**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 |

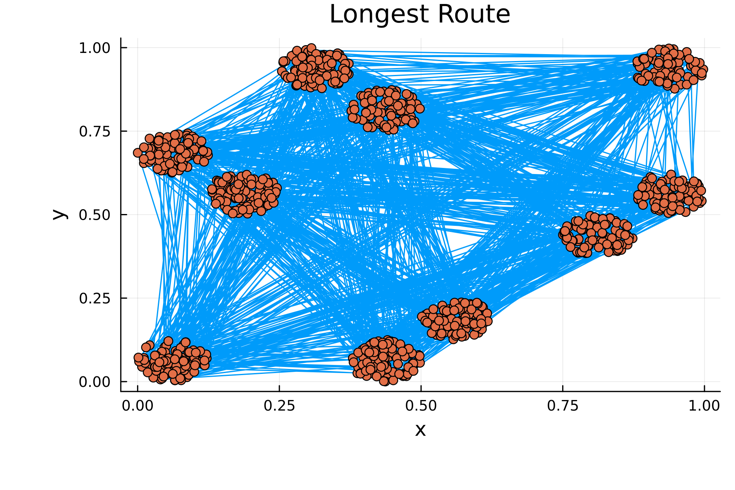
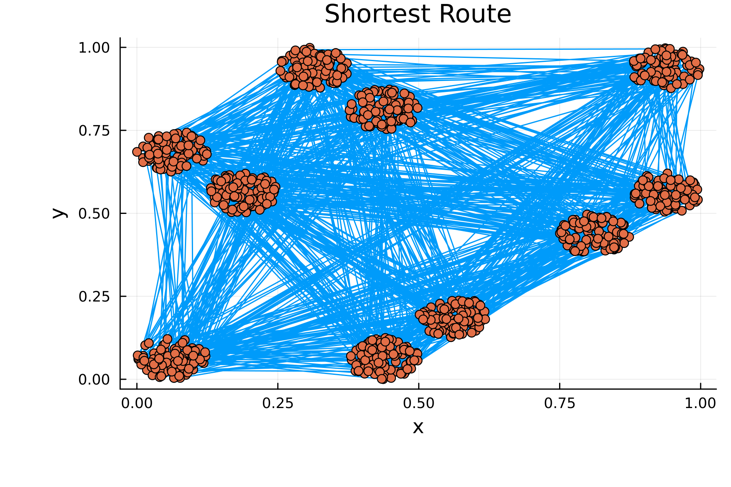
** **

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

**Theoretical Shortest Path:** 12.775068874879208

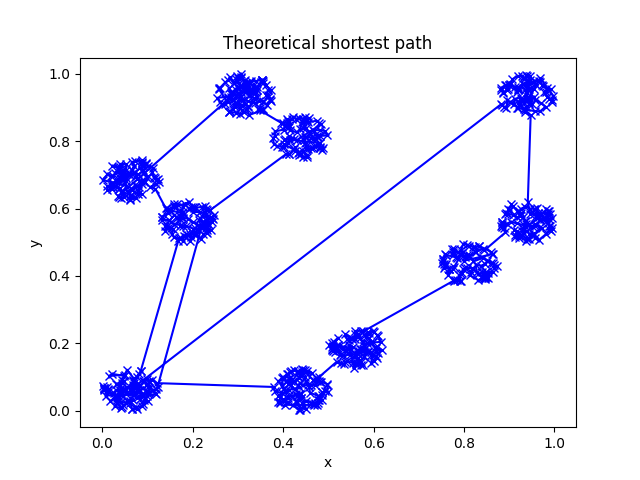


Figure 2. The theoretical shortest path found using Christofides’ algorithm.

**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>

1. **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.

