INF 421 PROJECT

Evolutionary Algorithms
Professor Martin Krejca

February 5, 2024

Gabriel Pereira de Carvalho

CONTENTS

1 INTRODUCTION

This report presents a solution to the programming project **Evolutionary Algorithms** for the course INF421: Design and Analysis of Algorithms at École Polytechnique. Each task is developed in a section of the report which also contains the code implemented using the Python programming language.

1.1 Instructions for running the project locally

The source code can be accessed on the project's Github repository. To execute it locally, clone the repository and install the project's dependencies.

```
git clone https://github.com/ArkhamKnightGPC/INF421.git
pip install requirements.txt
```

All the code can be found in the `code` folder in the repository.

```
cd code
```

Now, to generate the scatter plots for the empiric runtime analysis of the **OneMax** and **LeadingOnes** benchmark functions we run the `EmpiricRunTimes.py` file. The generated plots are saved in the `plots` folder.

```
python EmpiricRunTimes.py
```

Unit tests are also provided in the `unit_tests` folder.

2
TASK 1: INDIVIDUALS AND BENCHMARK FUNCTIONS

Write code that allows to use individuals as well as the three functions `OneMax`, `LeadingOnes`, and `Jump_k` . For individuals, do not use libraries but implement a data type that fully utilizes the memory. That is, do not store each bit value of an individual in a byte but in an actual bit.

The code for this project was developed with a strong respect for the SOLID design principles. For task1, taking the **single responsability principle** into account, three classes were developed: one defining the `Individual` data type, one providing implementations for the Benchmark Functions and one implementing the $(1+1)$ EA.

2.1 Individuals

Firstly, it is important to observe that the basic data types in Python use 1 byte of memory. Therefore, using a boolean variable for each bit value of an individual will not fully utilize memory. To do this, we use an array of integers, we each integer in the array represents 32 bit values.

The Individual class also provides auxiliary functions that will be used later on:

- a `get` function to retrieve a single bit;
- a `set` function to set a bit value to 1;
- a `reset` function to set a bit value to 0;
- a `flip` function to change the value of a single bit;
- a `count` function returning the number of bits equal to 1

```
class Individual:
    """
    Represents candidate solutions x = (x1, ..., xn)
    """
    def __init__(self, size):
        """
        Constructor for new Individual
        """
        # Number of integers necessary to represent all xi's
        necessary_integers = (size + 31) // 32
```

```python
        self.size = size
        self.bits = [0] * necessary_integers

    def get(self, idx):
        """
        Get bit at index idx
        """
        test_bit = self.bits[idx // 32] & (1 << (idx % 32))
        return 1 if test_bit > 0 else 0

    def set(self, idx):
        """
        Set bit at index idx to 1
        """
        self.bits[idx // 32] |= (1 << (idx % 32))

    def reset(self, idx):
        """
        Set bit at index idx to 0
        """
        self.bits[idx // 32] &= ~(1 << (idx % 32))

    def flip(self, idx):
        """
        Flip bit at index idx
        """
        bit_i = self.get(idx)
        if bit_i == 0:
            self.set(idx)
        else:
            self.reset(idx)

    def count(self):
        """
        Count number of bits equal to 1
        """
        result = 0
        for i in range(self.size):
            bit_i = self.get(i)
            result += bit_i
        return result
```

2.2 Benchmark Functions

```python
from Individual import Individual

def OneMax(individual):
    """
    Returns the number of 1s of the input
```

```
    """
    return individual.count()

def LeadingOnes(individual):
    """
    Returns the length of the longest consecutive prefix of 1s
    """
    n = individual.size
    result = 0
    for i in range(n):
        prefix_product = 1
        for j in range(0, i + 1):
            prefix_product *= individual.get(j)
        result += prefix_product
    return result

def JumpK(individual, k):
    """
    Analog to OneMax but penalizes individuals with a number of ones in n-k
    +1,...,n-1
    """
    n = individual.size
    one_max_x = OneMax(individual)
    if one_max_x <= n - k or one_max_x == n:
        return k + one_max_x
    return n - one_max_x
```

2.3 $(1 + 1)$ EA

Since the all-1s bit string is the unique global optimum of all three functions, we use a direct comparison all-1s bit string as our termination condition. A possible alternative is to impose a maximum number of iterations, but since we are interested in measuring performance against benchmark functions it is more interesting to let the EA reach the optimal solution. As mentioned in the statement the mutation rate adopted is $p = \frac{1}{n}$.

```
from Individual import Individual
import numpy as np
import random

def generateRandomOffspring(x, p):
    """
    Generate a copy of x flipping each bit independently with probability p
    """
    n = x.size
    y = Individual(n)
    for idx in range(n):
        xi = x.get(idx)
        rand_var = np.random.uniform(0, 1)
        #bit idx in y is 1 if and only if
```

```
        mutated_to_one = (rand_var < p and xi == 0) #mutated from 0 to 1 (with probability p)
        stayed_one = (rand_var >= p and xi == 1) #did not mutate, was already 1 (probability 1-p)
        if (mutated_to_one or stayed_one):
            y.set(idx)
    return y

def EvolutionaryAlgorithm(f, n):
    """
    (1+1) Evolutionary Algorithm
    """
    t = 0
    Pt = generateRandomOffspring(Individual(n), 0.5)  # random initial solution

    while Pt.count() < n:
        y = generateRandomOffspring(Pt, 1 / n)
        if f(y) > f(Pt):  # we pick solution that maximizes f
            Pt = y
        t += 1

    return Pt
```

3
Task 2: Runtime analysis

3.1 Theoretical run time upper bounds

Prove mathematically (preferably rather tight) upper bounds on the expected run time of the $(1+1)$ EA on `OneMax` and on `LeadingOnes`.

The method used for the proofs in this task is the classical fitness levels method (1).

