DHBW CAS
Duale Hochschule
Baden-Württemberg
Center for Advanced Studies

Computational Intelligence - Programmentwurf

Anwendung eines genetischen Algorithmus zur Optimierung von Produktionsplanung in einem Umfeld mit hoher Variantenvielfalt

DHBW CAS Heilbronn

Modul T3M40507.1 Maschinelles Lernen und Computational Intelligence

Ruslan Adilgereev, 7719665, Elektro- und Informationstechnik TMEIT23-W

Valentin Langenbacher, 1934264, Integrated Engineering TMIE22-W

Betreuer: Prof. Dr. Heinrich Braun

Abgabetermin: 30.10.23

Inhaltsverzeichnis

Abkürzungsverzeichnis

FSSP	Flow Shop Scheduling Problem
GAs	Genetische Algorithmen
KFZ	Kraftfahrzeug

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung in Optimierungsproblem

Die Produktion in der modernen Industrie steht heute mehr denn je vor komplexen Herausforderungen. Eine dieser Herausforderungen ist die Bewältigung der Variantenvielfalt, die aufgrund von individuellen Kundenanforderungen und der unterschiedlichen Produktquantitäten stetig zunimmt. Die Notwendigkeit, eine Vielzahl von Produktvarianten effizient zu steuern, wird besonders in Produktionsumgebungen deutlich, in denen die Balance zwischen Anpassungsfähigkeit und Effizienz von zentraler Bedeutung ist.

Betrachten wir beispielsweise die Produktionsplanung von Invertern für KFZ-E-Achsen. Hier stehen wir vor der komplexen Aufgabe, vier aufeinanderfolgende Produktionslinien zu koordinieren. Diese Produktionslinien sind nicht nur durch ihre Abfolge miteinander verbunden, sondern auch durch die Tatsache, dass sie Produkte mit unterschiedlichsten Anforderungen und Spezifikationen herstellen müssen. Diese hohe Variantenvielfalt, die durch spezifische Kundenanforderungen und unterschiedliche Produktionsmengen hervorgerufen wird, führt zu variierenden Komplexitäten und Produktionszeiten jeder einzelnen Linie. Ein Produkt mit geringerer Komplexität könnte beispielsweise schneller durch die Produktionslinien geführt werden als ein komplexeres Produkt.

Zur näheren Erläuterung sind nachfolgend die Erzeugnisse der jeweiligen Produktionslinien abgebildet.

