Group **MK2**

Artifical Inteligence

Laboratory Report

2. Evolutional Strategies

Wojciech Bieniek
Artur Oleksiński

Introduction

Evolutionary Strategies which is implemented in this project belongs to the family of genetic algorithms. This project takes (λ + γ) approach, this mean that for the next generation we take individuals from both parents and offsprings/children. Evolutionary Strategies take advantege of sigma-besed mustation proces which is the main drive for improvment in this implementation.

Algorithm

1. **Initialization** - Initialization of the base population (generation 1)
2. **Evaluation** - Evaluation of generation 1
3. **Selection** - Selection of "parents" for creating new generation (Rulette or Steady State)
4. **Generating new generation** - Generating the new generationfor future calculations with lambda \ gamma approach
5. **Evaluation** - Evaluating the most current generation
6. **Check if done** - Checking the stop condition
7. **Loop** - loop until stop condition is met

```
function EvolutionAlgorithm(
    data,
    population_quantity::Int=200,
    epsilon=0.000001,
    save_results::Bool=false,
    selection_method::String="RuletteSelection"
)

    top = Int(floor(population_quantity/10))
    generation = 1
    population = []
    data_quantity = length(data)

    initialize_population(population, population_quantity)
    evaluate_generation(data, population, population_quantity, data_quantity, generation)
    population[1] = sort_generation(population[1], population_quantity)

    while generation < population_quantity
        if selection_method == "RuletteSelection"
            selected = rulette_selection(population, generation)
        else
            selected = select_parents(population, generation)
        end

        next_generation = new_generation_evo(data, data_quantity, population, selected)
        generation +=1
        append!(population, next_generation)
        new_best = population[generation].individuals[1].fit
        best = mean([x.fit for x in population[generation-1].individuals[1:2]])

        if abs(new_best - best) > epsilon
            best = new_best
        else
            break
        end

        evaluate_generation(
            data,
            population,
            population_quantity,
            data_quantity,
            generation
        )
    end
```

Inintialization

Initialization is start of the program. It is a part where we create our “generation 1”. This is also the place for initialization of all necessary variables.

```
top = Int(floor(population_quantity/10))
generation = 1
population = []
data_quantity = length(data)
best = Inf

initialize_population(population, population_quantity)
evaluate_generation(data, population, population_quantity, data_quantity, generation)
```

Selection

Project includes two diffrent approaches to the selection of parents.

Rulette Selection

First impelementation is the **Rulette Selection** which uses the culmulative fitness value and fitness values of every individual to create chances of getting into maiting pool.
In our algorithm, as we want to minimize the fitness value (best case is 0), we are creating chances by deviding the best fitness value by fitness value of the candidate.
In this way we have chance equal 1 to choose the best candidate and other candidates heve equally fair chances (based on thir fitness value).

```
function rulette_selection(generation::Generation, desired_quantity::Int)
    result = Vector{Invi}()
    population = sort_generation(generation, desired_quantity)
    best = population.individuals[1].fit

    for individual in population.individuals
        probability = (best)/individual.fittness
        print(probability)
        chance = rand(Uniform(0.0, 1.0))
        if probability > chance
            append!(result, individual)
        end
    end

    # Just to assure there are at least 2 individuals in the new mating pool
    if length(result) < 2
        append!(result, population.individuals[best_index])
    end

    return Generation(result)
end
```

Steady State Selection

Second implementation is the **Solid State Selection**.
In this method we sort the whole population by fitness value and choose x of the best candidates. The x is dependent on the population size and in our implementation is calculated as population_quantity / 10 to get 10% of current generation.

```
function select_parents(population, generation, number)
    population[generation].individuals = sort(population[generation].individuals, by=v -> v
       return population[generation].individuals[1:number]
end
```

Generating the new Generation

After selecting the candidates for maiting pool we can proceed to creating new generation.
In our implementation we start with crossover of individuals in selection (mating pool).

```
function new_generation_evo(data, data_quantity, population, selected)
    # Make crossover based on selected data and generate 5*population quantity of children
    offspring = crossover(data, data_quantity, population, selected, rand(1:2))
    # Mutate all of children
    offspring = Generation(mutation(offspring))
    # Mutate parents
    selected = mutation(selected)
    # Evaluate new generation
    offspring = evaluate_generation(data, offspring, length(population[1].individuals), dat
    # Select the best from the population λ + γ and return
    return select_parents_generation(
        Generation(Vector{Invi}(vcat(selected, offspring.individuals))),
        length(population[1].individuals),
        20

    )
end
```

The crossover is randomized process of children production. We select pair of parents and exchange their genes in the chomosome. The part of chromosome being exchanged is uniformly random but always they exchange something.

```
function crossover(data, data_quantity, population, selected, separator)
    len_s = length(selected)
    len_p = length(population[1].individuals)
    offspring = []

    # We are generating 5*difference between the sizes of the base population
    for i in 1:len_p*5
        # Choosing the first parent randomly
        parent1 = rand(1:len_s)
        # Choosing the second parent randomly from population without parent1
        leftover = [r for r in 1:len_s-1 if r!=parent1]
        parent2 = rand(leftover)
        # Creating Child
        child = cross_two(data, data_quantity, selected[parent1], selected[parent2], separa
        # Adding the child to the offspring
        append!(offspring, child)
    end
    # Return the offspring
    return offspring
end
```

After succesfull crossover it is time to mutate both parents whom got into the mating pool and the freshly generated offspring.

