Prescriptive Analytics, Hausaufgabe 2

Henry Haustein

Aufgabe 1: First Things First

(a) den Code kann man mehr oder weniger aus dem Seminar abschreiben und ein bisschen anpassen

```
# Schreiben Sie hier Ihren Programmcode
from InputData import *
from OutputData import *
from EvaluationLogic import *

data = InputData("VFR10_10_10_SIST.json")
reihenfolge = [7, 6, 8, 10, 2, 9, 1, 4, 3, 5]
recommendedSolution = Solution(data.InputJobs, [x-1 for x in
    reihenfolge])
EvaluationLogic().DefineStartEnd(recommendedSolution)
print(recommendedSolution)
```

(b) auch hier nichts kompliziertes

```
# Schreiben Sie hier Ihren Programmcode
simpleSolution = Solution(data.InputJobs, range(10))
EvaluationLogic().DefineStartEnd(simpleSolution)
print(f"The objective delta is {recommendedSolution.Makespan -
     simpleSolution.Makespan}.")
```

(c) Wir gehen durch alle Jobs durch und schauen ob die Endzeit auf der letzten Maschine größer als der DueDate ist. Wenn ja, ist dieser Job verspätet.

```
# Schreiben Sie hier Ihren Programmcode
def CalculateTardyJobs(sol):
  tardyJobs = 0
  for job in sol.OutputJobs.values():
    end = job.EndTimes[-1]
    dueDate = job.DueDate
    if end > dueDate:
      tardyJobs += 1
  return tardyJobs

print(f"Tardy Jobs of simplesolution: {CalculateTardyJobs(
    simpleSolution)}.")
print(f"Tardy Jobs of recommendedSolution: {CalculateTardyJobs
    (recommendedSolution)}.")
```

Aufgabe 2: Zulässigkeitsprüfung

```
1  # Schreiben Sie hier Ihren Programmcode
2  def ValidateSolution(sol):
3    errors = []
4
5    # wrong number of IDs
6    if len(sol.Permutation) != 10:
7      errors.append("Detected wrong number of IDs in solution
           representation")
8
9    # duplicate IDs
10   if len(sol.Permutation) != len(set(sol.Permutation)):
11     errors.append("Detected duplicate IDs in solution
           representation")
12
13   # Überschneidung von Start- und Endzeit
14   ueberschneidung = False
15   for job in sol.OutputJobs.values():
16     for start, ende in zip(job.StartTimes, job.EndTimes):
17       if ende < start:
18         ueberschneidung = True
19         break
20   if ueberschneidung:
21     errors.append("Start- und Endzeiten überschneiden sich")
22
23   # Output
24   print("Reported Corruptions:")
25   if len(errors) == 0:
26     print("\t - No corruptions detected")
27   else:
28     for error in errors:
29       print("\t - " + error)
30
31 # Führen Sie im Anschluss den folgenden Code aus
32 ValidateSolution(simpleSolution)
33 ValidateSolution(Solution(data.InputJobs, [0, 3, 2, 3, 1, 4, 5,
       8]))
34 ValidateSolution(Solution(data.InputJobs, [0, 3, 2, 1, 4, 5, 8, 7,
        10, 9, 11, 12]))
```

Wir untersuchen hier die folgenden 3 Fehler, und sammeln alle Fehler in einer Liste, die wir am Ende ausgeben:

- Wenn die Permutation nicht 10 Zahlen enthält, so gibt es einen Fehler. Die 10 ist hier fest, weil ich weiß, dass es 10 Jobs gibt, aber wenn man eine andere json-Datei einließt, so kann es eventuell mehr Jobs geben. Dann müsste man die 10 hier anpassen (oder einfach besseren Code schreiben, der automatisch die korrekte Anzahl ermittelt unabhängig von der json-Datei).
- Eine sehr elegante Methode um auf Dopplungen in Listen zu prüfen, ist die Liste in ein set (= Menge) umzuwandeln. Ein set kann nämlich nur jeden Eintrag einmal enthalten und wirft deswegen Dopplungen direkt raus. Bei einem Vergleich der Länge des sets und der originalen Liste kann man

herausfinden ob und wie viele Dopplungen es gab.

- Um zu untersuchen, ob sich die Start- und Endzeiten eines Jobs überschneiden, gehen wir einfach durch alle Jobs durch und schauen, ob die Startzeit nach der Endzeit auf jeder Maschine ist. Sollte das der Fall sein, können wir die Suche abbrechen und ausgeben, dass es Überschneidungen gibt.

Zuletzt geben wir noch die Fehler alle aus. Das \t stellt einen Tab dar, somit ist der Output etwas eingerückt.