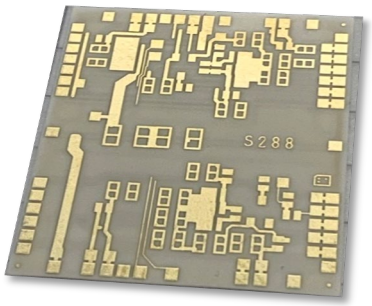

Abbildung 1: Logik-/Schaltungsmodul - Linie 1

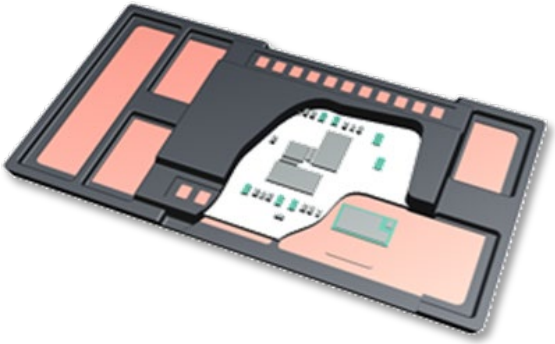

Abbildung 2: Power Modul - Linie 2

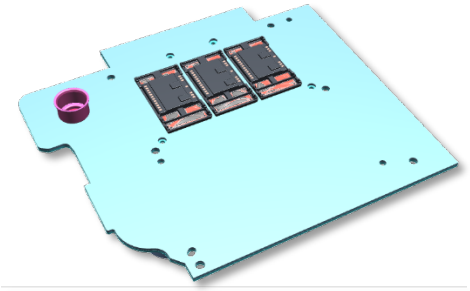

Abbildung 3: Kühlerbaugruppe - Linie 3

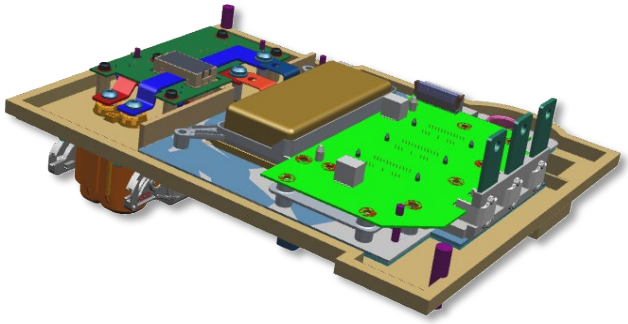

Abbildung 4: Gesamtbaugruppe "Inverter" zur Montage im KFZ/E-Achse - Linie 4

Je nach Kunde und Kundenspezifikation werden die einzelnen Linienerzeugnisse unterschiedlich komplex gefertigt. So erfordern bspw. bessere Eigenschaften des Logikmoduls eine längere Bearbeitungsdauer oder kundenspezifische Schnittstellen (z.B. Kühlwasserstutzen) der Gesamtbaugruppe erfordern zusätzliche Montageschritte.

Die Frage, die sich hier stellt, ist: Wie kann man in einer solchen Produktionsumgebung effizient steuern, um sowohl der hohen Variantenvielfalt gerecht zu werden als auch sicherzustellen, dass die Produktionslinien optimal ausgelastet sind? Wie lässt sich die Balance zwischen der schnellen Fertigung einfacher Produkte und der genauen Koordination komplexer Varianten finden?

In der vorliegenden Arbeit werden wir genau diesen Fragen nachgehen und versuchen, Ansätze und Strategien zu entwickeln, um die Produktionssteuerung bei hoher Variantenvielfalt zu optimieren.

In unserer Untersuchung der effizienten Produktionssteuerung bei Variantenvielfalt haben wir sechs spezifische Erzeugnisse identifiziert, die als P1 bis P6 bezeichnet werden. Jedes dieser Erzeugnisse weist unterschiedliche Anforderungen in Bezug auf die Stückzahl und die Herstellungskomplexität im Beispielmonat auf.

Laut den vorliegenden Daten variieren die geforderten Stückzahlen der Produkte beträchtlich, wobei P6 mit 814 Einheiten die höchste Nachfrage aufweist und P1 mit 139 Einheiten die geringste. Diese Unterschiede in der Stückzahl stellen bereits eine Herausforderung in Bezug auf die Produktionsplanung dar.

Stückzahlen	P1	139
	P2	729
	P3	382
	P4	382
	P5	468
	P6	814

Tabelle 1: Stückzahlen der Produktvarianten im Beispielmonat

Noch komplexer wird die Aufgabe, wenn man die Herstellungskomplexität betrachtet, die sich in den unterschiedlichen Durchlaufzeiten pro Fertigungslinie zeigt. Ein Blick auf die Taktzeiten offenbart, dass nicht alle Produkte gleich schnell produziert werden können. Je nach Produkt variieren die Taktzeiten pro Fertigungslinie zwischen 12,2 Sekunden und 160 Sekunden, wobei einige Produkte in bestimmten Linien schneller und in anderen langsamer hergestellt werden können.

		Linie1	Linie2	Linie3	Linie4
Taktzeit[sec]	P1	78,9	42,7	141	145
	P2	113,3	61,1	159	160
	P3	78,9	42,7	141	145
	P4	113,3	61,1	159	160
	P5	46,0	12,2	61,5	80
	P6	113,3	75	141	100

Tabelle 2: Taktzeiten der jeweiligen Produktvarianten und Produktionslinien

Die Herausforderung besteht nun darin, die bestmögliche Reihenfolge der Fertigungsaufträge zu bestimmen, um eine zeitlich optimale Auslastung der Produktionslinien zu gewährleisten. Ziel ist es, Engpässe zu vermeiden, die Kapazität der Produktionslinien optimal auszunutzen und die geforderten Stückzahlen pünktlich zu liefern.

