MECS4510 Evolutionary Computation, Fall 2023

HW1: Traveling Salesman Problem

Name: Bole (James) Pan

UNI: bp2632

Instructor: Prof. Hod Lipson

Date Submitted: September 27, 2023

Grace Hours Used: 17 hours

Grace Hours Remaining: 79 hours

1. General

Summary Result Table

Table 1. Results Summary

File Name	Category	Evaluations	Length
cities.txt	The shortest path	76010	523.7312659268067
cities.txt	The longest path	92641	616.6014859186349

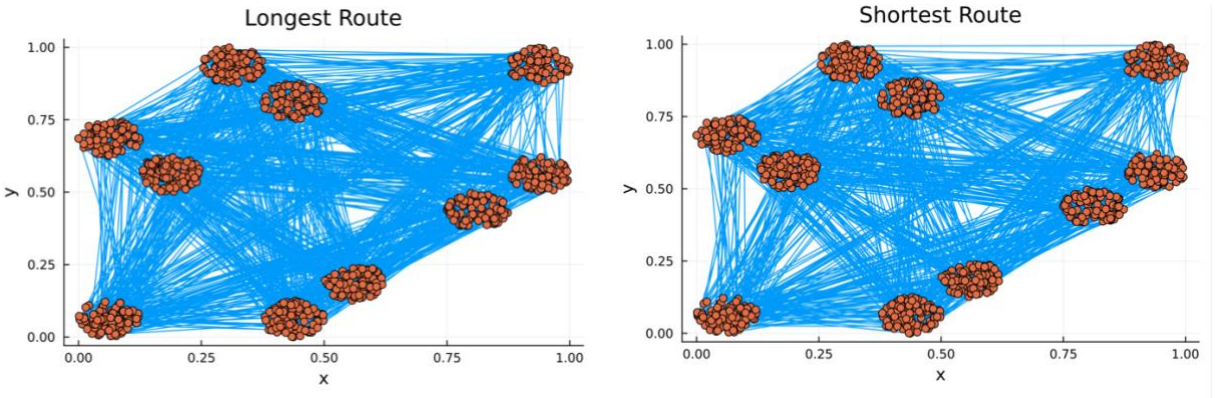

Figure 1. The shortest path (left) and the longest path (right) found

Theoretical Shortest Path: 12.775068874879208

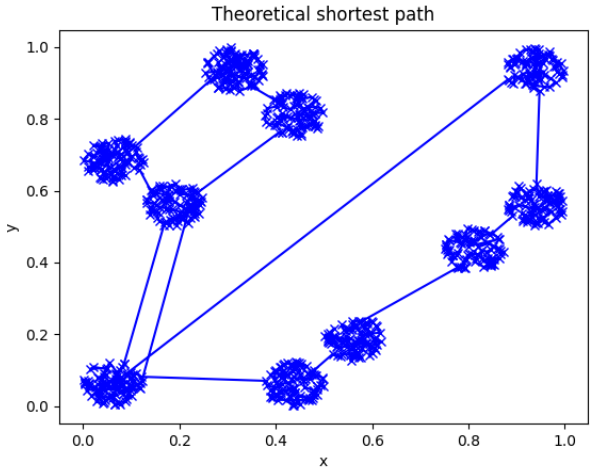

Figure 2. The theoretical shortest path found using Christofides' algorithm.

Movie of optimizing path of Random Search: one frame for every time path improves

https://drive.google.com/file/d/1T7Ytp0wu4ZlhIdmxli24-JA4Q-tCOssB/view?usp=sharing

2. Methods

- **Fitness:** we define fitness as the inverse of the total distance of a route. Therefore, the shorter the route, the higher the fitness.
- **Representation Used:** We represent each city as a tuple with two elements. A route, as called an individual, is an ordered array of all the cities. A population is an array of routes, or individuals. We are using direct representation, and more specifically, index representation.
    - **Mutation:** we use the swap mutation method. For an individual route, we loop through each city, and with a certain mutation rate we probabilistically swap this city with another random city.
    - **Crossover:** we use the order crossover method. For two parent routes, we randomly select a subset of cities from one parent, and make the child's corresponding segment the same as that of the parent, then fill in the rest of the cities in the order of the other parent.

