5 028	15
c-fat500-2	500	9 138	26	7 606	17
c-fat500-5	500	23 190	64	8 150	13
c-fat500-10	500	46 626	126	15 076	24
DSJC500.5	500	62 623	13	289 818	123
hamming8-4	256	20 863	16	9 443	18
C125.9	125	6 962	34	24 601	76

Таблица 1. Результаты тестирования.

В то же время, результаты на тестовых графах из библиотеки BHOSLIB не идеальны: 34/40 для *frb-40-19-x* (760 вершин, около 246000 ребер), 33/35 для *frb-35-17-x* (595 вершин, около 148000 ребер), 26/30 для *frb-30-15-x* (450 вершин, около 83000 ребер). Это говорит о том, что проблема «застревания» в локальном оптимуме остается открытой.

На некоторых классах задач алгоритм показал слабый результат: *san400_0.9_1* (400 вершин, 71819 ребер) – 83/100; *C500.9* (500 вершин, 112331 ребер) – 48/57; *brock400_4* (400 вершие, 59764 ребер) – 23/33; *gen400_p0.9_75.b* (400 вершин, 71819 ребер) – 53/75; *gen400_p0.9_65.b* (400 вершин, 71819 ребер) – 46/65; *gen400_p0.9_55.b* (400 вершин, 71819 ребер) – 46/55.

Заключение

Для решения поставленной задачи разработан и реализован соответствующий алгоритм (HGA-clique.rkt), система тестирования (ga-test.rkt) и программа для визуализации решения (draw.rkt), в сумме около 1000 строк на языке программирования scheme. Эффективность готового алгоритма проверена на множестве тестовых графов библиотеки BHOSLIB и международной конференции DIMACS. Для ряда из них удалось найти наилучшее решение, для нескольких, напротив, результаты оказались далекими от идеальных. Один из возможных путей развития получившегося алгоритма – применение более сложных, управляемых операторов мутации для решения проблемы «застревания» в локальном оптимуме. Так же, выполнение некоторых операций можно сделать параллельным, что значительно увеличит скорость вычислений.

Генетические алгоритмы успешно применяются для решения множества сложных задач комбинаторной оптимизации. Главное преимущество генетических алгоритмов заключается в том, что они просты и универсальны (подходят для решения широкого круга задач), естественно распараллеливаются. Среди недостатков – повторное вычисление функции приспособленности, «застревание» в локальном оптимуме, плохая масштабируемость, проблема выбора критерия остановки. Все эти проблемы так или иначе решаются. Выбор подходящих функций (мутации, скрещивания, отбора) и его математическое обоснование может оказаться весьма нетривиальной задачей. Если нужно быстро получить результат, можно применять генетический алгоритм.

Противники ГА утверждают, что при прочих равных условиях они проигрывают специальным алгоритмам, рассчитанным на конкретную задачу. Так, например, для решения задачи максимальной клики с успехом применяется множество других алгоритмов (динамическое программирование, локальный поиск, поиск с запретами, другие эвристические алгоритмы). Однако исследование этих алгоритмов и сравнение результатов выходит за рамки данной работы.

Список литературы

1. Carter R., Park K. How good are genetic algorithms at finding large cliques: an experimental study. – Boston University Computer Science Department, 1993.

2. Park K., Carter B. On the effectiveness of genetic search in combinatorial optimization //Proceedings of the 1995 ACM symposium on Applied computing. – ACM, 1995. – С. 329-336.

3. Johnson D. S., Trick M. A. Second DIMACS implementation challenge: cliques, coloring and satisfiability, volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science //American Mathematical Society. – 1996.

4. Marchiori E. Genetic, iterated and multistart local search for the maximum clique problem //Applications of Evolutionary Computing. – Springer Berlin Heidelberg, 2002. – С. 112-121.

5. Marchiori E. A simple heuristic based genetic algorithm for the maximum clique problem //SAC. – 1998. – Т. 98. – С. 366-373.

6. Fruchterman T. M. J., Reingold E. M. Graph drawing by force-directed placement //Software: Practice and experience. – 1991. – Т. 21. – №. 11. – С. 1129-1164.