Aufgabe 3: Die CDS Heuristik von Campell, Dudek & Smith

```
1  import pandas as pd
2
3  def CampellDudekSmith2(inputJobs):
4    # Matrix aufbauen
5    processingTimes = []
6    for job in inputJobs:
7      processingTimes.append([operation[1] for operation in job.
           Operations]) # Extrahieren der ProcessingTimes für jeden
           Job
8    df = pd.DataFrame(processingTimes)
9
10   # Variablen setzen
11   solutions = []
12   m = len(inputJobs[0].Operations) # Anzahl Maschinen
13   n = len(inputJobs) # Anzahl Jobs
14   p = m - 1
15   for k in range(1, p + 1):
16     processingTimesM1 = df.iloc[:,0:k].sum(axis = 1) # Zeilenweise
           Summe der ersten k Spalten
17     processingTimesM2 = df.iloc[:,n-k:n].sum(axis = 1) #
           Zeileweise Summe der letzten k Spalten
18     TwoMaschineProblem = pd.DataFrame([processingTimesM1,
           processingTimesM2]).transpose() # Zusammenfügen der beiden
           Spalten in ein DataFrame, Transponieren nötig, damit
           DataFrame zwei Spalten statt 2 Zeilen enthält
19     sol = Solution(inputJobs, solve2MaschineNJobsProblem(
           TwoMaschineProblem))
20     EvaluationLogic().DefineStartEnd(sol)
21     solutions.append(sol)
22
23   # finde beste Lösung
24   bestSol = None
25   for solution in solutions:
26     if bestSol == None or bestSol.Makespan > solution.Makespan:
27     bestSol = solution
28
29   return bestSol
30
31 def solve2MaschineNJobsProblem(df):
32   vorderePlaetze = []
33   hinterePlaetze = []
34   for n in range(len(df)):
```

```
35      min1 = df[0].min() # Minimum der ersten Spalte
36      min2 = df[1].min() # Minimum der zweiten Spalte
37      if min1 < min2:
38        # Ist das Minimum in der ersten Spalte zu finden, dann wird
              der Job vorne eingefügt
39        vorderePlaetze.append(df[0].idxmin())
40        df.drop(df[0].idxmin(), inplace = True) # Löschen des Jobs
41      else:
42        # Ist das Minimum in der zweiten Spalte zu finden, dann wird
              der Job hinten eingefügt
43        hinterePlaetze.append(df[1].idxmin())
44        df.drop(df[1].idxmin(), inplace = True) # Löschen des Jobs
45
46  return vorderePlaetze + hinterePlaetze[::-1]
47
48  print(CampellDudekSmith2(data.InputJobs))
49
50  # solve2MaschineNJobsProblem(pd.DataFrame
        ([[186,160],[152,272],[41,153],[61,175],[227,172],[67,219],[81,197],[175,219]
        ) # zum Testen aus dem Paper
```

Wir arbeiten das Flowchart aus dem Paper ab. Zuerst erstellen wir die Matrix der ProcessingTimes und speichern sie in der Variable `df`. Die Verwendung von Pandas bringt einige Vorteile, wie einfaches Auswählen von Spalten und Summation.

Im Paper kann man p Reihenfolgen berechnen lassen und wählt dann davon die beste aus. Man kann $p \leq m-1$ wählen, wir wählen $p = m-1$. Wir lassen dann k von 1 bis p laufen (`range(1,p+1)` liefert [1, 2, ..., p]) und für jedes k bauen wir ein 2-Maschinen-Problem. Die Lösung dieses Problems speichern wir dann in der `solutions`-Liste ab. Das k-te 2-Maschinen-Problem sieht so aus, dass wir die Summe der ersten k Spalten als M1 und die Summe der letzten k Spalten als M2 definieren. Haben wir alle k Lösungen, so können wir die beste Lösung anhand der Makespan suchen und zurückgeben.

Um das 2-Maschinen-Problem zu lösen (↗ Produktion und Logistik) erwarten wir ein Dataframe mit 2 Spalten (für die 2 Maschinen) und n Zeilen (für die Aufträge, die auf beiden Maschinen laufen müssen). Wir vergleichen die kleinsten Prozesszeiten der ersten und zweiten Maschine. Ist das Minimum auf der ersten Maschine, so wird der Auftrag vorne eingefügt; liegt das Minimum auf der zweiten Maschine, so wird der Auftrag hinten eingefügt. Anschließend wird der Auftrag gelöscht. Das Zusammensetzen der vorderen Plätze und der hinteren Plätze liefert dann die optimale Reihenfolge.