

Реализация методов оптимизации в Matlab

1 Метод имитации отжига

1.1 Задача размещения ферзей

Описание задачи: Необходимо расставить n ферзей на доске размера $n \times n$ так, чтобы они не били друг друга.

Реализация функции потерь: Для нашей задачи размещения ферзей в качестве функции потерь является число ферзей, которые бьют друг друга. Так как два ферзя не могут находиться в одной строке или столбце, можем сказать, что изначально рассматриваем только комбинации вида $q = (i_1, \dots, i_n)$, $i_j \neq i_r \forall j \neq r$, $i_k \in \{1, 2, \dots, n\}$, где n - размер доски/число ферзей.

Из вышесказанного следует, что ферзи могут бить друг друга только по диагонали, поэтому нам нужно пройти циклом по всем ферзям и посчитать число пересечений. Два ферзя стоят на одной диагонали, если модуль разности номеров строк равен модулю разности номеров столбцов. Реализуем эту функцию: на вход принимает два параметра q - расстановка, n - длина последовательности. Не реализована внутри функции, так как при реализации алгоритма мы можем сразу посчитать и не пересчитывать каждый раз при подсчете функции потерь.

```
1 function [h] = calculate(q, n)
2 h = 0;
3 for i = 1:n
4     for j = i+1:n
5         if abs(q(i) - q(j)) == abs(i-j)
6             h = h + 2;
7         end
8     end
9 end
```

По реализации: при каждом совпадении добавляем 2, так как каждый из двух бьет другого. Второй цикл начинается с $i+1$, так как до этого элементы уже будут учтены.

Реализация алгоритма: Наш алгоритм будет принимать на вход четыре параметра: q_0 - начальная инициализация (расстановка), n_{iter} - число итераций (можно брать побольше, так как остановится по достижению оптимума), T_0 - начальная температура (100 по умолчанию), α - коэффициент, который отвечает за скорость убывания температуры (решил взять 0.9 по умолчанию). Тогда код алгоритма выглядит следующим образом:

```
1 function [q] = heatalg(q_0, n_iter, T_0, alpha)
2 q_old = q_0;
3 n = length(q_old);
4 T = T_0;
5 for i = 1:n_iter
6     loss_old = calculate(q_old, n);
7     if loss_old == 0
8         q = q_old;
9         break
10    end
11    q_new = q_old;
12    ind = randi(numel(q_new),1,2);
13    q_new(ind) = q_new(ind([2,1]));
14    T = T * alpha;
15    loss_new = calculate(q_new, n);
16    delta_loss = loss_new - loss_old;
17    if delta_loss < 0
18        q_old = q_new;
19    end
20    if delta_loss >= 0
21        prob = exp(-delta_loss/T);
```

```
22         if probab > unifrnd(0,1)
23             q_old = q_new;
24         end
25     end
26 end
```

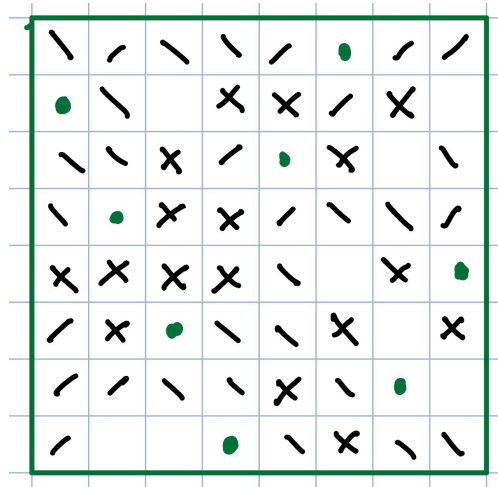
Комментарии к коду:

- Строки 1-4 отвечают за инициализацию
- Строки 5-10 задают цикл по числу итераций, считают лосс и проверяют, достигли ли мы оптимума, если да, то выходим из цикла
- Строки 11 - 16 пересчитывают вектор решений, на одной итерации алгоритма меняем местами два случайных элемента и уменьшаем температуру
- Строки 17 - 26 отвечают за то, что если значение функции стало меньше, то применяем новый вектор решений, иначе задаем вероятность того, что мы примем новый вектор, и потом сравниваем вероятность со случайной точкой на отрезке $[0, 1]$

