Genetische Algorithmen

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Quelle: Davic Kopec, Algorithmen in Python, Rheinwerk Computing

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Folien: https://delors.github.io/theo-algo-genetic_algorithms/folien.de.rst.html

https://delors.github.io/theo-algo-genetic_algorithms/folien.de.rst.html.pdf

Fehler melden:

https://github.com/Delors/delors.github.io/issues

1. Einführung

Genetische Algorithmen

- werden eingesetzt, wenn traditionelle Algorithmen nicht geeignet sind oder keine offensichtlichen Algorithmen existieren

  Z. B. im Bereich des Wirkstoffdesigns oder der Optimierung von Produktionsprozessen.

- benötigen im Prinzip nur eine Definition des Aufgabenziels
- (Implementierungen) sind in der Regel hochgradig spezialisiert; hier diskutieren wir eine allgemeine/generische Implementierung

(Biologische) Grundlagen

- Genetische Algorithmen basieren auf der Evolutionstheorie:

Selektion:	die besten Individuen überleben
Mutation:	zufällige Veränderungen
Rekombination:	Kreuzung von Individuen

- Terminologie:

Population:	Menge von Individuen, die auch Chromosomen genannt werden und Lösungskandidaten repräsentieren
Chromosomen:	Lösungskandidaten oder auch Individuen, die aus Genen bestehen
Gene:	Eigenschaften eines Lösungskandidaten
Fitness(-Funktion):	Bewertung eines Chromosoms (Lösungskandidaten)

D. h. auf der natürlichen Auslese und der Vererbung von Eigenschaften. Ein Individuum, das besser an die Umwelt angepasst ist, hat eine höhere Wahrscheinlichkeit, sich fortzupflanzen und seine Gene weiterzugeben. Schlechter angepasste Individuen sterben aus.

Klassisches Beispiel ist die Entwicklung von Resistenzen gegen Antibiotika bei Bakterien.

Grundidee

- ein genetischer Algorithmus durchläuft Generationen
- In jeder Generation werden die Individuen/Chromosomen bewertet und die besser angepassten Individuen (solche die *fitter* sind) mit einer höheren Wahrscheinlichkeit zum Überleben und zur Fortpflanzung ausgewählt
- in jeder Generation werden die Individuen mit gewissen Wahrscheinlichkeiten:
  1. rekombiniert (`crossover` von zwei Chromosomen) und
  2. mutiert (`mutate`)

Warnung

Genetische Algorithmen sind somit stochastische Algorithmen, d. h. sie liefern nicht immer das gleiche Ergebnis in der gleichen Zeitspanne und es gibt keine Garantie, dass sie das beste Ergebnis finden.

Genetische Algorithmen sind eine gute Wahl, wenn es keinen schnellen deterministischen Algorithmus gibt.

2. Grundlegendes Framework in Python

Basisklasse für Chromosomen

```python
class Chromosome(ABC):
    @abstractmethod
    def fitness(self) -> float:
        raise NotImplementedError

    @classmethod
    @abstractmethod
    def random_instance(cls: Type[T]) -> T:
        raise NotImplementedError

    @abstractmethod
    def crossover(self: T, other: T) -> tuple[T, T]:
        raise NotImplementedError

    @abstractmethod
    def mutate(self) -> None:
        raise NotImplementedError
```


Die Basisklasse für Chromosomen (https://delors.github.io/theo-algo-genetic_algorithms/code/lib/chromosome.py) definiert die Methoden, die jedes Chromosom implementieren muss: Eine Klassenmethode zum Erzeugen einer zufälligen Instanz (`random_instance`), und Instanzmethoden zum Mutieren einer Instanz (`mutate`), zum Kreuzen einer Instanz mit einer anderen (`crossover`) und schließlich zur Bewertung eines Chromosoms (`fitness`).