Let $(P^t)_{t\geq 0}$ represent the sequence of individuals in the population at each iteration of the algorithm, where

$$P^t = (P_1^t, \cdots, P_n^t) \in \{0,1\}^n \quad \forall t \geq 0.$$

We observe that $(P^t)_{t\geq 0}$ is a markov chain with state space $E = \{0,1\}^n$.

We consider a **fitness-based partition** of the state space $E = \bigcup_{i\in[0..n]} A_i$ where

$$\forall i \in [0..n] \quad A_i = \{x \in E \quad | \quad f(x) = i\}.$$

In the $(1+1)$ EA we note that $f(P^t) \geq \quad f(P^{t-1}) \quad \forall t \geq 1$. Thus, $(P^t)_{t\geq 0}$ is a **non-decreasing level process**.

Our strategy is to compute $\forall i \in [0..n-1]$ the probability p_i of leaving level A_i. Then, the expected number of iterations to leave level A_i is $\frac{1}{p_i}$.

Thus, we have the upper bound

$$\sum_{i=1}^{n-1} \frac{1}{p_i}$$

for the expected run time.

3.1.1 • Theoretical bound for OneMax

Let $\mathcal{P}(m,i,j)$ denote the probabilty of leaving level A_i and arriving at level A_j in an iteration for a problem size of m bits. We impose the following constraints

$$1 \leq m \leq n; \quad 0 \leq i < m; \quad i < j \leq m. \tag{1}$$

Let's discuss these coefficients can be calculated using dynamic programming. We start by defining the base cases $(m, 0, j)$. We have

$$\mathcal{P}(m,0,j) = \frac{\binom{m}{j} p^j (1-p)^{m-j}}{\sum_{k=0}^{m} \binom{m}{k} p^k (1-p)^{m-k}} \tag{2}$$

Now, we formulate our recurrence $\forall i \geq 1$.

$$\mathcal{P}(m,i,j) = p\mathcal{P}(m-1,i-1,j) + (1-p)\mathcal{P}(m-1,i-1,j-1) \tag{3}$$

To calculate p_i using our coefficients, we have

$$p_i = \sum_{j=i+1}^{n} \mathcal{P}(n,i,j). \tag{4}$$

We have a $O(n^3)$ algorithm to compute the coefficients p_i and thus the upper bound for the run time. Plotting the run time estimates for different values of n in figure 1, we conclude that the run time complexity is $O(n\log(n))$.

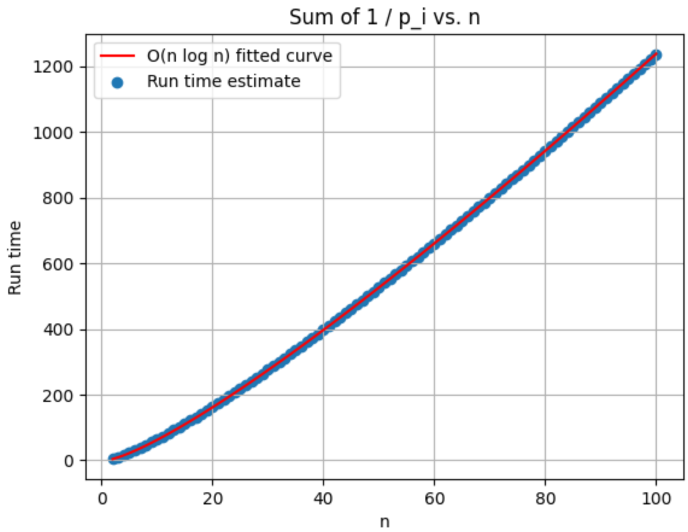

Figure 1: $O(n\log(n))$ fit for theoretical runtime curve

```python
import numpy as np
from math import comb
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def calculate_coefficients(n, p):
    P = [[[0.0] * (n + 1) for _ in range(n + 1)] for _ in range(n + 1)]

    # Base case: m
    for m in range(1, n + 1):
        for j in range(m + 1):
            denominator = sum(comb(m, k) * (p ** k) * ((1 - p) ** (m - k)) for k in range(m + 1))
            P[m][0][j] = comb(m, j) * (p ** j) * ((1 - p) ** (m - j)) / denominator

    # Recurrence relation
```

```python
    for m in range(2, n + 1):
        for i in range(1, m + 1):
            for j in range(i, m + 1):
                P[m][i][j] = p * P[m - 1][i - 1][j] + (1 - p) * P[m - 1][i - 1][j - 1]

    # Calculate p_i
    p_i = [0.0] * (n + 1)
    for i in range(n):
        for j in range(i + 1, n + 1):
            p_i[i] += P[n][i][j]

    return p_i

# Set up values for n
n_values = list(range(2, 101))  # Change the range as needed

# Calculate the sum for each n
sum_values = []
for n in n_values:
    coefficients = calculate_coefficients(n, 1 / n)
    sum_val = sum(1 / p if p != 0 else 0 for p in coefficients)  # Include i = 0
    sum_values.append(sum_val)

# Plot the sum as a function of n
plt.plot(n_values, sum_values)
plt.xlabel('n')
plt.ylabel('Sum')
plt.title('Sum of 1 / p_i vs. n')
plt.grid(True)
plt.savefig('../plots/OneMaxTheoreticalRunTime.png')

#function for fitting
def fit_function(y, a):
    return a * y * np.log(y)

# Perform curve fitting
popt, pcov = curve_fit(fit_function, n_values, sum_values, maxfev=10000) # maxfev increased for more iterations

# Plot the data and fitted curve
plt.scatter(n_values, sum_values, label='Run time estimate')
plt.plot(n_values, fit_function(np.array(n_values), *popt), 'r-', label='O(n log n) fitted curve')
plt.xlabel('n')
plt.ylabel('Run time')
plt.legend()
plt.grid(True)
plt.savefig('../plots/OneMaxTheoreticalRunTimeFit.png')
```

3.1.2 • Theoretical bound for LeadingOnes

For the `LeadingOnes` function, $\forall i \in [0..n-1]$, a necessary and sufficient condition for a mutation to leave level A_i is to keep bits $P_1^t, ..., P_i^t$ unchanged and to flip the bit P_{i+1}^t.

$$p_i = \frac{1}{n}\left(\frac{n-1}{n}\right)^i \tag{5}$$

Now, to estimate the run time

$$\begin{aligned}
\sum_{i=0}^{n-1} \frac{1}{p_i} &= \sum_{i=0}^{n-1} n\left(\frac{n}{n-1}\right)^i && (6)\\
&= n\sum_{i=0}^{n-1}\left(1+\frac{1}{n-1}\right)^i && (7)\\
&\leq n\sum_{i=0}^{n-1}\left(1+\frac{1}{n-1}\right)^{n-1} && (8)\\
&\leq n\sum_{i=0}^{n-1} e && (9)\\
&= en^2 && (10)
\end{aligned}$$

so the expected time complexity is $O(n^2)$.