7. Flatt M., Findler R. B. The Racket Guide. – 2013.

Приложение. Код программы

ga-test.rkt

```
#lang racket

(require racket/dict)
(require rackunit)
(require rackunit/text-ui)
;(require rackunit/gui)

;(require "HGA-clique.rkt")
(require "HGA-clique.rkt")

;образуют ли выданные вершины клику?
(define (is-clique? vertices graph)

  ;построим полный граф на данных вершинах
  (define (build-complete-graph complete-graph vertices)
    (if (null? vertices)
        complete-graph
        (build-complete-graph (append (map (lambda(vertice) (list (car vertices) vertice)) (cdr vertices)) complete-graph) (cdr vertices))))

  ;проверим, что в исходном графе есть все ребра полного графа на данных вершинах (в set поиск побыстрее)
  (let ((graph-set (list->set graph)))
    (andmap (lambda(edge) (or (set-member? graph-set edge) (set-member? graph-set (reverse edge)))) (build-complete-graph '() vertices))))

;читаем файл с набором тестов
;return: список путей к файлам с данными для тестов
(define (read-tests-file file-path)

  ;построчное чтение файла
  ;пустые строки и строки, начинающиеся с $ (комментарии) игнорируются
  ;return: список строк файла
  (define (read-file test-cases)

    (let ((next-line (read-line (current-input-port) 'any)))

      (if (eof-object? next-line)
          (filter-not (lambda(str) (or (zero? (string-length str)) (equal? #\$ (string-ref str 0)))) test-cases)
          (read-file (cons (string-trim next-line) test-cases)))))

  (with-input-from-file file-path #:mode 'text (lambda () (read-file '()))))

;читаем данные для теста из файла
;формат файла:
;;description
;;K - требуемый размер клики
;;граф в виде ((a b) (b c) ...)
(define (read-test-data test-file)
```

```
       #|(define (make-map edges)

         (define (make-map-proc n v-map edges)
           (cond ((null? edges) v-map)
                 ((dict-has-key? v-map (car edges)) (make-map-proc n v-map (cdr
edges)))
                 (else (make-map-proc (add1 n) (dict-set v-map (car edges) n) (cdr
edges)))))

         (make-map-proc 0 #hash() (remove-duplicates (flatten edges))))|#

       #|(define (map-graph edges)

         (let ((v-map (make-map edges)))
           (map (lambda(edge) (list (dict-ref v-map (first edge) -1) (dict-ref v-
map (second edge) -1))) edges)))|#

       (with-input-from-file test-file #:mode 'text
         (lambda () (list (read-line (current-input-port) 'any) (read) (read)))))

     (define (proc x) (length x))

     ;создаем тесты
     (define (do-test file-path)

       ;читаем из файла данные для теста, запускаем тест
       ;проверяем результат - выданный список вершин образует клику размера не
меньше, чем требовалось
       (define (create-test test-file)

         (let ((test-data (read-test-data test-file)))

           (lambda()

             (let* ((answer (HGA-MCP (first test-data) (second test-data) (third
test-data)))
                    (clique (cdr answer))
                    (ga-steps (car answer))
                    (clique-size (length (remove-duplicates clique)))
                    (vertices (length (remove-duplicates (flatten (third test-
data)))))
                    (edges (length (remove-duplicates (third test-data)))))

               (define test-desc (string-append (first test-data) " (" (number-
>string vertices) " vertices, " (number->string edges) " edges" ")"
                  " best known: " (number->string (second test-data))))

               (test-check
                 test-desc
                 (lambda(x y)
                   (printf "test: ~a \n" test-desc)
                   (printf "steps: ~a \n" (number->string ga-steps))
                   (printf "found: ~a is-clique: ~a \n" clique-size x))
                 (is-clique? clique (third test-data)) (<= (second test-data)
clique-size))))))
```

```
       (define ga-mcp-tests
         (test-suite
          "Genetic algorithm for maximum clique problem."
          (for-each (lambda(test-file) ((create-test test-file))) (read-tests-file
file-path))))

       ;запуск тестов с отображением результатов в текстовом режиме
       (run-tests ga-mcp-tests 'verbose))

       ;запуск тестов с отображением результатов в gui
       ;(test/gui ga-mcp-tests))

     (do-test "clique-test/test.txt")
```