Um dies zu erreichen, könnten mehrere Ansätze in Betracht gezogen werden, darunter die Priorisierung von Produkten mit höheren Stückzahlen oder die Optimierung der Reihenfolge basierend auf den Taktzeiten. Ein weiterer Ansatz könnte die gleichzeitige Produktion von

Produkten mit kürzeren und längeren Taktzeiten sein, um die Durchlaufzeiten insgesamt zu verkürzen.

Insgesamt wird die sorgfältige Planung und Koordination der Produktionslinien entscheidend sein, um sowohl die Stückzahl- als auch die Zeitanforderungen zu erfüllen und dabei eine hohe Effizienz zu gewährleisten.

2 Formale Modellierung

Folgende Tabellen sollen die Modellierung vereinfacht darstellen, mithilfe simplifizierter Taktzeiten:

	Linie 1 [ZE]	Linie 2 [ZE]	Linie 3 [ZE]	Stückzahl []
Produkt 1	3	1	1	4
Produkt 2	1	2	4	3
Produkt 3	5	3	3	2

Tabelle 3: Beispiel mit vereinfachten Taktzeiten

Ein Individuum mit einer Fertigstellung nach 31 Zeiteinheiten könnte folgend aussehen – Produkt 2 (gelb) wird zunächst in Linie 1 bearbeitet, sobald ein Stück fertig, wird dieses in Linie 2 weiterbearbeitet usw.:

<table>
<tr><td>1</td><td>2</td><td>3</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">3</td><td colspan="3">4</td><td colspan="5">1</td><td colspan="5">2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>x</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td>1</td><td>x</td><td>2</td><td>x</td><td>x</td><td>3</td><td>x</td><td>x</td><td>4</td><td>x</td><td>x</td><td>x</td><td>x</td><td colspan="3">1</td><td>x</td><td>x</td><td colspan="3">2</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>x</td><td>x</td><td>x</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td><td>1</td><td>2</td><td>3</td><td>4</td><td>x</td><td>x</td><td>x</td><td>x</td><td colspan="3">1</td><td>x</td><td>x</td><td colspan="3">2</td><td></td><td></td></tr>
</table>

Tabelle 4: Beispiel-Individuum #1

Dahingegen weist Individuum 2 eine Fertigstellung nach 37 Zeiteinheiten auf:

<table>
<tr><td colspan="5">1</td><td colspan="5">2</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">3</td><td colspan="3">4</td><td>1</td><td>2</td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td colspan="3">1</td><td colspan="3">2</td><td>1</td><td>2</td><td>x</td><td>3</td><td>x</td><td>x</td><td>4</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td colspan="3">1</td><td colspan="3">2</td><td>1</td><td>2</td><td>3</td><td>x</td><td>4</td><td>x</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td><td></td><td></td><td></td></tr>
</table>

Tabelle 5: Beispiel-Individuum #2

Die zuvor beschriebenen Überlegungen und Kriterien für die zeitlich optimale Auslastung der Produktionslinien lassen sich in ein formales Modell überführen. Dieses Modell dient als Basis für weiterführende Optimierungsverfahren und -techniken.

Entscheidungsvariable:

Die Hauptentscheidungsvariable ist die Reihenfolge, in der die Produkte die Prozesse durchlaufen. Die Codierung dieser Reihenfolge, bezeichnet als x, wird durch eine Liste von Zahlen dargestellt. Zum Beispiel bedeutet $\mathrm{x} = [0,1,0,3,1,2]$ die Reihenfolge der Produkte: Produkt0, Produkt1, Produkt0, Produkt3, Produkt1 und Produkt2.

Zielfunktion:

Das primäre Ziel des Modells ist es, die Gesamtzeit T zu minimieren, die benötigt wird, um alle Produkte durch die Fertigungslinien zu führen. Die Formulierung der Zielfunktion kann wie folgt dargestellt werden:

$$\min(\mathrm{T}) = \min(\max(\mathrm{t}_{i-1,j}, \mathrm{t}_{i,j-1}) + \mathrm{P}_{i.j})$$

Hierbei gibt die Hilfsmatrix P die Bearbeitungszeiten der Produkte in einer Matrixform wieder. Das Hauptziel hinter dieser Zielfunktion ist es, sicherzustellen, dass die Produktionslinien so weit wie möglich für Ingenieurarbeiten verfügbar sind und die geforderten Stückzahlen zeitlich optimal verarbeitet werden.

