Übung 05

Optimierung von Ablaufplanungen

Aufgabe 1: Optimierung der Montage

Cyber Systems steht vor der Herausforderung, die Montage ihrer neuesten Endoskelett-Arme zu optimieren. In der Endmontage gibt es eine einzelne Station, die für die finale Qualitätsprüfung und Kalibrierung zuständig ist. Für eine Charge von 6 Armen sind die Bearbeitungszeiten (in Stunden) für diese Station bekannt. Die Aufträge sind in der Reihenfolge ihres Eintreffens (FCFS) nummeriert.

Auftrag (Arm-ID)	Bearbeitungszeit a_p (Stunden)
A001	3
A002	5
A003	2
A004	8
A005	4
A006	6

Ihre Aufgaben:

1. Ermitteln Sie die Auftragsreihenfolge nach der FCFS-Regel (First Come, First Served). Berechnen Sie für diese Reihenfolge:
   - Den Fertigstellungszeitpunkt F_p für jeden Auftrag.
   - Die Durchlaufzeit D_p für jeden Auftrag (da alle Aufträge zum Zeitpunkt 0 eintreffen, gilt $D_p = F_p$).
   - Die mittlere Durchlaufzeit $\bar{D}$.
2. Ermitteln Sie die Auftragsreihenfolge nach der KOZ-Regel (Kürzeste Operationszeit-Regel, auch SPT-Regel). Berechnen Sie für diese Reihenfolge ebenfalls F_p, D_p und $\bar{D}$.
3. Vergleichen Sie die Ergebnisse der FCFS- und KOZ-Regel. Welche Regel führt zu einer geringeren mittleren Durchlaufzeit?
4. Diskutieren Sie kurz, warum die KOZ-Regel in Bezug auf die mittlere Durchlaufzeit optimal ist, aber welche potenziellen Nachteile sie haben könnte.

Lösungshinweise:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

plt.rcdefaults()
plt.style.use('tableau-colorblind10')
```

```python
# Auftragsdaten
data_task1 = {
    'Auftrag': ['A001', 'A002', 'A003', 'A004', 'A005', 'A006'],
    'Bearbeitungszeit_ap': [3, 5, 2, 8, 4, 6]
}
df_task1 = pd.DataFrame(data_task1)

print("Ursprüngliche Auftragsdaten:")
print(df_task1)

def calculate_schedule_metrics(df, order_col_name="Auftrag"):
    """Berechnet Fertigstellungszeit, Durchlaufzeit und mittlere
Durchlaufzeit."""
    df_calc = df.copy()
    df_calc['Fertigstellungszeit_Fp'] =
df_calc['Bearbeitungszeit_ap'].cumsum()
    # Da alle Aufträge zum Zeitpunkt 0 eintreffen, ist Durchlaufzeit =
Fertigstellungszeit
    df_calc['Durchlaufzeit_Dp'] = df_calc['Fertigstellungszeit_Fp']
    mean_flow_time = df_calc['Durchlaufzeit_Dp'].mean()
    return df_calc, mean_flow_time

# 1. FCFS-Regel (Aufträge sind bereits in FCFS-Reihenfolge)
df_fcfs = df_task1.copy()
df_fcfs_metrics, mean_flow_time_fcfs = calculate_schedule_metrics(df_fcfs)

print("\n1. FCFS-Regel:")
print(df_fcfs_metrics)
print(f"Mittlere Durchlaufzeit (FCFS): {mean_flow_time_fcfs:.2f} Stunden")

# 2. KOZ-Regel (SPT - Shortest Processing Time)
df_koz =
df_task1.sort_values(by='Bearbeitungszeit_ap').reset_index(drop=True)
df_koz_metrics, mean_flow_time_koz = calculate_schedule_metrics(df_koz)

print("\n2. KOZ-Regel (Kürzeste Operationszeit):")
print(df_koz_metrics)
print(f"Mittlere Durchlaufzeit (KOZ): {mean_flow_time_koz:.2f} Stunden")

print(f"""
3. Vergleich der Ergebnisse:
Die FCFS-Regel führt zu einer mittleren Durchlaufzeit von
{mean_flow_time_fcfs:.2f} Stunden.
Die KOZ-Regel führt zu einer mittleren Durchlaufzeit von
{mean_flow_time_koz:.2f} Stunden.
Die KOZ-Regel ist in diesem Fall besser und reduziert die mittlere
Durchlaufzeit.

4. Diskussion zur KOZ-Regel:
Die KOZ-Regel minimiert die mittlere Durchlaufzeit (und damit auch die
mittlere Wartezeit und den mittleren Bestand), weil sie dafür sorgt, dass
```

```
Aufträge schnell abgeschlossen werden und die Anzahl der wartenden Aufträge
rasch reduziert wird. Jeder Auftrag, der früher fertiggestellt wird, trägt
weniger zur Summe der Durchlaufzeiten bei.
Potenzielle Nachteile:
- Aufträge mit langer Bearbeitungszeit könnten sehr lange warten müssen
("Aushungern" langer Aufträge), was zu einer hohen Varianz der
Durchlaufzeiten führen kann.
- Wenn Liefertermine eine Rolle spielen, werden diese von der KOZ-Regel
nicht berücksichtigt, was zu Verspätungen bei wichtigen Aufträgen führen
kann, auch wenn diese eine lange Bearbeitungszeit haben.
""")