3.2 Empirical run time estimates

Complement your theoretical bounds with empirical results and compare them.

In order to estimate the expected runtime empirically, we use the **law of large numbers**. Let $k \in \mathbb{N}$ and $T_1, ..., T_k$ represented run time measurements. We suppose the run times are independent and identically distributed. Then, by the law of large numbers,

$$\frac{T_1 + ... + T_k}{k} \xrightarrow[k \to +\infty]{} \mathbb{E}[T_1] \tag{11}$$

For practical reasons, we adopt $k = 30$.

The empirical results were plotted in a scatter plot together with a curve representing the theoretical run time bound. Analysing the plots in figures 2 and 3, we conclude that for `OneMax` we observe indeed that the run time is a linear function of the problem size and for `LeadingOnes` we observe a quadratic curve. Thus, the theoretical run time estimates are accurate and fit well with the empirical results.

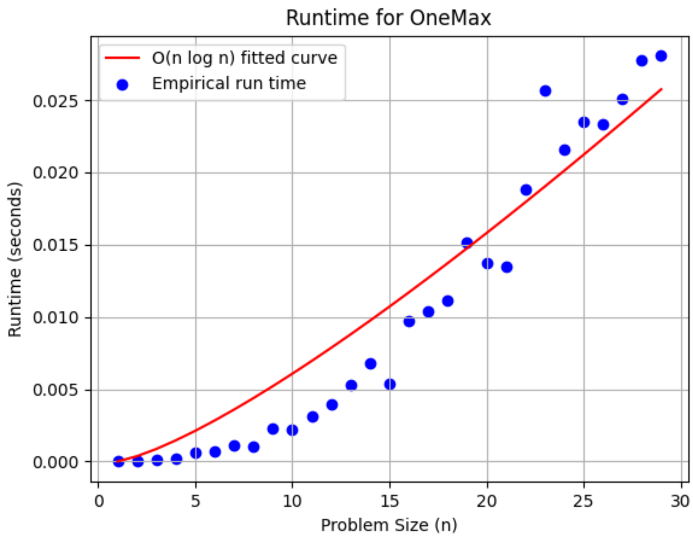

Figure 2: Empiric runtime analysis for OneMax

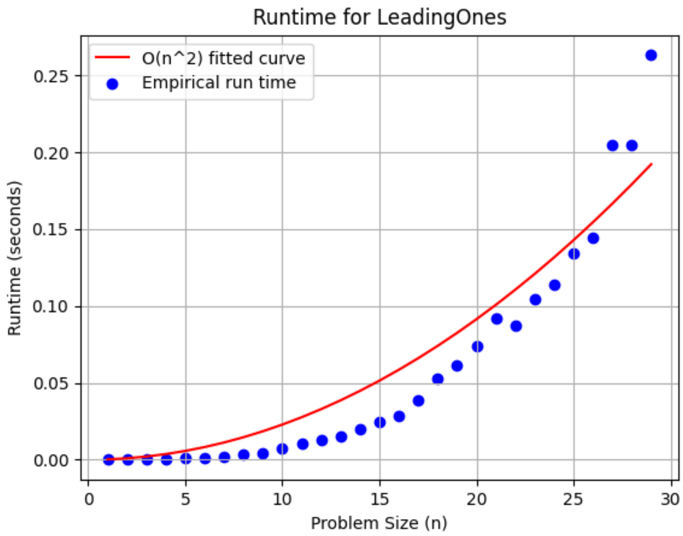

Figure 3: Empiric runtime analysis for LeadingOnes

3.3 Empirical run time tests for the $(\mu + 1)$ EA

Furthermore, run empirical tests for the $(\mu + 1)$ EA on `OneMax` with various, self-chosen values of μ. Visualize the expected run time. What do you see? What μ would you recommend?

In your report, do not forget to add a brief discussion about the parameter choices you made yourself, especially the number of tries for each value of μ you chose.

The code for the $(\mu + 1)$ EA is very similar to the $(1 + 1)$ EA with the following changes:

- termination condition must be checked for all μ individuals in the population;
- to determine if offspring is accepted, we must compare it to the least fit individual in the population.

There are two major computational costs associated with using $\mu > 1$:

- Sorting the population by fitness, which must be done at each iteration in the algorithm description.
- Computing the fitness value for the entire population at each iteration.