Nebenbedingungen:

Es gibt einige Einschränkungen, die berücksichtigt werden müssen:

Die Reihenfolge der Prozesse muss beibehalten werden. Dies wird formal als

$\mathrm{t}_{\mathrm{Ende,ij}} \leq \mathrm{t}_{\mathrm{Anfang,i+1,j}}$ ausgedrückt.

Die Reihenfolge der Produkte sollte in allen Prozessen gleich sein, d.h.

$x_{\mathrm{i,j}} = \mathrm{x}_{\mathrm{i,j+1}}$, für alle i und j.

3 Skizzierung Lösungsansatz: Genetische Algorithmen

3.1 Einleitung und Motivation

Die Optimierung von Produktionsprozessen ist in der modernen Industrie ein zentrales Anliegen. Mit dem Aufkommen der Industrie 4.0 und dem zunehmenden Wettbewerb stehen Unternehmen vor der Herausforderung, ihre Produktionseffizienz stetig zu erhöhen, während sie gleichzeitig flexibel auf Marktanforderungen reagieren müssen. In diesem Kontext werden Optimierungsalgorithmen immer wichtiger, um komplexe Produktionsaufgaben effizient und effektiv zu lösen.

Ein spezielles Problem, das in Produktionsumgebungen auftritt, ist die Planung von Produktionsreihenfolgen auf mehreren, miteinander verbundenen Produktionslinien, auch bekannt als das Flow Shop Scheduling Problem (FSSP). Dieses Problem ist NP-schwer, was bedeutet, dass es keine effiziente Lösung gibt, die in polynomialer Zeit gefunden werden kann. Daher ist der Einsatz von heuristischen Methoden zur Annäherung an eine optimale Lösung in vielen Fällen gerechtfertigt.

Genetische Algorithmen (GAs) bieten hierfür eine vielversprechende Lösungsstrategie. Sie sind nicht nur flexibel genug, um eine Vielzahl von Problemstellungen abzudecken, sondern haben sich auch in der Praxis als robust und effizient erwiesen. Die Anwendung von GAs im Kontext von Produktionsprozessen ist besonders interessant, da sie die Möglichkeit bieten, eine optimale oder nahezu optimale Reihenfolge der Produktionsaufträge zu finden, die den gesamten Durchlauf durch die Produktionslinien minimiert.

Die Relevanz der Optimierung mittels GAs wird zusätzlich durch die steigende Komplexität moderner Produktionsumgebungen erhöht. Die Produktvielfalt, kundenspezifische Anforderungen und variierende Losgrößen führen zu einem hohen Grad an Variabilität, der ohne intelligente Planungsalgorithmen kaum effizient zu bewältigen ist. In diesem Sinne bieten GAs eine ausgezeichnete Möglichkeit, die Effizienz der Produktion unter Berücksichtigung von Komplexität und Variabilität signifikant zu steigern.

3.2 Modellierung von Genotyp und Phänotyp

Im Kontext genetischer Algorithmen stellt die Modellierung von Genotyp und Phänotyp eine entscheidende Voraussetzung für den Erfolg der Optimierung dar. Der Genotyp repräsentiert die genetische Struktur eines Individuums und dient als Datengrundlage für die Optimierung.

In unserem Fall des Flow Shop Scheduling Problems (FSSP) wird der Genotyp als eine Liste von Produktionsaufträgen dargestellt. Diese Liste enthält die Reihenfolge, in der verschiedene Produkte die verschiedenen Produktionslinien durchlaufen sollen. Dabei wird jedes Produkt durch einen eindeutigen Index repräsentiert, und die Reihenfolge der Indizes in der Liste gibt die Produktionsreihenfolge an.