- **Random Search:** Randomly generate a permutation of the cities and calculate the total distance. Repeat this process for a number of times and record the best result.

- **Random Mutation Hill Climbing:** Randomly generate a permutation of the cities and calculate the total distance. Then, for a number of times, apply mutation on the route and calculate the fitness of that route. If the new fitness is higher, keep the new route. Repeat this process for a number of times and record the best result.

- **EA variation and selection methods used:** We use the roulette selection method. From the initial population, we first select a small population of elites. The higher the fitness, the more likely an individual would be selected for this group. We put all our elites in the new population. Then, until the new population reaches the predetermined population size, we randomly select two individuals from the elites and apply crossover to produce a child, apply mutation on the child, and put it into the new population. This process, going from the initial population to the new population, is repeated for a number of times (called generations).

- **Analysis of Performance:**
    - In terms of finding the shortest path, in 100k generations, with mutation rate of 5%, selection rate of 30%, and population size of 100, we found that evolutionary algorithm performs the best. Random search follows with significantly lower performance, while random mutation hill climbing performing the worst. Hill climber also has the greatest error of the mean at each generation.
    - The best result found from genetic algorithm is very high compared to the theoretical shortest path. It’s suspected that all three algorithms have fallen into a local minimum, and was thus not able to make too much process after around 50k generations. This could potentially be remedied by adding diversity to the

populations. How to optimize genetic algorithm's performance will be the subject of later explorations.

- In terms of finding the longest path, with the same parameters, we still found that evolutionary algorithm performs the best. Random search follows, and then random mutation hill climbing. This agrees well with the results from the shortest path.

- **Methods compared:** we compare the performance of random search, random mutation hill climbing, and EA on the two tasks of finding the longest and the shortest paths.

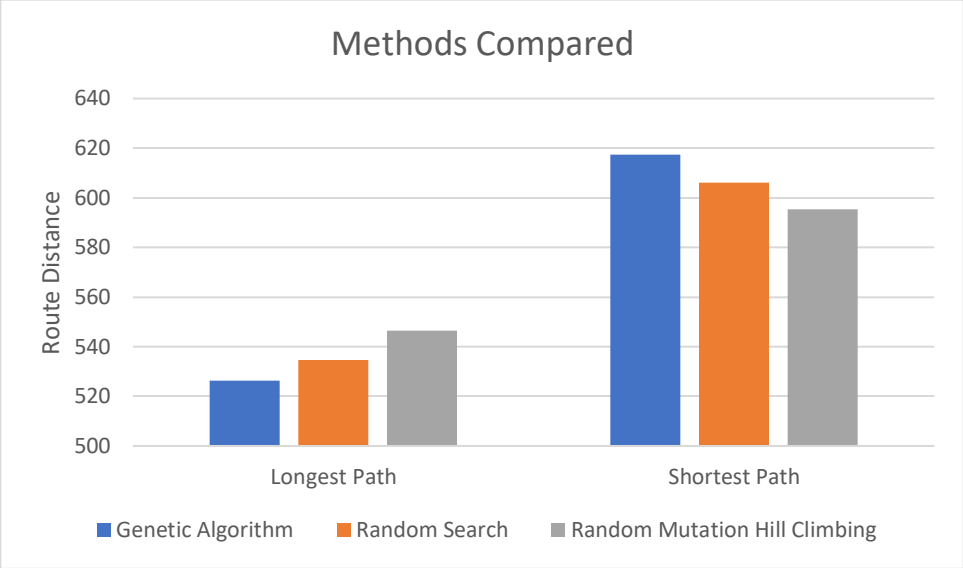

Figure 3. Comparison of methods. Genetic Algorithm displayed the best overall performance.