Примеры реализации:

```
1 >> q_0 = [1 2 3 4 5 6 7 8]
2 >> heatalg(q_0, 1000, 100, 0.9)
3 ans =
4      2      4      6      8      3      1      7      5
5
6 >> q_0 = [1 2 3 4 5 6 7 8 9 10 11 12 13]
7 >> heatalg(q_0, 1000, 100, 0.9)
8 ans =
9      7     10     12      3      6     13     11      8      1      4
      2      5      9
```

Изобразим картинку для первого примера, чтобы убедиться, что все верно:



1.2 Минимизация недифференцируемой функции

Описание задачи: В данном упражнении перед нами стоит задача найти минимум следующей функции:

$$f(x) = x^2 \cdot (2 + |\sin(a \cdot x)|), \quad a > 0 \quad f(x) \xrightarrow{x} \min$$

Несложно увидеть, что минимум функции достигается в нуле, однако градиентные методы тут бесполезны, так как функция является недифференцируемой. Поэтому мы воспользуемся методом отжига.

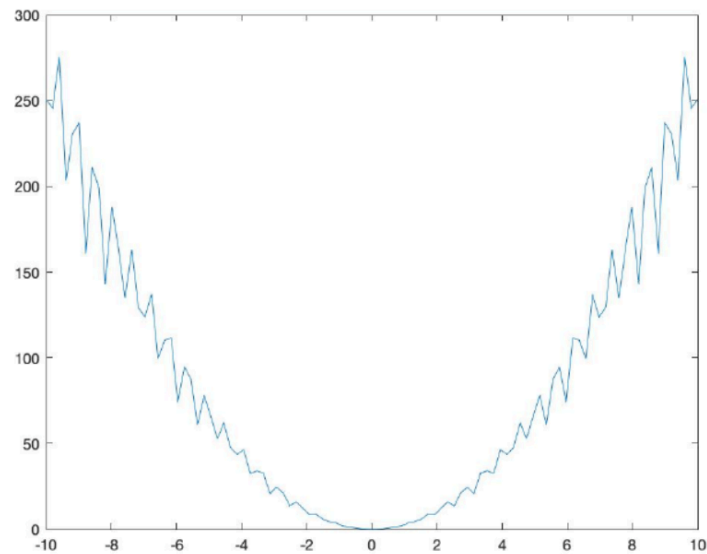
Реализация функции потерь: Сначала напомним простой код, который будет вычислять значение функции в заданной точке при фиксированном a - параметр, который мы задаем сами (по умолчанию зададим $a = 10$).

```
1 function [f] = loss(x, a)
2 f = x^2 * (2 + abs(sin(a*x)));
3 end
```

Также визуализируем данную функцию простой генерацией семплов из матлаба:

```
1 >> x = linspace(-10, 10)
```

```
2 >> y = arrayfun(@loss, x)
3 >> plot(x, y)
```



Реализация алгоритма: Наш алгоритм будет принимать на вход четыре параметра: a - параметр функции, n_{iter} - число итераций (1000 по умолчанию), T_0 - начальная температура (100 по умолчанию), α - коэффициент, который отвечает за скорость убывания температуры (0.95 по умолчанию). Тогда код алгоритма выглядит следующим образом:

```
1 function [x] = new_heatalg(a, n_iter, T_0, alpha)
2 x_old = unifrnd(-10, 10);
3 T = T_0;
4 for i = 1:n_iter
5     loss_old = loss(x_old, a);
6     x_new = x_old + unifrnd(-1, 1);
7     T = T * alpha;
8     loss_new = loss(x_new, a);
9     delta_loss = loss_new - loss_old;
10    if delta_loss < 0
```

```
11         x_old = x_new;
12     end
13     if delta_loss >= 0
14         prob = exp(-delta_loss/T);
15         if prob > unifrnd(0,1)
16             x_old = x_new;
17         end
18     end
19 x = x_old;
20 end
```