In order to minimize the impact of these two operations, we use the **priority queue** data structure to store the population (implemented in the **heapq** module in python).
In the priority queue, we store pairs $(f(x), x)$ where x represents an individual in the population. This way we compute the fitness of each element only once when adding it to queue and then we can retrieve this value in the future without having to compute it again.
An important point is that we create a comparator for the **Individual** data type in order to use this approach. Since this is merely to break ties between individuals with same fitness, and we don't distinguish between elements with same fitness in the $(\mu + 1)$ EA the exact content of this comparator is not important.

```
def __lt__(self, other):
    if(self.size < other.size):
        return True
    else:
        return False
```

Below is the code for the $(\mu + 1)EA$, which was added to `EA.py`.

```
from Individual import Individual
import numpy as np
import random
```

```python
import heapq

def generateRandomOffspring(x, p):
    """
    Generate a copy of x flipping each bit independently with probability p
    """
    n = x.size
    y = Individual(n)
    for idx in range(n):
        xi = x.get(idx)
        rand_var = np.random.uniform(0, 1)
        #bit idx in y is 1 if and only if
        mutated_to_one = (rand_var < p and xi == 0) #mutated from 0 to 1 (
    with probability p)
        stayed_one = (rand_var >= p and xi == 1) #did not mutate, was
    already 1 (probability 1-p)
        if (mutated_to_one or stayed_one):
            y.set(idx)
    return y

def EvolutionaryAlgorithm(f, n):
    """
    (1+1) Evolutionary Algorithm
    """
    t = 0
    Pt = generateRandomOffspring(Individual(n), 0.5)  # random initial
    solution

    while Pt.count() < n:
        y = generateRandomOffspring(Pt, 1 / n)
        if f(y) > f(Pt):  # we pick solution that maximizes f
            Pt = y
        t += 1

    return Pt

def EvolutionaryAlgorithm2(f, n, mu):
    """
    (mu + 1) Evolutionary Algorithm
    """
    t = 0
    Pt = []

    for _ in range(mu):# we must create a random initial population of size
    mu
        individual = generateRandomOffspring(Individual(n), 0.5)
        pair_fitness_individual = (f(individual), individual) #we create
    pair (fitness, individual)
        Pt.append(pair_fitness_individual)

    heapq.heapify(Pt) # we transform population into a priority queue

```

```
    while True:

        most_fit_individual = max(Pt)
        if(most_fit_individual[1].count() == n):
            return most_fit_individual[1]

        offspring = generateRandomOffspring(random.choice(Pt)[1], 1 / n)

        heapq.heappushpop(Pt, (f(offspring), offspring)) #we add offspring
 to the priority queue and pop least fit individual
        t += 1
```

In testing, the problem size was fixed $n = 30$. We tested μ from 1 to 30. Same as before, in order to plot the run times we performed an average over 30 different execution for each value of μ.

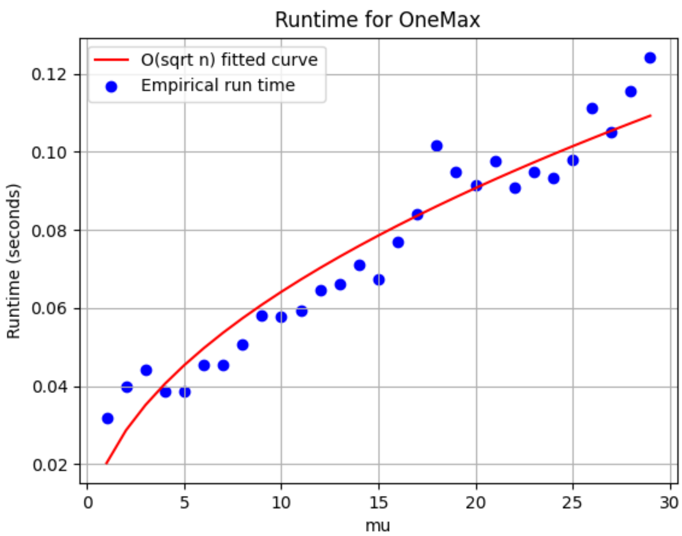

Figure 4: Run time for $\mu \in [1..30]$

In figure 4, we observe that the run time is an increasing function of μ. In order to study this dependency, different fit functions were experimented, and we conclude that the run time is $O(\sqrt{\mu})$ for a fixed n.

3.4 Code for empirical run time plots

```
import BenchmarkFunctions
import EA
import time
import matplotlib.pyplot as plt
import numpy as np
```

```python
from scipy.optimize import curve_fit
from statistics import mean

def PlotOneMaxRunTime():
    """
    Generates scatter plot for empirical run time analysis using (1+1) EA
    and the OneMax benchmark function
    """
    nvals = range(1, 30)
    run_times = []

    for n in nvals:

        number_of_trials = 30
        current_trial = 1
        trial_run_times = []

        while(current_trial <= number_of_trials):

            start_time = time.process_time() #we measure start time before
    running the EA
            solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.OneMax, n
    )
            end_time = time.process_time() #we measure end time after
    running the EA

            trial_run_times.append(end_time - start_time)
            current_trial += 1

        run_times.append(mean(trial_run_times))

    #function for fitting
    def fit_function(y, a):
        return a * y * np.log(y) #Theoretical analysis points to n log(n)
    complexity

    # Perform curve fitting
    popt, pcov = curve_fit(fit_function, nvals, run_times, maxfev=10000) #
    maxfev increased for more iterations

    # we make a scatter plot for run times
    plt.scatter(nvals, run_times, color='blue', marker='o', label='Empirical
     run time')
    plt.plot(nvals, fit_function(np.array(nvals), *popt), 'r-', label='O(n
    log n) fitted curve')
    plt.xlabel('Problem Size (n)')
    plt.ylabel('Runtime (seconds)')
    plt.title('Runtime for OneMax')
    plt.legend()
    plt.grid(True)

    #save plot in a png
```

```
    plt.savefig('../plots/OneMaxRunTime.png')
    return

def PlotLeadingOnesRunTime():
    """
    Generates scatter plot for empirical run time analysis using (1+1) EA and the LeadingOnes benchmark function
    """
    nvals = range(1, 30)
    run_times = []

    for n in nvals:

        number_of_trials = 30
        current_trial = 1
        trial_run_times = []

        while(current_trial < number_of_trials):

            start_time = time.process_time() #we measure start time before running the EA
            solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.LeadingOnes, n)
            end_time = time.process_time() #we measure end time after running the EA

            trial_run_times.append(end_time - start_time)#we use average of measurements as theestimator
            current_trial += 1

        run_times.append(mean(trial_run_times))#we use average of measurements as theestimator

    #function for fitting
    def fit_function(y, a):
        return a * y * y #Theoretical analysis points to n^2 complexity

    # Perform curve fitting
    popt, pcov = curve_fit(fit_function, nvals, run_times, maxfev=10000) # maxfev increased for more iterations

    # we make a scatter plot for run times
    plt.scatter(nvals, run_times, color='blue', marker='o', label='Empirical run time')
    plt.plot(nvals, fit_function(np.array(nvals), *popt), 'r-', label='O(n^2) fitted curve')
    plt.xlabel('Problem Size (n)')
    plt.ylabel('Runtime (seconds)')
    plt.title('Runtime for LeadingOnes')
    plt.legend()
    plt.grid(True)

