INF 421 PROJECT

Evolutionary Algorithms
Professeur Martin Krejca

February 3, 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

```python
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
        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(1, 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 $p = \frac{1}{n}$.

```python
from Individual import Individual
import numpy as np

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)
```

```python
        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 f(Pt) < 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 $f(P^t) \geq \ f(P^{t-1}) \quad \forall t \geq 1$. 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. Plotting the run time estimates for different values of n in figure 1, we conclude that the run time complexity is $O(n\log(n))$.

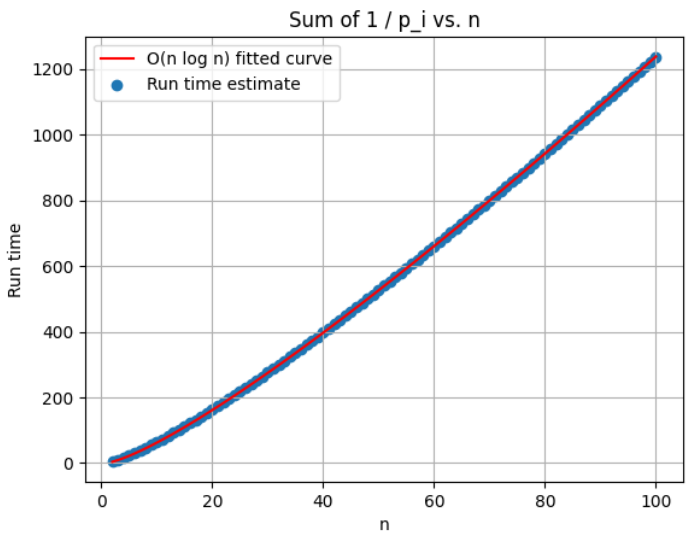

Figure 1: $O(n\log(n))$ fit for theoretical runtime curve

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.

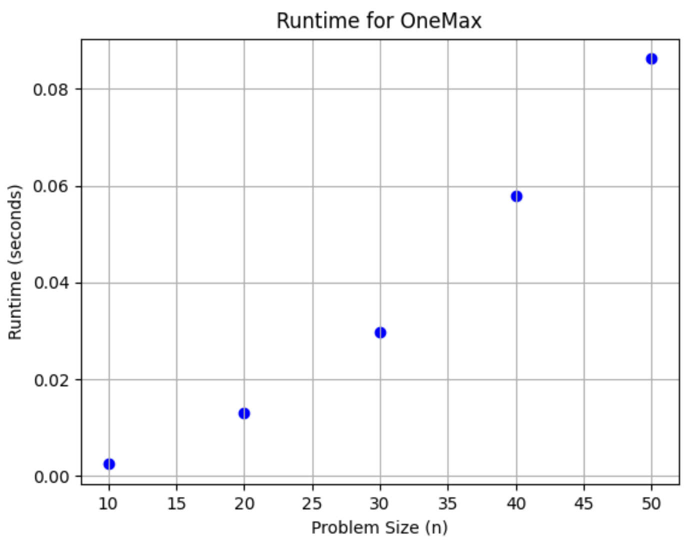

Figure 2: Empiric runtime analysis for OneMax

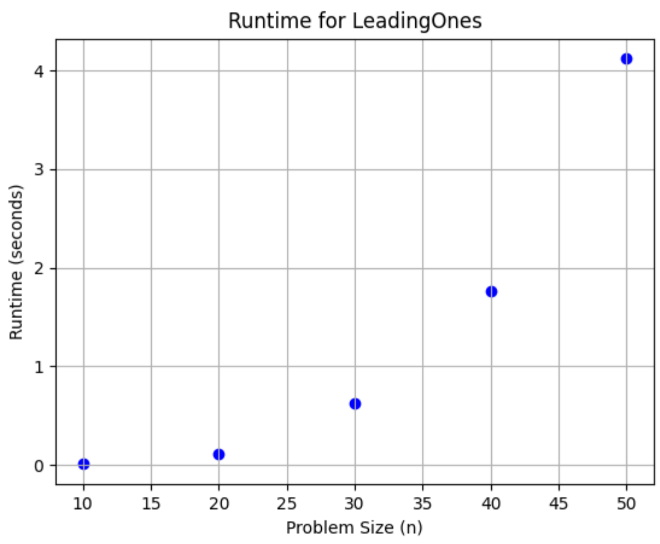

Figure 3: Empiric runtime analysis for LeadingOnes

```python
import Individual
import BenchmarkFunctions
import EA
import time
import matplotlib.pyplot as plt
from statistics import mean

def PlotOneMaxRunTime():
    '''
    Generates scatter plot for empirical run time analysis using (1+1) EA
   and the OneMax benchmark function
    '''
    nvals = [10, 20, 30, 40, 50]
    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))

    # Plotting the histogram
    plt.scatter(nvals, run_times, color='blue', marker='o')
    plt.xlabel('Problem Size (n)')
    plt.ylabel('Runtime (seconds)')
    plt.title('Runtime for OneMax')
    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 = [10, 20, 30, 40, 50]
    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

    # Plotting the histogram
    plt.scatter(nvals, run_times, color='blue', marker='o')
    plt.xlabel('Problem Size (n)')
    plt.ylabel('Runtime (seconds)')
    plt.title('Runtime for LeadingOnes')
    plt.grid(True)

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

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.

4
TASK 3

5 TASK 4

6
TASK 5

REFERENCES

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

[2] Wikipedia. Abel's Inequality. https://en.wikipedia.org/wiki/Abel%27s_inequality