`mutate`: führt eine kleine zufällige Änderung (an den Genen) durch

`crossover`: kombiniert zwei Chromosomen zu einem neuen Chromosom; d. h. mischt zwei Lösungen

`fitness`: bewertet die eigene Fitness

`random_instance`: erzeugt eine zufällige Instanz; wird nur einmal am Anfang benötigt, um die Population zu initialisieren

Ablauf von genetischen Algorithmen

1. Erzeugen einer zufälligen Population

2. Miss die *Fitness* der Individuen, wenn einer den Zielwert erreicht, beende den Algorithmus und gib das Individuum zurück
3. Wähle einige Individuen aus, die sich fortpflanzen - bevorzuge die Fitteren mit einer höheren Wahrscheinlichkeit
4. Kombiniere die ausgewählten Individuen, um neue Individuen zu erzeugen
5. Mutiere einige Individuen, um die neue Generation zu vervollständigen

6. Wiederhole die Schritte 2-5 für eine bestimmte Anzahl von Generationen

Parameter von genetischen Algorithmen

- Größe der Population
- Design der ersten Population (rein zufällig oder basierend auf einer Heuristik)
- Wahl des Schwellenwertes, der angibt, wann der Algorithmus beendet wird
- Auswahl der Chromosomen, die sich fortpflanzen
- Wie und mit welcher Wahrscheinlichkeit die Rekombination erfolgt (`crossover`)
- Wie und mit welcher Wahrscheinlichkeit eine Mutation erfolgt (`mutate`)
- Wie viele Generationen max. durchlaufen werden

Es gibt zwei typische Strategien zur Auswahl der Chromosomen, die sich fortpflanzen: die *Tournament Selection* und die *Roulette Wheel Selection*.

Selektionsstrategien von Chromosomen

Bestimmung der Chromosomen, die überleben und sich ggf. fortpflanzen.

Auswahlstrategien

***Tournament Selection*:**

Wähle zufällig einige Chromosomen aus und wähle das beste(/die besten) Chromosom(en) aus dieser Gruppe:

```
    def _pick_tournament(
            self, num_participants: int) -> tuple[C, C]:
        participants: list[C] = \
            choices(self._population, k=num_participants)
        return tuple(
            nlargest(
                2,
                participants, key=self._fitness_key))
```

***Roulette Wheel Selection*:**

Jedes Chromosom wird mit einer Wahrscheinlichkeit ausgewählt, die proportional zu seiner Fitness ist (die Fitness wird in eine Wahrscheinlichkeit umgerechnet).

```
    def _pick_roulette(
            self, wheel: list[float]) -> tuple[C, C]:
        c: tuple[C, C] = \
            tuple(choices(
                    self._population,
                    weights=wheel, k=2))
        return c
```

Vorgehen

1. Für jedes gewählte Chromosomenpaar bestimme, ob diese rekombiniert werden sollen. Wenn ja, führe die Rekombination durch. Wenn nicht, kopiere die Chromosomen einfach in die neue Generation.
2. Wiederhole die Selektion und ggf. Rekombination, bis die gewünschte Anzahl von Chromosomen ausgewählt wurde

Basisklasse des genetischen Algorithmus

```python
C = TypeVar("C", bound=Chromosome)  # type of the chromosomes


class GeneticAlgorithm(Generic[C]):
    SelectionType = Enum("SelectionType", "ROULETTE TOURNAMENT")

    def __init__(
        self,
        initial_population: list[C],
        threshold: float,
        max_generations: int = 100,
        mutation_chance: float = 0.01,
        crossover_chance: float = 0.7,
        selection_type: SelectionType = SelectionType.TOURNAMENT,
    )
```

Hauptmethode des genetischen Algorithmus

```
    def run(self) -> C:
        best: C = max(self._population, key=self._fitness_key)
        for generation in range(self._max_generations):
            # early exit if we beat threshold
            if best.fitness() ⩾ self._threshold:
                return best
            print(
                f"Generation {generation} Best {best.fitness()} Avg {
                    mean(map(self._fitness_key, self._population))
                }")
            self._reproduce_and_replace() # selection and crossover
            self._mutate()
            highest: C = max(self._population, key=self._fitness_key)
            if highest.fitness() > best.fitness():
                best = highest  # found a new best
        return best  # best we found in _max_generations
```


`fitness_key` ist eine Referenz auf die Methode, die die Fitness eines Chromosoms berechnet.