```

```
    #save plot in a png
    plt.savefig('../plots/LeadingOnesRunTime.png')
    return

def PlotMuPlusOneEAOneMax():
    """
    Generates scatter plot for empirical run time analysis using (mu + 1) EA and the OneMax benchmark function
    """
    n = 30 #we keep n constant and vary only mu
    mu_vals = range(1, 30)
    run_times = []

    for mu in mu_vals:

        number_of_trials = 30
        current_trial = 1
        trial_run_times = []

        while(current_trial < number_of_trials):

            start_time = time.process_time() #we measure start time before running the EA
            solution = EA.EvolutionaryAlgorithm2(BenchmarkFunctions.OneMax, n, mu)
            end_time = time.process_time() #we measure end time after running the EA

            trial_run_times.append(end_time - start_time)#we use average of measurements as theestimator
            current_trial += 1

        run_times.append(mean(trial_run_times))#we use average of measurements as theestimator

    #function for fitting
    def fit_function(y, a):
        return a * np.sqrt(y) #Theoretical analysis points to n log(n) complexity

    # Perform curve fitting
    popt, pcov = curve_fit(fit_function, mu_vals, run_times, maxfev=10000) # maxfev increased for more iterations

    # we make a scatter plot
    plt.scatter(mu_vals, run_times, color='blue', marker='o', label='Empirical run time')
    plt.plot(mu_vals, fit_function(np.array(mu_vals), *popt), 'r-', label='O(sqrt n) fitted curve')
    plt.xlabel('mu')
    plt.ylabel('Runtime (seconds)')
    plt.title('Runtime for OneMax')
```

```
    plt.grid(True)
    plt.legend()

    #save plot in a png
    plt.savefig('../plots/OneMaxRunTime2.png')
    return
```

4
TASK 3

4.1 Plateau of the Jump_k benchmark function

Let $k \in [1..n]$. Assume that you start the $(1+1)$ EA on the plateau of Jump_k. Prove mathematically the expected number of iterations until the global optimum is created for the first time via standard bit mutation.

Let x be an individual such that $\texttt{OneMax}(x) = n - k$. The probability that x mutates to the global optimum is

$$\mathfrak{p} = p^k(1-p)^{n-k} \tag{12}$$

Thus, the expected number of iterations until the global optimum is created (assuming $p = \frac{1}{n}$) is

$$\frac{1}{\mathfrak{p}} = \frac{1}{p^k(1-p)^{n-k}} = \frac{n^n}{(n-1)^{n-k}} \tag{13}$$

5 TASK 4

5.1 Implementation $(\mu + \lambda)$ GA

Implement the $(\mu + \lambda)$ GA such that it can be run on pseudo-boolean functions.

Let's discuss some implementation details of the $(\mu+\lambda)$ genetic algorithm. The structure is similar to that of the $(\mu+1)$ EA, we keep the population in a priority queue of pairs $(f(x), x)$ and at each iteration we check the fittest individual in the priority queue to evaluate the termination condition.

The process of generating the offspring at each iteration is, however, very different. We use a `for` loop to generate λ different individuals in the offspring. At each iteration, a uniform random variable `branch_decider` determines if the new individual will be generated by recombination (with probability p_c) or by the standard EA procedure.

The offspring is stored in an array and, after all λ individuals have been generated, we perform the **heappushpop** operation (which adds and right after removes least fit element in the priority queue) from the python **heapq** module to each of them. At the end, the population has the μ fittest individuals generated so far. This is more efficient than storing population and offspring in an array and performing a sort operation.

```
import random
import heapq
from Individual import Individual
from EA import generateRandomOffspring

def GeneticAlgorithm(f, n, k, mu, lamb, pc):
    """
    (mu + lambda) Genetic Algorithm with recombination rate pc for Task4
    """
    t = 0
    Pt = []

    for _ in range(mu):# we must create a random initial population of size mu
        individual = generateRandomOffspring(Individual(n), 0.5)
        pair_fitness_individual = (f(individual, k), individual) #we create pair (fitness, individual)
        Pt.append(pair_fitness_individual)

    heapq.heapify(Pt) # we transform population into a priority queue

    while True:

```

```
        most_fit_individual = max(Pt)
        if(most_fit_individual[1].count() == n):
            return most_fit_individual[1]

        offspring = []

        for _ in range(lamb): #in each iteration, we will generate an individual in the offspring
            branch_decider = random.uniform(0, 1)

            if(branch_decider < pc): #we perform recombination with probability pc
                individual1 = random.choice(Pt)[1]
                individual2 = random.choice(Pt)[1]
                new_individual = Individual(n)
                for i in range(n):
                    #for each bit, we chose uniformly at random between bit value in individual1 and individual2
                    bit_value = random.choice([individual1.get(i), individual2.get(i)])
                    if(bit_value == 1): #if chosen value is 1, we set bit in offspring
                        new_individual.set(i)
            else: #else we do normal EA iteration
                offspring.append(generateRandomOffspring(random.choice(Pt)[1], 1 / n))

        while(len(offspring) > 0):
            candidate_individual = offspring.pop(0)
            #we add candidate_individual to the priority queue and pop least fit individual
            heapq.heappushpop(Pt, (f(candidate_individual, k), candidate_individual))

        t += 1
```