HGA-clique.rkt

```
#lang racket

;fitness proportionate selection, also known as roulette wheel selection
(define (roulette-wheel-selection items)

  (define (select random-number items)
    (let ((random-number (- random-number (caar items))))
      (if (> random-number 0) (select random-number (cdr items)) (car items))))

  (let ((sumw (foldl + 0 (map car items))))
    (cond ((null? items) '())
          ((<= sumw 0) (pick-random items))
          (else (select (* (random) sumw) items)))))

(define (fifty-fifty . something) (<= (random) 0.5))

(define (build-initial-population population-size chromosome-size)

  (define (build-population population)
    (if (<= population-size (length population))
        population
        (let ((vertices (filter-map (lambda(x) (if (<= (random) 0.2);(/ (sqrt
chromosome-size) chromosome-size))
          (arithmetic-shift 1 x) #f)) (range chromosome-size))))
          (build-population (cons (foldl bitwise-ior 0 vertices) population)))))

  (build-population '()))

;get vertices from chromosome, e.g. 10101 -> (1 3 4)
(define (extract-vertices chromosome chromosome-size)
  (filter-map (lambda(vertice) (if (bitwise-bit-set? chromosome vertice) vertice
#f)) (range chromosome-size)))

(define (extract-vertices-2 chromosome chromosome-size)
  (extract-vertices (bitwise-not chromosome) chromosome-size))

;return true, if given chromosome is clique in graph, false otherwise
```

```
(define (is-clique? chromosome chromosome-size graph-matrix)

  (andmap (lambda(x) (or (not (bitwise-bit-set? chromosome x))
                         (equal? chromosome (bitwise-and chromosome (vector-ref
graph-matrix x)))))
          (range chromosome-size)))

(define (is-clique?-2 chromosome chromosome-size graph-matrix)

  (foldl + 0 (filter-map (lambda(x) (if (not (bitwise-bit-set? chromosome x))
                                        #f
                                        (integer-count (bitwise-and chromosome
(bitwise-not (vector-ref graph-matrix x))))))
                         (range chromosome-size))))

;f(chromosome) = number of vertices in given chromosome, if it's a clique, 0
otherwise
(define (get-fitness chromosome chromosome-size graph-matrix)
  (if (is-clique? chromosome chromosome-size graph-matrix) (integer-count
chromosome) 0))

;f(chromosome) = N_vertices * exp(- N_missing_edges / c)
;so if given chromosome is a clique, f(chromosome) = N_vertices
(define (get-fitness-2 chromosome chromosome-size graph-matrix)
  (/ (/ (* (integer-count chromosome) (integer-count chromosome)) 2) (add1 (is-
clique?-2 chromosome chromosome-size graph-matrix))))
  ;(* (integer-count chromosome)
     ;(exp (/ (* 1.0 (- (is-clique?-2 chromosome chromosome-size graph-matrix)))
100))))

;uniform crossover
(define (ux-crossover father mother chromosome-size)

    (let* ((intersection (bitwise-and father mother))
           (difference (bitwise-xor father mother))
           (cross (filter-map
                   (lambda(x) (if (and (fifty-fifty) (bitwise-bit-set? difference
x)) (arithmetic-shift 1 x) #f))
                   (range chromosome-size)))
           (crossed (foldl bitwise-ior 0 cross)))

        (list (bitwise-ior intersection crossed) (bitwise-ior intersection
(bitwise-xor difference crossed)))))

;x, y = random, random; chromosome[x], chromosome[y] = chromosome[y], chromosome[x]
(define (swap-mutatio chromosome chromosome-size)

  (let ((x (random chromosome-size)) (y (random chromosome-size)))

    (set-bit (set-bit chromosome x (bitwise-bit-set? chromosome y)) y (bitwise-bit-
set? chromosome x))))

;return number of vertices in given chromosome
(define (integer-count chromosome)
  (count (lambda(x) (bitwise-bit-set? chromosome x)) (range (integer-length
chromosome))))
```

```
;x = random; chromosome[x] = not chromosome[x]
(define (inverse-mutatio chromosome chromosome-size)
  (flip-bit chromosome (random chromosome-size)))

;chromosome[idx] = 0
(define (down-bit chromosome idx)
  (bitwise-and chromosome (bitwise-not (arithmetic-shift 1 idx))))

;chromosome[idx] = 1
(define (up-bit chromosome idx)
  (bitwise-ior chromosome (arithmetic-shift 1 idx)))

;chromosome[idx] = not chromosome[idx]
(define (flip-bit chromosome idx)
  (if (bitwise-bit-set? chromosome idx) (down-bit chromosome idx) (up-bit
chromosome idx)))

;chromosome[idx] = value
(define (set-bit chromosome idx value)
  (if value (up-bit chromosome idx) (down-bit chromosome idx)))

;sort vertices by order
(define (sort-vertices vertices graph-order graph-matrix)
  (sort (if (null? vertices) (range graph-order) vertices)
        #:key (lambda(vertice) (integer-count (vector-ref graph-matrix vertice)))
<))

;step0:relax
(define (relax chromosome graph-matrix graph-order)

  (define (remove chromosome vertices)

    (if (null? vertices) chromosome
        (if (and (bitwise-bit-set? chromosome (car vertices)) (< (random) 0.5))
            (remove (down-bit chromosome (car vertices)) (cdr vertices))
            (remove chromosome (cdr vertices)))))

  (define (add chromosome vertices)

    (if (null? vertices) chromosome
        (if (and (not (bitwise-bit-set? chromosome (car vertices))) (< (random)
0.1))
            (add (up-bit chromosome (car vertices)) (cdr vertices))
            (add chromosome (cdr vertices)))))

  ;chromosome)
  ;(let ((vertices (sort-vertices
  ;                 (build-list (quotient graph-order 20) (lambda(x) (random graph-
order)))
  ;                 graph-order graph-matrix)))
  ;  (add (remove chromosome vertices) (reverse vertices))))

  (let ((vertices (sort-vertices ;(range graph-order)
                   (build-list (quotient graph-order 10) ;(inexact->exact (round (*
2 (sqrt graph-order))))
```

```
                              (lambda(x) (random graph-order)))
                    graph-order graph-matrix)))
    (add (remove chromosome (take vertices (quotient (length vertices) 2)))
         (take-right vertices (quotient (length vertices) 2)))))


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

;E.Marchiore heorustic first step : repair
(define (repair chromosome graph-matrix graph-order)

  (define (repair-proc chromosome vertices)
    (if (null? vertices)
        chromosome
        (let ((i (pick-random vertices)))
          (if (< (random) 0.1)
              (repair-proc (down-bit chromosome i) (remove i vertices))
              (let ((new-chromosome (bitwise-and chromosome (vector-ref graph-
matrix i))))
                (repair-proc new-chromosome (remove i (extract-vertices new-
chromosome graph-order))))))))

  (repair-proc chromosome (extract-vertices chromosome graph-order)))

;step2 extend
(define (extend chromosome graph-matrix graph-order)

  (define (extend-proc chromosome vertices)

    (cond ((null? vertices) chromosome)
          ((bitwise-bit-set? chromosome (car vertices)) chromosome)
          (else (let ((new-chromosome (up-bit chromosome (car vertices))))
