



INF 421 PROJECT

Evolutionary Algorithms
Professor Martin Krejca

February 5, 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 run time upper bounds	8
3.1.1	Theoretical bound for OneMax	8
3.1.2	Theoretical bound for LeadingOnes	11
3.2	Empirical run time estimates	11
3.3	Empirical run time tests for the $(\mu + 1)$ EA	13
3.4	Code for empirical run time plots	16
4	Task 3	20
4.1	Plateau of the Jump_k benchmark function	20
5	Task 4	21
5.1	Implementation $(\mu + \lambda)$ GA	21
6	Task 5	23
6.1	Modifications to $(\mu + \lambda)$ GA	23
6.2	Empirical tests for the $(\mu + \lambda)$ GA	25
6.3	Tests with identical setups	28
6.4	Code used for tests on the $(\mu + \lambda)$ GA	28

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

```

```
15         mutated_to_one = (rand_var < p and xi == 0) #mutated from 0 to 1 (
with probability p)
16         stayed_one = (rand_var >= p and xi == 1) #did not mutate, was
already 1 (probability 1-p)
17         if (mutated_to_one or stayed_one):
18             y.set(idx)
19         return y
20
21 def EvolutionaryAlgorithm(f, n):
22     """
23     (1+1) Evolutionary Algorithm
24     """
25     t = 0
26     Pt = generateRandomOffspring(Individual(n), 0.5) # random initial
solution
27
28     while Pt.count() < n:
29         y = generateRandomOffspring(Pt, 1 / n)
30         if f(y) > f(Pt): # we pick solution that maximizes f
31             Pt = y
32         t += 1
33
34     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

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

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.

3.1.1 • THEORETICAL BOUND FOR **ONEMAX**

Let $\mathcal{P}(m, i, j)$ denote the probability 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

$$1 \leq m \leq n; \quad 0 \leq i < m; \quad i < j \leq m. \quad (1)$$

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

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

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

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

To calculate p_i using our coefficients, we have

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

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 $O(n \log(n))$.

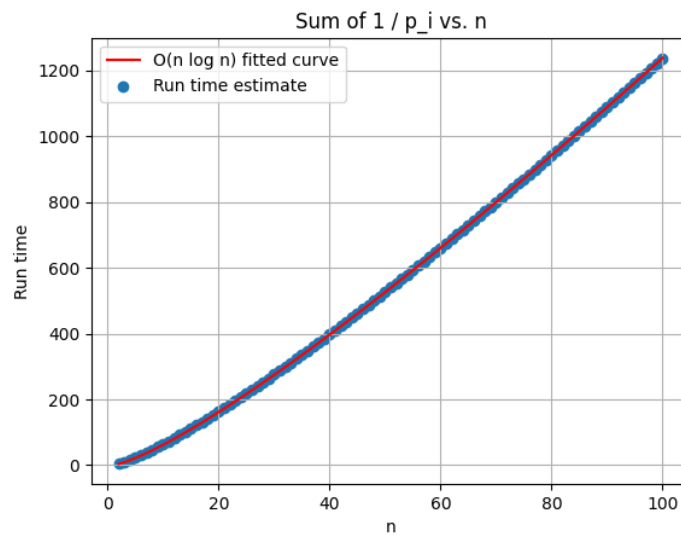


Figure 1: $O(n \log(n))$ fit for theoretical runtime curve

```

1 import numpy as np
2 from math import comb
3 import matplotlib.pyplot as plt
4 from scipy.optimize import curve_fit
5
6 def calculate_coefficients(n, p):
7     P = [[[0.0] * (n + 1) for _ in range(n + 1)] for _ in range(n + 1)]
8
9     # Base case: m
10    for m in range(1, n + 1):
11        for j in range(m + 1):
12            denominator = sum(comb(m, k) * (p ** k) * ((1 - p) ** (m - k))
13    for k in range(m + 1))
14    P[m][0][j] = comb(m, j) * (p ** j) * ((1 - p) ** (m - j)) /
15    denominator
16
17    # Recurrence relation