Der Phänotyp hingegen ist die manifeste Eigenschaft des Genotyps, das heißt, er repräsentiert die tatsächliche, in der Realität beobachtbare Ausprägung des Genotyps (siehe Abbildung 5). Für das FSSP ist der Phänotyp der sogenannte "Makespan", also die gesamte benötigte Zeit für die Fertigstellung aller Produkte auf der letzten Produktionslinie. Um vom Genotyp zum Phänotyp zu gelangen, wird eine spezielle Übersetzungsfunktion benötigt. In unserer Implementierung wird dies durch die Funktion **order_to_time** und **fitness_calc** erledigt. Die **order_to_time** Funktion nimmt den Genotyp als Eingabe und transformiert ihn in eine Matrix, die die Produktionszeiten für jedes Produkt auf jeder Linie enthält. Die **fitness_calc** Funktion nimmt diese Matrix und berechnet den Makespan.

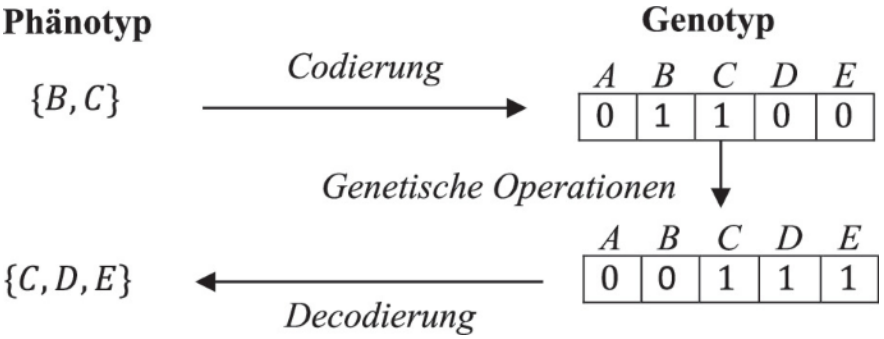

Abbildung 5: Genotyp und Phänotyp

3.3 Fitness-Funktion

Die Fitness-Funktion ist ein entscheidender Bestandteil im genetischen Algorithmus und dient zur Quantifizierung des Phänotyps eines Individuums. In unserem speziellen Fall des Flow Shop Scheduling Problems ist der Phänotyp der Makespan, der die gesamte Produktionsdauer für alle Produkte auf der letzten Produktionslinie angibt. Diese Dauer wird durch die Formel $min(T) = min(max(t_{i-1,j}, t_{i,j-1}) + P_{i,j})$ berechnet, wobei T den Makespan und P die Bearbeitungszeiten repräsentiert. Diese Formel ist in der **fitness_calc** Funktion implementiert, die eine Matrix von Produktionszeiten akzeptiert und den Makespan als Ausgabe liefert. Durch die Quantifizierung des Phänotyps in einer einzigen Metrik ermöglicht die Fitness-Funktion die Selektion der besten Individuen für die nächste Generation, was entscheidend für die Effizienz des gesamten Optimierungsprozesses ist.

3.4 Selektion, Crossover und Mutation

In genetischen Algorithmen sind die Mechanismen der Selektion, Crossover und Mutation entscheidend für die Generierung neuer, potenziell besserer Lösungen für das Optimierungsproblem. Des Weiteren gibt es weitere Methoden zur Evolution, auf die nicht näher eingegangen wird (siehe Abbildung 6).

Selektion: Im Kontext Ihres Projekts erfolgt die Selektion der besten Individuen einer Generation auf Grundlage der Fitness-Funktion, die den Makespan minimiert. In Ihrer Implementierung werden die besten 20% der Individuen für die nächste Generation ausgewählt. Dieser selektive Druck fördert die Weitergabe von vorteilhaften Eigenschaften und ist ein Schlüsselaspekt für die Effizienz des Algorithmus.

Crossover: Normalerweise ist der Crossover-Schritt dafür verantwortlich, neue Individuen durch die Kombination der Genotypen der Eltern zu erzeugen. In unserem Fall ist die Anwendung des Crossover-Mechanismus jedoch problematisch. Da die Anzahl der Produkte (Stückzahl) in der Produktionsreihenfolge beibehalten werden muss, führt ein herkömmliches Crossover zu inkonsistenten Lösungen. Daher wurde in Ihrer Implementierung auf den Crossover-Schritt verzichtet, was eine interessante Anpassung des Standardverfahrens darstellt.