                  (extend-proc (if (is-clique? new-chromosome graph-order graph-
matrix) new-chromosome chromosome) (cdr vertices))))))

  (extend-proc chromosome (shuffle (extract-vertices-2 chromosome graph-order))))


(define (HGA-MCP output-file task graph)

  (define (best-fitness population) (car (argmax car population)))
  (define (worst-fitness population) (car (argmin car population)))
  (define (average-fitness population) (inexact->exact (round (/ (foldl + 0 (map
car population)) 1.0 (length population)))))

  (define (apply-HA population graph-matrix graph-order)
    ;population)
    (map (lambda(chromosome) (extend (repair (relax chromosome graph-matrix graph-
order) graph-matrix graph-order) graph-matrix graph-order)) population))

  (define (apply-crossover population cross-rate chromosome-size)

    (define (build-new-population new-population population-size)
      (if (<= population-size (length new-population))
          new-population
```

```
          (let ((father (roulette-wheel-selection population)) (mother (roulette-
wheel-selection population)))
            (build-new-population (append new-population (ux-crossover (cdr father)
(cdr mother) chromosome-size)) population-size))))

    (build-new-population '() (* cross-rate (length population))))

  (define (apply-mutatio population mutatio-probability chromosome-size)
    (map (lambda(chromosome) (if (< (random) mutatio-probability)
                                 (inverse-mutatio chromosome chromosome-size)
                                 chromosome))
         population))

  (define (apply-selection population select-number)

    (define (make-selection new-population)
      (if (<= select-number (length new-population))
          new-population
          (make-selection (cons (roulette-wheel-selection population) new-
population))))

    ;(printf "selection0:") (print (length (remove-duplicates population)))
(newline)
    ;(printf "selection1:") (print (length (remove-duplicates (make-selection
'()))))) (newline)
    (make-selection (list (argmax car population))))

  (define (map-fitness chromosome-size population graph-matrix)

    ;(print (length (remove-duplicates population))) (newline)
    ;(for-each (lambda(x) (printf "~b ~a\n" x (get-fitness-2 x chromosome-size
graph-matrix))) population )

    (map (lambda(chromosome) (cons (get-fitness chromosome chromosome-size graph-
matrix) chromosome)) population))

  (define initial-size 100)
  (define cross-rate 1)
  (define mutatio-probability 0.1)
  ;(define elites 2)
  (define ga-steps 200)

  (define (find-maximum-clique graph-order population steps graph-matrix fitness-
array)

    (cond ((or (< steps 0) (>= (car (argmax car population)) task))

           ;(printf "best:~a found:~a steps:~a\n" task (argmax car population)
steps)

           (with-output-to-file (string-append "out/" output-file ".out")

             (lambda()
               (printf "~a\n" (list initial-size mutatio-probability (* initial-size
cross-rate 1/2)))
```

```
              (printf "~a\n" (extract-vertices (cdr (argmax car population)) graph-
order))
              (printf "~a\n" fitness-array)
              (printf "~a\n" graph))
             #:mode 'text #:exists 'replace)

            (cons steps (extract-vertices (cdr (argmax car population)) graph-
order))
            ;(print (argmax car population))(newline)
            )
           (else
            ;(printf "begin:") (print (length (remove-duplicates population)))
(newline)
            (let ((new-cross-population (append
                                         (apply-crossover population cross-rate
graph-order)
                                         (build-initial-population (/ initial-size
2) graph-order))))
              ;(printf "cross:") (print (length (remove-duplicates new-cross-
population))) (newline)
              ;(print new-cross-population)(newline)
              (let ((new-mut-population ;(apply-mutatio new-cross-population
mutatio-probability graph-order)))
                (apply-HA (apply-mutatio new-cross-population mutatio-
probability graph-order) graph-matrix graph-order)))

               (let ((new-fit-population (map-fitness graph-order
                                                      new-mut-population
                                                      ;(append new-mut-population
                  ;(apply-HA (build-initial-population initial-size graph-order)
graph-matrix graph-order))
                                                      graph-matrix)))
                 ;(printf "mutatio:") (print (length (remove-duplicates new-mut-
population))) (newline)
                 ;(print (append population new-mut-population))(newline)
                 (let ((new-sel-population (apply-selection (append population new-
fit-population) initial-size)))
                   ;(printf "selection:") (print (length (remove-duplicates new-
sel-population))) (newline)

                   ;(print (apply-selection (append population new-mut-population)
initial-size))
                   ;(print new-sel-population)
                   (printf "~a :: best: ~a worst: ~a total: ~a different: ~a\n"
                           steps (car (argmax car new-sel-population)) ;(caar (sort
new-sel-population #:key car >))
                           (car (argmin car new-sel-population)) ;(car (last (sort
new-sel-population #:key car >)))
                           (foldl + 0 (map car new-sel-population))
                           (length (remove-duplicates new-sel-population)))

                   (find-maximum-clique graph-order new-sel-population (sub1 steps)
graph-matrix
                                        ;(append fitness-array (list (best-fitness
population)
                                        ; (average-fitness population) (worst-
fitness population)))  ))))))))
```

```
                                        (append fitness-array (list (map car new-
sel-population)))  ))))))))

  (define (build-graph-matrix graph-set graph-order)

    (define (make-line i)

      (foldl bitwise-ior
             0
             (filter-map (lambda(j) (if (or (equal? i j)
                                            (set-member? graph-set (list i j))
                                            (set-member? graph-set (list j i)))
                                        (arithmetic-shift 1 j)
                                        #f))
                         (range graph-order))))

    (build-vector graph-order make-line))

  (time (let* ((graph-order (length (remove-duplicates (flatten graph))))
         (initial-population (build-initial-population initial-size graph-order))
         (graph-matrix (build-graph-matrix (list->set graph) graph-order))
         (init-pop-fitness (map-fitness graph-order initial-population graph-
matrix)))

      (find-maximum-clique graph-order init-pop-fitness ga-steps graph-matrix
'()))))

(provide HGA-MCP)
```