3. Performance Curves

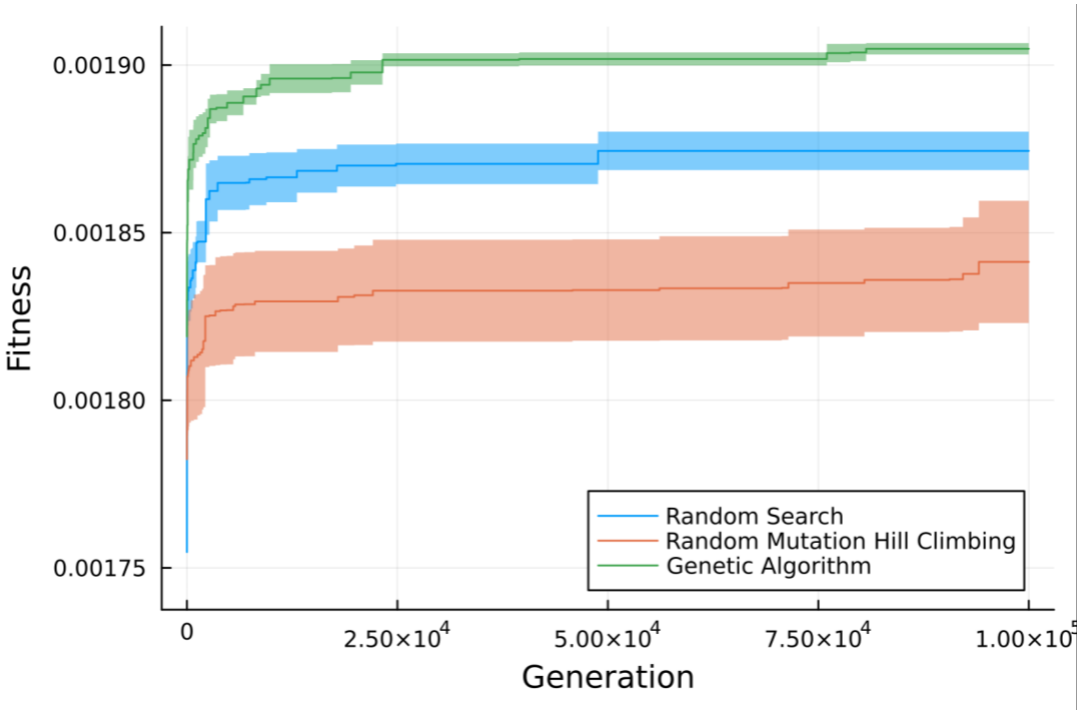

Figure 4. The shortest path learning curves of random search, hill climber, and EA (ribbons represent the error of the mean in each generation, averaged on four runs)