Mutation: Die Mutation ist in der Implementierung weiterhin vorhanden und spielt eine wichtige Rolle bei der Exploration des Lösungsraums. Bei einer vorgegebenen Mutationsrate von 1 werden zufällige Änderungen am Genotyp vorgenommen, wobei die Stückzahl der Produkte unverändert bleibt. Diese Mutationen tragen dazu bei, den Algorithmus aus lokalen Optima herauszuführen und eine breitere Suche im Lösungsraum zu ermöglichen.

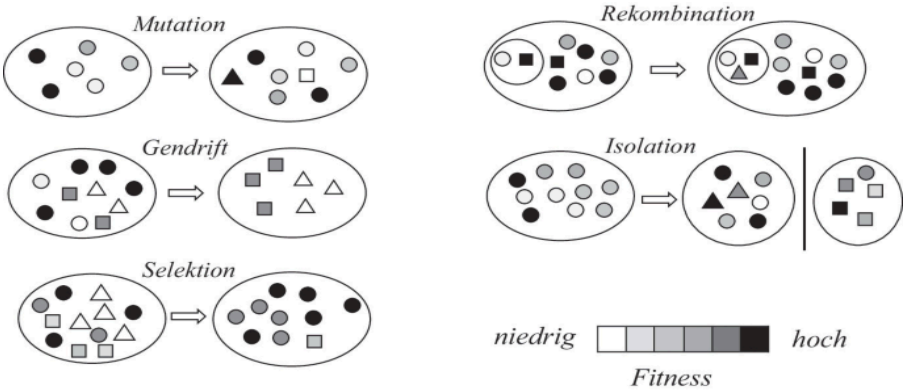

Abbildung 6: Methoden der Evolution

4 Python-Umsetzung

Datenvorbereitung

Das 2D-NumPy-Array **P_pcs_time** dient als Grundlage für die gesamte Simulation. Es enthält die Stückzahlen der Produkte sowie deren Produktionszeiten auf den verschiedenen Bändern. Dieses Array wird sowohl für die Erstellung der Anfangspopulation als auch für die Berechnung der Fitness eines jeden Individuums verwendet.

Individuenerzeugung und Anfangspopulation

Die Funktion **generate_individuum()** erstellt ein zufälliges Individuum, das eine bestimmte Reihenfolge der Produkte repräsentiert. Jedes Individuum wird durch eine permutierte Liste der Produktindizes dargestellt. Dabei wird die Stückzahl der Produkte berücksichtigt, sodass jedes Produkt entsprechend seiner Stückzahl in der Liste erscheint.

Nach der Generierung der Individuen wird die Anfangspopulation erstellt. Die Fitness jedes Individuums wird berechnet und in einem Dictionary, **pop_fitness**, gespeichert. Dieses Dictionary dient als schnelle Referenz für spätere Operationen wie Selektion und Mutation.

Genetische Operatoren

In diesem speziellen Fall wurde nur ein genetischer Operator, die Mutation, implementiert. Die Funktion **mutate()** nimmt ein Individuum und führt eine zufällige Permutation seiner Elemente durch. Dies geschieht nur, wenn eine generierte Zufallszahl kleiner als die vorgegebene Mutationsrate ist. Die Anzahl der zu mutierenden Elemente wird durch **mutation_count** festgelegt.

Die Crossover-Funktion wurde bewusst ausgelassen, da ein Crossover die Stückzahl der Produkte im Endresultat verändern würde, was in diesem Anwendungsfall nicht zulässig ist.

Evolutionärer Algorithmus

Der Hauptalgorithmus befindet sich im main-Block und steuert den gesamten Evolutionsprozess. In jeder Generation wird die Population anhand ihrer Fitness sortiert. Die besten Individuen (Top 20%, gesteuert durch **top_individuals_ratio**) werden ausgewählt und bilden die Grundlage für die nächste Generation. Diese Individuen werden mutiert und wieder in die Population eingefügt, bis die Population für die nächste Generation komplett ist.