draw.rkt

```
#lang racket/gui

(require racket/gui/base)
(require racket/draw)
(require racket/dict)
(require plot)
(require plot/utils)

;main window size
(define window_width 950)
(define window_height 700)
;canvas size
(define picture_width 950)
(define picture_height 3500)
;graph picture size
(define graph_width 900)
(define graph_height 600)
;plot picture size
(define plot_width 800)
(define plot_height 550)

;this is main window proc
(define (main-window)
```

```
  (define frame (new frame% [label "Visualization"] [style '(no-resize-border
toolbar-button)]
                     [width window_width] [height window_height])) ;make main
window

  (send frame create-status-line) ;make status line
  (send frame set-status-text "Click menu File -> Load and open data file!")

  (define canvas (new canvas% [parent frame] [style '(vscroll border)]
                      [paint-callback (lambda(canvas dc)
                                        (send dc set-smoothing 'smoothed)
                                        (send dc draw-bitmap picture 0 0))])) ;make
main canvas
  (send canvas init-auto-scrollbars picture_width picture_height 0.0 0.0) ;show
vertical scroll-bar

  (define picture (send canvas make-bitmap picture_width picture_height)) ;make
main bitmap picture
  (define dc (new bitmap-dc% [bitmap picture])) ;get drawing context from main
bitmap picture
  (send dc set-smoothing 'smoothed)
  (send dc set-background "white")

  (define menu-bar (new menu-bar% [parent frame] [demand-callback
(lambda(m)(void))])) ;make menu bar
  (define file-menu (new menu% [label "File"] [parent menu-bar] [help-string "Fain
menu"])) ;make menu-item "File"

  (define (on-menu-save m-item c-event) ;save current picture to file
    (let ((filepath (put-file "Save picture!" frame "." "picture" "jpg")))
      (if filepath
          (begin
            (send picture save-file filepath 'jpeg 100)
            (send frame set-status-text (string-append "Picture saved to " (path-
>string filepath))))
          #f)))
  (define menu-save (new menu-item% [label "Save"] [parent file-menu]
                         [callback on-menu-save] [help-string "Save picture on
disk!"])) ;make menu item File->Save

  (define (on-menu-load m-item c-event) ;load data from file and begin drawing on
picture
    (let ((filepath (get-file "Load data file!" frame "." "data" "txt")))
      (define (call-load-and-draw) (load-and-draw frame misc-menu dc filepath))
      (if filepath (thread call-load-and-draw) #f)))
  (define menu-load (new menu-item% [label "Load"] [parent file-menu]
                         [callback on-menu-load] [help-string "Load data from
disk!"])) ;make menu item File->Load

  (define misc-menu (new menu% [label "Miscellaneous"] [parent menu-bar] [help-
string "Miscellaneous menu"])) ;make menu-item "Miscellaneous"

  (define about-string (string-append "Visualization of a heuristic based
genetic\n"
                                      "algorithm for the maximum clique problem.\n"
```

```
                                        "\nFaculty of Computational Mathematics and
Cybernetics\n"
                                        "Moscow State University"))
  (define (on-menu-about m-item c-event) (message-box "About..." about-string frame
'(ok no-icon)))
  (define menu-misc-about (new menu-item% [label "About..."] [parent misc-menu]
                          [callback on-menu-about] [help-string "Show about message
box!"])) ;make menu item Misc->About...

  (define misc-menu-sep (new separator-menu-item% [parent misc-menu]))

  (define (on-menu-2dplot m-item c-event) (send canvas scroll #f (/ 350
picture_height)))
  (define menu-misc-2dplot (new menu-item% [label "Show 2d generation-fitness
plot"] [parent misc-menu]
                          [callback on-menu-2dplot] [help-string "Scroll to 2d
plot!"])) ;make menu item Misc->Show 2d plot
  (define (on-menu-3dplot m-item c-event) (send canvas scroll #f (/ 1150
picture_height)))
  (define menu-misc-3dplot (new menu-item% [label "Show 3d generation-fitness
plot"] [parent misc-menu]
                          [callback on-menu-3dplot] [help-string "Scroll to 3d
plot!"])) ;make menu item Misc->Show 3d plot
  (define (on-menu-splot m-item c-event) (send canvas scroll #f (/ 2050
picture_height)))
  (define menu-misc-splot (new menu-item% [label "Show solution plot"] [parent
misc-menu]
                          [callback on-menu-splot] [help-string "Scroll to solution
plot!"])) ;make menu item Misc->Show solution plot

  (send frame show #t))

(define (load-and-draw frame misc-menu drawing-context filepath)
  (with-input-from-file filepath #:mode 'text
    (lambda ()

      (send frame set-status-text (string-append "Loading data from " (path->string
filepath)))
      (send drawing-context set-background "white")
      (send drawing-context clear)

      (let ((header (read)) (clique (read)) (fitness-array (map (lambda(f-arr)
(sort f-arr <)) (read))) (graph (read)))

        (send frame set-status-text
              (string-append "Loading data from " (path->string filepath) ":
header"))
        (draw-header drawing-context filepath header clique fitness-array graph)
        (send frame refresh)

        (send frame set-status-text
              (string-append "Loading data from " (path->string filepath) ":   2d
plot"))
        (simple-plot drawing-context fitness-array plot_width plot_height)
        (send frame refresh)

        (send frame set-status-text
```

```
              (string-append "Loading data from " (path->string filepath) ":   3d
plot"))
        (smart-plot drawing-context fitness-array plot_width plot_height)
        (send frame refresh)

        (send frame set-status-text
              (string-append "Loading data from " (path->string filepath) ":
solution plot"))
        (solution-plot drawing-context frame filepath clique graph graph_width
graph_height)
        (send frame refresh)

        (define 3dplot-frame (smart-plot-2 drawing-context fitness-array plot_width
plot_height))
        (define (on-menu-zzz m-item c-event) (send 3dplot-frame show #t))
        (define menu-sep3dplot (new menu-item% [label "Show 3d generation-fitness
plot in separate window"] [parent misc-menu]
                                    [callback on-menu-zzz] [help-string "Show 3d plot
in separate window!"])) ;make menu item Misc->Show separate 3d plot

        (send frame set-status-text
              (string-append "Loading data from " (path->string filepath) ":
loaded successfully"))))))

;draw summary, e.g. graph size, clique size, ga steps, etc.
(define (draw-header drawing-context filepath header clique fitness-array graph)
  ;(send drawing-context set-brush "white" 'solid)
  ;(send drawing-context draw-rectangle 0 0 picture_width picture_height)
