Prescriptive Analytics, Seminar 3

Henry Haustein

```python
# files from first session
from InputData import InputData
from OutputData import OutputJob
# Additionally
import numpy
```

Aufgabe 1: Codierung und Bewertung einer Lösung

(a) Klasse `Solution`

```python
class Solution:
  OutputJobs = {}
  Makespan = None
  TotalTardiness = None
  TotalWeightedTardiness = None
  Permutation = None

  def __init__(self, inputJobs, reihenfolge):
    self.Makespan = -1
    self.TotalTardiness = -1
    self.TotalWeightedTardiness = -1
    self.Permutation = reihenfolge

    for idx, value in enumerate(inputJobs):
      self.OutputJobs[idx] = OutputJob(value)

  def __str__(self):
    return f"Solution({self.OutputJobs = }, {self.Makespan =
        }, {self.TotalTardiness = }, {self.
        TotalWeightedTardiness = }, {self.Permutation = })"

#### Führen Sie im Anschluss folgenden Code aus ####
data = InputData("InputFlowshopSIST.json")
Permutation = [x-1 for x in [6,5,7,4,8,3,9,2,10,1,11]]
DevilSolution = Solution(data.InputJobs, Permutation)
print(DevilSolution)
```

(b) Klasse `EvaluationLogic`. Diese Klasse ist ziemlich sinnlos, sie enthält keine Attribute, sondern nur Funktionen

```
class EvaluationLogic:
  def DefineStartEnd(self, currentSolution):
    #####
    # schedule first job: starts when finished at previous stage
    firstJob = currentSolution.OutputJobs[currentSolution.Permutation[0]]
    firstJob.EndTimes = numpy.cumsum([firstJob.ProcessingTime(x) for x in range(len(firstJob.EndTimes))])
    firstJob.StartTimes[1:] = firstJob.EndTimes[:-1]
    #####
    # schedule further jobs: starts when finished at previous stage and the predecessor is no longer on the considered machine
    for j in range(1,len(currentSolution.Permutation)):
      currentJob = currentSolution.OutputJobs[currentSolution.Permutation[j]]
      previousJob = currentSolution.OutputJobs[currentSolution.Permutation[j-1]]
      # first machine
      currentJob.StartTimes[0] = previousJob.EndTimes[0]
      currentJob.EndTimes[0] = currentJob.StartTimes[0] + currentJob.ProcessingTime(0)
      # other machines
      for i in range(1,len(currentJob.StartTimes)):
        currentJob.StartTimes[i] = max(previousJob.EndTimes[i], currentJob.EndTimes[i-1])
        currentJob.EndTimes[i] = currentJob.StartTimes[i] + currentJob.ProcessingTime(i)
    #####
    # Save Makespan and return Solution
    currentSolution.Makespan = currentSolution.OutputJobs[currentSolution.Permutation[-1]].EndTimes[-1]

EvaluationLogic().DefineStartEnd(DevilSolution)
print(DevilSolution.Makespan)
```

(c) Die Dokumentation des csv-Moduls gibt es hier: https://docs.python.org/3/library/csv.html

```
import csv

def WriteSolToCsv(self):
  with open("output.csv", "w") as file:
    writer = csv.writer(file)
    writer.writerow(["Machine", "Job", "Start_Setup", "End_Setup", "Start", "End"])
    for maschine in range(5):
      for jobID in self.Permutation:
        currentJob = self.OutputJobs[jobID]
        writer.writerow([maschine + 1, jobID, currentJob.StartSetups[maschine], currentJob.EndSetups[maschine], currentJob.StartTimes[maschine],
```

```
                currentJob.EndTimes[maschine]])
11
12 setattr(Solution, "WriteSolToCsv", WriteSolToCsv)
13
14 DevilSolution.WriteSolToCsv()
```

(d) Super simpel, wenn man weiß, was man machen muss. Ein bisschen Dokumentation hätte dem externen Quellcode schon gut getan

```
1 from Gantt import *
2
3 GanttChartFromCsv("output.csv", "gantt.html")
```

Aufgabe 2: Dispatching Rules zur Erstellung von Lösungen

(a) Es gibt viele Funktionen die eine zufällige Permutation erzeugen. Ich habe mich hier für `numpy.random.shuffle()` entschieden.

```
1  def ROS(anzahlFolgen, inputData):
2    bestSolution = None
3    reihenfolge = numpy.arange(11)
4    for i in range(anzahlFolgen):
5      numpy.random.shuffle(reihenfolge)
6      sol = Solution(inputData.InputJobs, reihenfolge)
7      EvaluationLogic().DefineStartEnd(sol)
8      if bestSolution == None or bestSolution.Makespan > sol.
           Makespan:
9        bestSolution = sol
10
11   print("Reihenfolge " + str(bestSolution.Permutation) + "
         ergibt Makespan " + str(bestSolution.Makespan))
12   return sol
13
14 ROS(50, data)
```