6
TASK 5

For at least three values of n (at least 100, preferably far larger), of k (do not go larger than 6 or 7 here, but larger than 1), of μ (at least $\lfloor \ln(n) \rfloor$), of λ (containing the value 1), and of p_c (containing the value 1), measure the diversity of the $(\mu + \lambda)$ GA on `Jump_k`, initialized such that the initial population $P^{(0)}$ only contains individuals on the plateau, chosen uniformly at random. Stop the algorithm once the diversity for the maximum distance is sufficiently high (or after a maximum number of iterations).
For each parameter combination, pick one of the runs and visualize the diversity of the population over time. Please also mark the iteration when the optimum was found for the first time.
Please do not forget to briefly discuss your parameter choices, especially how many times you ran each setup. Furthermore, state whether plots for identical setups (of which you only show one in the report) are qualitatively the same. What do you see? Is there a correlation between some of the parameters and the number of iterations required in order to get to a certain level of diversity?

6.1 Modifications to $(\mu + \lambda)$ GA

Firstly, let's discuss the necessary modifications to the $(\mu + \lambda)$ GA that are described in the problem statement. Instead of starting with a population picked uniformly at random over $\{0,1\}^n$, we pick individuals in the plateau $\{x \text{ such that } \texttt{OneMax}(x) = n - k\}$. Then, we modify the termination condition to stop the algorithm once a fourth of pairs $(x, y) \in P^t \times P^t$ is $\geq 2k$ (or after a maximum number of iterations).

We create a function to generate individuals on the plateau uniformly at random. Then we add functions to compute the Hamming distance between two individuals, the diversity in the population for the maximum distance $2k$ and a boolean function for the termination condition.

```python
import random
import heapq
import numpy as np
from itertools import combinations
from Individual import Individual
from EA import generateRandomOffspring

def GenerateOnPlateau(n, k):
    """
    Creates individual with k ones. Positions are chosen uniformly at random
    """
    random_permutation = np.random.permutation(range(1, n)) #we pick first k
    numbers in a random permutation of [1..n]
```

```
    individual = Individual(n)
    for i in range(k):
        bit_idx = random_permutation[i]
        individual.set(bit_idx)
    return individual

def HammingDistance(individual1, individual2):
    n = individual1.size
    dist = 0
    for i in range(n):
        bit1 = individual1.get(i)
        bit2 = individual2.get(i)
        dist += (bit1 + bit2)%2
    return dist

def Diversity(Pt, k):
    diversity = 0
    #we compute hamming distance between all pairs of individuals in the population
    for individual1, individual2 in combinations(Pt, 2):
        dist = HammingDistance(individual1[1], individual2[1])
        if(dist == 2*k):
            diversity += 1 #add to diversity pairs with maximum distance
    return diversity

def GeneticAlgorithm2(f, n, k, mu, lamb, pc, max_iter = 100):
    """
    (mu + lambda) Genetic Algorithm with recombination rate pc for Task5
    """
    t = 0
    Pt = []
    diversities = []
    found_optimum = -1

    for _ in range(mu):# we must create a random initial population of size mu
        individual = GenerateOnPlateau(n, k)
        pair_fitness_individual = (f(individual, k), individual) #we create pair (fitness, individual)
        Pt.append(pair_fitness_individual)

    heapq.heapify(Pt) # we transform population into a priority queue

    while True:

        if(found_optimum == -1):#let's check if optimum has been found
            most_fit_individual = max(Pt)
            if(most_fit_individual[1].count() == n):
                found_optimum = t #we store first iteration where maximum has been found

        diversity = Diversity(Pt, k) #we compute diversity of population
```

```
        diversities.append(diversity)
        total_pairs = (n*(n-1))/2
        if diversity >= total_pairs/4: #if diversity >= a fourth of total pairs, we break
            return diversities, found_optimum

        if(t >= max_iter): #if we reach max iterations, we also break
            return diversities, found_optimum

        offspring = []

        for _ in range(lamb): #in each iteration, we will generate an individual in the offspring
            branch_decider = random.uniform(0, 1)

            if(branch_decider < pc): #we perform recombination with probability pc
                individual1 = random.choice(Pt)[1]
                individual2 = random.choice(Pt)[1]
                new_individual = Individual(n)
                for i in range(n):
                    #for each bit, we chose uniformly at random between bit value in individual1 and individual2
                    bit_value = random.choice([individual1.get(i), individual2.get(i)])
                    if(bit_value == 1): #if chosen value is 1, we set bit in offspring
                        new_individual.set(i)
            else: #else we do normal EA iteration
                offspring.append(generateRandomOffspring(random.choice(Pt)[1], 1 / n))

        while(len(offspring) > 0):
            candidate_individual = offspring.pop(0)
            #we add candidate_individual to the priority queue and pop least fit individual
            heapq.heappushpop(Pt, (f(candidate_individual, k), candidate_individual))

        t += 1
```

6.2 Empirical tests for the $(\mu + \lambda)$ GA

We used the following parameter values for testing:

```
 test_values = [(300, 4, 5, 1, 0), #base values
           (300, 5, 5, 1, 0), (300, 6, 5, 1, 0), #testing influence of k
           (300, 4, 10, 1, 0), (300, 4, 15, 1, 0), #testing influence of mu
           (300, 4, 5, 5, 0), (300, 4, 5, 10, 0), #testing influence of lambda
```

```
                (300, 4, 5, 1, 0.5), (300, 4, 5, 1, 1)] #testing influence of pc
```

The Github repository for the project contains five complete series of tests each containing all 9 plots.

In the first 3 sets, the maximum number of iterations set for testing was max_iter = 10000 and in the last 3 max_iter = 100 was used to allow for better visualisation.

In this report, we show only series 4. Please check the project repository for the full set of tests.