Anhang

Python Code

```
import numpy as np
import matplotlib.pyplot as plt
import time

# P_pcs_time = np.array([[139, 729, 382, 382, 468, 814],  ## Stückzahl der Produkte
#                        [79,113, 78, 113, 46, 113],  ## Produktionszeit auf Band 1
#                        [43, 61, 42, 61,12,75], ## Produktionszeit auf Band 2
#                        [141,159,141,159,62,141], ## Produktionszeit auf Band 3
#                        [145,160,145,160,80,100]])  ## Produktionszeit auf Band 4)


P_pcs_time = np.array([ [1,2,3,3], ## Stückzahl der Produkte
                        [2,1,3,1], ## Produktionszeit auf Band 1
                        [5,1,2,1]])## Produktionszeit auf Band 2

def fitness_calc(P):
    """
    Calculate the total time for products to pass through all bands.

    Parameters:
    - P: 2D numpy array where rows represent bands and columns represent products.

    Returns:
    - Total time for all products to pass through the last band.
    """

    num_bands, num_products = P.shape  # Number of bands and products
    t = np.zeros((num_bands, num_products))  # Initialize the time matrix

    # For the first band
    t[0, 0] = P[0, 0]  # The first element is the same as the first element of P
    for j in range(1, num_products):  # The first row is the cumulative sum of the first row
of P
        t[0, j] = t[0, j - 1] + P[0, j]

    # For the subsequent bands
    for i in range(1, num_bands):  # For each row
        t[i, 0] = t[i - 1, 0] + P[i, 0]  # The first element is the sum of the first element
of the previous row
                                         # and the first element of P
        for j in range(1, num_products):  # For each column
            t[i, j] = max(t[i - 1, j], t[i, j - 1]) + P[i, j]

    return t[num_bands - 1, num_products - 1]  # Return the last element of the time matrix


def generate_individuum(P_pcs_time):
    # Create an empty list
    array = []
    n_band = P_pcs_time.shape[0] - 1

    # Create a product order list based on the quantity of each product
    product_order_list = []
    for i in range(P_pcs_time.shape[1]):
        product_order_list.extend([i] * P_pcs_time[0, i])

    # Shuffle the product order list to get the shuffled product order
    shuffled_product_order = np.random.permutation(product_order_list)

    # Create the shuffled processing time representation based on the shuffled product order
    for j in range(n_band):
        for product_index in shuffled_product_order:
            array.append(P_pcs_time[j + 1, product_index])

    # Convert the array to a 2D matrix
    shuffled_array = np.array(array).reshape(-1, len(shuffled_product_order))
```

```
    return shuffled_array, shuffled_product_order

## Nicht anwendbar, da die Stückzahl bei der Crossover Methode nicht beachtet wird.
def single_point_crossover_1D(A, B, p):  # A, B: 1D numpy arrays
    if p <= 0:
        p = np.random.randint(1,A.size)
    child1 = A  # np.concatenate((A[:p], B[p:]))
    child2 = B  # np.concatenate((B[:p], A[p:]))
    return child1, child2

def order_to_time(product_order):
    array = []
    n_band = P_pcs_time.shape[0] - 1

    # Create the processing time representation based on the given product order
    for j in range(n_band):
        for product_index in product_order:
            array.append(P_pcs_time[j + 1, product_index])

    # Convert the array to a 2D matrix
    time matrix = np.array(array).reshape(-1, len(product order))

    return time_matrix

def mutate(individuum, mutation_rate):
    """Mutation_Count gibt die Anzahl der Mutation an die stattfinden soll.
    Mutation_Rate gibt die Wahrscheinlichkeit an, dass eine Mutation stattfindet."""
    if np.random.rand() < mutation_rate:
        for i in range(mutation_count):
            # Wähle zwei zufällige Indizes zum Austauschen
            idx1, idx2 = np.random.randint(0, len(individuum), 2)
            # Tausche die Elemente an diesen Indizes