# Einfache Gantt-Chart Visualisierung (konzeptionell)
def plot_gantt(df, title, ax, order_col='Auftrag',
time_col='Bearbeitungszeit_ap', completion_col='Fertigstellungszeit_Fp'):
    start_time = 0
    for i, row in df.iterrows():
        ax.barh(row[order_col], row[time_col], left=start_time,
edgecolor='black', color='skyblue')
        start_time = row[completion_col]
    ax.set_xlabel("Zeit (Stunden)")
    ax.set_title(title)
    ax.grid(True, axis='x')
    ax.invert_yaxis() # Aufträge von oben nach unten

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(7, 5), sharex=True)
plot_gantt(df_fcfs_metrics, 'Gantt-Diagramm (FCFS-Regel)', ax1)
plot_gantt(df_koz_metrics, 'Gantt-Diagramm (KOZ-Regel)', ax2)
fig.suptitle('Vergleich FCFS vs. KOZ bei Cyber Systems', fontsize=14)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

```
Ursprüngliche Auftragsdaten:
  Auftrag  Bearbeitungszeit_ap
0    A001                    3
1    A002                    5
2    A003                    2
3    A004                    8
4    A005                    4
5    A006                    6

1. FCFS-Regel:
  Auftrag  Bearbeitungszeit_ap  Fertigstellungszeit_Fp  Durchlaufzeit_Dp
0    A001                    3                       3                 3
1    A002                    5                       8                 8
2    A003                    2                      10                10
3    A004                    8                      18                18
4    A005                    4                      22                22
5    A006                    6                      28                28
Mittlere Durchlaufzeit (FCFS): 14.83 Stunden
```

```
2. KOZ-Regel (Kürzeste Operationszeit):
  Auftrag  Bearbeitungszeit_ap  Fertigstellungszeit_Fp  Durchlaufzeit_Dp
0    A003                    2                       2                 2
1    A001                    3                       5                 5
2    A005                    4                       9                 9
3    A002                    5                      14                14
4    A006                    6                      20                20
5    A004                    8                      28                28
Mittlere Durchlaufzeit (KOZ): 13.00 Stunden

3. Vergleich der Ergebnisse:
Die FCFS-Regel führt zu einer mittleren Durchlaufzeit von 14.83 Stunden.
Die KOZ-Regel führt zu einer mittleren Durchlaufzeit von 13.00 Stunden.
Die KOZ-Regel ist in diesem Fall besser und reduziert die mittlere
Durchlaufzeit.

4. Diskussion zur KOZ-Regel:
Die KOZ-Regel minimiert die mittlere Durchlaufzeit (und damit auch die
mittlere Wartezeit und den mittleren Bestand), weil sie dafür sorgt, dass
Aufträge schnell abgeschlossen werden und die Anzahl der wartenden Aufträge
rasch reduziert wird. Jeder Auftrag, der früher fertiggestellt wird, trägt
weniger zur Summe der Durchlaufzeiten bei.
Potenzielle Nachteile:
- Aufträge mit langer Bearbeitungszeit könnten sehr lange warten müssen
("Aushungern" langer Aufträge), was zu einer hohen Varianz der
Durchlaufzeiten führen kann.
- Wenn Liefertermine eine Rolle spielen, werden diese von der KOZ-Regel
nicht berücksichtigt, was zu Verspätungen bei wichtigen Aufträgen führen
kann, auch wenn diese eine lange Bearbeitungszeit haben.
```