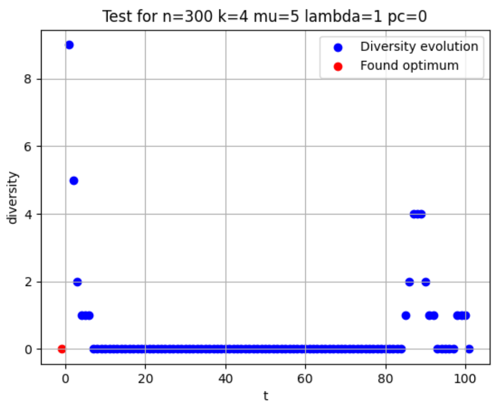

Figure 5: Test for $n = 300$, $k = 4$, $\mu = 5$, $\lambda = 1$, $p_c = 0$

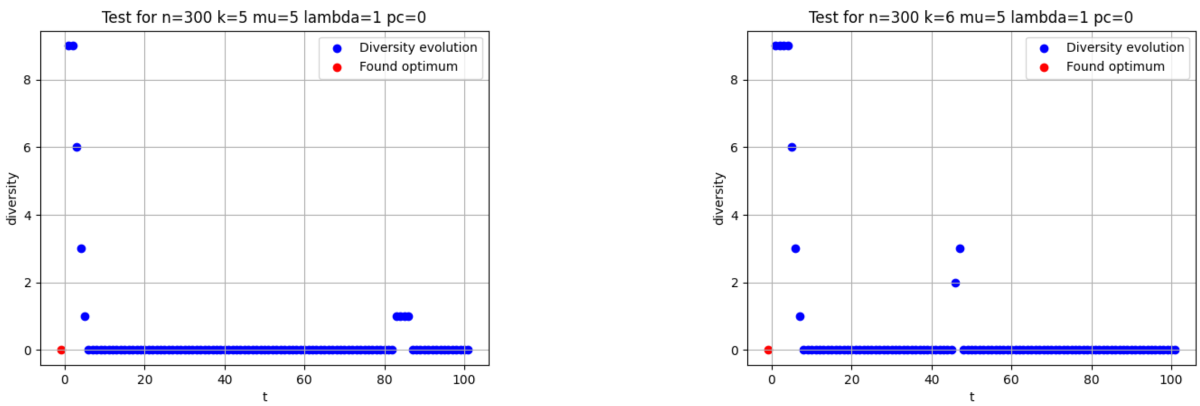

Figure 6: Varying value of k

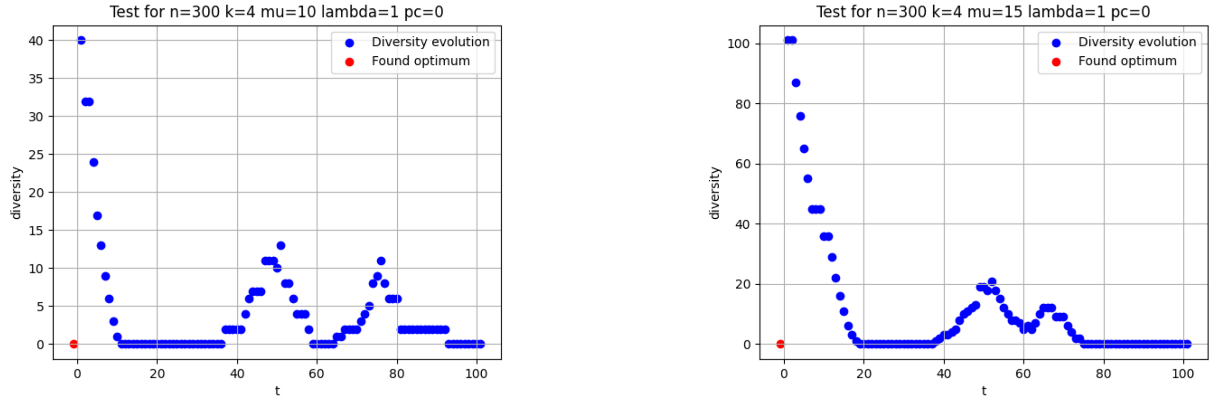

Figure 7: Varying value of μ

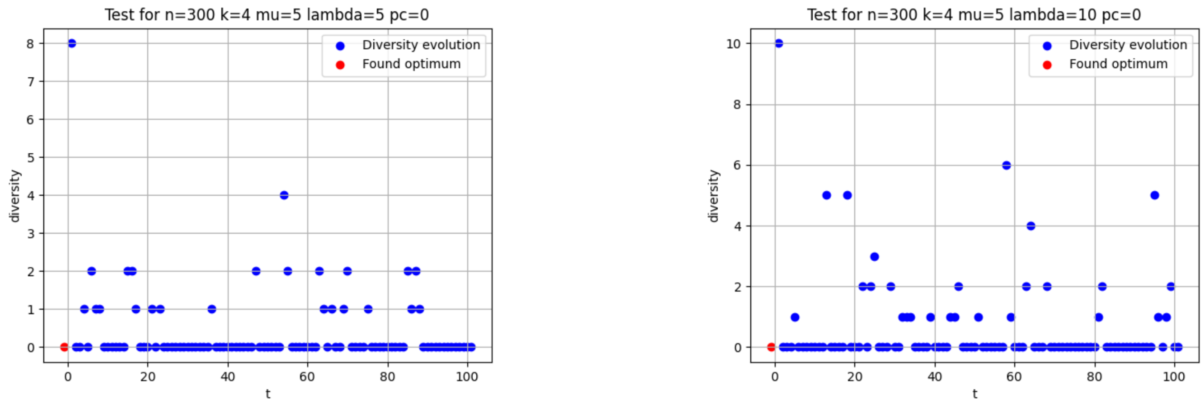

Figure 8: Varying value of *lambda*

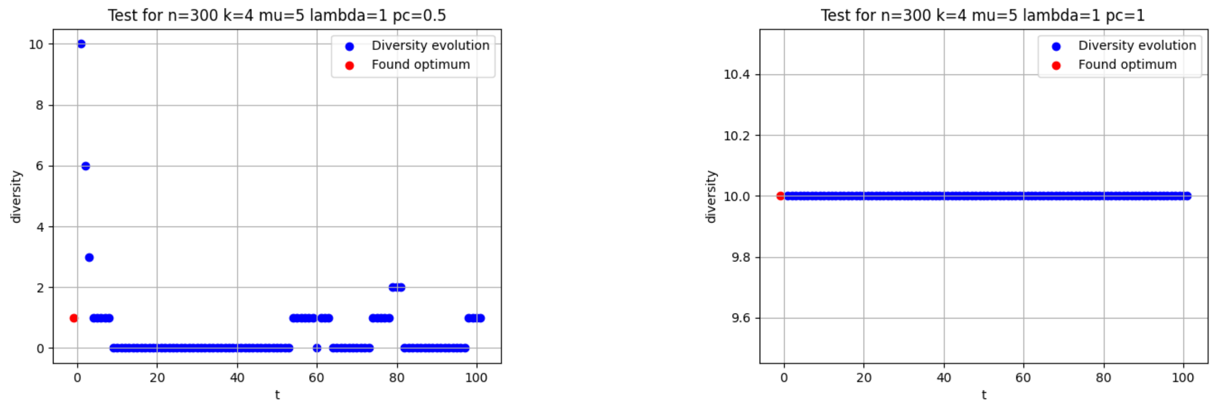

Figure 9: Varying value of p_c

We conclude that ...

6.3 Tests with identical setups

In this section, let's compare plots with identical parameters in the three series presented in the Github repository for the project.

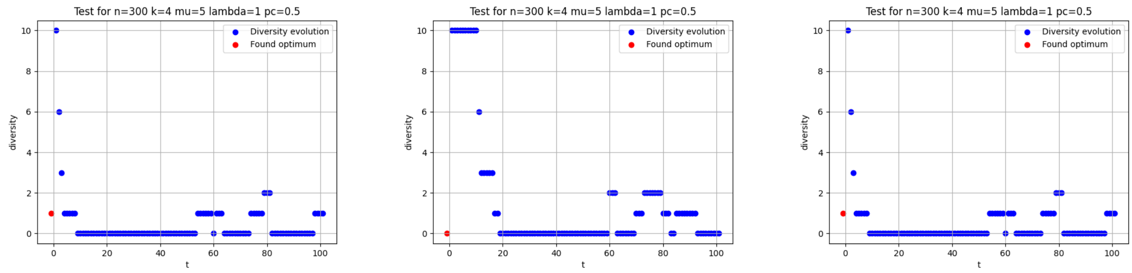

Figure 10: Tests for $n = 300$, $k = 4$, $\mu = 5$, $\lambda = 1$, $p_c = 0.5$

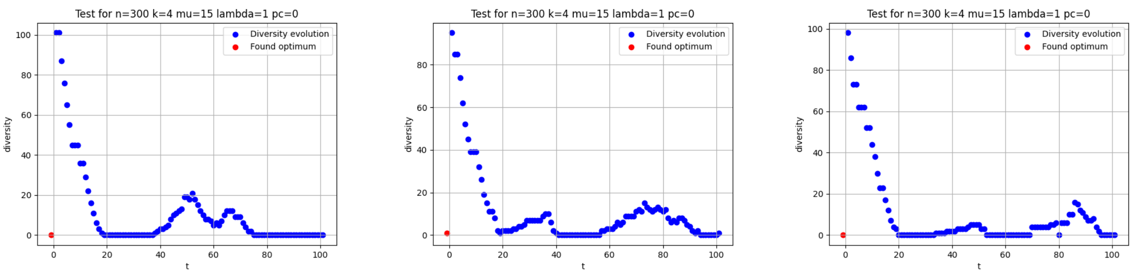

Figure 11: Tests for $n = 300$, $k = 4$, $\mu = 15$, $\lambda = 1$, $p_c = 0.5$

We conclude that ...

6.4 Code used for tests on the $(\mu + \lambda)$ GA

```
import time
import numpy as np
from statistics import mean
import matplotlib.pyplot as plt
import Individual
import GA_task5
import BenchmarkFunctions

def PlotGeneticAlgorithmDiversities():
    """
    Generates scatter plot for empirical run time analysis using (mu +
   lambda) GA and the Jumpk benchmark function
```

```
    """
    #we set parameter values for our tests
