Описание версии (v.0.0.4)

Table of Contents

Структура версии

Файл	Классы
Agent.py	AgentController, Agent
Genome.py	Genome
Core.py	Основной скрипт

Описание Классов

Agent.py

AgentController

Конструктор класса AgentController работает по данному алгоритму:

- При создании передаются атрибуты - **count_of_agent, answer**.
  - **count_of_agent** - количество агентов в популяции.
  - **answer** - конечный ответ, который нужно найти.
- Инициализация переменных и присвоение значений, создание начальной популяции и промежуточной:
  Программная реализация:

```
self.answer = answer
self.ID_of_population = 0
self.max_of_population = count_of_agent
self.main_population = []
self.intermediate_population = []
self.general_fitness = self.calculate_general_fitness()
self.spawn_start_population(len(answer), count_of_agent)
```

Метод	Атрибуты	Описание	В зн
update()	-	Вызывает заданный набор команд каждую итерацию основного цикла	-
spawn_start_population()	genome_length(int) count_of_agent(int)	Создаёт начальную популяцию из count_of_agent особей с геномом длины genome_length	-
check_answer()	-	Проверяет найден ли итоговый ответ	Ес - И

Метод	Атрибуты	Описание	Во зн
calculate_general_fitness()	-	Вычисляет общее значение приспособленности популяции Однако в данный момент не используется(будет исправлено в ближайшее время)	О пр по да по
create_intermediate_population()	-	Создаёт промежуточную популяцию особей используя турнирный отбор **Перед завершением очищает основную популяцию**	-
create_new_population()	-	Создаёт основную популяцию скрещивая случайные агетов из промежуточной популяции **Перед завершением очищает промежуточную популяцию**	-

- **create_intermediate_population()** - создаёт промежуточную популяцию по данному алгоритму:
  - Из основной популяции выбираются агенты в количестве по формуле **max(2, int(0.02 * len(self.main_population)))**.
  - Группа сортируется по приспособленности каждого агента.
  - Выбирается лучшая особь и добавляется в промежуточную популяцию.
  - Шаги повторяются пока не будет набрана популяция необходимого размера.
  - Обнуляется основная популяция.

    Программная реализация:

```
  def create_intermediate_population(self):
      tournament_size = max(2, int(0.02 * len(self.main_population)))
      while (len(self.intermediate_population)) < self.max_of_population:
          tournament = rnd.sample(self.main_population, tournament_size)
          winner = min(tournament, key=lambda agent:
agent.hamming_distance)
          self.intermediate_population.append(winner)
      self.main_population.clear()
```

- **create_new_population()** - создаёт новую основную популяцию из промежуточной по данному алгоритму:
  - Выбираются случайные агенты из промежуточной популяции.
  - Выбранные агенты скрещиваются.
  - Потомок добавляется в основную популяцию.
  - Шаги повторяются пока не будет набрана популяция необходимого размера.
  - Обнуляется промежуточная популяция.

    Программная реализация:

```
  def create_new_population(self):
      while len(self.main_population) < self.max_of_population:
          parent1 = rnd.choice(self.intermediate_population)
          parent2 = rnd.choice(self.intermediate_population)
          child = parent1.reproduce(parent2)
          self.main_population.append(child)
      self.main_population.sort(key=lambda agent: agent.hamming_distance)
```

```
        self.intermediate_population.clear()
```

Agent

Конструктор класса AgentController работает по данному алгоритму:

- При создании передаются атрибуты - **genome_length, answer, parent1, parent2**.
  - **genome_length** - длина генома агента.
  - **answer** - эталонный ответ.
  - **parent1** - первый родитель.
  - **parent2** - второй родитель.
- Инициализация генома и сохранение :

```
    self.genome = Genome.Genome(genome_length, parent1, parent2).genome
```

- Сохранение эталонного ответа, для передачи в следующее поколение(стоит избавится от этого пункта и переработать):

```
    self.answer = answer
```

- Вычисление приспособленности агента:

```
    self.hamming_distance = self.calculate_fitness()
```