```

```

16     for m in range(2, n + 1):
17         for i in range(1, m + 1):
18             for j in range(i, m + 1):
19                 P[m][i][j] = p * P[m - 1][i - 1][j] + (1 - p) * P[m - 1][i -
20                     1][j - 1]
21
22     # Calculate p_i
23     p_i = [0.0] * (n + 1)
24     for i in range(n):
25         for j in range(i + 1, n + 1):
26             p_i[i] += P[n][i][j]
27
28     return p_i
29
30 # Set up values for n
31 n_values = list(range(2, 101)) # Change the range as needed
32
33 # Calculate the sum for each n
34 sum_values = []
35 for n in n_values:
36     coefficients = calculate_coefficients(n, 1 / n)
37     sum_val = sum(1 / p if p != 0 else 0 for p in coefficients) # Include i
38     = 0
39     sum_values.append(sum_val)
40
41 # Plot the sum as a function of n
42 plt.plot(n_values, sum_values)
43 plt.xlabel('n')
44 plt.ylabel('Sum')
45 plt.title('Sum of 1 / p_i vs. n')
46 plt.grid(True)
47 plt.savefig('../plots/OneMaxTheoreticalRunTime.png')
48
49 #function for fitting
50 def fit_function(y, a):
51     return a * y * np.log(y)
52
53 # Perform curve fitting
54 popt, pcov = curve_fit(fit_function, n_values, sum_values, maxfev=10000) #
55     maxfev increased for more iterations
56
57 # Plot the data and fitted curve
58 plt.scatter(n_values, sum_values, label='Run time estimate')
59 plt.plot(n_values, fit_function(np.array(n_values), *popt), 'r-', label='O(n
60     log n) fitted curve')
61 plt.xlabel('n')
62 plt.ylabel('Run time')
63 plt.legend()
64 plt.grid(True)
65 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, \dots, P_i^t unchanged and to flip the bit P_{i+1}^t .

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

Now, to estimate the run time

$$\sum_{i=0}^{n-1} \frac{1}{p_i} = \sum_{i=0}^{n-1} n \left(\frac{n}{n-1} \right)^i \quad (6)$$

$$= n \sum_{i=0}^{n-1} \left(1 + \frac{1}{n-1} \right)^i \quad (7)$$

$$\leq n \sum_{i=0}^{n-1} \left(1 + \frac{1}{n-1} \right)^{n-1} \quad (8)$$

$$\leq n \sum_{i=0}^{n-1} e \quad (9)$$

$$= en^2 \quad (10)$$

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

$$\frac{T_1 + \dots + T_k}{k} \xrightarrow[k \rightarrow +\infty]{} \mathbb{E}[T_1] \quad (11)$$

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.

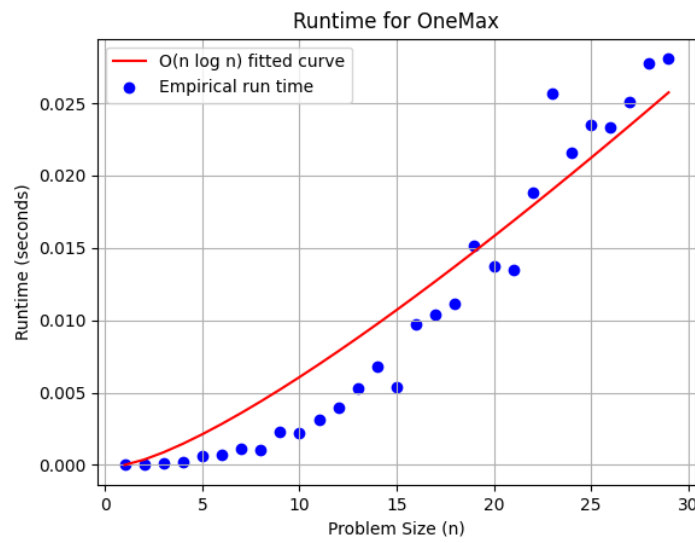


Figure 2: Empiric runtime analysis for OneMax

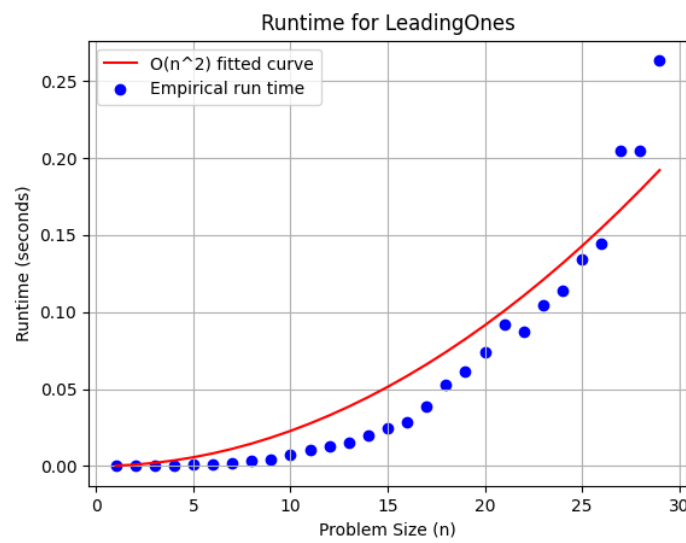


Figure 3: Empiric runtime analysis for LeadingOnes

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 in order to eliminate the least fit individual, which must be done at each iteration in the algorithm description.
- Computing the fitness values of 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 as a min heap 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. And we can remove least fit individual by looking at the top of the min heap.

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.