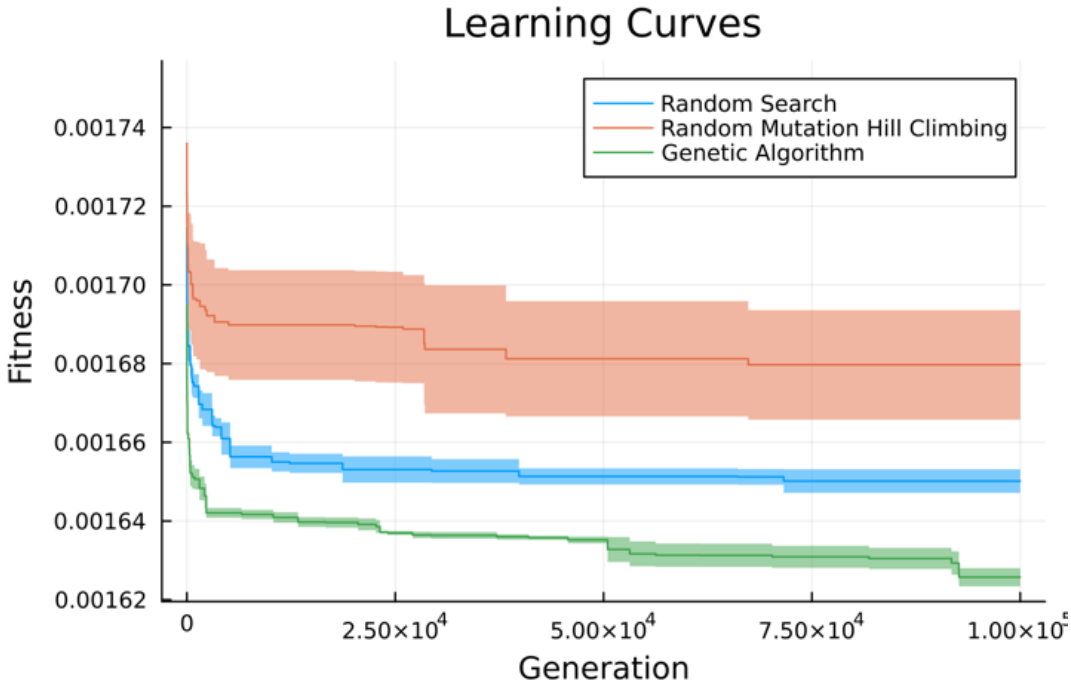

Figure 5. The longest path learning curves of random search, hill climber, and EA

Appendix: Code

- only included code for finding the shortest paths, which is only slightly different from the code for finding the longest paths

TSP_genetic_algorithm_methods.jl

```
using Random

function distance(city1, city2)
    return sqrt((city1[1] - city2[1])^2 + (city1[2] - city2[2])^2)
end

# we define the fitness of a route as the inverse of its length
function fitness(route)
    route_length = sum(distance(route[i], route[i+1]) for i in 1:length(route)-1)
    route_length += distance(route[length(route)], route[1])
    return 1 / route_length
end

# we define mutation as the random swapping of two cities in a route
function mutate(route, mutation_rate=0.05)
    for i in 1:length(route)
        if rand() < mutation_rate
            j = rand(1:length(route))
            city1, city2 = route[i], route[j]
            route[i], route[j] = city2, city1
        end
    end
    return route
end

# we use ordered crossover to create a child from two parents
function crossover(parent1, parent2)

    idx1, idx2 = sort(rand(1:length(parent1), 2))
    subset_parent1 = parent1[idx1:idx2]

    offspring = [(-1.0,-1.0) for _ in 1:length(parent1)]
```

```
    offspring[idx1:idx2] = subset_parent1

    j = 1
    for i in 1:length(parent1)
        if i < idx1 || i > idx2
            while parent2[j] in subset_parent1
                j += 1
            end
            offspring[i] = parent2[j]
            j += 1
        end
    end
    return offspring
end

function roulette_selection(population, selection_rate=0.5)
    total_fitness = sum(fitness.(population))
    selected = []

    while length(selected) < length(population) * selection_rate
        accumulator = 0.0
        random_val = rand() * total_fitness
        for individual in population
            accumulator += fitness(individual)
            if accumulator > random_val
                push!(selected, individual)
                break
            end
        end
    end
    return selected
end

function breed(selected, population_size, mutation_rate = 0.05)
    # Create a new generation by breeding the selected individuals
    new_population = copy(selected)
    while length(new_population) < population_size
        parent1, parent2 = rand(selected), rand(selected)
        child = crossover(parent1, parent2)
        push!(new_population, mutate(child, mutation_rate))
    end
```

```
    return new_population
end

# a helper method for the genetic algorithm
function determine_maximum_fitness(population)
    maximum_fitness = 0.0
    best_route = population[1]
    best_individual_index = 1
    i=1
    for individual in population
        fitness_value = fitness(individual)
        if fitness_value > maximum_fitness
            maximum_fitness = fitness_value
            best_route = individual
            best_individual_index = i
        end
        i += 1
    end
    return best_individual_index, best_route, maximum_fitness
end

function genetic_algorithm(coordinates, generations, population_size = 100, selection_rate=0.3, mutation_rate=0.05)
    population = [shuffle(coordinates) for _ in 1:population_size]
    best_route = population[1]
    best_fitness = fitness(best_route)
    best_index = 1
    fitness_history = []
    for index in 1:generations
        selected = roulette_selection(population, selection_rate)
        population = breed(selected, population_size, mutation_rate)
        _, population_best_route, population_best_fitness = determine_maximum_fitness(population)
        if population_best_fitness > best_fitness
            best_route = population_best_route
            best_fitness = population_best_fitness
            best_index = index
        end
        push!(fitness_history, best_fitness)
    end
    return best_index, best_route, fitness_history
end
```

```
function random_search(coordinates, generations)
    route = shuffle(copy(coordinates))
    best_fitness = fitness(route)
    best_route = copy(route)
    fitness_history = Float64[]
    for _ in 1:generations
        shuffle!(route)
        if fitness(route) > best_fitness
            best_fitness = fitness(route)
            best_route = copy(route)
        end
        push!(fitness_history, best_fitness)
    end
    return best_route, fitness_history
end

function random_mutation_hill_climbing(coordinates, generations, mutation_rate=0.05) # check using 0.05, 0.1, 0.3, 0.5
    route = shuffle(copy(coordinates))
    best_fitness = fitness(route)
    best_route = copy(route)
    fitness_history = []
    for _ in 1:generations
        mutated_route = mutate(copy(route), mutation_rate)
        if fitness(mutated_route) > best_fitness
            best_fitness = fitness(mutated_route)
            best_route = copy(mutated_route)
        end
        push!(fitness_history, best_fitness)
    end
    return best_route, fitness_history
end
```