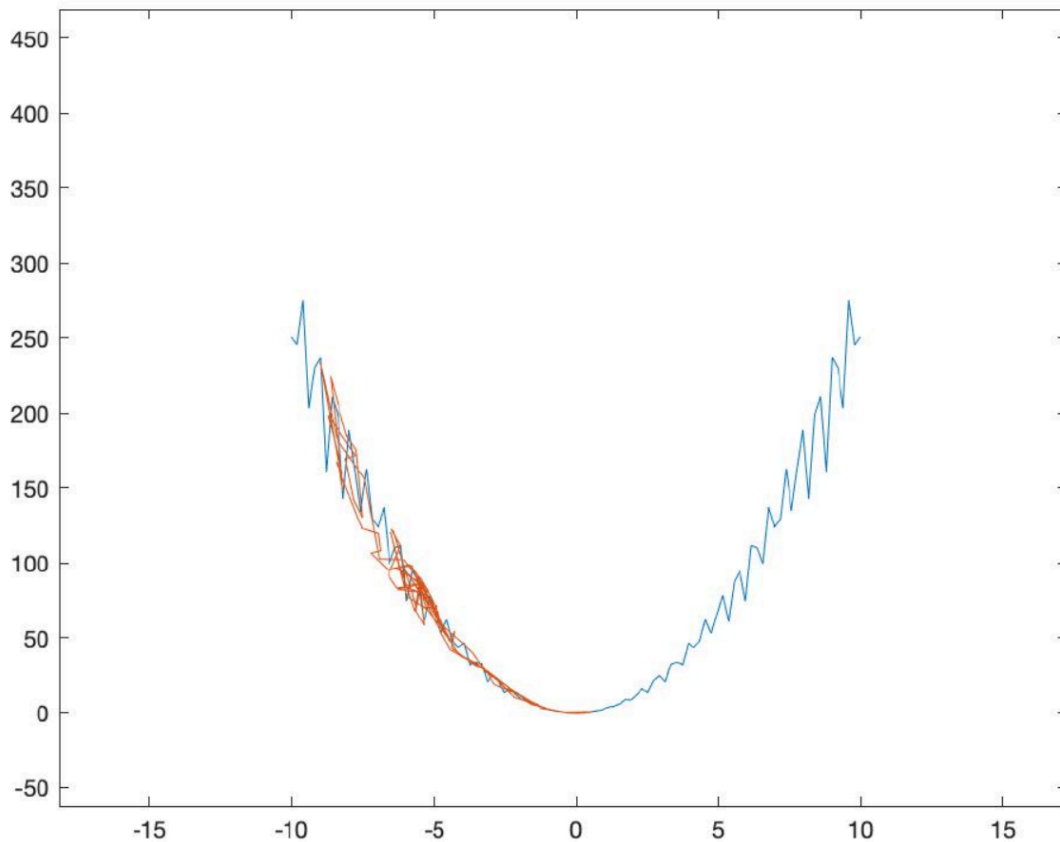
Комментарии к коду:

- Строки 1-3 отвечают за инициализацию
- Строки 4-9 отвечают за подсчет лосса, создание новой точки и уменьшение температуры (в цикле). Новая точка получается как $x_k = x_{k-1} + \xi_k$, $\xi_k \sim \text{Unif}([-1, 1])$
- Строки 10 - 20 Сравниваем значение лосса со старым и аналогично задаче 1.1 считаем вероятность принять новую точку, если значение функции выросло

Пример реализации:

```
1 >> new_heatalg(10, 1000, 100, 0.95)
2 ans = 1.0587e-04
3 >> loss(ans, 10)
4 ans = 2.2427e-08
```

Как мы видим, алгоритм работает успешно и при 1000 итераций выдает значение очень близкое к $x_{min} = 0$, при этом значение функции также около нуля, что является минимумом данной функции. График иллюстрирует оранжевым цветом путь алгоритма по шагам.