The mutation is done with the gen-support variables $\sigma_{a,b,c}$, and the constants τ_1, and τ_2.

```
# Mutation based on the taus and Normal random distributions
function mutation(offspring, gens_count=3)
    # Length of given ofspring vector
    nr_of_genes = length(offspring[1].chromosome)
    len = length(offspring)
    # For every individual
    for i in 1:len
        # calculate τ₁ for given chromosome
        gen_tau_1 = exp(rand(Normal(0, τ₁)))
        # For every gen in chromosome
        for gen in 1:nr_of_genes

            # Mutate every gene
            offspring[i].chromosome[gen] = offspring[i].chromosome[gen] + rand(Normal(
                0,
                offspring[i].σ[gen]
            ))
            # Mutate every σ
            offspring[i].σ[gen] = offspring[i].σ[gen]*gen_tau_1*exp(rand(Normal(0, τ₂)))

        end
    end
    # REsturn the offspring after mutation
    return offspring
end
```

After this, new generation is being evaluated for the fitness values, sorted and choppend to the proper population size.

Stop condition

The mechanism which main purpouse is to prevent the algorithm from going into infinity.
There are two things which can stop main loop:

1. Checking of the result improvements.

When the algorithm is showing any progress then we are stoping the loop.
We use the absolute value of subraction of best fitnes value from current generation from the best fitness value from previous generation.

2. Generation limit

To avoid too long execution times which could took infinitly lon,g taking into account the random nature of the algorithm, we cap the number of generations. If generation > max then end.

Experimentation

With epsilon equal to **1e-6**
Data set nr **14**

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	100	0.255943	“Steady State”	2.113s	45
1	100	0.256956	“Steady State”	1.272s	36
2	100	0.256939	“Steady State”	1.274s	32
3	100	0.256940	“Steady State”	1.547s	44
4	100	0.256754	“Steady State”	1.350s	37
5	100	0.256976	“Steady State”	1.301s	24
6	100	0.256939	“Steady State”	1.890s	38
7	100	0.256944	“Steady State”	1.814s	59
8	100	0.256939	“Steady State”	1.729s	41
9	100	0.256939	“Steady State”	1.792s	28
10	100	0.256942	“Steady State”	1.608s	31

Mean Fitness	Mean Time	Mean Generations
0.256946	1.47417	34.5833

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	100	0.256941	“Rulette”	2.478s	48
1	100	0.256939	“Rulette”	1.418s	46
2	100	0.256943	“Rulette”	1.500s	35
3	100	0.256939	“Rulette”	1.636s	56
4	100	0.256939	“Rulette”	1.359s	40
5	100	0.256940	“Rulette”	1.749s	51
6	100	0.256940	“Rulette”	1.943s	64
7	100	0.256942	“Rulette”	1.614s	57
8	100	0.256939	“Rulette”	1.818s	59
9	100	0.256941	“Rulette”	1.312s	40
10	100	0.256940	“Rulette”	1.623s	56

Mean Fitness	Mean Time	Mean Generations
0.256940	1.5375	46

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	200	0.256941	“Steady State”	2.349s	32
1	200	0.256939	“Steady State”	1.814s	31
2	200	0.257233	“Steady State”	1.840s	23
3	200	0.256939	“Steady State”	1.994s	26
4	200	0.256975	“Steady State”	1.730s	22
5	200	0.256941	“Steady State”	1.686s	31
6	200	0.256939	“Steady State”	2.186s	29
7	200	0.256939	“Steady State”	1.873s	29
8	200	0.256942	“Steady State”	1.469s	28
9	200	0.256939	“Steady State”	2.057s	36
10	200	0.256939	“Steady State”	2.117s	34

Mean Fitness	Mean Time	Mean Generations
0.256970	1.91955	29.1818

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	200	0.256940	“Rulette”	3.749s	77
1	200	0.256940	“Rulette”	2.742s	56
2	200	0.256739	“Rulette”	2.784s	40
3	200	0.256939	“Rulette”	2.488s	44
4	200	0.256950	“Rulette”	2.351s	33
5	200	0.256939	“Rulette”	1.987s	42
6	200	0.256939	“Rulette”	2.986s	52
7	200	0.256939	“Rulette”	2.183s	38
8	200	0.256939	“Rulette”	2.602s	46
9	200	0.256941	“Rulette”	2.274s	36
10	200	0.256940	“Rulette”	2.582s	45

Mean Fitness	Mean Time	Mean Generations
0.256940	2.61164	46.2727

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	500	0.256939255	“Steady State”	3.210s	20
1	500	0.256939145	“Steady State”	3.264s	26
2	500	0.256939695	“Steady State”	3.180s	28
3	500	0.256939617	“Steady State”	3.177s	24
4	500	0.256939134	“Steady State”	2.906s	24
5	500	0.256938702	“Steady State”	3.360s	25
6	500	0.256939630	“Steady State”	3.495s	26
7	500	0.256938980	“Steady State”	2.769s	28
8	500	0.256940246	“Steady State”	2.871s	22
9	500	0.256939078	“Steady State”	3.454s	26
10	500	0.256944352	“Steady State”	2.285s	22

Mean Fitness	Mean Time	Mean Generations
0.256939803	3.08827	24.6364

Tries	Population Szie	Fitness	Selection Method	Time	Generations
Warmup	500	0.256938655	“Rulette”	4.914s	45
1	500	0.256939160	“Rulette”	4.684s	38
2	500	0.256940051	“Rulette”	4.990s	37
3	500	0.256939112	“Rulette”	4.660s	38
4	500	0.256938990	“Rulette”	3.934s	36
5	500	0.256939483	“Rulette”	5.350s	38
6	500	0.256939325	“Rulette”	4.470s	39
7	500	0.256938644	“Rulette”	4.594s	37
8	500	0.256939366	“Rulette”	4.276s	41
9	500	0.256942457	“Rulette”	4.936s	33
10	500	0.256939113	“Rulette”	4.144s	34

Mean Fitness	Mean Time	Mean Generations
0.256939487	4.632	37.8182