Durchführung der Mutationen

```
94      def _mutate(self) -> None:
95          for individual in self._population:
96              if random() < self._mutation_chance:
97                  individual.mutate()
```

Durchführung der Selektion und Vererbung

```
69      def _reproduce_and_replace(self) -> None:
70          new_population: list[C] = []
71          # Selektiere und Kreuze Individuen, bis die neue Population steht
72          while len(new_population) < len(self._population):
73              # Wahl der zwei Eltern gemäß Selektionsmethode
74              # (Roulette oder Turnier)
```

```
82              # Kreuze ggf. die Eltern
83              if random() < self._crossover_chance:
84                  new_population.extend(parents[0].crossover(parents[1]))
85              else:
86                  new_population.extend(parents)
87          # Falls die Populationsgröße ungerade ist,
88          # entferne das letzte Individuum
89          if len(new_population) > len(self._population):
90              new_population.pop()
91          self._population = new_population  # replace reference
```

```
73              # Wahl der zwei Eltern gemäß Selektionsmethode
74              # (Roulette oder Turnier)
75              if self._selection_type == \
76                  GeneticAlgorithm.SelectionType.ROULETTE:
77                  parents: tuple[C, C] = self._pick_roulette(
78                      [x.fitness() for x in self._population]
79                  )
80              else:
81                  parents = \
82                      self._pick_tournament(len(self._population) // 2)
```

Die **Herausforderung bei der Entwicklung von genetischen Algorithmen** ist somit zweigeteilt:

1. eine passende Formulierung für das Problem zu entwickeln und
2. danach die Parameter so zu wählen, dass der Algorithmus in einer akzeptablen Zeit eine gute Lösung findet.

3. Beispiel: SEND + MORE = MONEY

SEND+MORE=MONEY[1]

- Klassisches Problem der Kryptographie
- Jeder Buchstabe repräsentiert eine Ziffer von 0 bis 9
- Keine Ziffer darf doppelt vorkommen

```
  S E N D
+ M O R E
---------
M O N E Y
```

- Welcher Buchstabe steht für welchen Wert?

[1] Wir verwenden hier das Beispiel lediglich zur Demonstration der Technik. Das Problem kann auch anders gelöst werden!

SEND+MORE=MONEY - Umsetzung

Chromosomen ermöglichen Zuordnung von Buchstaben zu Ziffern

Idee: Verwendung von Listenindizes zur Repräsentation der Ziffern: [0,...,9]. Die Werte in der Liste entsprechen den Buchstaben; Beispiel:

Zuordnung von Ziffern zu Buchstaben:

Index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
letters = [	O,	M,	Y,	␣,	␣,	E,	N,	D,	R,	S	]

Für die Lösung würde sich ergeben:

S	E	N	D	M	O	R	Y	_	_
9	5	6	7	1	0	8	2	3	4

Somit kann durch die Verschiebung der Buchstaben in der Liste der assoziierte Wert geändert werden.

1

Klasse für Chromosomen

```
1 class SendMoreMoney(Chromosome):
2     def __init__(self, letters: list[str]) -> None:
3         self.letters: list[str] = letters
```

2

Erzeugen von Individuen bzw. Chromosomen

Idee: beliebige Verwürfelung der Buchstaben in der Liste.

```
1     def random_instance(cls) -> Self:
2         letters = [
3             "S", "E", "N", "D", "M", "O", "R", "Y",
4             " ", " "]
5         shuffle(letters)
6         return SendMoreMoney(letters)
```

3

Mutation eines Chromosoms

Idee: zufälliger paarweiser Austausch von zwei Buchstaben in der Liste.

```
    def mutate(self) -> None: # swap two letters' locations
        idx1, idx2 = sample(range(len(self.letters)), k=2)
        self.letters[idx1], self.letters[idx2] = \
            self.letters[idx2], self.letters[idx1]
```


Vererbung

Idee: pro Nachkommen sicherstellen, dass einige Buchstaben Indizes (d. h. Wertzuordnungen) von einem Elternteil und einige vom anderen Elternteil stammen.

