Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 12

по курсу «Методы оптимизации»

«Реализация генетического алгоритма для восстановления картинки»

Студент группы ИУ9-81Б Окутин Д.А.

Преподаватель Посевин Д. П.

Москва 2025

1 Задание

Реализовать генетический алгоритм для восстановления рисунка на закрашенной области, визуализировать процесс поиска на графиках.

2 Реализация

Исходный код программмы представлен в листинге 1.

Листинг 1: code

```
using Random
using Plots
using Colors

function random_gene()
    numbers = [5, 15, 31]  #
    return rand(numbers)
end

function generate_p_matrix(rows, cols)
    #
    matrix = fill(31, (rows, cols))

    #                                                    "   "
    for i in 1:rows
        matrix[i, 1] = 5  #

        matrix[i, cols] = 5  #

        if i == rows  #

            for j in 1:cols
                matrix[i, j] = 5
            end
        end
    end

    matrix[rows,1]=15
    matrix[rows,rows]=15

    return matrix
end

function generate_matrix(rows, cols)
```

```
32      #
33      matrix = fill(31, (rows, cols))
34
35      #                                                       "   "
36      for i in 1:rows
37          for j in 1:cols
38              matrix[i, j] = random_gene()
39          end
40      end
41
42      return matrix
43  end
44
45  function init_population(pop_size, n)
46      [generate_matrix(n,n) for _ in 1:pop_size]
47  end
48
49  function fitness_boolean(ind, target)
50      diff = 0
51      for i in eachindex(ind)
52          diff += count_ones(ind[i]      target[i])
53      end
54      diff
55  end
56
57  function fitness_euclidean(ind, target)
58      sqrt(sum((ind .- target).^2))
59  end
60
61  function tournament_selection(pop, fitnesses, tournament_size=2)
62      indices = rand(1:length(pop), tournament_size)
63      best = indices[1]
64      for idx in indices
65          if fitnesses[idx] < fitnesses[best]
66              best = idx
67          end
68      end
69      pop[best]
70  end
71
72  function crossover_gene_level(parent1, parent2)
73      n = size(parent1, 1)
74      child = similar(parent1)
75      for i in 1:n, j in 1:n
76          child[i, j] = rand() < 0.5 ? parent1[i, j] : parent2[i, j]
77      end
```

```
78      child
79  end
80
81  function crossover_vector_level(parent1, parent2)
82      n = size(parent1, 1)
83      child = similar(parent1)
84      for i in 1:n
85          child[i, :] = rand() < 0.5 ? parent1[i, :] : parent2[i, :]
86      end
87      child
88  end
89
90  function mutate!(ind, mutation_rate)
91      n = size(ind, 1)
92      for i in 1:n, j in 1:n
93          if rand() < mutation_rate
94              ind[i, j] = random_gene()
95          end
96      end
97  end
98
99  function genetic_algorithm(target; pop_size=50, max_generations=500,
        mutation_rate=0.01, epsilon=0.0,
100                            fitness_func=:boolean, crossover_method=:gene
        )
101     n = size(target, 1)
102     custom_gradient = cgrad([:blue, :green, :red])
103     population = init_population(pop_size, n)
104     best_fitness_history = Float64[]
105     generation = 0
106     best_fitness = Inf
107     current_best = 0
108     anim = Animation()
109     while generation < max_generations && best_fitness > epsilon
110         fitnesses = [(fitness_func == :boolean ? fitness_boolean(ind,
        target) : fitness_euclidean(ind, target)) for ind in population]
111         current_best_idx = argmin(fitnesses)
112         current_best = population[current_best_idx]
113         best_fitness = fitnesses[current_best_idx]
114         push!(best_fitness_history, best_fitness)
115         p1 = heatmap(current_best, c=custom_gradient, title="Generation:
         $generation,                    : $best_fitness", colorbar=false,
        xticks=false, yticks=false, aspect_ratio=1)
116         p2 = heatmap(target, c=custom_gradient, title="Target", colorbar
        =false, xticks=false, yticks=false, aspect_ratio=1)
117         p = plot(p1, p2, layout=(1,2), size=(1200,600))
```

```
118         frame(anim, p)
119         new_population = []
120         for i in 1:pop_size
121             parent1 = tournament_selection(population, fitnesses)
122             parent2 = tournament_selection(population, fitnesses)
123             child = crossover_method == :gene ? crossover_gene_level(
    parent1, parent2) : crossover_vector_level(parent1, parent2)
124             mutate!(child, mutation_rate)
125             push!(new_population, child)
126         end
127         population = new_population
128         generation += 1
129     end
130     final_plot = plot(heatmap(current_best, c=custom_gradient, title="
    Final Generation: $generation,                    : $best_fitness",
     colorbar=false, xticks=false, yticks=false, aspect_ratio=1),
131                       heatmap(target, c=custom_gradient, title="Target",
     colorbar=false, xticks=false, yticks=false, aspect_ratio=1),
132                       layout=(1,2), size=(1200,600))
133     for i in 1:50
134         frame(anim, final_plot)
135     end
136     return best_fitness_history, anim
137 end
138
139 function run_experiment(n, fitness_func, crossover_method)
140     target = generate_p_matrix(n,n)
141     println("                                                $n $n,
                                : $(fitness_func),
                               : $(crossover_method)")
142     fitness_history, anim = genetic_algorithm(target, pop_size=100,
    max_generations=500, mutation_rate=0.05, epsilon=0.0, fitness_func=
    fitness_func, crossover_method=crossover_method)
143     println("      -                      ",length(fitness_history),"
          $n $n,                                 : $(fitness_func),
                                          : $(crossover_method)")
144     gif(anim, "GA_$(n)x$(n)_$(fitness_func)_$(crossover_method).gif",
    fps=15)
145     p_convergence = plot(fitness_history, xlabel="                    ",
    ylabel="                        ", title="                      : $n $n,
    $(fitness_func), $(crossover_method)")
146     savefig(p_convergence, "Convergence_$(n)x$(n)_$(fitness_func)_$(
    crossover_method).png")
147 end
148
149 sizes = [3, 5, 9, 21]
```

```
# sizes = [3]
fitness_types = [:boolean, :euclidean]
crossover_methods = [:gene,:vector]

for n in sizes
    for fit in fitness_types
        for method in crossover_methods
            run_experiment(n, fit, method)
        end
    end
end
```