```
1 def __lt__(self, other):
2     if(self.size < other.size):
3         return True
4     else:
5         return False
```

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

```
1 from Individual import Individual
2 import numpy as np
```

```

3 import random
4 import heapq
5
6 def generateRandomOffspring(x, p):
7     """
8     Generate a copy of x flipping each bit independently with probability p
9     """
10    n = x.size
11    y = Individual(n)
12    for idx in range(n):
13        xi = x.get(idx)
14        rand_var = np.random.uniform(0, 1)
15        #bit idx in y is 1 if and only if
16        mutated_to_one = (rand_var < p and xi == 0) #mutated from 0 to 1 (
with probability p)
17        stayed_one = (rand_var >= p and xi == 1) #did not mutate, was
already 1 (probability 1-p)
18        if (mutated_to_one or stayed_one):
19            y.set(idx)
20    return y
21
22 def EvolutionaryAlgorithm(f, n):
23     """
24     (1+1) Evolutionary Algorithm
25     """
26    t = 0
27    Pt = generateRandomOffspring(Individual(n), 0.5) # random initial
solution
28
29    while Pt.count() < n:
30        y = generateRandomOffspring(Pt, 1 / n)
31        if f(y) > f(Pt): # we pick solution that maximizes f
32            Pt = y
33        t += 1
34
35    return Pt
36
37 def EvolutionaryAlgorithm2(f, n, mu):
38     """
39     (mu + 1) Evolutionary Algorithm
40     """
41    t = 0
42    Pt = []
43    most_fit_individual = (0, Individual(n)) #used to store solution
44
45    for _ in range(mu):# we must create a random initial population of size
mu
46        individual = generateRandomOffspring(Individual(n), 0.5)
47        fitness = f(individual)
48        pair_fitness_individual = (fitness, individual) #we create pair (
fitness, individual)
49

```

```

50     if(fitness > most_fit_individual[0]):
51         most_fit_individual = pair_fitness_individual
52
53     Pt.append(pair_fitness_individual)
54
55     heapq.heapify(Pt) # we transform population into a priority queue
56
57     while True:
58
59         if(most_fit_individual[1].count() == n):
60             return most_fit_individual[1]
61
62         offspring = generateRandomOffspring(random.choice(Pt)[1], 1 / n)
63         fitness = f(offspring)
64         pair_fitness_offspring = (fitness, offspring)
65
66         if(fitness > most_fit_individual[0]):
67             most_fit_individual = pair_fitness_offspring
68
69         heapq.heappushpop(Pt, pair_fitness_offspring) #we add offspring to
70         the priority queue and pop least fit individual
71         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 .

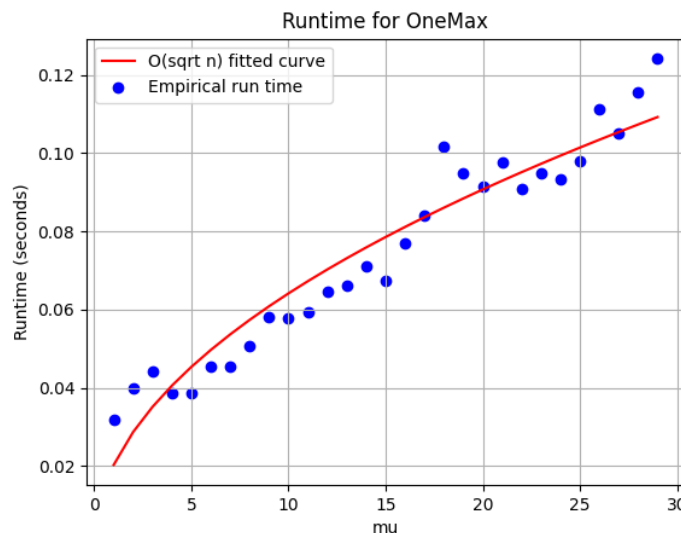


Figure 4: Run time for $\mu \in [1..30]$

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

```

1 import BenchmarkFunctions
2 import EA
3 import time
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from scipy.optimize import curve_fit
7 from statistics import mean
8
9 def PlotOneMaxRunTime():
10     """
11     Generates scatter plot for empirical run time analysis using (1+1) EA
12     and the OneMax benchmark function
13     """
14     nvals = range(1, 30)
15     run_times = []
16
17     for n in nvals:
18         number_of_trials = 30
19         current_trial = 1
20         trial_run_times = []
21
22         while(current_trial <= number_of_trials):
23
24             start_time = time.process_time() #we measure start time before
25             running the EA
26             solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.OneMax, n
27             )
28             end_time = time.process_time() #we measure end time after
29             running the EA
30
31             trial_run_times.append(end_time - start_time)
32             current_trial += 1
33
34             run_times.append(mean(trial_run_times))
35
36     #function for fitting
37     def fit_function(y, a):
38         return a * y * np.log(y) #Theoretical analysis points to n log(n)
39         complexity
40
41     # Perform curve fitting
42     popt, pcov = curve_fit(fit_function, nvals, run_times, maxfev=10000) #
43     maxfev increased for more iterations
44
45     # we make a scatter plot for run times
46     plt.figure()
47     plt.scatter(nvals, run_times, color='blue', marker='o', label='Empirical