  (send drawing-context set-font (make-font #:size 20 #:family 'system #:weight
'normal #:underlined? #t #:smoothing 'partly-smoothed))
  (send drawing-context draw-text (string-append "Summary for file " (path->string
filepath)) 20 20)
  (send drawing-context set-font (make-font #:size 16 #:family 'system #:weight
'normal #:smoothing 'partly-smoothed))
  (send drawing-context draw-text
        (string-append "Graph order: |V(G)| = " (number->string (length (remove-
duplicates (flatten graph))))) 50 60)
  (send drawing-context draw-text (string-append "Graph size: |E(G)| = " (number-
>string (length (remove-duplicates graph)))) 50 80)

  (define (makeclique-string)
    (define str (string-join (map number->string (sort (set->list clique) <)) ","))
    (list->string (map (lambda(x y) (if (and (equal? x #\,) (zero? (modulo y 10)))
#\linefeed x))
      (string->list str) (range (string-length str)))))

  (send drawing-context draw-text
        (string-append "Clique size: |V(C)| = " (number->string (set-count
clique))) 50 110)
  (send drawing-context draw-text
        (string-append "Clique: V(C) = { " (string-join (map number->string (sort
(set->list clique) <)) ",") " }") 50 130)
  (send drawing-context draw-text
        (string-append "GA steps: " (number->string (length fitness-array))) 50
160)
  (send drawing-context draw-text (string-append "Population size: " (number-
>string (first header))) 50 180)
```

```
  (send drawing-context draw-text (string-append "Mutation probability: " (number-
>string (second header))) 50 200)
  (send drawing-context draw-text (string-append "Crossovers: " (number->string
(third header))) 50 220))



;2d fitness plot
(define (simple-plot drawing-context fitness-array width height)

  (send drawing-context set-font (make-font #:size 20 #:family 'system #:weight
'normal #:underlined? #t #:smoothing 'partly-smoothed))
  (send drawing-context draw-text "Generation-fitness 2d plot" 20 300)

  (define (best-fitness step) (if (>= step (length fitness-array)) -1 (argmax
values (list-ref fitness-array step))))
  (define (worst-fitness step) (if (>= step (length fitness-array)) -1 (argmin
values (list-ref fitness-array step))))
  (define (average-fitness step) (if (>= step (length fitness-array)) -1
                                     (/ (foldl + 0 (list-ref fitness-array step))
1.0 (length (list-ref fitness-array step)))))

  (define x-max0 (sub1 (length fitness-array)))
  (define x-mids0 (linear-seq 0 x-max0 (add1 x-max0)))
  (define y-max0 (foldl max -1 (flatten fitness-array)))
  (define y-min0 (foldl min y-max0 (flatten fitness-array)))

  (plot-x-label "generation")
  (plot-y-label "fitness")
  (plot-font-size 16)
  (label-point-size 0.00001)

  (plot/dc (list (tick-grid)

                 (lines-interval (for/list ([x (in-list x-mids0)]) (vector x (best-
fitness x)))
                                 (for/list ([x (in-list x-mids0)]) (vector x
(worst-fitness x)))
                                 #:x-min -2 #:x-max (add1 x-max0)
                                 #:y-min y-min0 #:y-max (* y-max0 1.2)
                                 #:color 6 #:alpha 0.7 #:style 'crossdiag-hatch)

                 (lines (for/list ([x (in-list x-mids0)]) (vector x (best-fitness
x)))
                        #:x-min 0 #:x-max (add1 x-max0)
                        #:y-min y-min0 #:y-max y-max0
                        #:color "black" #:width 0.8 #:label "best fitness" #:style
'solid)

                 (lines (for/list ([x (in-list x-mids0)]) (vector x (worst-fitness
x)))
                        #:x-min 0 #:x-max (add1 x-max0)
                        #:y-min y-min0 #:y-max y-max0
                        #:color "red" #:width 0.8 #:label "worst fitness")

                 (lines (for/list ([x (in-list x-mids0)]) (vector x (average-
fitness x)))
                        #:x-min 0 #:x-max (add1 x-max0)
```

```
                        #:y-min y-min0 #:y-max y-max0
                        #:color "blue" #:width 0.8 #:label "average fitness")

                  #|(points (append (for/list ([x (in-list x-mids0)]) (vector x
(best-fitness x)))
                                   (for/list ([x (in-list x-mids0)]) (vector x
(average-fitness x)))
                                   (for/list ([x (in-list x-mids0)]) (vector x
(worst-fitness x))))
                            #:x-min -1 #:x-max (add1 x-max0)
                            #:y-min (sub1 y-min0) #:y-max (add1 y-max0)
                            #:fill-color "black" #:size 1.5 #:sym '4star #:color
"black" #:line-width 1)|#

                  (for/list ([x (in-list (filter (lambda(x) (zero? (remainder x
100))) x-mids0))])
                    (point-label (vector x (best-fitness x)) #:alpha 0.3 #:color
"black" #:size 10
                                 #:angle (if (< (best-fitness x) (best-fitness (add1
x))) (- 0.2) (+ 0.2))
                                 #:anchor (if (< (best-fitness x) (best-fitness
(add1 x))) 'top 'bottom)))

                  (for/list ([x (in-list (filter (lambda(x) (zero? (remainder (+ 30
x) 100))) x-mids0))])
                    (point-label (vector x (worst-fitness x)) #:alpha 0.3 #:color
"black" #:size 10
                                 #:angle (if (< (worst-fitness x) (worst-fitness
(add1 x))) (- 0.2) (+ 0.2))
                                 #:anchor (if (< (worst-fitness x) (worst-fitness
(add1 x))) 'top 'bottom)))

                  (for/list ([x (in-list (filter (lambda(x) (zero? (remainder (+ 90
x) 100))) x-mids0))])
                    (point-label (vector x (average-fitness x)) #:alpha 0.3 #:color
"black" #:size 10
                                 #:angle (if (< (average-fitness x) (average-fitness
(add1 x))) (- 0.2) (+ 0.2))
                                 #:anchor (if (< (average-fitness x) (average-
fitness (add1 x))) 'top 'bottom)))
                                 ) drawing-context 50 370 width height))

;3d fitness plot in separate window
(define (smart-plot-2 drawing-context fitness-array width height)

  (define frame (new frame% [label "Visualization"] [style '(no-resize-border
toolbar-button)]
                     [width width] [height height])) ;make main window

  (define (average-fitness step) (if (>= step (length fitness-array)) -1
                                     (/ (foldl + 0 (list-ref fitness-array step))
1.0 (length (list-ref fitness-array step)))))

  (define x-max (length fitness-array))
  (define y-max (length (car fitness-array)))
  (define z-max (foldl max -1 (flatten fitness-array)))
  (define z-min (foldl min z-max (flatten fitness-array)))
```

```
  (define (norm2 x y) (list-ref (list-ref fitness-array x) y))
  (define x-ivls (bounds->intervals (linear-seq 0 x-max (add1 x-max))))
  (define y-ivls (bounds->intervals (linear-seq 1 (add1 y-max) (add1 y-max))))
  (define x-mids (linear-seq 0 x-max (add1 x-max)))