2 Алгоритм роя частиц

2.1 Две функции

Постановка задачи и реализация loss функций: В данном упражнении перед нами стоит задача найти минимум функции Розенброка:

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \rightarrow \min_{x_1, x_2}$$

Несложно увидеть, что минимум функции достигается в точке $(1, 1)$, однако градиентные методы очень тяжело справляются с задачей минимизации из-за своеобразных линий уровня функции - овражная функция. Реализуем подсчет данной функции в matlab в векторном виде: $x = (x_1, x_2)$.

```
1 function [func] = loss(x, type)
2 if type == 1
```

```

3     func = (1 - x(:, 1)).^2 + 100 * (x(:, 2) - x(:, 1).^2).^2;
4 end
5 if type == 2
6     func = 0.01 .* (x(:, 1) - 0.5) .^ 2 + abs(x(:, 1).^2 - x(:,
        2)) + abs(x(:, 1).^2 - x(:, 3));
7 end

```

Функция Розенброка занумерована первым типом, параллельно будем писать код для второй функции, которая сложнее, чем первая:

$$f(x_1, x_2, x_3) = 0.01(x_1 - 0.5)^2 + |x_1^2 - x_2| + |x_1^2 - x_3| \rightarrow \min_{x_1, x_2, x_3}$$

Минимум этой функции находится в точке $x = (0.5, 0.25, 0.25)$

Реализация алгоритма Алгоритм будет принимать на вход шесть параметров: α, β, γ - параметры функции, отвечающие за веса при пересчете вектора скорости, n_{iter} - число итераций (1000 по умолчанию), m - число частиц (агентов), $type$ - тип функции - принимает значения 0 или 1. Тогда код алгоритма выглядит следующим образом:

```

1 function [answer] = particle_swarm(alpha, beta, gamma, n_iter,
    m, type)
2 if type == 1
3     n_args = 2;
4     x = unifrnd(-10, 10, m, n_args);
5     v = unifrnd(-3, 3, m, n_args);
6 end
7 if type == 2
8     n_args = 3;
9     x = zeros(m, n_args);
10    x(:, 1) = unifrnd(-0.2, 1, m, 1);
11    x(:, 2) = unifrnd(-0.3, 1, m, 1);
12    x(:, 3) = unifrnd(-0.5, 1, m, 1);
13    v = unifrnd(-0.05, 0.05, m, n_args);

```



```
14 end
15 p = x;
16 [~, ind] = min(loss(p, type));
17 J = x(ind, :);
18 for i = 1:n_iter
19     xi_1 = unifrnd(0, 1, m, n_args);
20     xi_2 = unifrnd(0, 1, m, n_args);
21     v = alpha .* v + beta .* (xi_1 .* (p - x)) + gamma .* xi_2
        .* (J - x);
22     x = x + v;
23     for j = 1:m
24         if loss(x(j, :), type) < loss(p(j, :), type)
25             p(j, :) = x(j, :);
26         end
27     end
28     [~, ind] = min(loss(p, type));
29     J = x(ind, :);
30 answer = J;
31 end
```

Комментарии к коду: так как число параметров и инициализируемые значения у двух функций будут отличаться, поэтому в начале приходится расписывать нулевой шаг алгоритма отдельно в зависимости от значения переменной `type`.

- Строки 1 - 17 инициализируют параметры алгоритма в зависимости от типа функции и определяем изначальный минимум функции
- Строки 18 - 22 вычисляют новые точки путем изменения вектора скорости
- Строки 23 - 27 заменяют историю лучшего положения точки, если новое положение точки оказалось лучше, чем все старые

- Строки 28-29 ищут лучшую точку среди истории и записывают в ответ по этому индексу текущее значение точки
- Выходя из цикла записываем ответ

Выводы: Пример реализации для первой функции:

```
1 >> ans_first = particle_swarm(0.95, 0.2, 0.2, 10000, 100, 1)
2 ans_first =
3      1      1
```