```



```

    run_time')
43     plt.plot(nvals, fit_function(np.array(nvals), *popt), 'r-', label='O(n
log n) fitted curve')
44     plt.xlabel('Problem Size (n)')
45     plt.ylabel('Runtime (seconds)')
46     plt.title('Runtime for OneMax')
47     plt.legend()
48     plt.grid(True)
49
50     #save plot in a png
51     plt.savefig('../plots/OneMaxRunTime.png')
52     return
53
54 def PlotLeadingOnesRunTime():
55     """
56     Generates scatter plot for empirical run time analysis using (1+1) EA
and the LeadingOnes benchmark function
57     """
58     nvals = range(1, 30)
59     run_times = []
60
61     for n in nvals:
62
63         number_of_trials = 30
64         current_trial = 1
65         trial_run_times = []
66
67         while(current_trial < number_of_trials):
68
69             start_time = time.process_time() #we measure start time before
running the EA
70             solution = EA.EvolutionaryAlgorithm(BenchmarkFunctions.
LeadingOnes, n)
71             end_time = time.process_time() #we measure end time after
running the EA
72
73             trial_run_times.append(end_time - start_time)#we use average of
measurements as theestimator
74             current_trial += 1
75
76             run_times.append(mean(trial_run_times))#we use average of
measurements as theestimator
77
78     #function for fitting
79     def fit_function(y, a):
80         return a * y * y #Theoretical analysis points to n^2 complexity
81
82     # Perform curve fitting
83     popt, pcov = curve_fit(fit_function, nvals, run_times, maxfev=10000) #
maxfev increased for more iterations
84
85     # we make a scatter plot for run times

```

```

86 plt.figure()
87 plt.scatter(nvals, run_times, color='blue', marker='o', label='Empirical
    run time')
88 plt.plot(nvals, fit_function(np.array(nvals), *popt), 'r-', label='O(n
    ^2) fitted curve')
89 plt.xlabel('Problem Size (n)')
90 plt.ylabel('Runtime (seconds)')
91 plt.title('Runtime for LeadingOnes')
92 plt.legend()
93 plt.grid(True)
94
95 #save plot in a png
96 plt.savefig('../plots/LeadingOnesRunTime.png')
97 return
98
99 def PlotMuPlusOneEAOneMax():
100     """
101     Generates scatter plot for empirical run time analysis using (mu + 1) EA
    and the OneMax benchmark function
102     """
103     n = 30 #we keep n constant and vary only mu
104     mu_vals = range(1, 30)
105     run_times = []
106
107     for mu in mu_vals:
108
109         number_of_trials = 30
110         current_trial = 1
111         trial_run_times = []
112
113         while(current_trial < number_of_trials):
114
115             start_time = time.process_time() #we measure start time before
    running the EA
116             solution = EA.EvolutionaryAlgorithm2(BenchmarkFunctions.OneMax,
    n, mu)
117             end_time = time.process_time() #we measure end time after
    running the EA
118
119             trial_run_times.append(end_time - start_time)#we use average of
    measurements as theestimator
120             current_trial += 1
121
122             run_times.append(mean(trial_run_times))#we use average of
    measurements as theestimator
123
124     #function for fitting
125     def fit_function(y, a):
126         return a * np.sqrt(y) #Theoretical analysis points to n log(n)
    complexity
127
128     # Perform curve fitting

```

```
129     popt, pcov = curve_fit(fit_function, mu_vals, run_times, maxfev=10000) #
130     maxfev increased for more iterations
131
132     # we make a scatter plot
133     plt.figure()
134     plt.scatter(mu_vals, run_times, color='blue', marker='o', label='
Empirical run time')
135     plt.plot(mu_vals, fit_function(np.array(mu_vals), *popt), 'r-', label='0
(sqrt n) fitted curve')
136     plt.xlabel('mu')
137     plt.ylabel('Runtime (seconds)')
138     plt.title('Runtime for OneMax')
139     plt.grid(True)
140     plt.legend()
141
142     #save plot in a png
143     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 $\mathbf{OneMax}(x) = n - k$. The probability that x mutates to the global optimum is

$$\mathbf{p} = p^k (1 - p)^{n-k} \quad (12)$$

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

$$\frac{1}{\mathbf{p}} = \frac{1}{p^k (1 - p)^{n-k}} = \frac{n^n}{(n - 1)^{n-k}} \quad (13)$$

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.