TSP_learning_curve_plotting.jl

```
using Plots
using Plots.PlotMeasures
using Statistics
using Distributed
using JLD2

number_of_workers = 4
```

```
addprocs(number_of_workers)

@everywhere include("TSP_genetic_algorithm_methods.jl")

# import the data from text document
@everywhere begin
    coordinates = []
    open("code_hw1/cities.txt") do file
        for line in eachline(file)
            x, y = split(line, ',')
            push!(coordinates, (parse(Float64, x), parse(Float64, y)))
        end
    end
end

generation = 100000
rs_best_fitness_history_four_workers = []
rmch_best_fitness_history_four_workers = []
ga_best_fitness_history_four_workers = []
ga_best_route_four_workers = []
ga_index_list = []

for p in workers()
    print("p: ", p, "\n")
    rs_best_route, rs_fitness_history = fetch(@spawnat p random_search(coordinates, generation))
    rmch_best_route, rmch_fitness_history = fetch(@spawnat p random_mutation_hill_climbing(coordinates, generation))
    ga_best_index, ga_best_route, ga_fitness_history = fetch(@spawnat p genetic_algorithm(coordinates, generation))
    push!(rs_best_fitness_history_four_workers, rs_fitness_history)
    push!(rmch_best_fitness_history_four_workers, rmch_fitness_history)
    push!(ga_best_fitness_history_four_workers, ga_fitness_history)
    push!(ga_best_route_four_workers, ga_best_route)
    push!(ga_index_list, ga_best_index)
end
rmprocs(workers())

best_worker, overall_best_route, maximum_fitness = determine_maximum_fitness(ga_best_route_four_workers)
shortest_distance = 1/maximum_fitness
println("The shortest distance is: ", shortest_distance)
println("Generation when this is found: ", ga_index_list[best_worker])

rs_best_fitness_history_four_workers = (hcat(rs_best_fitness_history_four_workers...))'
```

```
average_rs_fitness_history = reshape(mean(rs_best_fitness_history_four_workers, dims=1),(generation,))
error_rs_fitness_history = reshape(std(rs_best_fitness_history_four_workers, dims=1)/sqrt(number_of_workers),(generation,))

rmch_best_fitness_history_four_workers = (hcat(rmch_best_fitness_history_four_workers...))'
average_rmch_fitness_history = reshape(mean(rmch_best_fitness_history_four_workers, dims=1),(generation,))
error_rmch_fitness_history = reshape(std(rmch_best_fitness_history_four_workers, dims=1)/sqrt(number_of_workers),(generation,))

ga_best_fitness_history_four_workers = (hcat(ga_best_fitness_history_four_workers...))'
average_ga_fitness_history = reshape(mean(ga_best_fitness_history_four_workers, dims=1),(generation,))
error_ga_fitness_history = reshape(std(ga_best_fitness_history_four_workers, dims=1)/sqrt(number_of_workers),(generation,))

# save the arrays to a file
@save "result_data.jld2" average_rs_fitness_history error_rs_fitness_history average_rmch_fitness_history
error_rmch_fitness_history average_ga_fitness_history error_ga_fitness_history overall_best_route

x_values = collect(1:generation)
plot(x_values, average_rs_fitness_history, ribbon=error_rs_fitness_history, legend=true, xlabel="Generation", ylabel="Fitness",
dpi=300, size=(600, 400), left_margin = 40px, bottom_margin = 40px, label = "Random Search")
plot!(x_values, average_rmch_fitness_history, ribbon= error_rmch_fitness_history, legend=true, label = "Random Mutation Hill
Climbing")
plot!(x_values, average_ga_fitness_history, ribbon= error_ga_fitness_history, legend=true, label = "Genetic Algorithm")
savefig("learning_curve.png")

# plot the best route
x_coordinates = [x[1] for x in overall_best_route]
x_coordinates = vcat(x_coordinates, x_coordinates[1])
y_coordinates = [x[2] for x in overall_best_route]
y_coordinates = vcat(y_coordinates, y_coordinates[1])
plot(x_coordinates, y_coordinates, legend=false, xlabel="x", ylabel="y", dpi=300, size=(600, 400), left_margin = 40px,
bottom_margin = 40px, title="Shortest Route")
scatter!(x_coordinates, y_coordinates, legend=false, xlabel="x", ylabel="y", title="Shortest Route")
savefig("ga_shortest_route.png")
```