Как мы видим, для первой функции все работает отлично и при малых значениях числа итераций, однако вторая функция гораздо сложнее, поэтому для успешного применения и стабильного достижения оптимума напишем функцию `multistart`, которая делает несколько запусков и выбирает наилучший из них.

```
1 function [answer] = multistart(n_times)
2 loss_best = 1000;
3 x = zeros(n_times, 3);
4 for i = 1:n_times
5     x = particle_swarm(0.6, 0.3, 0.3, 10000, 100, 2);
6     if loss(x, 2) < loss_best
7         loss_best = loss(x, 2);
8         answer = x;
9     end
10 end
```

Теперь протестируем работу функции `multistart` (параметры α, β, γ) подобраны путем перебора нескольких вариантов, однако благодаря тому, что мы делаем много запусков и берем лучший результат, их выбор не имеет большого значения.

```
1 >> ans_second = multistart(100)
2 ans_second =
3      0.5002      0.2502      0.2502
```

Как мы видим, путем создания дополнительной функции, удастся достичь очень стабильных и точных результатов.

3 Генетический алгоритм

3.1 Непрерывная функция

Постановка задачи: Необходимо найти минимум следующей функции:

$$F(x) = \sum_{k=1}^n x_k^2 \cdot (1 + |\sin(100x_k)|)$$

Функция задана на \mathbb{R}_n , по умолчанию будем рассматривать $n = 5$. Особенность функции в том, что она имеет множество локальных минимум и один глобальный в $x = 0$, в который мы и хотим попасть. Реализация кода функции:

```

1 function [y] = func1(x)
2     y = (x.^2) .* (1 + abs(sin(100 .* x)));
3     y = sum(y, 2);
4 end

```

Создание функции приспособленности: Для всех задач оптимизации при помощи генетического алгоритма нам понадобится функция приспособленности, с помощью которой мы будем ранжировать пул особей:

$$Fit(x) = \frac{1}{1 + F(x)}$$

где $F(x)$ - минимизируемая функция. В случае неотрицательной функции получаем удобную величину для ранжирования, так как принимает значения от 0 до 1, кроме того, функция важна для скрещивания и получения новых особей, с помощью нее вычисляем вероятности:

$$p = \frac{Fit(x)}{Fit(x) + Fit(y)}, \quad q = \frac{Fit(y)}{Fit(x) + Fit(y)}$$

В результате, имея двух особей x и y можем получить новую особь z путем сравнения p с ξ_i - равномерно распределенной с. в. Если $\xi_i \leq p$, то i -я компонента новой особи равна

x_i , иначе y_i . Таким образом, чем более приспособлена особь, тем больше генов войдут в ее потомка. Реализуем функцию в коде:

```
1 function [y] = fit_func(x)
2     y = 1 ./ (1 + x);
3 end
```

Реализация алгоритма: Возьмем число особей $M = 1000$, число худших убиваемых особей $M_c = 200$, количество итераций алгоритма $L = 1000$

```
1 function [x] = genetic_algorithm(m, m_c, l, n)
2     x = rand(m, n) .* 20 - 10;
3     for iter_ = 1:l
4         [~, indexes] = sort(fit_func(func1(x)), 'descend');
5         x = x(indexes, :);
6         ind = randi([1, m], [m-m_c-1, 1]);
7         xi = rand(m-m_c-1, n);
8         fit = fit_func(func1(x));
9         probas = fit(2:m-m_c) ./ (fit(2:m-m_c) + fit(ind));
10        mask = 1 * (probas >= xi);
11        x(2:m-m_c, :) = mask .* x(2:m-m_c, :) + (1 - mask) .* x
            (ind, :);
12        x(m-m_c+1:m, :) = rand(m_c, n) .* 20 - 10;
13    end
14    x = x(1, :);
15 end
```

Комментарии к коду:

- Строки 1-2 инициализируем начальных особей из равномерного распределения от -10 до 10
- Строки 3-5 сортируют точки по значению функции (первая точка - наилучшая)

особь)

- Строка 6 выбирает индексы случайных особей для скрещивания
- Строки 7-9 рассчитывают значение функции приспособленности и вероятность того, что мы возьмем i -ю компоненту из новой особи
- Строки 10-11 осуществляют скрещивание особей по описанной ранее схеме - применяем ко всем особям кроме наилучшей и 200 наихудших
- Строка 12 заменяет худшие особи на случайные
- Строки 13-15 берут лучшую особь в качестве ответа и возвращают ответ

Пример работы алгоритма:

```

1 >> genetic_algorithm(1000, 200, 1000, 5)
2 ans =
3      0.0266      0.0194     -0.0009      0.0008     -0.0088

```

Как мы видим, мы находимся в окрестности оптимального решения, если увеличить число итераций до 10000, то значений будут порядка $1e-4$, то есть очевидно, что алгоритм сходится к минимуму функции.