```

1 import random
2 import heapq
3 from Individual import Individual
4 from EA import generateRandomOffspring
5
6 def GeneticAlgorithm(f, n, k, mu, lamb, pc):
7     """
8     (mu + lambda) Genetic Algorithm with recombination rate pc for Task4
9     """
10    t = 0
11    Pt = []
12    most_fit_individual = (0, Individual(n)) #used to store solution
13
14    for _ in range(mu):# we must create a random initial population of size
mu
15        individual = generateRandomOffspring(Individual(n), 0.5)
16        fitness = f(individual, k)
17        pair_fitness_individual = (fitness, individual) #we create pair (
fitness, individual)
18        if(fitness > most_fit_individual[0]):
19            most_fit_individual = pair_fitness_individual
20    Pt.append(pair_fitness_individual)
21

```

```

22     heapq.heapify(Pt) # we transform population into a priority queue
23
24     while True:
25
26         if (most_fit_individual[1].count() == n):
27             return most_fit_individual
28
29         offspring = []
30
31         for _ in range(lamb): #in each iteration, we will generate an
individual in the offspring
32             branch_decider = random.uniform(0, 1)
33
34             if (branch_decider < pc): #we perform recombination with
probability pc
35                 individual1 = random.choice(Pt)[1]
36                 individual2 = random.choice(Pt)[1]
37                 new_individual = Individual(n)
38                 for i in range(n):
39                     #for each bit, we chose uniformly at random between bit
value in individual1 and individual2
40                     bit_value = random.choice([individual1.get(i),
individual2.get(i)])
41                     if (bit_value == 1): #if chosen value is 1, we set bit in
offspring
42                         new_individual.set(i)
43                 else: #else we do normal EA iteration
44                     offspring.append(generateRandomOffspring(random.choice(Pt)
[1], 1 / n))
45
46                 while (len(offspring) > 0):
47                     candidate_individual = offspring.pop(0)
48                     fitness = f(candidate_individual, k)
49                     candidate_pair = (fitness, candidate_individual)
50
51                     if (fitness > most_fit_individual[0]):
52                         #we update most fit individual if necessary
53                         most_fit_individual = candidate_pair
54
55                     #we add candidate_individual to the priority queue and pop least
fit individual
56                     heapq.heappushpop(Pt, candidate_pair)
57
58     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 $\{x \text{ such that } \text{OneMax}(x) = n - k\}$. 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.

```

1 import random
2 import heapq
3 import numpy as np
4 from itertools import combinations
5 from Individual import Individual
6 from EA import generateRandomOffspring
7
8 def GenerateOnPlateau(n, k):
9     """
10     Creates individual with k ones. Positions are chosen uniformly at random
11     """
12     random_permutation = np.random.permutation(range(0, n)) #we pick first n
    -k numbers in a random permutation of [1..n]
```

```

13     individual = Individual(n)
14     for i in range(n-k):
15         bit_idx = random_permutation[i]
16         individual.set(bit_idx)
17     return individual
18
19 def HammingDistance(individual1, individual2):
20     n = individual1.size
21     dist = 0
22     for i in range(n):
23         bit1 = individual1.get(i)
24         bit2 = individual2.get(i)
25         dist += (bit1 + bit2)%2
26     return dist
27
28 def Diversity(Pt, k):
29     diversity = 0
30     #we compute hamming distance between all pairs of individuals in the
    population
31     for individual1 in Pt:
32         for individual2 in Pt:
33             dist = HammingDistance(individual1[1], individual2[1])
34             if(dist == 2*k):
35                 diversity += 1 #add to diversity pairs with maximum distance
36     return diversity/2 #we divide by 2 because each pair is counted twice
37
38 def GeneticAlgorithm2(f, n, k, mu, lamb, pc, max_iter = 50):
39     """
40     (mu + lambda) Genetic Algorithm with recombination rate pc for Task5
41     """
42     t = 0
43     Pt = []
44     diversities = []
45     found_optimum = -1 #used to store time we find optimum
46
47     for _ in range(mu):# we must create a random initial population of size
    mu
48         individual = GenerateOnPlateau(n, k)
49         fitness = f(individual, k)
50         pair_fitness_individual = (fitness, individual) #we create pair (
    fitness, individual)
51         Pt.append(pair_fitness_individual)
52
53     heapq.heapify(Pt) # we transform population into a priority queue
54
55     while True:
56
57         diversity = Diversity(Pt, k) #we compute diversity of population
58         diversities.append(diversity)
59         total_pairs = (n*(n-1))/2
60         if diversity >= total_pairs/4: #if diversity >= a fourth of total
    pairs, we break