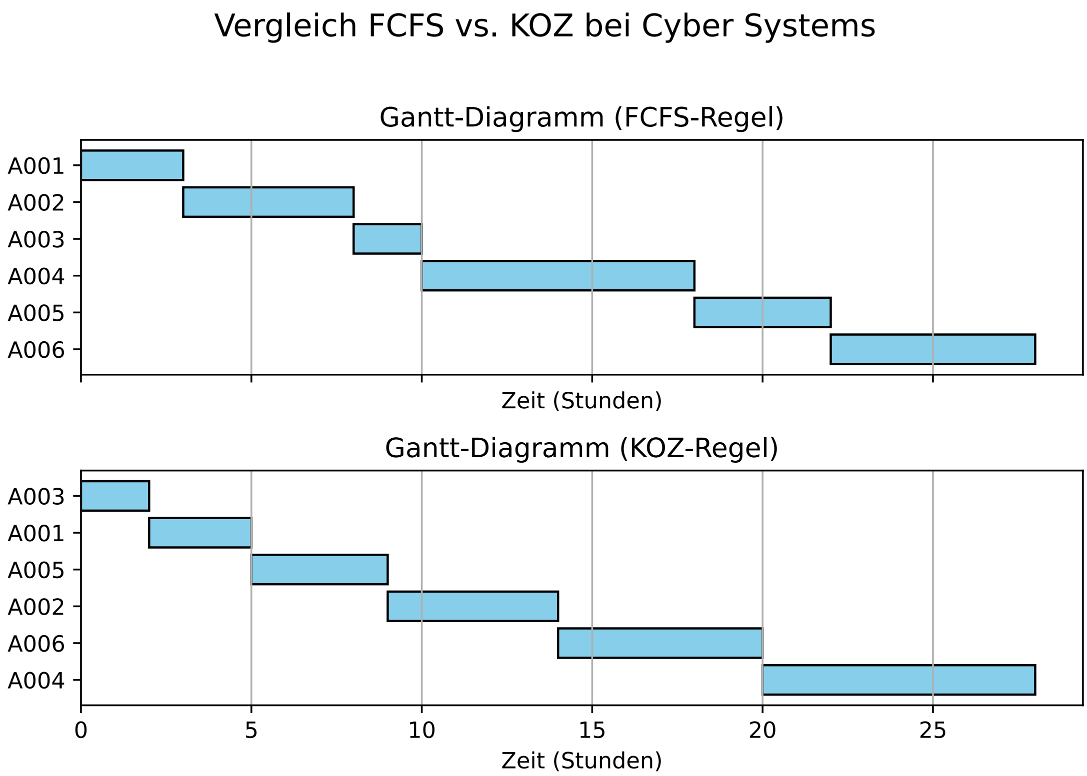

Figure 1: Bearbeitungsplan Endoskelett-Arme

Aufgabe 2: Einhaltung von Produktionsfristen

Wey Corp. steht unter Druck, kritische Navigationskomponenten für ihre nächste Generation von Frachtern der "Nostromos"-Klasse zu fertigen. Jeder Komponententyp durchläuft eine spezielle Endmontage- und Teststation. Für die aktuelle Produktionswoche liegen fünf dringende Aufträge vor, jeweils mit bekannter Bearbeitungszeit an dieser Station und einem festen Auslieferungstermin (Plantermin). Das Management möchte die Anzahl der verspäteten Aufträge minimieren. Alle Aufträge sind zu Beginn der Woche (Zeitpunkt 0) verfügbar.

Auftrag (Komponente)	Bearbeitungszeit a_p (Tage)	Plantermin LT_p (Tag)
C01	3	7
C02	5	10
C03	2	5
C04	6	12
C05	4	8

Ihre Aufgaben:

1. Sortieren Sie die Aufträge zunächst nach der Liefertermin-Regel (EDD - Earliest Due Date). Erstellen Sie einen Plan und ermitteln Sie für jeden Auftrag den Fertigstellungszeitpunkt F_p und die Verspätung $V_p = \max(0, F_p - LT_p)$. Wie viele Aufträge sind verspätet?
2. Wenden Sie nun das Hodgson-Verfahren an, um die Anzahl der verspäteten Aufträge zu minimieren.

3. Erstellen Sie den finalen Ablaufplan. Berechnen Sie für diesen Plan:
   - Den Fertigstellungszeitpunkt F_p für jeden Auftrag.
   - Die Verspätung V_p für jeden Auftrag.
   - Die Gesamtzahl der verspäteten Aufträge.
   - Die maximale Verspätung.
4. Vergleichen Sie das Ergebnis des Hodgson-Verfahrens mit dem der reinen Liefertermin-Regel hinsichtlich der Anzahl verspäteter Aufträge.

Lösungshinweise:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

plt.rcdefaults()
plt.style.use('tableau-colorblind10')

# Auftragsdaten
data_task2 = {
    'Auftrag': ['C01', 'C02', 'C03', 'C04', 'C05'],
    'Bearbeitungszeit_ap': [3, 5, 2, 6, 4],
    'Plantermin_LTp': [7, 10, 5, 12, 8]
}
df_task2 = pd.DataFrame(data_task2)

print("Ursprüngliche Auftragsdaten:")
print(df_task2)

def calculate_schedule_details(df_schedule):
    """Berechnet Fp, Vp für einen gegebenen Plan."""
    df_calc = df_schedule.copy()
    df_calc['Fertigstellungszeit_Fp'] =
df_calc['Bearbeitungszeit_ap'].cumsum()
    df_calc['Verspaetung_Vp'] = (df_calc['Fertigstellungszeit_Fp'] -
df_calc['Plantermin_LTp']).clip(lower=0)
    num_tardy_jobs = (df_calc['Verspaetung_Vp'] > 0).sum()
    max_tardiness = df_calc['Verspaetung_Vp'].max()
    total_tardiness = df_calc['Verspaetung_Vp'].sum()
    return df_calc, num_tardy_jobs, max_tardiness, total_tardiness

# 1. Reine Liefertermin-Regel (EDD)
df_edd = df_task2.sort_values(by='Plantermin_LTp').reset_index(drop=True)
df_edd_details, num_tardy_edd, max_tard_edd, total_tard_edd =
calculate_schedule_details(df_edd)

print("\n1. Reine Liefertermin-Regel (EDD):")
print(df_edd_details)
print(f"Anzahl verspäteter Aufträge (EDD): {num_tardy_edd}")
print(f"Maximale Verspätung (EDD): {max_tard_edd}")
print(f"Gesamte Verspätung (EDD): {total_tard_edd}")

# 2. Hodgson-Verfahren
```

```
print("\n2. Hodgson-Verfahren (Schrittweise):")

# Initialisierung
jobs_df = df_task2.copy()
R_set_indices = list(jobs_df.index) # Indizes der Aufträge
S_set_indices = []                   # Indizes der zurückgestellten
Aufträge

iteration = 0
while True:
    iteration += 1
    print(f"\nIteration {iteration}:")

    if not R_set_indices:
        print("R_set ist leer. Stoppe.")
        break

    # Aktuelle R_set Aufträge nach EDD sortieren
    current_R_jobs =
jobs_df.loc[R_set_indices].sort_values(by='Plantermin_LTp').reset_index(drop=False)