test_mutation.jl (Test file for verifying methods' correctness)

```
using Test
include("TSP_genetic_algorithm_methods.jl")

@testset "Test distance" begin
    city1 = (0, 3)
    city2 = (4, 0)  # Change square brackets to parentheses
    @test distance(city1, city2) ≈ 5.0
```

```
    #write more tests here
    city3 = (1.1, 2.2)  # Change square brackets to parentheses
    city4 = (4.1, 6.2)  # Change square brackets to parentheses
    @test distance(city3, city4) ≈ 5.0
end

@testset "Test fitness" begin
    route = [(0, 0), (0, 1), (1, 1), (1, 0)]  # Change square brackets to parentheses
    @test fitness(route) ≈ 1/4

    #write more tests here
    route2 = [(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)]  # Change square brackets to parentheses
    @test fitness(route2) ≈ 1/4
end

@testset "Test mutate" begin
    route = [(5, 6), (7, 8), (1, 2), (3, 4)]
    initial_route = copy(route)
    mutated_route = mutate(route, 0.0)
    @test mutated_route == initial_route

    #write more tests here
    mutated_route2 = mutate(route, 1.0)
    @test mutated_route2 != initial_route
end

@testset "Test crossover" begin
    parent1 = [(0, 0), (0, 1), (1, 1), (1, 0)]  # Change square brackets to parentheses
    parent2 = [(1, 0), (1, 1), (0, 0), (0, 1)]  # Change square brackets to parentheses
    child = crossover(parent1, parent2)
    @test length(child) == length(parent1)
    @test sort(child) == sort(parent1)

    parent1 = [(1, 2), (3, 4), (5, 6), (7, 8)]  # Change square brackets to parentheses
    parent2 = [(5, 6), (7, 8), (1, 2), (3, 4)]  # Change square brackets to parentheses
    child = crossover(parent1, parent2)
    @test length(child) == length(parent1)
    @test sort(child) == sort(parent1)

    parent1 = [(8, 7), (6, 5), (4, 3), (2, 1)]  # Change square brackets to parentheses
```

```
    parent2 = [(2, 1), (4, 3), (6, 5), (8, 7)]  # Change square brackets to parentheses
    child = crossover(parent1, parent2)
    @test length(child) == length(parent1)
    @test sort(child) == sort(parent1)
end

@testset "Test roulette selection" begin
    population = [[(0,1),(0,-1),(1,0),(-1,0)],[(0,1),(0,-1),(1,0),(-1,0)]]
    new_population = roulette_selection(population, 0.5, 1)
    @test length(new_population) == length(population)

end

@testset "Test determine maximum_fitness" begin
    population = [[(0,1),(0,-1),(1,0),(-1,0)],[(0,1),(0,-1),(1,0),(-1,0)]]
    best_worker, overall_best_route, maximum_fitness = determine_maximum_fitness(population)
    @test best_worker == 1
    @test overall_best_route == [(0,1),(0,-1),(1,0),(-1,0)]
    @test maximum_fitness == 1/4
end
```