```



```

61         return diversities, found_optimum
62
63     if(t >= max_iter): #if we reach max iterations, we also break
64         return diversities, found_optimum
65
66     t += 1
67     offspring = []
68
69     for _ in range(lamb): #in each iteration, we will generate an
individual in the offspring
70         branch_decider = random.uniform(0, 1)
71
72         if(branch_decider < pc): #we perform recombination with
probability pc
73             individual1 = random.choice(Pt)[1]
74             individual2 = random.choice(Pt)[1]
75             new_individual = Individual(n)
76             for i in range(n):
77                 #for each bit, we chose uniformly at random between bit
value in individual1 and individual2
78                 bit_value = random.choice([individual1.get(i),
individual2.get(i)])
79                 if(bit_value == 1): #if chosen value is 1, we set bit in
offspring
80                     new_individual.set(i)
81                     offspring.append(new_individual)
82             else: #else we do normal EA iteration
83                 offspring.append(generateRandomOffspring(random.choice(Pt)
[1], 1 / n))
84
85         while(len(offspring) > 0):
86             candidate_individual = offspring.pop(0)
87             fitness = f(candidate_individual, k)
88
89             if(candidate_individual.count() == n and found_optimum == -1):
90                 #optimum must come from the offspring at some point
91                 found_optimum = t
92
93             #we add candidate_individual to the priority queue and pop least
fit individual
94             heapq.heappushpop(Pt, (fitness, candidate_individual))

```

6.2 EMPIRICAL TESTS FOR THE $(\mu + \lambda)$ GA

We used the following parameter values for testing:

```

1     test_values = [(300, 4, 5, 1, 0), #base values
2                   (300, 5, 5, 1, 0), (300, 6, 5, 1, 0), #testing influence of k
3                   (300, 4, 10, 1, 0), (300, 4, 15, 1, 0), #testing influence of mu

```

```

4      (300, 4, 5, 5, 0), (300, 4, 5, 10, 0), #testing influence of
lambda
5      (300, 4, 5, 1, 0.5), (300, 4, 5, 1, 1)] #testing influence of pc

```

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

The maximum number of iterations set for testing was $\text{max_iter} = 100$ to allow for better visualisation. For certain setups, we observe that the global optimum is not found. The setups used were empirically chosen because they generated the optimum solution rather quickly.

In particular, when the recombination rate p_c was 0, the algorithm never found a solution for $\text{max_iter} = 100$. Which supports the idea that recombination is essential to finding the solution for the `Jump_k` function. The number of iterations required also increases rapidly in function of k . The plots stick to smaller values in order to find solutions for $\text{max_iter} = 100$.

In this report, we show only one series of plots as indicated in the problem statement. Please check the project repository for the full set of tests.

