МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Звіт
до лабораторної роботи з дисципліни
«Теорія алгоритмів та структур даних»

Виконав: Пфайфер В.В.
Група: ТР - 35
Прийняв: Андрущак В.С.

Львів 2021

Мета роботи: знайти та оцінити довільний алгоритм на складність.
Назва алгоритму: Genetic Algorithm (посилання на код даного алгоритму)
Опис алгоритму:
Генети́чний алгори́тм (*genetic algorithm*) — алгоритм, який відтворює процес еволюції. Особливістю генетичного алгоритму є акцент на використання оператора «схрещення», який виконує операцію рекомбінацію рішень-кандидатів, роль якої аналогічна ролі схрещення в живій природі.

Вручну, або випадковим чином, в масиві створюється деяка кількість початкових елементів «осіб» (початкова популяція - N_POPULATION). Після цього вибираються особи, допущені до схрещення (*селекція - N_SELECTED*). До осіб застосовується «генетичні оператори» (в більшості випадків це оператор схрещування (crossover) і оператор мутації (mutation, ймовірність мутації - MUTATION_PROBABILITY)), створюючи таким чином наступне покоління осіб. Особи наступного покоління також оцінюються застосуванням генетичних операторів і виконується селекція і мутація. Так моделюється еволюційний процес, що продовжується декілька життєвих циклів (*поколінь*), поки не буде виконано критерій зупинки алгоритму. Таким критерієм є знаходження потрібного набору хромосом.

```python
# Select, Crossover and Mutate a new population
def select(parent_1: tuple[str, float]) -> list[str]:
    """Select the second parent and generate new population"""
    pop = []
    # Generate more child proportionally to the fitness score
    child_n = int(parent_1[1] * 100) + 1
    child_n = 10 if child_n >= 10 else child_n
    for _ in range(child_n):
        parent_2 = population_score[random.randint(0, N_SELECTED)][0]
        child_1, child_2 = crossover(parent_1[0], parent_2)
        # Append new string to the population list
        pop.append(mutate(child_1))
        pop.append(mutate(child_2))
    return pop

def crossover(parent_1: str, parent_2: str) -> tuple[str, str]:
    """Slice and combine two string in a random point"""
    random_slice = random.randint(0, len(parent_1) - 1)
    child_1 = parent_1[:random_slice] + parent_2[random_slice:]
    child_2 = parent_2[:random_slice] + parent_1[random_slice:]
    return (child_1, child_2)

def mutate(child: str) -> str:
    """Mutate a random gene of a child with another one from the list"""
    child_list = list(child)
    if random.uniform(0, 1) < MUTATION_PROBABILITY:
        child_list[random.randint(0, len(child)) - 1] = random.choice(genes)
    return "".join(child_list)
```

Рис.1 Функції вибору, кросоверу, мутації

Робота алгоритму:

1. Підключаємо наступні бібліотеки: time (для визначення часу, затраченого на виконання процесу), os. psutil (для визначення затраченої оперативної пам'яті на виконання процесу), matplotlib (для побудови графіків).

2. Ініціалізація початкової популяції, к-сть елементів з кожної генерації, які підлягають еволюції, ймовірність мутації серед елементів.

3. Перевіряємо, чи к-сть вибраних елементів для еволюції не перевищує загальну чисельність. Перевіряємо ціль на сторонні гени (ті, що не містяться у списку доступних)

4. Генеруємо гени для всієї популяції. Добавляємо цикл перевірки популяції

на ідеальне співпадіння з кінцевою ціллю (функція evaluate). Кожні 10 циклів - вивід загальної чисельності (яка при кожному пробігу збільшується), номер циклу (покоління), бал найкращого співпадіння з ціллю (вибирається елемент з найбільшим збігом генів - виводиться його результат), та його набір генів.

5. В список для наступних еволюцій вносяться елементи з найбільшим балом, решту не беруть участь.

6. Функції вибору (вибір "батька", генерація "дітей"), кросоверу (гени двох батьків "ріжуться" навпіл та об'єднуються. Присвоюються "дітям"), мутації (якщо ймовірність мутації виконується, тоді один із генів "дітей" змінюється на випадковий зі списку доступних).

7. Добавляємо обмеження на максимальний бал співпадіння, щоб процес не тривав вічно. (...)