    # Fertigstellungszeiten und Verspätungen für current_R_jobs berechnen
    current_R_jobs['Fp_temp'] =
current_R_jobs['Bearbeitungszeit_ap'].cumsum()
    current_R_jobs['Vp_temp'] = (current_R_jobs['Fp_temp'] -
current_R_jobs['Plantermin_LTp']).clip(lower=0)

    print("Aktuelle Reihenfolge in R (nach EDD sortiert mit Fp, Vp):")
    print(current_R_jobs[['Auftrag', 'Bearbeitungszeit_ap',
'Plantermin_LTp', 'Fp_temp', 'Vp_temp']])

    # Ersten verspäteten Auftrag finden
    first_tardy_job_series = current_R_jobs[current_R_jobs['Vp_temp'] > 0]

    if first_tardy_job_series.empty:
        print("Keine verspäteten Aufträge in R_set gefunden. Hodgson-
Algorithmus abgeschlossen für R_set.")
        break

    first_tardy_job_index_in_current_R = first_tardy_job_series.index[0] #
Index in current_R_jobs DataFrame
    first_tardy_job_original_index =
current_R_jobs.loc[first_tardy_job_index_in_current_R, 'index']
    print(f"Erster verspäteter Auftrag:
{current_R_jobs.loc[first_tardy_job_index_in_current_R, 'Auftrag']}")

    # Teilmenge bis einschließlich des ersten verspäteten Auftrags
    # Indices in current_R_jobs
    subset_indices_in_current_R =
current_R_jobs.index[:first_tardy_job_index_in_current_R + 1]
    subset_jobs = current_R_jobs.loc[subset_indices_in_current_R]
    print("Betrachte Teilmenge für Entfernung (bis einschl. erster
```

```
Verspäteter):")
    print(subset_jobs[['Auftrag', 'Bearbeitungszeit_ap']])

    # Auftrag mit längster Bearbeitungszeit in dieser Teilmenge
    job_to_remove_idx_in_subset =
subset_jobs['Bearbeitungszeit_ap'].idxmax() # Index in current_R_jobs (via
subset_jobs)
    job_to_remove_original_idx =
current_R_jobs.loc[job_to_remove_idx_in_subset, 'index'] # Original-Index
in df_task2

    print(f"Entferne Auftrag '{jobs_df.loc[job_to_remove_original_idx,
'Auftrag']}' (Bearbeitungszeit: {jobs_df.loc[job_to_remove_original_idx,
'Bearbeitungszeit_ap']})")

    # Aus R_set entfernen und zu S_set hinzufügen
    R_set_indices.remove(job_to_remove_original_idx)
    S_set_indices.append(job_to_remove_original_idx)
    print(f"Aktuelles R_set (Original Indizes): {R_set_indices}")
    print(f"Aktuelles S_set (Original Indizes): {S_set_indices}")

# Finale Reihenfolge erstellen
final_R_jobs = jobs_df.loc[R_set_indices].sort_values(by='Plantermin_LTp')
# S_set Aufträge z.B. nach ursprünglichem Liefertermin oder KOZ anhängen
# Hier: nach Liefertermin sortiert für Konsistenz (oder KOZ, oder wie sie
entfernt wurden)
final_S_jobs = jobs_df.loc[S_set_indices].sort_values(by='Plantermin_LTp')

final_schedule_hodgson = pd.concat([final_R_jobs,
final_S_jobs]).reset_index(drop=True)

df_hodgson_details, num_tardy_hodgson, max_tard_hodgson, total_tard_hodgson
= calculate_schedule_details(final_schedule_hodgson)

print("\n3. & 4. Finaler Ablaufplan nach Hodgson-Verfahren:")
print(df_hodgson_details)
print(f"Anzahl verspäteter Aufträge (Hodgson): {num_tardy_hodgson}")
print(f"Maximale Verspätung (Hodgson): {max_tard_hodgson}")
print(f"Gesamte Verspätung (Hodgson): {total_tard_hodgson}")

print("""
5. Vergleich:
Die reine Liefertermin-Regel (EDD) führte zu {num_tardy_edd} verspäteten
Aufträgen.
Das Hodgson-Verfahren führt zu {num_tardy_hodgson} verspäteten Aufträgen.
Das Hodgson-Verfahren ist darauf ausgelegt, die *Anzahl* der verspäteten
Aufträge zu minimieren, was hier gelungen ist.
Es kann jedoch vorkommen, dass die maximale oder gesamte Verspätung dadurch
nicht zwingend minimal wird oder sich sogar erhöht.
""".format(num_tardy_edd=num_tardy_edd,
num_tardy_hodgson=num_tardy_hodgson))
```

```
# Gantt-Chart für Hodgson-Plan
fig_task2, ax_task2 = plt.subplots(figsize=(7, 5))
start_time_hodgson = 0
for i, row in df_hodgson_details.iterrows():
    ax_task2.barh(row['Auftrag'], row['Bearbeitungszeit_ap'],
left=start_time_hodgson, edgecolor='black', color='darkcyan')
    # Plantermin als vertikale Linie
    ax_task2.vlines(row['Plantermin_LTp'], ymin=i-0.4, ymax=i+0.4,
color='red', linestyle='--', label='Plantermin' if i == 0 else "")
    # Fertigstellungstermin markieren
    ax_task2.text(row['Fertigstellungszeit_Fp'] + 0.1, i,
f"Fp={row['Fertigstellungszeit_Fp']}", va='center', color='blue')
    if row['Verspaetung_Vp'] > 0:
         ax_task2.text(row['Fertigstellungszeit_Fp'] + 0.1, i-0.25,
f"Vp={row['Verspaetung_Vp']}", va='center', color='red', fontsize=9)
    start_time_hodgson = row['Fertigstellungszeit_Fp']

ax_task2.set_xlabel("Zeit (Tage)")
ax_task2.set_title('Gantt-Diagramm (Hodgson-Verfahren)')
ax_task2.grid(True, axis='x')
ax_task2.invert_yaxis()
handles, labels = ax_task2.get_legend_handles_labels() # Für einmalige
Legende
if handles: # Nur wenn Label gesetzt wurde
    by_label = dict(zip(labels, handles))
    ax_task2.legend(by_label.values(), by_label.keys())
plt.tight_layout()
plt.show()
```