Beispiel (verkürzt):

```
Elternteil 1 = [a, b, c, d, e]  # d.h. ein Chromosom
Elternteil 2 = [c, a, b, e, d]  # d.h. ein Chromosom
```

gewählte Indizes seien: 0 und 2

```
Nachkomme 1 = [a, c, b, d, e]  # im Wesentlichen Elternteil 1,
                                aber Position von "b" vom 2ten übernommen
                                (dadurch Anpassung der Pos. von "c" notw.)
Nachkomme 2 = [a, c, b, e, d]  # im Wesentlichen Elternteil 2,
                                aber Position von "a" vom 1ten übernommen

                                (dadurch Anpassung der Pos. von "c" notw.)
```


Vererbung - Implementierung

```
    def crossover(self, other: Self) -> tuple[Self, Self]:
        child1: SendMoreMoney = deepcopy(self)
        child2: SendMoreMoney = deepcopy(other)
        idx1, idx2 = sample(range(len(self.letters)), k=2)
        l1, l2 = child1.letters[idx1], child2.letters[idx2]
        child1.letters[child1.letters.index(l2)], \
        child1.letters[idx2] = \
            child1.letters[idx2], l2

        child2.letters[child2.letters.index(l1)], \
        child2.letters[idx1] = \
            child2.letters[idx1], l1

        return child1, child2
```


Fitness-Funktion

Idee: Eine Lösung ist besser, je näher die Summe der Ziffern von SEND und MORE an MONEY ist. D. h.:

$$\text{minimiere}(|money - (send + more)|)$$

Feststellung: Das Ziel des generischen genetischen Algorithmus ist es, die Fitness-Funktion zu maximieren. Wir müssen somit unser Minimierungsproblem in ein Maximierungsproblem umwandeln:

$$\text{maximiere}\left(\frac{1}{|money - (send + more)| + 1}\right)$$

Beispiel: Sei die Differenz 1, dann ist die Fitness $1/2$; bei einer Differenz von 0 ist die Fitness 1 und somit maximal.

Fitness-Funktion - Implementierung

```
1     def fitness(self) -> float:
2         s: int = self.letters.index("S")
3         e: int = self.letters.index("E")
4         n: int = self.letters.index("N")
5         d: int = self.letters.index("D")
6         m: int = self.letters.index("M")
7         o: int = self.letters.index("O")
8         r: int = self.letters.index("R")
9         y: int = self.letters.index("Y")
10        send: int = s * 1000 + e * 100 + n * 10 + d
11        more: int = m * 1000 + o * 100 + r * 10 + e
12        money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
13        return 1 / (abs(money - (send + more)) + 1)
```


Initialisierung

```
1     initial_population: list[SendMoreMoney] = \
2         [SendMoreMoney.random_instance() for _ in range(1000)]
3     ga: GeneticAlgorithm[SendMoreMoney] = \
4         GeneticAlgorithm(
5             initial_population=initial_population,
6             threshold=1.0,
7             max_generations = 1000,
8             mutation_chance = 0.2,
9             crossover_chance = 0.7,
```

```
            selection_type= \
                GeneticAlgorithm.SelectionType.ROULETTE)
    result: SendMoreMoney = ga.run()
    print(result)
```

9

Exemplarische Ausführungen zeigt stochastische Natur

1. Lauf (237 Generationen)

```
Generation 0 Best 0.03333333333333333 Avg 0.0001322689088530895
⋮
Generation 236 Best 0.5 Avg 0.26552306542712417
9567 + 1085 = 10652 Difference: 0
```

2. Lauf (2 Generationen)

```
Generation 0 Best 0.025 Avg 0.0001658511923601707
Generation 1 Best 0.3333333333333333 Avg 0.003446183841665406
7429 + 814 = 8243 Difference: 0
```

3. Lauf (4 Generationen)

```
Generation 0 Best 0.006896551724137931 Avg 0.00012684576568285674
⋮
Generation 3 Best 0.5 Avg 0.01426133638316889
5731 + 647 = 6378 Difference: 0
```

10

Übung

Gruppenzuteilung

Entwickeln Sie einen genetischen Algorithmus, der eine sehr gute Aufteilung von Personen (Studierenden) auf eine feste Anzahl an Gruppen findet, basierend auf den Präferenzen der Personen.