            individuum[idx1], individuum[idx2] = individuum[idx2], individuum[idx1]
    return individuum

if __name__ == "__main__":
    summe = 0
    start = time.time()
    generation_count = 1000  # Anzahl der Generationen
    pop_count = 100          # Anzahl der Individuuen pro Population
    mutation_rate = 1     # Wahrscheinlichkeit für eine Mutation (zwischen 0 und 1)
    mutation_count = 5      # Anzahl der Mutationselemente
    top_individuals_ratio = 0.2  # Anteil der besten Individuen, die in die nächste Generation
übernommen werden in % (zwischen 0 und 1)
    all_fit_values = []     # Liste, um die Fitnesswerte aller Individuen über alle
Generationen zu speichern
    all_order_values = []   # Liste, um die Reihenfolge aller Individuen über alle
Generationen zu speichern
    pop_order = {}
    pop_fitness = {}

    plt.ion()
    fig, ax = plt.subplots()
    # Anpassungen an der X- und Y-Achse, Titel, Legende usw.
    ax.set_xlabel('Individuum Nummer')
    ax.set_ylabel('Fitness')
    ax.set_title('Fitness über die Individuuen')
    ax.grid(True)
    line, = ax.plot([])     # Leerer Plot, der später aktualisiert wird

    """Greedy Verfahren"""
    sorted_P_pcs_time = np.argsort(P_pcs_time[1])
    sorted_M = P_pcs_time[:, sorted_P_pcs_time]
    # Extracting the repetition counts from the first row
    repetition_counts_first_row = sorted_M[0]
    greedy_order = [value for value, count in zip(sorted_P_pcs_time,
repetition_counts_first_row) for _ in range(count)]
    print(f"Fitness mit Greedy Verfahren: {fitness_calc(order_to_time(greedy_order))}:
{greedy_order}")
    """**************************"""

    """Startpopulation bilden"""
    for i in range(pop_count):
        ind_time, ind_order = generate_individuum(P_pcs_time)
        pop_fitness[i] = fitness_calc(ind_time)
```

```
        pop_order[i] = ind_order

    for generation in range(generation_count):
        # Sortieren aller Fitnesswerte dieser Population mit der Reihenfolge
        sorted_keys = sorted(pop_fitness, key = pop_fitness.get) # type: ignore
        sorted_pop_order = {key: pop_order[key] for key in sorted_keys}
        top_individuals = [sorted_pop_order[key] for key in sorted_keys[:int(pop_count *
top_individuals_ratio)]]
        top_individuals_dict = {i: indiv for i, indiv in enumerate(top_individuals)}

        k = len(top_individuals_dict)
        while k < pop_count:
            i, j = np.random.randint(0, len(top_individuals_dict), size=2)
            child1, child2 = single_point_crossover_1D(top_individuals_dict[i],
                                                       top_individuals_dict[j],
                                                       np.random.randint(0,
len(top_individuals_dict[0])-1))
            top_individuals_dict[k] = mutate(child1, mutation_rate)
            top_individuals_dict[k + 1] = mutate(child2, mutation_rate)
            k += 2

        for i in range(len(top individuals dict)):
            pop_order[i] = top_individuals_dict[i]
            current_fitness = fitness_calc(order_to_time(top_individuals_dict[i]))
            pop_fitness[i] = current_fitness
            all_fit_values.append((current_fitness, pop_order[i]))   # Add the current fitness
value directly
            all_fit_values.sort(key=lambda x: x[0], reverse=True)
            x_data = range(len(all_fit_values))                      # X-data for the plot
            y_data = [value[0] for value in all_fit_values]          # Extract the fitness
values for Y-data
            line.set_xdata(x_data)                                   # Update X-data
            line.set ydata(y data)              # Y-Daten aktualisieren

        ax.relim()  # Grenzen neu berechnen
        ax.autoscale_view()  # Skaliert die Achse
        plt.draw()  # Zeichnet den aktualisierten Plot
        plt.pause(0.01)
    print(f"Fitness mit Genetischen Algorithmen: {all fit values[-1]}")
    plt.ioff()
    ende = time.time()
    print('{:5.3f}s'.format(ende - start))
    plt.show()
```