```
Ursprüngliche Auftragsdaten:
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp
0     C01                    3               7
1     C02                    5              10
2     C03                    2               5
3     C04                    6              12
4     C05                    4               8

1. Reine Liefertermin-Regel (EDD):
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp  Fertigstellungszeit_Fp  \
0     C03                    2               5                       2
1     C01                    3               7                       5
2     C05                    4               8                       9
3     C02                    5              10                      14
4     C04                    6              12                      20

   Verspaetung_Vp
0               0
1               0
2               1
3               4
4               8
```

```
Anzahl verspäteter Aufträge (EDD): 3
Maximale Verspätung (EDD): 8
Gesamte Verspätung (EDD): 13

2. Hodgson-Verfahren (Schrittweise):

Iteration 1:
Aktuelle Reihenfolge in R (nach EDD sortiert mit Fp, Vp):
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp  Fp_temp  Vp_temp
0     C03                    2               5        2        0
1     C01                    3               7        5        0
2     C05                    4               8        9        1
3     C02                    5              10       14        4
4     C04                    6              12       20        8
Erster verspäteter Auftrag: C05
Betrachte Teilmenge für Entfernung (bis einschl. erster Verspäteter):
  Auftrag  Bearbeitungszeit_ap
0     C03                    2
1     C01                    3
2     C05                    4
Entferne Auftrag 'C05' (Bearbeitungszeit: 4)
Aktuelles R_set (Original Indizes): [0, 1, 2, 3]
Aktuelles S_set (Original Indizes): [4]

Iteration 2:
Aktuelle Reihenfolge in R (nach EDD sortiert mit Fp, Vp):
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp  Fp_temp  Vp_temp
0     C03                    2               5        2        0
1     C01                    3               7        5        0
2     C02                    5              10       10        0
3     C04                    6              12       16        4
Erster verspäteter Auftrag: C04
Betrachte Teilmenge für Entfernung (bis einschl. erster Verspäteter):
  Auftrag  Bearbeitungszeit_ap
0     C03                    2
1     C01                    3
2     C02                    5
3     C04                    6
Entferne Auftrag 'C04' (Bearbeitungszeit: 6)
Aktuelles R_set (Original Indizes): [0, 1, 2]
Aktuelles S_set (Original Indizes): [4, 3]

Iteration 3:
Aktuelle Reihenfolge in R (nach EDD sortiert mit Fp, Vp):
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp  Fp_temp  Vp_temp
0     C03                    2               5        2        0
1     C01                    3               7        5        0
2     C02                    5              10       10        0
Keine verspäteten Aufträge in R_set gefunden. Hodgson-Algorithmus
abgeschlossen für R_set.

3. & 4. Finaler Ablaufplan nach Hodgson-Verfahren:
```

```
  Auftrag  Bearbeitungszeit_ap  Plantermin_LTp  Fertigstellungszeit_Fp  \
0     C03                    2               5                       2   
1     C01                    3               7                       5   
2     C02                    5              10                      10   
3     C05                    4               8                      14   
4     C04                    6              12                      20   

   Verspaetung_Vp  
0               0  
1               0  
2               0  
3               6  
4               8  
Anzahl verspäteter Aufträge (Hodgson): 2
Maximale Verspätung (Hodgson): 8
Gesamte Verspätung (Hodgson): 14