movie_of_random_search.jl (for generating the movie)

```
using Random
using Statistics
using Plots

coordinates = []
open("cities.txt") do file
    for line in eachline(file)
        x, y = split(line, ',')
        push!(coordinates, (parse(Float64, x), parse(Float64, y)))
    end
end

function distance(city1, city2)
    return sqrt((city1[1] - city2[1])^2 + (city1[2] - city2[2])^2)
end

# we define the fitness of a route its total length
function fitness(route)
    route_length = sum(distance(route[i], route[i+1]) for i in 1:length(route)-1)
```

```
    route_length += distance(route[length(route)], route[1])
    return route_length
end

generations = 100000

function random_search(coordinates, generations)
    route = shuffle(copy(coordinates))
    best_fitness = fitness(route)
    best_route = copy(route)
    fitness_history = Float64[]
    for index in 1:generations
        shuffle!(route)
        if fitness(route) < best_fitness
            best_fitness = fitness(route)
            best_route = copy(route)
            plot([best_route[i][1] for i in 1:length(best_route)], [best_route[i][2] for i in 1:length(best_route)], label="best route_$(index);
length = $(best_fitness)", title="Random Search")
            scatter!([best_route[i][1] for i in 1:length(best_route)], [best_route[i][2] for i in 1:length(best_route)],label="")
            savefig("movie/random_search_$(index).png")
        end
        #plot the best route
        push!(fitness_history, best_fitness)
    end
    return best_route, fitness_history
end

best_route, fitness_history = random_search(coordinates, generations)
plot(fitness_history, legend=false, xlabel="Generation", ylabel="Fitness", dpi=300, size=(600, 400), title="Random Search")
```

optimal_path.py (for finding the theoretical shortest path)

```


def tsp(data)
    # Implementation of Christofides algorithm for finding optimal path in graph
    # from Andrew Zhuravchak - student of CS@UCU
    # link: https://github.com/Retsediv/ChristofidesAlgorithm


# read in data from cities.txt
def read_data(file_name):
    data = []
```

```
    with open(file_name) as f:
        # append [x,y] coordinates of each city to data
        for line in f:
            x, y = line.strip().split(',')
            data.append([float(x), float(y)])
    return data

data = read_data("code_hw1/cities.txt")
length, path = tsp(data)
print("Length: ", length)

# plot the path
import matplotlib.pyplot as plt
x = []
y = []
for i in path:
    x.append(data[i][0])
    y.append(data[i][1])
plt.plot(x, y, 'xb-')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Theoretical shortest path')
plt.savefig('latex_hw1/theoretical_shortest.png')
```