8. Будуємо графіки складності алгоритму (за часом та використаною пам'яттю)

Вивід:

```
Generation: 420
Total Population:86518
Best score: 77.0
Best string: This is a genetic algorithmsto evaluate, combine, evolve, and mutate a string!

Generation: 430
Total Population: 88578
Target: This is a genetic algorithm to evaluate, combine, evolve, and mutate a string!
Elapsed time: 45.046875 43.65625
Elapsed time during the whole program in seconds: 1.390625
Memory Usage: 88203264
```

Рис.2 Вивід роботи алгоритму

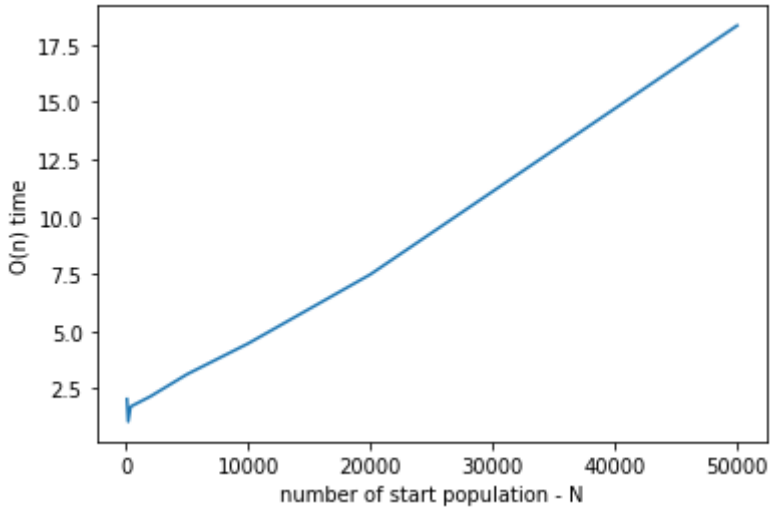

Рис.3 Графік складності алгоритму за витраченим часом

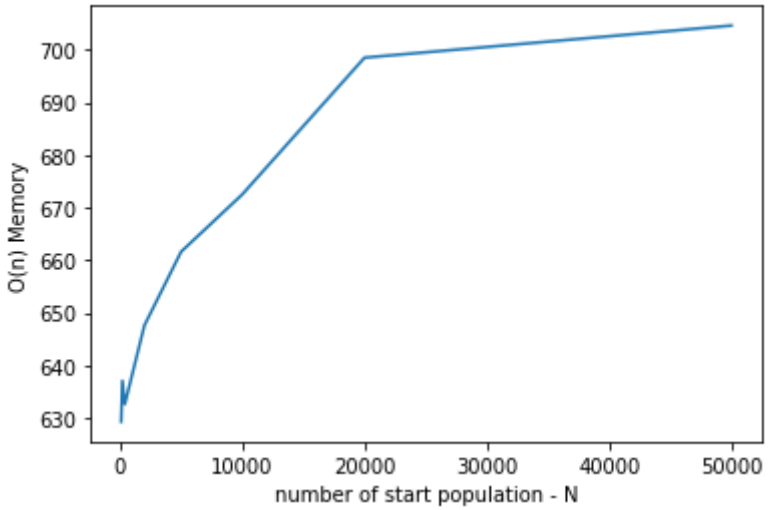

Рис.4 Графік складності алгоритму за використаною пам'яттю

Висновок:

Складність даного алгоритму (за часом), описує функція лінійна функція O(n), проте можливий варіант функції квадратичної O(n^2) при великому значенні популяції, та сталому значенні к-сті елементів вибраних для еволюції.

Складність даного алгоритму (за використаною пам'яттю), описує логарифмічна функція O(log n).

З цього слідує, що алгоритм є не надто ефективний за часом, проте достатньо ефективний за використаною пам'яттю. При оцінюванні чисельності популяції що сягає 1/10000 частину населення (7594000), при вибірці в 100 елементів для еволюції - алгоритм виконується за 24 секунди.

З цього випливає, що при лінійній функції складності даний алгоритм проаналізує населення Землі (при тій самій вибірці елементів) за 69,45

годин (майже за 3 дні). Хоча, к-сть елементів для еволюції також впливає на кінцевий результат, тому можливі відхилення в часі.

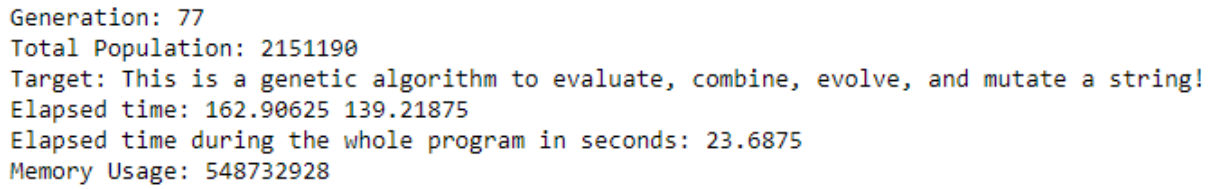

Рис.5 Вивід результату співпадіння при 1/10000 населення Землі