1. **Performance Curves**

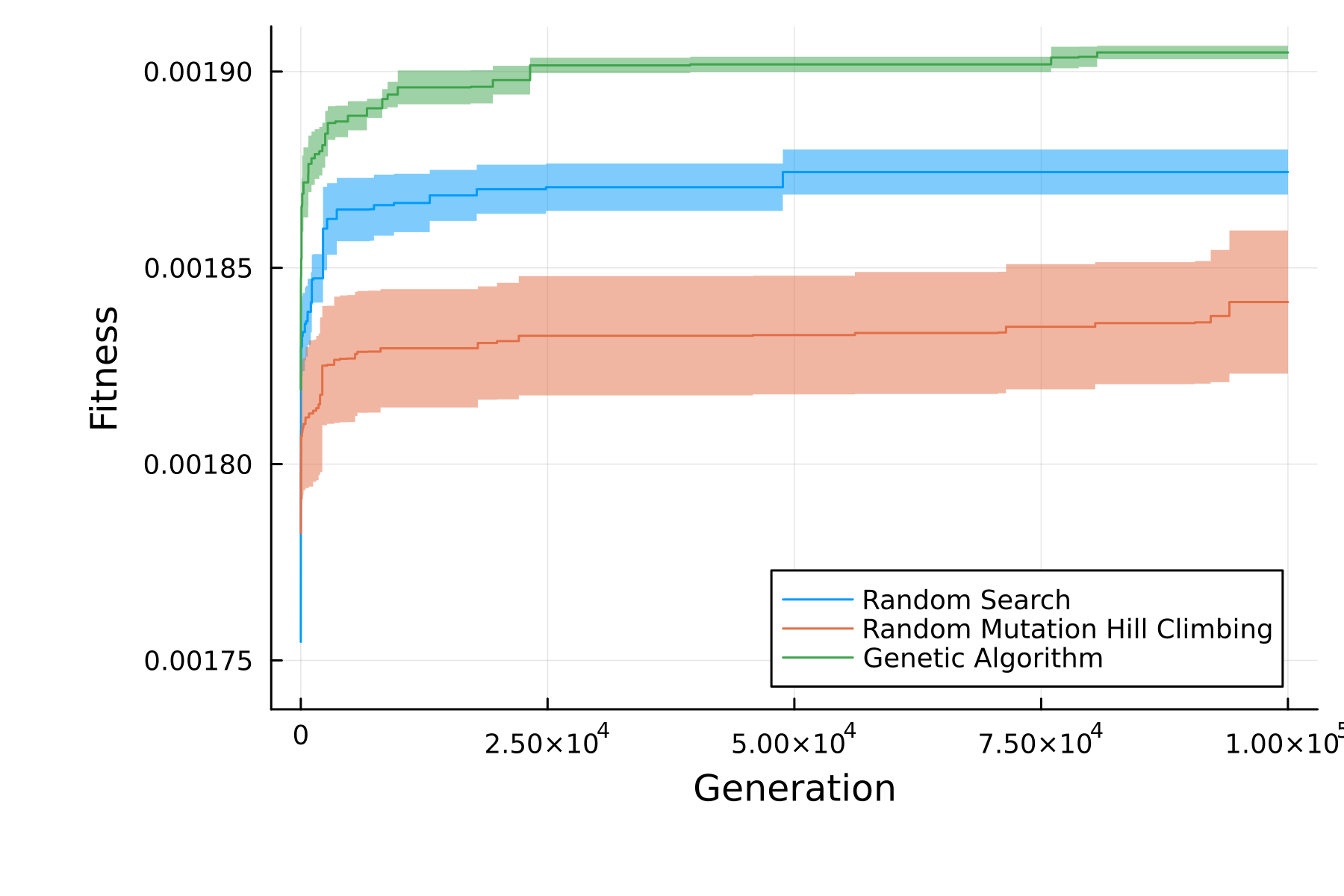
****

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)

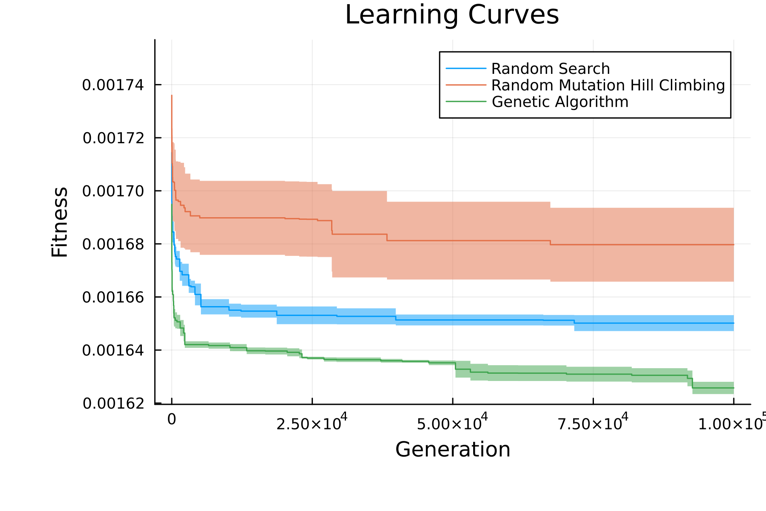


Figure 5. The longest path learning curves of random search, hill climber, and EA

**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')