5. Vergleich:
Die reine Liefertermin-Regel (EDD) führte zu 3 verspäteten Aufträgen.
Das Hodgson-Verfahren führt zu 2 verspäteten Aufträgen.
Das Hodgson-Verfahren ist darauf ausgelegt, die *Anzahl* der verspäteten
Aufträge zu minimieren, was hier gelungen ist.
Es kann jedoch vorkommen, dass die maximale oder gesamte Verspätung dadurch
nicht zwingend minimal wird oder sich sogar erhöht.
```

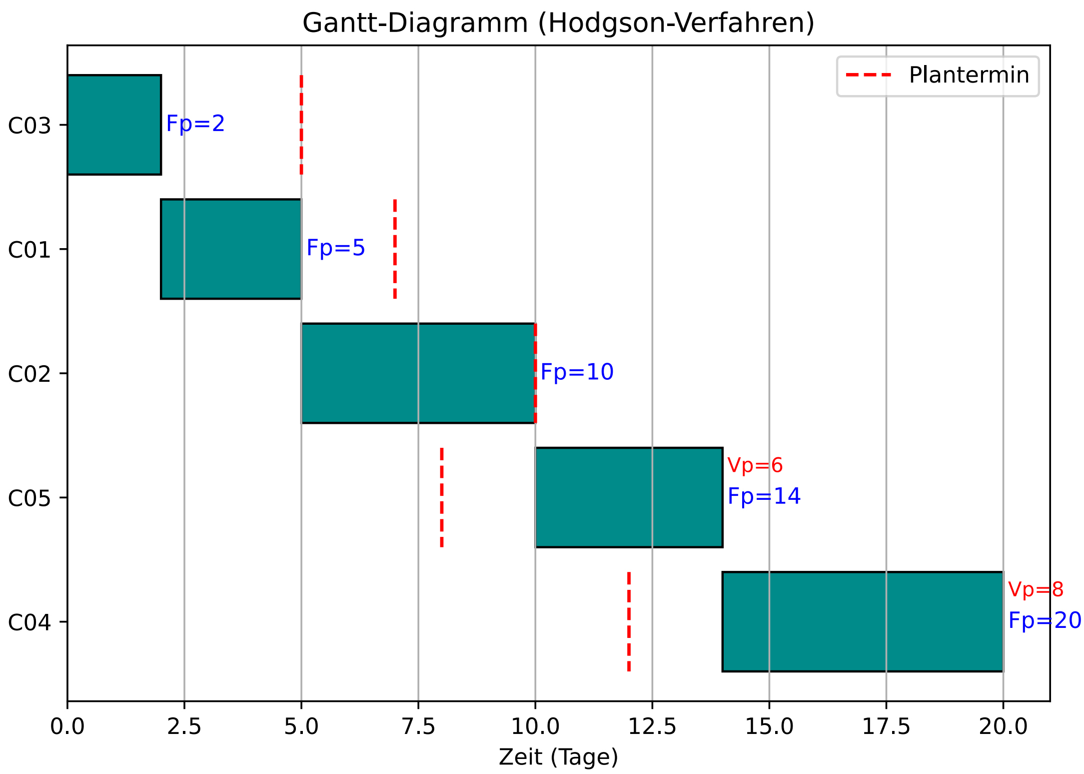

Figure 2: Produktionsplan Komponenten

Aufgabe 3: Produktionsoptimierung

Johnson-Industries arbeitet an der Fertigung von Schlüsselkomponenten für ein neues, fortschrittliches Verteidigungssystem, Projekt "Aegis". Jede Komponente muss zwei Hauptproduktionsstufen durchlaufen: Zuerst eine Präzisionsbearbeitung auf Maschine A und anschließend eine komplexe Montage auf Maschine B. Die Aufträge können nicht überholt werden und die Reihenfolge der Bearbeitung ist auf beiden Maschinen gleich (Flow Shop). Ziel ist es, die Gesamtfertigungszeit für alle anstehenden Komponenten (den Makespan) zu minimieren.

Für die anstehende Produktionscharge sind die Bearbeitungszeiten (in Stunden) für fünf Komponenten bekannt:

Komponente (ID)	Bearbeitungszeit Maschine A (Stunden)	Bearbeitungszeit Maschine B (Stunden)
ARC-01	5	2
REP-02	1	6
UNI-03	9	7
THR-04	3	8
STA-05	10	4

Ihre Aufgaben:

1. Wenden Sie den Johnson-Algorithmus an, um die optimale Auftragsreihenfolge zu bestimmen, die den Makespan minimiert. Dokumentieren Sie Ihre Schritte zur Herleitung der Reihenfolge. Geben Sie die finale, optimale Auftragsreihenfolge an.
2. Erstellen Sie einen detaillierten Belegungsplan (Gantt-Diagramm) für die ermittelte Reihenfolge. Zeichnen Sie die Belegung für Maschine A und Maschine B.
3. Berechnen Sie den minimalen Makespan (Gesamtfertigungszeit) für diese Auftragscharge.

Lösungshinweise:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

plt.rcdefaults()
plt.style.use('tableau-colorblind10')

# Auftragsdaten
data_task3 = {
    'Auftrag': ['ARC-01', 'REP-02', 'UNI-03', 'THR-04', 'STA-05'],
    'Zeit_M1': [5, 1, 9, 3, 10],
    'Zeit_M2': [2, 6, 7, 8, 4]
}
df_task3 = pd.DataFrame(data_task3)

print("Ursprüngliche Auftragsdaten:")
print(df_task3)
```

```
# Johnson-Algorithmus
def johnson_algorithm(df_jobs):
    jobs = df_jobs.copy()
    jobs['Original_Index'] = jobs.index
    n = len(jobs)

    sequence = [None] * n
    front_ptr = 0
    back_ptr = n - 1

    processed_indices = []

    print("\nSchritte des Johnson-Algorithmus:")
    step = 1
    while front_ptr <= back_ptr and len(processed_indices) < n:
        # Filter out already processed jobs
        remaining_jobs =
jobs[~jobs['Original_Index'].isin(processed_indices)].copy()
        if remaining_jobs.empty:
            break

        # Create a list of (time, machine_type (0 for M1, 1 for M2),
original_index) for all remaining operations
        operations = []
        for _, job_row in remaining_jobs.iterrows():
            operations.append((job_row['Zeit_M1'], 0,
job_row['Original_Index']))
            operations.append((job_row['Zeit_M2'], 1,
job_row['Original_Index']))

        operations.sort()  # Sort by time, then machine_type (M1 before M2
for ties in time)

        # Find the first operation belonging to an unprocessed job
        chosen_op_time, chosen_op_machine, chosen_op_job_original_idx =
None, None, None
        for op_time, op_machine, op_job_idx in operations:
            if op_job_idx not in processed_indices:
                chosen_op_time = op_time
                chosen_op_machine = op_machine
                chosen_op_job_original_idx = op_job_idx
                break

        job_to_schedule_name = jobs.loc[chosen_op_job_original_idx,
'Auftrag']

        if chosen_op_machine == 0:  # Min time is on Machine 1
            sequence[front_ptr] = chosen_op_job_original_idx
            print(f"Schritt {step}: Kürzeste Zeit {chosen_op_time}
(Maschine A) für Auftrag {job_to_schedule_name}. Platziere vorn an Position
{front_ptr+1}.")
```

```
            front_ptr += 1
        else:  # Min time is on Machine 2
            sequence[back_ptr] = chosen_op_job_original_idx
            print(f"Schritt {step}: Kürzeste Zeit {chosen_op_time}
(Maschine B) für Auftrag {job_to_schedule_name}. Platziere hinten an
Position {back_ptr+1}.")
            back_ptr -= 1

        processed_indices.append(chosen_op_job_original_idx)
        step += 1

    ordered_df =
jobs.set_index('Original_Index').loc[sequence].reset_index(drop=True)
    return ordered_df

df_johnson_sequence = johnson_algorithm(df_task3)
print("\n2. Finale optimale Auftragsreihenfolge nach Johnson:")
print(df_johnson_sequence[['Auftrag', 'Zeit_M1', 'Zeit_M2']])

# 3. & 4. Belegungsplan und Makespan
def calculate_makespan_and_gantt(df_sequence):
    n_jobs = len(df_sequence)
    m1_finish_times = [0] * n_jobs
    m2_start_times = [0] * n_jobs
    m2_finish_times = [0] * n_jobs

    # Machine 1
    current_m1_time = 0
    for i in range(n_jobs):
        current_m1_time += df_sequence.loc[i, 'Zeit_M1']