(b) Wir kombinieren `ROS` mit `Itertools` um alle Permutationen durchlaufen zu lassen:

```
1  from itertools import permutations
2
3  def checkAllPermutations(inputData):
4    bestSolution = None
5    allPermutations = permutations(range(11))
6    print("Rechenzeit: " + str(len(allPermutations) *
         0.0011259999999992942 / 60) + " Minuten")
7    for reihenfolge in allPermutations:
8      sol = Solution(inputData.InputJobs, reihenfolge)
9      EvaluationLogic().DefineStartEnd(sol)
10     if bestSolution == None or bestSolution.Makespan > sol.
           Makespan:
11       bestSolution = sol
12
```

```
13    print("Reihenfolge " + str(bestSolution.Permutation) + "
          ergibt Makespan " + str(bestSolution.Makespan))
14    return sol
```

Um die Rechenzeit halbwegs exakt abgeben zu können, müssen wir ermitteln, wie lange es für eine Permutation dauert. Dafür gibt es `time`:

```
1  import time
2
3  start = time.process_time()
4  reihenfolge = range(11)
5  sol = Solution(data.InputJobs, reihenfolge)
6  EvaluationLogic().DefineStartEnd(sol)
7  print(time.process_time() - start)
```

(c) First-come-first-serve ist ziemlich einfach. Die Reihenfolge ist einfach 1, 2, 3, ...

```
1  def FCFS(inputData):
2    reihenfolge = range(11)
3    sol = Solution(inputData.InputJobs, reihenfolge)
4    EvaluationLogic().DefineStartEnd(sol)
5
6    return sol
7
8  FCFSSol = FCFS(data)
9  print(FCFSSol)
```

Für Shortest-Processing-Time müssen wir erst alle ProcessingTimes für jeden Job aufaddieren und danach sortieren:

```
1  def SPT(inputData):
2    procTimes = {}
3    for job in inputData.InputJobs:
4      procTimes[job.JobId] = sum(value[1] for value in job.
           Operations)
5    reihenfolge = {k-1: v for k, v in sorted(procTimes.items(),
         key=lambda item: item[1])}
6    print(reihenfolge)
7    sol = Solution(inputData.InputJobs, list(reihenfolge.keys())
         )
8    EvaluationLogic().DefineStartEnd(sol)
9
10   return sol
11
12 SPTSol = SPT(data)
13 print(SPTSol)
```

Longest-Processing-Time ist fast das selbe, nur, dass wir hier anders herum sortieren:

```
1  def LPT(inputData):
2    procTimes = {}
3    for job in inputData.InputJobs:
4      procTimes[job.JobId] = sum(value[1] for value in job.
           Operations)
```

```
    reihenfolge = {k-1: v for k, v in sorted(procTimes.items(),
        key=lambda item: item[1], reverse = True)}
    print(reihenfolge)
    sol = Solution(inputData.InputJobs, list(reihenfolge.keys())
        )
    EvaluationLogic().DefineStartEnd(sol)

    return sol

LPTSol = LPT(data)
print(LPTSol)
```

(d) NEH-Heuristik war schon gegeben

(e) Wieder so eine völlig sinnlose Klasse mit nur einer Funktion

```
class ConstructiveHeuristic:
  inputData = None

  def __init__(self, inputData):
    self.inputData = inputData

  def useRule(self, name):
    if name  == "ros":
      return ROS(10, self.inputData)
    if name == "fcfs":
      return FCFS(self.inputData)
    if name == "spt":
      return SPT(self.inputData)
    if name == "lpt":
      return LPT(self.inputData)
    if name == "neh":
      return NEH(self.inputData)
```

Aufgabe 3: Liefertreue

(a) schon gegeben

(b) DueDates sammeln und nach diesen sortieren

```
def EDD(inputData):
  dueTimes = {}
  for job in inputData.InputJobs:
    dueTimes[job.JobId] = job.DueDate
  reihenfolge = {k-1: v for k, v in sorted(dueTimes.items(),
      key=lambda item: item[1])}
  print(reihenfolge)
  sol = Solution(inputData.InputJobs, list(reihenfolge.keys())
      )
  EvaluationLogic().DefineStartEnd(sol)
  EvaluationLogic().CalculateTardiness(sol)

```

```
  return sol

SPTSol = SPT(data)
print(SPTSol)
```