

INF 421 PROJECT

Evolutionary Algorithms
Professeur Martin Krejca

February 2, 2024



Gabriel Pereira de Carvalho



CONTENTS

1	Introduction	3
1.1	Instructions for running the project locally	3
2	Task 1: Individuals and Benchmark Functions	4
2.1	Individuals	4
2.2	Benchmark Functions	5
2.3	(1 + 1) EA	6
3	Task 2: Runtime Analysis	8
3.1	Theoretical bound for <code>OneMax</code>	8
3.2	Theoretical bound for <code>LeadingOnes</code>	8
4	Task 3	13
5	Task 4	14
6	Task 5	15

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.

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

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

```
1 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.

```
1 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 responsibility 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, where 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

```

1 class Individual:
2     """
3     Represents candidate solutions x = (x1, ..., xn)
4     """
5     def __init__(self, size):
6         """
7         Constructor for new Individual
8         """
9         # Number of integers necessary to represent all xi's
10        necessary_integers = (size + 31) // 32

```

```

11     self.size = size
12     self.bits = [0] * necessary_integers
13
14     def get(self, idx):
15         """
16         Get bit at index idx
17         """
18         test_bit = self.bits[idx // 32] & (1 << (idx % 32))
19         return 1 if test_bit > 0 else 0
20
21     def set(self, idx):
22         """
23         Set bit at index idx to 1
24         """
25         self.bits[idx // 32] |= (1 << (idx % 32))
26
27     def reset(self, idx):
28         """
29         Set bit at index idx to 0
30         """
31         self.bits[idx // 32] &= ~(1 << (idx % 32))
32
33     def flip(self, idx):
34         """
35         Flip bit at index idx
36         """
37         bit_i = self.get(idx)
38         if bit_i == 0:
39             self.set(idx)
40         else:
41             self.reset(idx)
42
43     def count(self):
44         """
45         Count number of bits equal to 1
46         """
47         result = 0
48         for i in range(self.size):
49             bit_i = self.get(i)
50             result += bit_i
51         return result

```

2.2 BENCHMARK FUNCTIONS

```

1 from Individual import Individual
2
3 def OneMax(individual):
4     """
5     Returns the number of 1s of the input

```

```

6     """
7     return individual.count()
8
9 def LeadingOnes(individual):
10    """
11    Returns the length of the longest consecutive prefix of 1s
12    """
13    n = individual.size
14    result = 0
15    for i in range(n):
16        prefix_product = 1
17        for j in range(1, i + 1):
18            prefix_product *= individual.get(j)
19        result += prefix_product
20    return result
21
22 def JumpK(individual, k):
23    """
24    Analog to OneMax but penalizes individuals with a number of ones in n-k
25    +1,...,n-1
26    """
27    n = individual.size
28    one_max_x = OneMax(individual)
29    if one_max_x <= n - k or one_max_x == n:
30        return k + one_max_x
31    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 $p = \frac{1}{n}$.

```

1 from Individual import Individual
2 import numpy as np
3
4 def generateRandomOffspring(x, p):
5     """
6     Generate a copy of x flipping each bit independently with probability p
7     """
8     n = x.size
9     y = Individual(n)
10    for idx in range(n):
11        xi = x.get(idx)
12        rand_var = np.random.uniform(0, 1)
13        #bit idx in y is 1 if and only if
14        mutated_to_one = (rand_var < p and xi == 0) #mutated from 0 to 1 (
15    with probability p)

```