        m1_finish_times[i] = current_m1_time

    # Machine 2
    m2_start_times[0] = m1_finish_times[0] # First job on M2 starts after
it finishes M1
    m2_finish_times[0] = m2_start_times[0] + df_sequence.loc[0, 'Zeit_M2']

    for i in range(1, n_jobs):
        m2_start_times[i] = max(m1_finish_times[i], m2_finish_times[i-1])
        m2_finish_times[i] = m2_start_times[i] + df_sequence.loc[i,
'Zeit_M2']

    makespan = m2_finish_times[-1]

    gantt_data = []
    m1_time = 0
    m2_time = 0
    for i in range(n_jobs):
        job_name = df_sequence.loc[i, 'Auftrag']

        # M1
        start_m1 = m1_time
```

```
        duration_m1 = df_sequence.loc[i, 'Zeit_M1']
        gantt_data.append({'Auftrag': job_name, 'Maschine': 'Maschine A',
'Start': start_m1, 'Dauer': duration_m1, 'Ende': start_m1 + duration_m1})
        m1_time += duration_m1

        # M2
        start_m2 = m2_start_times[i]
        duration_m2 = df_sequence.loc[i, 'Zeit_M2']
        gantt_data.append({'Auftrag': job_name, 'Maschine': 'Maschine B',
'Start': start_m2, 'Dauer': duration_m2, 'Ende': start_m2 + duration_m2})

    df_gantt = pd.DataFrame(gantt_data)
    return makespan, df_gantt

makespan_johnson, df_gantt_johnson =
calculate_makespan_and_gantt(df_johnson_sequence)

print(f"\n4. Minimaler Makespan: {makespan_johnson} Stunden")

print("\n3. Belegungsdaten für Gantt-Diagramm:")
print(df_gantt_johnson)

# Gantt-Chart Visualisierung
fig_task3, ax_task3 = plt.subplots(figsize=(7, 5))
colors = {'Maschine A': 'skyblue', 'Maschine B': 'lightcoral'}

job_order_for_plot = df_johnson_sequence['Auftrag'].tolist()

for i, job_name in enumerate(job_order_for_plot):
    # Maschine A
    job_m1_data = df_gantt_johnson[(df_gantt_johnson['Auftrag'] ==
job_name) & (df_gantt_johnson['Maschine'] == 'Maschine A')].iloc[0]
    ax_task3.barh(i - 0.2, job_m1_data['Dauer'], left=job_m1_data['Start'],
color=colors['Maschine A'], edgecolor='black', height=0.4, align='center')
    ax_task3.text(job_m1_data['Start'] + job_m1_data['Dauer']/2, i - 0.2,
f"{job_m1_data['Dauer']}", va='center', ha='center', color='black',
fontsize=8)

    # Maschine B
    job_m2_data = df_gantt_johnson[(df_gantt_johnson['Auftrag'] ==
job_name) & (df_gantt_johnson['Maschine'] == 'Maschine B')].iloc[0]
    ax_task3.barh(i + 0.2, job_m2_data['Dauer'], left=job_m2_data['Start'],
color=colors['Maschine B'], edgecolor='black', height=0.4, align='center')
    ax_task3.text(job_m2_data['Start'] + job_m2_data['Dauer']/2, i + 0.2,
f"{job_m2_data['Dauer']}", va='center', ha='center', color='black',
fontsize=8)

# Create custom y-labels for jobs to represent the sequence
y_labels_display = [f"{job}" for job in job_order_for_plot]
y_ticks_positions = np.arange(len(job_order_for_plot))

ax_task3.set_yticks(y_ticks_positions)
```

```python
ax_task3.set_yticklabels(y_labels_display)
ax_task3.invert_yaxis() # Display jobs from top to bottom in sequence order

ax_task3.set_xlabel("Zeit (Stunden)")
ax_task3.set_ylabel("Auftrag (Bearbeitungsreihenfolge)")
ax_task3.set_title(f'Gantt-Diagramm Johnson-Industries (Makespan:
{makespan_johnson}h)')

# Create custom legend
from matplotlib.patches import Patch
legend_elements = [Patch(facecolor=colors['Maschine A'], edgecolor='black',
label='Maschine A'),
                   Patch(facecolor=colors['Maschine B'], edgecolor='black',
label='Maschine B')]
ax_task3.legend(handles=legend_elements, loc='upper right')

ax_task3.grid(True, axis='x', linestyle=':')
plt.tight_layout()
plt.show()
```

```
Ursprüngliche Auftragsdaten:
  Auftrag  Zeit_M1  Zeit_M2
0  ARC-01        5        2
1  REP-02        1        6
2  UNI-03        9        7
3  THR-04        3        8
4  STA-05       10        4

Schritte des Johnson-Algorithmus:
Schritt 1: Kürzeste Zeit 1 (Maschine A) für Auftrag REP-02. Platziere vorn
an Position 1.
Schritt 2: Kürzeste Zeit 2 (Maschine B) für Auftrag ARC-01. Platziere
hinten an Position 5.
Schritt 3: Kürzeste Zeit 3 (Maschine A) für Auftrag THR-04. Platziere vorn
an Position 2.
Schritt 4: Kürzeste Zeit 4 (Maschine B) für Auftrag STA-05. Platziere
hinten an Position 4.
Schritt 5: Kürzeste Zeit 7 (Maschine B) für Auftrag UNI-03. Platziere
hinten an Position 3.

2. Finale optimale Auftragsreihenfolge nach Johnson:
  Auftrag  Zeit_M1  Zeit_M2
0  REP-02        1        6
1  THR-04        3        8
2  UNI-03        9        7
3  STA-05       10        4
4  ARC-01        5        2

4. Minimaler Makespan: 30 Stunden

3. Belegungsdaten für Gantt-Diagramm:
```

```
  Auftrag    Maschine  Start  Dauer  Ende
0  REP-02  Maschine A      0      1     1
1  REP-02  Maschine B      1      6     7
2  THR-04  Maschine A      1      3     4
3  THR-04  Maschine B      7      8    15
4  UNI-03  Maschine A      4      9    13
5  UNI-03  Maschine B     15      7    22
6  STA-05  Maschine A     13     10    23
7  STA-05  Maschine B     23      4    27
8  ARC-01  Maschine A     23      5    28
9  ARC-01  Maschine B     28      2    30
```