Метод	Атрибуты	Описание	Возвращаемое значение
calculate_fitness()	-	Вычисляет значение приспособленности агента	Количество ошибок(расстояние Хэмминга)

Метод	Атрибуты	Описание	Возвращаемое значение
reproduce()	partner(Agent)	Создаёт нового агента на основе генов данной особи и её партнёра	-

- **calculate_fitness()** - вычисляет приспособленость агента.
  Возвращает количество ошибок(различий с эталонным ответом) в виде расстояния Хэмминга. Чем ближе возвращаемое значение к нулю, тем более прсипособлен агент.
  Программная реализация:

```
def calculate_fitness(self):
    errors = 0
    for i in range(len(self.genome)):
        if self.genome[i] != self.answer[i]:
            errors+=1

    return errors
```

- **reproduce()** - создаёт новую особь на основе геномов родителей.
  Программная реализация:

```
def reproduce(self, partner):
      return Agent(len(self.genome), self.answer, self, partner)
```

Genome.py

Genome

Конструктор класса Genome работает по данному алгоритму:

- При создании передаются атрибуты - **length, parent1, parent2**.
  - **length** - необходимая длина генома.
  - **parent1, parent2** - родители нового агента. По умолчанию - None.
- Проверяется есть ли родители у агента.
- Если их нет, либо отсутсвует один из них, то создаётся новая особь.
  Иначе вызывается функция **inheritance()** и в неё передаются геномы родителей.
- По необходимости, полученный в предыдущем шаге, геном мутирует с помощью функции **mutate()**.

Метод	Атрибуты	Описание	Возвращаемое значение
create_genome()	length(int)	Создаёт геном длины length	-
inheritance()	parent1(Agent) parent2(Agent)	Функция создания генома из геномов родителей	child_genome(Genome)
mutate()	genome(Genome) probability(float)	Функция мутации каждого гена с вероятностью probability(по умолчанию - 0.01)	genome(Genome)

- **inheritance()** - создаёт новый геном из геномов родителей. Использует двухточечный кроссинговер.
  Программная реализация:

```
def inheritance(self, parent1, parent2):
    crossover_point1 = rnd.randint(1, len(parent1.genome) - 1)
```

```
        crossover_point2 = rnd.randint(crossover_point1, len(parent1.genome))

        child_genome = (
            parent1.genome[:crossover_point1] +
            parent2.genome[crossover_point1:crossover_point2] +
            parent1.genome[crossover_point2:]
        )
        return child_genome
```

- **mutate()** - проходит по всем аллелям генома и с некоторой вероятностью меняет их.
  По умолчанию вероятность мутации - 0.01(10%).
  Аллели меняются на случайные из константы ALPHABET.
  Программная реализация:

```
def mutate(self, genome, probability=0.01):
  for i in range(len(genome)):
      if (rnd.random() <= probability):
          genome[i] = rnd.choice(alphabet)

  return genome
```

Core.py

Core.py - основной файл запуска алгоритма. В нём описываются константы, создаются контроллеры агентов, вводится необходимый ответ, длина генома, пропивается основной цикл.

Константы	Описание
COUNT_OF_AGENTS	Количество особей в популяции
GENERATION_THRESHOLD	Порог поколений при достижении которого симуляция останавливается

Addition

Вернуться в оглавление

Цели даной версии:

- [x] Реализовать конкретную фитнесс функцию.
- [x] Реализовать турнирный отбор.
- [x] Реализовать ГА для поиска строки.
  - [ ] Оптимизировать этот алгоритм.
- [ ] Реализовать рулеточный отбор.
- [ ] Реализовать влияние общей приспособленности популяции.

Отличия от прошлх версий:

1. Отброшено множество ненужных функций.
2. Удалена визуализация и pygame.
3. Из симуляции жизни алгоритм переработан в приближённый к классическим ГА.
4. Убран интерфейс(только консоль).

Roadmap:

1. Реализовать поиск корней квадратный уравнений.
   - [ ] Рализация данной задачи в v.0.0.5 как основной цели
   - [ ] Обучение алгоритма v.0.0.4 для распознования и решения уравнений
2. Оптимизация алгоритма поиска строки.