  (define y-mids (linear-seq 0 y-max (add1 y-max)))

  (plot-title "Generation-fitness 3d plot")
  (plot-x-label "generation")
  (plot-y-label "individual")
  (plot-z-label "fitness")
  (plot-z-far-label "fitness")
  (plot-font-size 16)
  (plot3d-angle 30)
  (plot3d-altitude 35)
  (plot-legend-anchor 'top-right)
  ;(plot3d-ambient-light 1)
  (plot3d-frame (list (rectangles3d (append*
                                     (for/list ([y-ivl (in-list y-ivls)] [y (in-list
y-mids)])
                                       (for/list ([x-ivl (in-list (filter-map
(lambda(x y) (if (even? y) x #f)) x-ivls (range x-max)))]
                                                  [x (in-list (filter-map (lambda(x y)
(if (even? y) x #f)) x-mids (range (add1 x-max))))])
                                         (define z (norm2 x y))
                                         (vector x-ivl y-ivl (ivl 0 z)))))
                                    #:x-min 0 #:x-max x-max
                                    #:y-min 0 #:y-max (add1 y-max)
                                    #:z-min z-min #:z-max (* 2 z-max)
                                    #:alpha 1  #:style 'solid
                                    #:color "gray" #:line-color "black" #:line-width
0.5 ;#:line-style '
                                    #:label "fitness")
                      (rectangles3d (append*
                                     (for/list ([y-ivl (in-list y-ivls)] [y (in-list
y-mids)])
                                       (for/list ([x-ivl (in-list (filter-map
(lambda(x y) (if (odd? y) x #f)) x-ivls (range x-max)))]
                                                  [x (in-list (filter-map (lambda(x y)
(if (odd? y) x #f)) x-mids (range (add1 x-max))))])
                                         (define z (norm2 x y))
                                         (vector x-ivl y-ivl (ivl 0 z)))))
                                    #:x-min 0 #:x-max x-max
                                    #:y-min 0 #:y-max (add1 y-max)
                                    #:z-min z-min #:z-max (* 2 z-max)
                                    #:alpha 1  #:style 'solid
                                    #:color 4 #:line-color "black" #:line-width 0.5
;#:line-style '
                                    #:label "fitness")
                      (rectangles3d (append
                                     (for/list ([x-ivl (in-list x-ivls)]
                                                [x (in-list x-mids)])
                                       (vector x-ivl (ivl 0 1) (ivl 0 (average-fitness
x)))))
                                    #:x-min 0 #:x-max x-max
                                    #:y-min 0 #:y-max (add1 y-max)
                                    #:z-min z-min #:z-max (* 2 z-max)
                                    #:alpha 2/4  #:style 'solid
```

```
                                  #:color 2 #:line-color "black" #:line-width 0.5
;#:line-style '
                                  #:label "average fitness"))))

;3d fitness plot
(define (smart-plot drawing-context fitness-array width height)

  (send drawing-context set-font (make-font #:size 20 #:family 'system #:weight
'normal #:underlined? #t #:smoothing 'partly-smoothed))
  (send drawing-context draw-text "Generation-fitness 3d plot" 20 950)

  (define (average-fitness step) (if (>= step (length fitness-array)) -1
                                     (/ (foldl + 0 (list-ref fitness-array step))
1.0 (length (list-ref fitness-array step)))))

  (define x-max (length fitness-array))
  (define y-max (length (car fitness-array)))
  (define z-max (foldl max -1 (flatten fitness-array)))
  (define z-min (foldl min z-max (flatten fitness-array)))

  (define (norm2 x y) (list-ref (list-ref fitness-array x) y))
  (define x-ivls (bounds->intervals (linear-seq 0 x-max (add1 x-max))))
  (define y-ivls (bounds->intervals (linear-seq 1 (add1 y-max) (add1 y-max))))
  (define x-mids (linear-seq 0 x-max (add1 x-max)))
  (define y-mids (linear-seq 0 y-max (add1 y-max)))

  ;(plot-title "Evolution")
  (plot-x-label "generation")
  (plot-y-label "individual")
  (plot-z-label "fitness")
  (plot-z-far-label "fitness")
  (plot-font-size 16)
  (plot3d-angle 30)
  (plot3d-altitude 35)
  (plot-legend-anchor 'top-right)
  ;(plot3d-ambient-light 1)
  (plot3d/dc (list (rectangles3d (append*
                                  (for/list ([y-ivl (in-list y-ivls)] [y (in-list
y-mids)])
                                    (for/list ([x-ivl (in-list (filter-map
(lambda(x y) (if (even? y) x #f)) x-ivls (range x-max)))]
                                                [x (in-list (filter-map (lambda(x y)
(if (even? y) x #f)) x-mids (range (add1 x-max))))])
                                      (define z (norm2 x y))
                                      (vector x-ivl y-ivl (ivl 0 z)))))
                                  #:x-min 0 #:x-max x-max
                                  #:y-min 0 #:y-max (add1 y-max)
                                  #:z-min z-min #:z-max (* 2 z-max)
                                  #:alpha 1  #:style 'solid
                                  #:color "gray" #:line-color "black" #:line-width
0.5 ;#:line-style '
                                  #:label "fitness")
                   (rectangles3d (append*
                                  (for/list ([y-ivl (in-list y-ivls)] [y (in-list
y-mids)])
                                    (for/list ([x-ivl (in-list (filter-map
(lambda(x y) (if (odd? y) x #f)) x-ivls (range x-max)))]
```

```
                                                [x (in-list (filter-map (lambda(x y)
(if (odd? y) x #f)) x-mids (range (add1 x-max))))])
                                      (define z (norm2 x y))
                                      (vector x-ivl y-ivl (ivl 0 z)))))
                                #:x-min 0 #:x-max x-max
                                #:y-min 0 #:y-max (add1 y-max)
                                #:z-min z-min #:z-max (* 1.5 z-max)
                                #:alpha 1  #:style 'solid
                                #:color 4 #:line-color "black" #:line-width 0.5
;#:line-style '
                                #:label "fitness")
                    (rectangles3d (append
                                   (for/list ([x-ivl (in-list x-ivls)]
                                              [x (in-list x-mids)])
                                     (vector x-ivl (ivl 0 1) (ivl 0 (average-fitness
x)))))
                                #:x-min 0 #:x-max x-max
                                #:y-min 0 #:y-max (add1 y-max)
                                #:z-min z-min #:z-max (* 1.5 z-max)
                                #:alpha 2/4  #:style 'solid
                                #:color 2 #:line-color "black" #:line-width 0.5
;#:line-style '
                                #:label "average fitness")) drawing-context 50