3 Результаты

Результаты запуска представлены на рисунках 1.

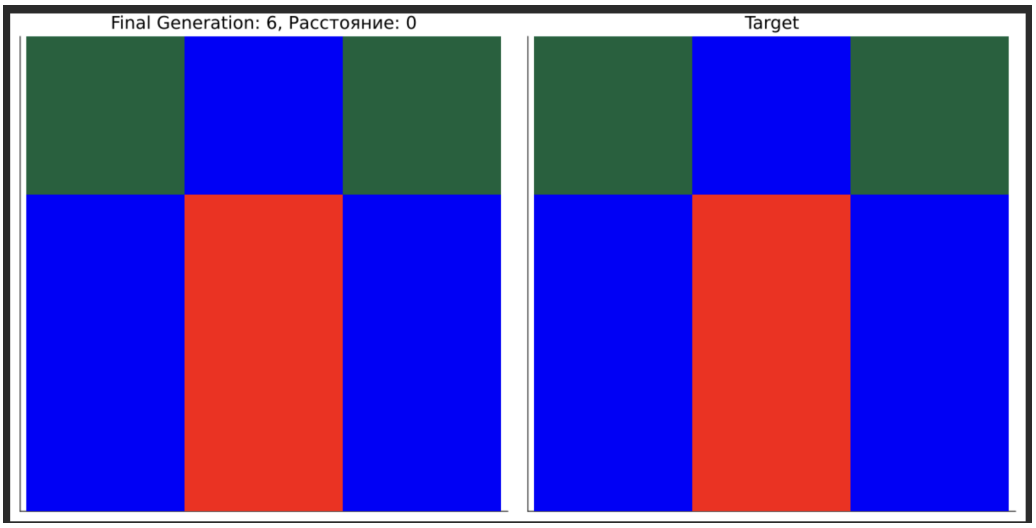

Рис. 1 — Визуализация 3x3

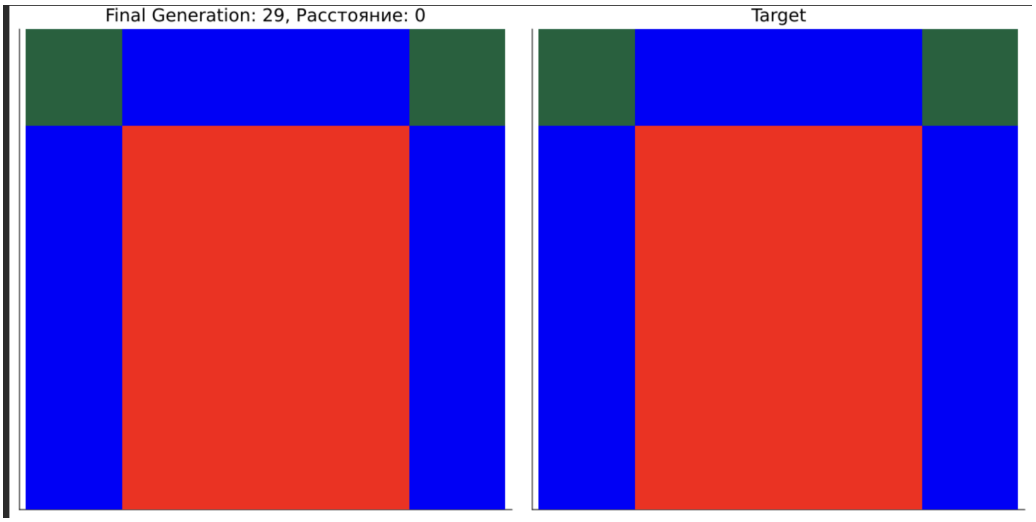

Рис. 2 — Визуализация 5x5

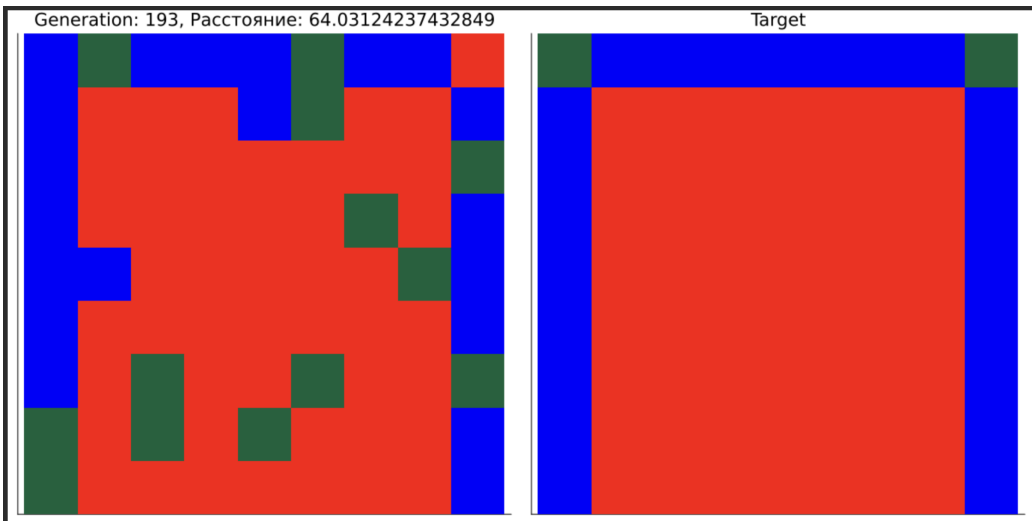

Рис. 3 — Визуализация 9x9

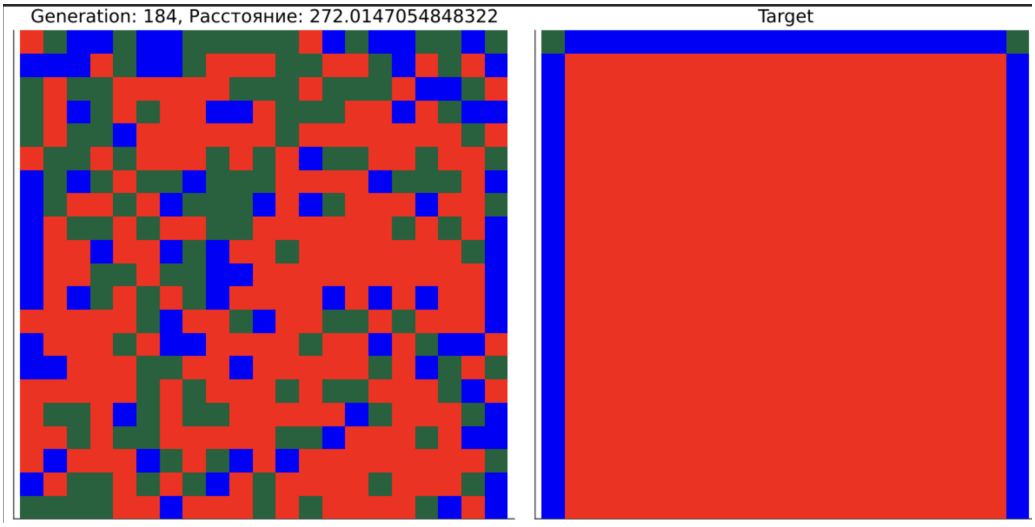

Рис. 4 — Визуализация 21x21

4 Выводы

В результате данной лабораторной работы был реализован генетический алгоритм для восстановления рисунка из случайной последовательности, на матрицах размерности 3x3 и 5x5 алгоритм показал хорошую сходимость, однако, на матрицах больше размера алгоритм не сходился. Возможно, из-за особенности рисунка, возможно из-за большого количества вариаций различных цветов.