    test_values = [(300, 4, 5, 1, 0), #base values
            (300, 5, 5, 1, 0), (300, 6, 5, 1, 0), #testing influence of k
            (300, 4, 10, 1, 0), (300, 4, 15, 1, 0), #testing influence of mu
            (300, 4, 5, 5, 0), (300, 4, 5, 10, 0), #testing influence of
   lambda
            (300, 4, 5, 1, 0.5), (300, 4, 5, 1, 1)] #testing influence of pc

    plot_number=1

    for n, k, mu, lamb, pc in test_values:

        diversities, found_optimum = GA_task5.GeneticAlgorithm2(
   BenchmarkFunctions.JumpK, n, k, mu, lamb, pc)

        # we make a scatter plot
        plt.figure()
        plt.scatter(range(1, len(diversities)+1), diversities, color='blue',
    marker='o', label='Diversity evolution')
        plt.scatter(range(found_optimum, found_optimum+1), diversities[
   found_optimum], color='red', marker='o', label='Found optimum')
        plt.xlabel('t')
        plt.ylabel('diversity')
        plt.title(f'Test for n={n} k={k} mu={mu} lambda={lamb} pc={pc}')
        plt.grid(True)
        plt.legend()

        #save plot in a png
        plt.savefig(f'../plots/GAplots6/GAtest{plot_number}.png')
        plot_number+=1

PlotGeneticAlgorithmDiversities()
```

REFERENCES

[1] Doerr, B., Kötzing, T. Lower Bounds from Fitness Levels Made Easy. Algorithmica (2022). https://doi.org/10.1007/s00453-022-00952-w