3.2 Задача разбиения множества

Постановка задачи: Пусть задано конечное множество из натуральных чисел A , необходимо разбить A на два подмножества K_1, K_2 такие, что

$$A = K_1 \cup K_2, K_1 \cap K_2 = \emptyset$$

таким образом, чтобы

$$H(K_1, K_2) = \left| \sum_{a_k \in K_1} a_k - \sum_{a_m \in K_2} a_m \right| \rightarrow \min$$

В целом, задача решается только полным перебором, однако с помощью генетического алгоритма можно решить значительно быстрее. Пусть $|A| = n$, в нашем случае возьмем

$n = 5000$, чтобы реализовать функции и работу алгоритма, представим решение $x = (x_1, \dots, x_n)$, в котором каждая компонента либо 0, либо 1, в зависимости от того, в какое подмножество попадает элемент. Кроме того, в примере множество $A = \{1, \dots, 5000\}$, поэтому сумму можно посчитать как сумму индексов элементов, которые относятся к конкретному подмножеству. Тогда реализация функции потерь выглядит следующим образом:

```
1 function [result] = func2(x)
2     loss = [];
3     for i = 1:size(x, 1)
4         a_1 = find(x(i,:) == 1);
5         a_0 = find(x(i,:) == 0);
6         loss(i) = abs(sum(a_1) - sum(a_0));
7     end
8     result = loss;
9 end
```

В данной функции мы проходимся циклом по всем особям и считаем сумму согласно тому, как мы ранее описали. Функция приспособленности не отличается от задачи 3. 1.

Реализация алгоритма: Для данной задачи возьмем $M = 1000, M_c = 200, L = 100, n = 5000$. Посмотрим, сойдется ли алгоритм за 100 итераций.

```
1 function [result] = genetic_algorithm_2(m, m_c, l, n)
2     x = binornd(1, 0.5, m, n);
3     for iter_ = 1:l
4         [~, indexes] = sort(fit_func(func2(x)), 'descend');
5         x = x(indexes, :);
6         ind = randi([1, m], [m-m_c-1, 1]);
7         xi = rand(m-m_c-1, n);
8         fit = fit_func(func2(x));
9         probas = (fit(2:m-m_c) ./ (fit(2:m-m_c) + fit(ind))).';
```

```

10     mask = 1 * (probas >= xi);
11     x(2:m-m_c, :) = mask .* x(2:m-m_c, :) + (1 - mask) .* x
        (ind, :);
12     x(m-m_c+1:m,:) = binornd(1, 0.5, m_c, n);
13 end
14 result = x(1,:);
15 end

```

Комментарии к коду: реализация алгоритма ничем не отличается от задачи 3. 1., кроме того, что теперь мы сэмплируем новые точки из биномиального распределения: выбор k успехов из n испытаний, где вероятность успеха равна 0.5. Таким образом, получаем векторы такого вида, который и ожидает увидеть на входе наша функция потерь.

Пример работы алгоритма:

```

1 >> x = genetic_algorithm_2(1000, 200, 300, 5000);
2 >> func2(x)
3 ans =
4     0

```

Бывает, что алгоритм скатывается в локальные минимумы, где значение функции потерь равно 2 или 4, однако это можно решить, несколько раз запустив алгоритм, либо же увеличив число особей / число итераций. В данном случае присутствует trade-off между стабильным достижением глобального минимума и временем работы алгоритма.