1000 width height))

(define (solution-plot drawing-context frame filepath clique graph graph_width
graph_height)

  (define (force-directed width height graph)

    (define (integer-count chromosome)
      (count (lambda(x) (bitwise-bit-set? chromosome x)) (range (integer-length
chromosome))))

    (define (build-graph-matrix graph-set graph-order)

      (define (make-line i)

        (foldl bitwise-ior
               0
               (filter-map (lambda(j) (if (or (equal? i j) (set-member? graph-set
(list i j)) (set-member? graph-set (list j i)))
                                          (arithmetic-shift 1 j)
                                          #f))
                           (range graph-order))))

      (build-vector graph-order make-line))

    (define (make-random-points points graph-order)

      (if (< graph-order 0)
          points
          (make-random-points (dict-set points graph-order (list (cons (random
width) (random height)) '(0 . 0))) (sub1 graph-order))))

    (define (force-directed-drawing graph-matrix points graph-order)
```

```
      (define (vec-mod v) (sqrt (+ (sqr (car v)) (sqr (cdr v)))))
      (define (vec-norm v) (vec-div v (vec-mod v)))
      (define (vec-sub v1 v2) (cons (- (car v1) (car v2)) (- (cdr v1) (cdr v2))))
      (define (vec-add v1 v2) (cons (+ (car v1) (car v2)) (+ (cdr v1) (cdr v2))))
      (define (vec-div v x) (cons (/ (car v) (+ x 0.0001)) (/ (cdr v) (+ x
0.0001))))
      (define (vec-mul v x) (cons (* (car v) x) (* (cdr v) x)))
      (define (vec-min v x) (if (< (vec-mod v) x) (vec-mod v) x))

      (define t 20.0)
      (define cooling 0.3)
      (define Ck 1.0)
      (define k (* Ck (sqrt (/ (* width height 1.0) graph-order))))

      (define CR 0.1)
      (define (f-r x) (/ (* CR k k) x))
      (define CA 1.0)
      (define (f-a x) (/ (* CA x x) k))

      (define (repulsive-forces points)

        (define (zero-disp points vertices)
          (if (null? vertices)
              points
              (zero-disp (dict-update points (car vertices) (lambda(value) (list
(car value) '(0 . 0)))) (cdr vertices))))

        (define (check-forces points v vertices)

          (cond ((null? vertices) points)
                (equal? v (car vertices) (check-forces points v (cdr vertices)))
                (else
                 (let* ((v-pos (car (dict-ref points v)))
                        (v-disp (cdr (dict-ref points v)))
                        (u-pos (car (dict-ref points (car vertices))))
                        (delta (vec-sub v-pos u-pos)) ;!!!!
                        (v-new-disp (vec-add v-disp (vec-mul (vec-norm delta) (f-r
(vec-mod delta))))))

                   (check-forces (dict-set points v (list v-pos v-new-disp) v (cdr
vertices)))))))

        (define (loop points vertices)

          (if (null? vertices)
              points
              (loop (check-forces points (car vertices) (range graph-order)) (cdr
vertices))))

        (loop (zero-disp points (range graph-order)) (range graph-order)))

      (define (attractive-forces points)

        (define (check-forces points graph)
          (if (null? graph)
              points
```

```
              (let* ((v (first (first graph)))
                     (u (second (first graph)))
                     (v-pos (car (dict-ref points v)))
                     (u-pos (car (dict-ref points u)))
                     (delta (vec-sub v-pos u-pos))
                     (disp-vec (vec-mul (vec-norm delta) (f-a (vec-mod delta))))
                     (v-disp (vec-sub (second (dict-ref points v)) disp-vec))
                     (u-disp (vec-add (second (dict-ref points u)) disp-vec)))

                ;(if (not (bitwise-bit-set? (vector-ref graph-matrix v) u)) (check-
forces points (cdr graph))
                    (check-forces (dict-set (dict-set points u (list u-pos u-disp))
v (list v-pos v-disp)) (cdr graph)))))
;)

        (check-forces points graph))

      (define (limit-displacement graph-order points t)

        (define (limit-proc points vertices)

          (if (null? vertices)
              points
              (let* ((v-pos (first (dict-ref points (car vertices))))
                     (v-disp (second (dict-ref points (car vertices))))
                     (v-new-pos (vec-add v-pos (vec-mul (vec-norm v-disp) (vec-min
v-disp t)))))

                (limit-proc (dict-set points (car vertices) (list (cons (max 0 (min
width (car v-new-pos))) (max 0 (min height (cdr v-new-pos)))) '(0 . 0)))
                            (cdr vertices)))))

        (limit-proc points (range graph-order)))

      (define (fd-loop step t points)

        ;(printf "~a\n" step)
        (send frame set-status-text
              (string-append "Loading data from " (path->string filepath) ":
force-directed placement (" (number->string step) " steps left)"))

        (if (<= step 0) points
            (fd-loop (sub1 step) (* cooling t) (limit-displacement graph-order
(attractive-forces (repulsive-forces points)) t))))

      (fd-loop 50 t points))

    (let* ((graph-order (length (remove-duplicates (flatten graph))))
           (graph-matrix (build-graph-matrix (list->set graph) graph-order)))

      (force-directed-drawing graph-matrix (make-random-points #hash() (sub1 graph-
order)) graph-order)))


  (define (sol-plot points x-offset y-offset)
```

```
    (send frame set-status-text (string-append "Loading data from " (path->string
filepath) ":   solution plot"))
    (send drawing-context set-font (make-font #:size 20 #:family 'system #:weight
'normal #:underlined? #t #:smoothing 'partly-smoothed))
    (send drawing-context draw-text "Solution plot" 20 1670)

    (for-each (lambda(edge)

                (if (and (set-member? clique (first edge)) (set-member? clique
(second edge)))
                    (send drawing-context set-pen "red" 0.2 'solid)
                    (send drawing-context set-pen (make-color 105 105 105 0.9) 0.06
'solid))
                (send drawing-context draw-line
                      (+ x-offset (car (first (dict-ref points (first edge)))))
                      (+ y-offset (cdr (first (dict-ref points (first edge)))))
                      (+ x-offset (car (first (dict-ref points (second edge)))))
                      (+ y-offset (cdr (first (dict-ref points (second edge))))))))
              graph)

    (dict-for-each points (lambda(key value)
                            (if (set-member? clique key)
                                (send drawing-context set-pen "red" 9 'solid)
                                (send drawing-context set-pen "blue" 7 'solid))
                            (send drawing-context draw-point (+ x-offset (car
(first value))) (+ y-offset (cdr (first value)))))))

  (sol-plot (force-directed graph_width graph_height graph) 5 1700))

(main-window)
```