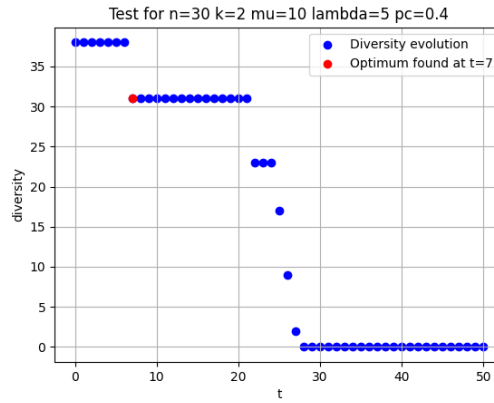


Figure 5: Test for $n = 300$, $k = 4$, $\mu = 5$, $\lambda = 1$, $p_c = 0$

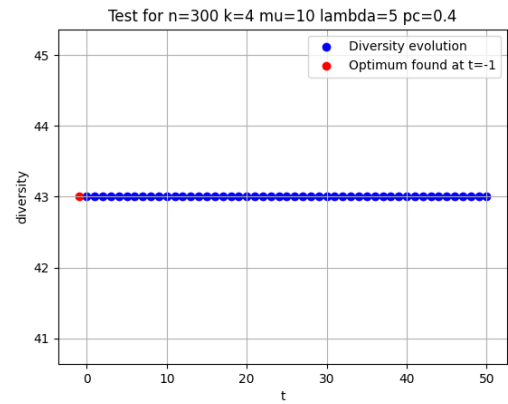
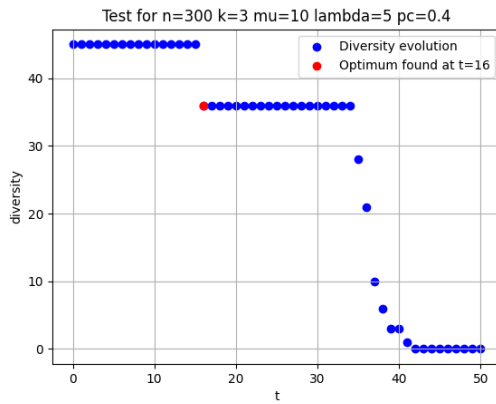
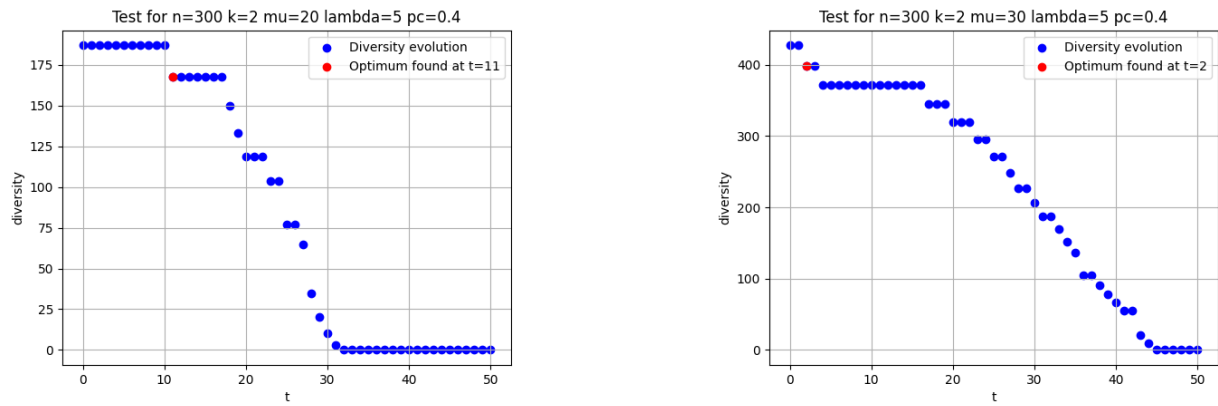
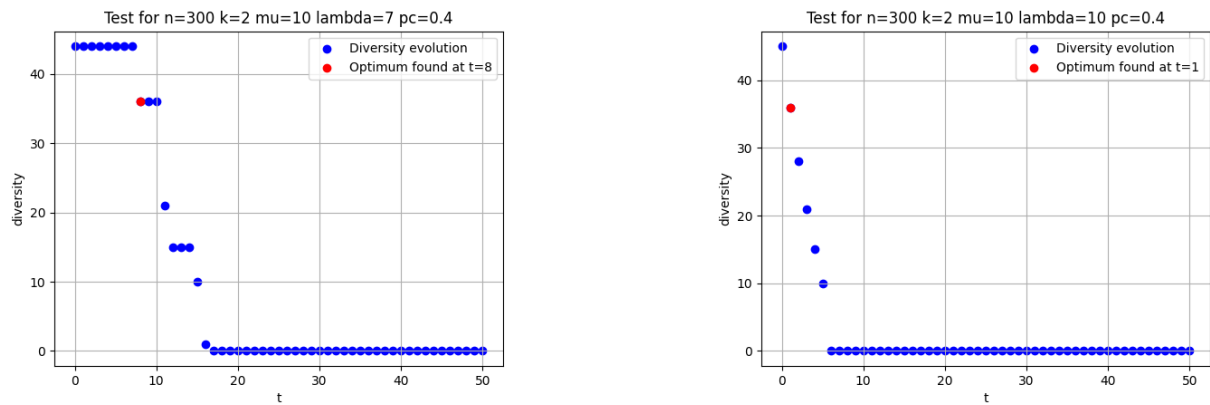
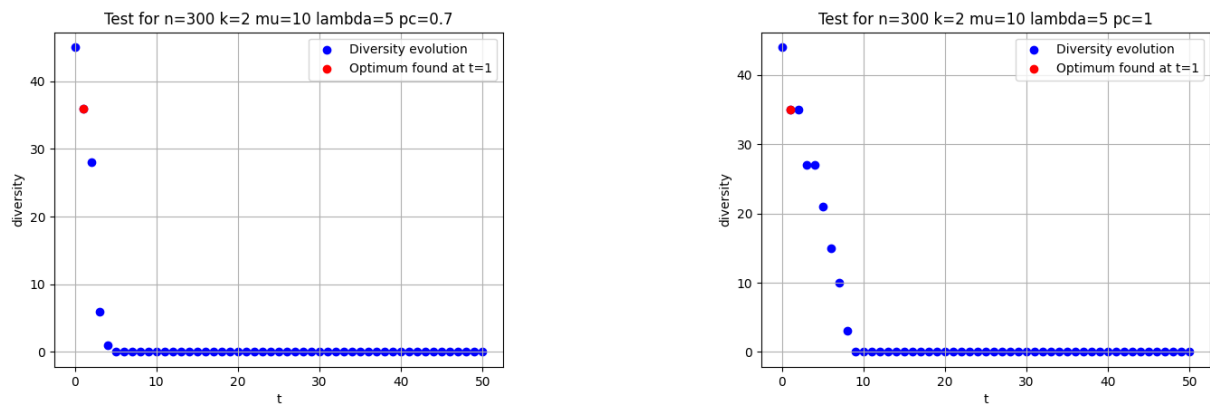


Figure 6: Varying value of k

Figure 7: Varying value of μ Figure 8: Varying value of λ 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.