3.3 Задача линейного программирования

Постановка задачи: Задача производства n товаров: T_1, \dots, T_n , для товаров требуется m ресурсов. $A = \{(a_{ij})\}$ - кол-во i -го ресурса, необходимое для производства j -го товара, $a_{ij} \geq 0$. Вектор $b = (b_1, \dots, b_m)^T$ - ограничение ресурсов. Необходимо определить план производства товаров $x = (x_1, \dots, x_n)^T$, $x_j \geq 0$, при этом должны учесть, что некоторые товары могут быть только в целочисленном количестве. Также для каждого товара заданы цены $c = (c_1, \dots, c_n)$, цены положительные. Необходимо максимизиро-

вать следующую функцию:

$$F(x) = c_1x_1 + \dots + c_nx_n \rightarrow \max_x$$

При условиях:

$$a_{11}x_1 + \dots + a_{1n}x_n \leq b_1$$

...

$$a_{m1}x_1 + \dots + a_{mn}x_n \leq b_m$$

Реализация условий задачи: Сначала создадим функцию, которая будет определять, выполняются ли для товаров ресурсные ограничения:

```
1 function result = constraint(x, A, b)
2     result = all((x * A.' <= b), 2);
3 end
```

Данная функция просто проверяет, что все m условий выполняются.

Будем учитывать ограничения в функции потерь следующим образом: так как кол-во всех товаров неотрицательно и цены положительные, установим значение $F = 0$, если ресурсные ограничения для набора товаров не выполняются.

```
1 function vals = func_3(x, c, A, b)
2     vals = x * c.';
3     vals(constraint(x, A, b) == 0) = 0;
4 end
```

Кроме того, чтобы мы могли генерировать и натуральные, и просто неотрицательные числа, будем по особому генерировать наборы товаров - зададим `int_idx -> list`, который хранит все индексы товаров, количество которых может принимать только натуральные значения.

```
1 function x = create_random_x(int_idx, m, n)
2     x = rand(m, n) * 100;
3     x(:, int_idx) = randi([0, 100], m, length(int_idx));
```



```
4 end
```

Сначала заполняем все равномерным распределением от 0 до 100, затем заменяем количества у целочисленных товаров на случайные целые числа от 0 до 100.

Реализация алгоритма: не отличается от предыдущих задач, единственное отличие в том, что так как у нас теперь задача максимизации, вместо fit функции будем использовать исходную функцию и в формулу для подсчета вероятностей добавим $\text{eps} = 1e-5$, так как в случае, если для обоих особей не выполнены ресурсные ограничения возникает в знаменателе $p = 0/0$, так как значения функции равны нулю.

```
1 function x = genetic_algorithm_3(m, m_c, l, n, c, A, b, int_idx
  )
2     x = create_random_x(int_idx, m, n);
3     for iteration = 1:l
4         [~, indexes] = sort(func_3(x, c, A, b), 'descend');
5         x = x(indexes, :);
6         ind = randi([1, m], [m-m_c-1, 1]);
7         xi = rand(m - m_c - 1, n);
8         fit = func_3(x, c, A, b);
9         probas = fit(2:m - m_c) ./ ((fit(2:m - m_c) + fit(ind))
              + 1e-5);
10        mask = probas >= xi;
11        x(2:m - m_c, :) = mask .* x(2:m - m_c, :) + (1 - mask)
              .* x(ind, :);
12        x(m - m_c + 1:m, :) = create_random_x(int_idx, m_c, n);
13    end
14    x = x(1, :);
15 end
```

Пример работы алгоритма - зададим следующий пример:

$$A = \begin{pmatrix} 40 & 30 \\ 20 & 30 \end{pmatrix}, \quad b = (4000, 3000)^T, \quad c = (30, 40)^T, \quad \text{int_idx} = [1]$$

Известно что для таких параметров истинный оптимум $x = (50, 200/3)$. Проверим наш алгоритм:

```
1 >> A = [[40, 30]; [20, 30]];
2 >> b = [4000, 3000];
3 >> c = [30, 40];
4 >> int_idx = [1];
5 >> genetic_algorithm_3(1000, 200, 500, 2, c, A, b, int_idx)
6 ans =
7      50.0000      66.6619
```

Действительно, получаем значения практически неотличимые от истинного оптимума.