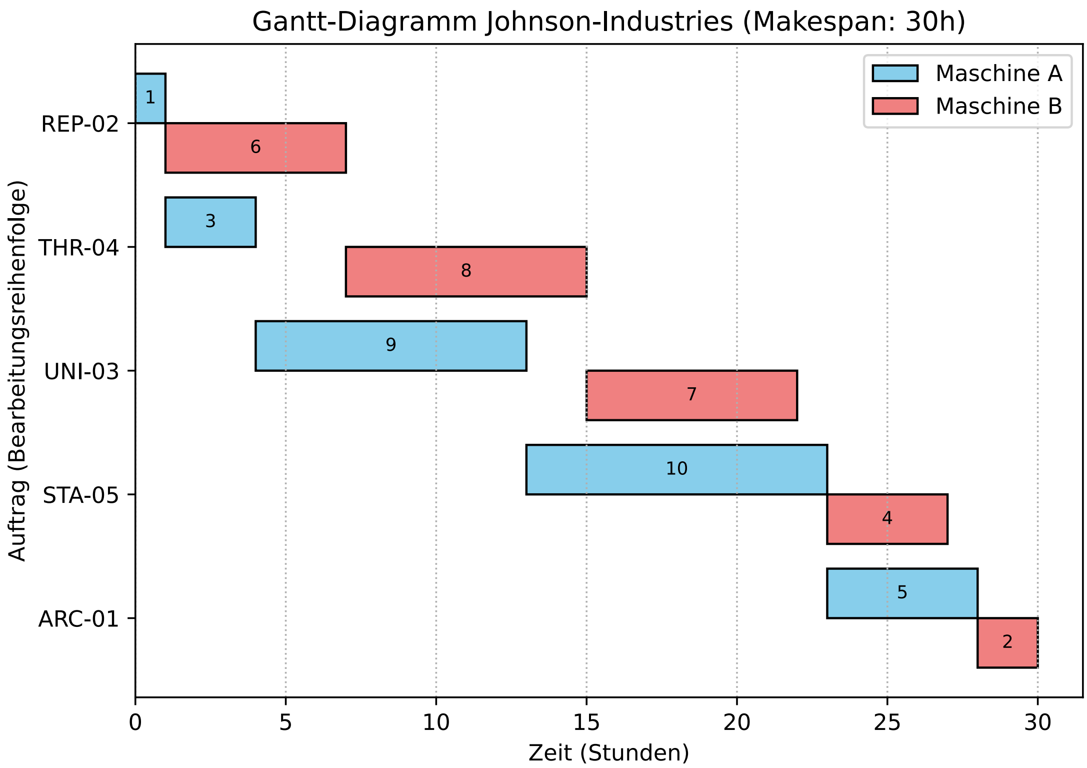

Figure 3: Produktionsplan Johnson-Industries (Johnson-Algorithmus)