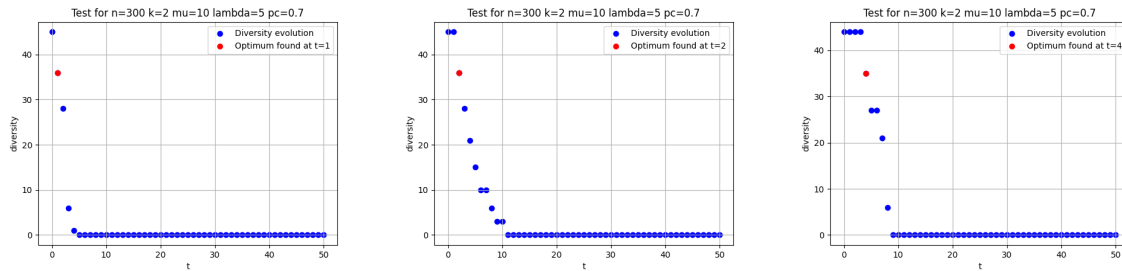


Figure 10: Tests for $n = 300$, $k = 4$, $\mu = 5$, $\lambda = 1$, $p_c = 0.5$

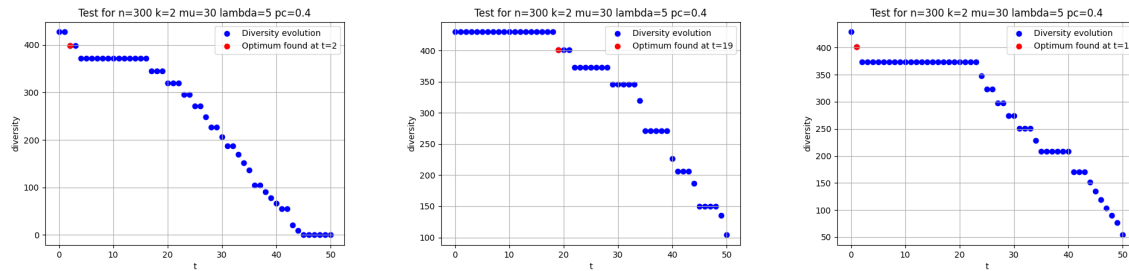


Figure 11: Tests for $n = 300$, $k = 4$, $\mu = 15$, $\lambda = 1$, $p_c = 0.5$

We conclude that ...

6.4 CODE USED FOR TESTS ON THE $(\mu + \lambda)$ GA

```

1 import time
2 import numpy as np
3 from statistics import mean
4 import matplotlib.pyplot as plt
5 import GA_task5
6 import BenchmarkFunctions
7
8 def PlotGeneticAlgorithmDiversities():
9     """
10     Generates scatter plot for empirical run time analysis using (mu +
11     lambda) GA and the Jumpk benchmark function
12     """

```

```

12     #we set parameter values for our tests
13     test_values = [(30, 2, 10, 5, 0.4), #base values n=300, k=4, mu=10,
14                   (300, 3, 10, 5, 0.4), (300, 4, 10, 5, 0.4), #testing influence
of k
15                   (300, 2, 20, 5, 0.4), (300, 2, 30, 5, 0.4), #testing influence
of mu
16                   (300, 2, 10, 7, 0.4), (300, 2, 10, 10, 0.4), #testing influence
of lambda
17                   (300, 2, 10, 5, 0.7), (300, 2, 10, 5, 1)] #testing influence of
pc
18
19     plot_number=1
20
21     for n, k, mu, lamb, pc in test_values:
22
23         diversities, found_optimum = GA_task5.GeneticAlgorithm2(
BenchmarkFunctions.JumpK, n, k, mu, lamb, pc)
24
25         # we make a scatter plot
26         plt.figure()
27         plt.scatter(range(len(diversities)), diversities, color='blue',
marker='o', label='Diversity evolution')
28         plt.scatter(found_optimum, diversities[found_optimum], color='red',
marker='o', label=f'Optimum found at t={found_optimum}')
29         plt.xlabel('t')
30         plt.ylabel('diversity')
31         plt.title(f'Test for n={n} k={k} mu={mu} lambda={lamb} pc={pc}')
32         plt.grid(True)
33         plt.legend()
34
35         #save plot in a png
36         plt.savefig(f'../plots/GAplots3/GAtest{plot_number}.png')
37         plot_number+=1

```

REFERENCES

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