Im Template ist eine initiale Aufgabenstellung hinterlegt, die es zu lösen gilt: Verteilung von 16 Studierenden auf 4 Gruppen inkl. Bewertungsmatrix (jeder Studierende hat jeden anderen mit Werten von 1 bis 10 bewertet).

Basis (Achten Sie auf die Verzeichnisse)

https://delors.github.io/theo-algo-genetic_algorithms/code/lib/chromosome.py

https://delors.github.io/theo-algo-genetic_algorithms/code/lib/genetic_algorithm.py

https://delors.github.io/theo-algo-genetic_algorithms/code/group_assignment_template.py

Denken Sie daran, dass Sie die Parameter bzgl. `mutation_chance`, `crossover_chance`, `selection_type`, `initial_population` und `max_generations` anpassen können.

Sie können für die Darstellung der Mitglieder einer Gruppe auch eine andere Darstellung als eine Liste von Listen verwenden. Sie müssen dann nur ggf. auch die `__str__` Methode anpassen. Es steht Ihnen natürlich auch frei Konzepte wie Memoization zu verwenden, um die Fitness-Funktion zu beschleunigen.

Sieger ist, wer in einem akzeptablen Zeitrahmen (3 Minuten) die beste Lösung findet.

Gruppenzuteilung

Entwickeln Sie einen genetischen Algorithmus, der eine sehr gute Aufteilung von Personen (Studierenden) auf eine feste Anzahl an Gruppen findet, basierend auf den Präferenzen der Personen.

Im Template ist eine initiale Aufgabenstellung hinterlegt, die es zu lösen gilt: Verteilung von 16 Studierenden auf 4 Gruppen inkl. Bewertungsmatrix (jeder Studierende hat jeden anderen mit Werten von 1 bis 10 bewertet).

Basis (Achten Sie auf die Verzeichnisse)

https://delors.github.io/theo-algo-genetic_algorithms/code/lib/chromosome.py

https://delors.github.io/theo-algo-genetic_algorithms/code/lib/genetic_algorithm.py

https://delors.github.io/theo-algo-genetic_algorithms/code/group_assignment_template.py

Übung

Ausgeglichene Gruppenzuteilung

Passen Sie Ihren genetischen Algorithmus aus der vorherigen Übung so an, dass die Gruppen alle in etwa die gleiche Glücklichkeit aufweisen. D. h. eine Zuteilung, bei der die Gruppenglücklichkeiten z. B. 92 + 91 + 73 + 89 = 345 sind, die aber größere Unterschiede zwischen den Gruppen hat, sollte vermieden werden. Eine Verteilung z. B. mit den Gruppenglücklichkeiten: 80, 84, 80, 80 = 324 sollte bevorzugt werden.

Hinweis

Sie können den bisher errechneten Wert zum Beispiel dadurch anpassen, dass Sie von dem Wert die Summe der absoluten Abweichungen vom Durchschnitt abziehen. Ggf. ist es erforderlich den Wert auch noch zu skalieren, damit steigende Abweichungen stärker ins Gewicht fallen.

Bedenken sie, dass sie ggf. den Threshold anpassen müssen.

Ausgeglichene Gruppenzuteilung

Passen Sie Ihren genetischen Algorithmus aus der vorherigen Übung so an, dass die Gruppen alle in etwa die gleiche Glücklichkeit aufweisen. D. h. eine Zuteilung, bei der die Gruppenglücklichkeiten z. B. 92 + 91 + 73 + 89 = 345 sind, die aber größere Unterschiede zwischen den Gruppen hat, sollte vermieden werden. Eine Verteilung z. B. mit den Gruppenglücklichkeiten: 80, 84, 80, 80 = 324 sollte bevorzugt werden.

Hinweis

Sie können den bisher errechneten Wert zum Beispiel dadurch anpassen, dass Sie von dem Wert die Summe der absoluten Abweichungen vom Durchschnitt abziehen. Ggf. ist es erforderlich den Wert auch noch zu skalieren, damit steigende Abweichungen stärker ins Gewicht fallen.

Bedenken sie, dass sie ggf. den Threshold anpassen müssen.