```
15         stayed_one = (rand_var >= p and xi == 1) #did not mutate, was
already 1 (probability 1-p)
16         if (mutated_to_one or stayed_one):
17             y.set(idx)
18         return y
19
20 def EvolutionaryAlgorithm(f, n):
21     """
22     (1+1) Evolutionary Algorithm
23     """
24     t = 0
25     Pt = generateRandomOffspring(Individual(n), 0.5) # random initial
solution
26
27     while f(Pt) < n:
28         y = generateRandomOffspring(Pt, 1 / n)
29         if f(y) > f(Pt): # we pick solution that maximizes f
30             Pt = y
31         t += 1
32
33     return Pt
```

3

TASK 2: RUNTIME ANALYSIS

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

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

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

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

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 THEORETICAL BOUND FOR **OneMax**

For the **OneMax** function, $\forall i \in [0..n-1]$, in order for a mutation to leave level A_i the number of 1 bits must have increased after the mutation.

Suppose $0 \leq k \leq i$ bits that were already 1 got flipped and that $k \leq j \leq n-i$ bits that were 0 got flipped. The new total of 1 bits is $i - k + j > i$.

$$p_i = \sum_{k=0}^i \left(\sum_{j=k}^{n-i} \left(\frac{1}{n} \right)^{k+j} \left(\frac{n-1}{n} \right)^{n-k-j} \right) \quad (1)$$

3.2 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, \dots, P_i^t unchanged and to flip the bit P_{i+1}^t .

$$p_i = \frac{1}{n} \left(\frac{n-1}{n} \right)^i \quad (2)$$

Now, to estimate the run time

$$\sum_{i=0}^{n-1} \frac{1}{p_i} = \sum_{i=0}^{n-1} n \left(\frac{n}{n-1} \right)^i \quad (3)$$

$$= n \sum_{i=0}^{n-1} \left(1 + \frac{1}{n-1} \right)^i \quad (4)$$

$$\leq n \sum_{i=0}^{n-1} \left(1 + \frac{1}{n-1} \right)^{n-1} \quad (5)$$

$$\leq n \sum_{i=0}^{n-1} e \quad (6)$$

$$= en^2 \quad (7)$$

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

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, \dots, T_k represented run time measurements. We suppose the run times are independent and identically distributed. Then, by the law of large numbers,

$$\frac{T_1 + \dots + T_k}{k} \xrightarrow[k \rightarrow +\infty]{} \mathbb{E}[T_1] \quad (8)$$

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 1 and 2, 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.

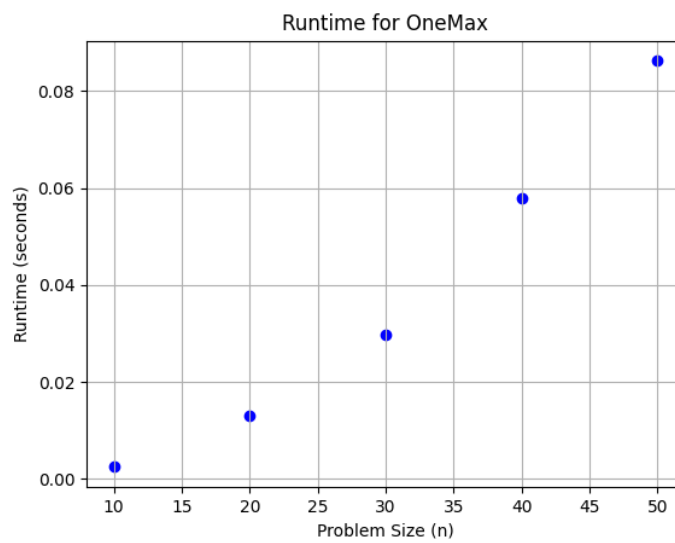


Figure 1: Empiric runtime analysis for OneMax

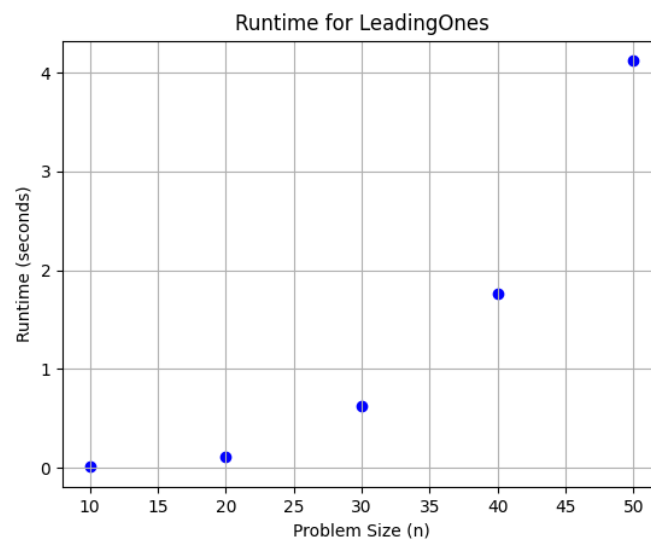


Figure 2: Empiric runtime analysis for LeadingOnes

```

1 import Individual
2 import BenchmarkFunctions
3 import EA
4 import time
5 import matplotlib.pyplot as plt
6 from statistics import mean
7
8 def PlotOneMaxRunTime():

```

```

9      '''
10     Generates scatter plot for empirical run time analysis using (1+1) EA
    and the OneMax benchmark function
11     '''
12     nvals = [10, 20, 30, 40, 50]
13     run_times = []
14
15     for n in nvals:
16
17         number_of_trials = 30
18         current_trial = 1
19         trial_run_times = []
20
21         while(current_trial <= number_of_trials):
22
23             start_time = time.process_time() #we measure start time before
    running the EA
24             solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.OneMax, n
    )
25             end_time = time.process_time() #we measure end time after
    running the EA
26
27             trial_run_times.append(end_time - start_time)
28             current_trial += 1
29
30             run_times.append(mean(trial_run_times))
31
32     # Plotting the histogram
33     plt.scatter(nvals, run_times, color='blue', marker='o')
34     plt.xlabel('Problem Size (n)')
35     plt.ylabel('Runtime (seconds)')
36     plt.title('Runtime for OneMax')
37     plt.grid(True)
38
39     #save plot in a png
40     plt.savefig('../plots/OneMaxRunTime.png')
41     return
42
43 def PlotLeadingOnesRunTime():
44     '''
45     Generates scatter plot for empirical run time analysis using (1+1) EA
    and the LeadingOnes benchmark function
46     '''
47     nvals = [10, 20, 30, 40, 50]
48     run_times = []
49
50     for n in nvals:
51
52         number_of_trials = 30
53         current_trial = 1
54         trial_run_times = []
55

```

```

56         while(current_trial < number_of_trials):
57
58             start_time = time.process_time() #we measure start time before
running the EA
59             solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.
LeadingOnes, n)
60             end_time = time.process_time() #we measure end time after
running the EA
61
62             trial_run_times.append(end_time - start_time)#we use average of
measurements as theestimator
63             current_trial += 1
64
65             run_times.append(mean(trial_run_times))#we use average of
measurements as theestimator
66
67             # Plotting the histogram
68             plt.scatter(nvals, run_times, color='blue', marker='o')
69             plt.xlabel('Problem Size (n)')
70             plt.ylabel('Runtime (seconds)')
71             plt.title('Runtime for LeadingOnes')
72             plt.grid(True)
73
74             #save plot in a png
75             plt.savefig('../plots/LeadingOnesRunTime.png')
76             return
77
78 #PlotOneMaxRunTime()
79 PlotLeadingOnesRunTime()

```

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