Aufgabe 2: Vollgeladen

Team-ID: 01048

Team: Lorian

Bearbeiter/-innen dieser Aufgabe:
Lorian Linnertz

17. November 2021

Inhaltsverzeichnis

Lösungsidee

Meine Lösungsidee ist, dass man durch alle Möglichkeiten durch iteriert und anschliessend die beste Route als Lösung nehmen kann. Das Problem ist jedoch, dass die Möglichkeiten exponentiel zunehmen und somit die Berechnungsdauer ins ewige steigen kann. Um dies zu verhindern limitier man die Anzahl der Iterationen, normaler Wert ist 10(Dies reicht für die gegeben Aufgaben aus, alles über diesem Wert kommt zum gleichen Resultat braucht jedoch wesntlich mehr rechenleistung). Bedeutet dass für jeden Tag 10 Möglichkeiten berechnet werden. Die 10 Möglichkeiten werden im ersten durch lauf anhand ihrer Bewertung aus gesucht, bedeutet dass die 10 am besten bewertetsten Hotels genommen werden. Sollte der erste Durchlauf ohne Erfolg sein(also falls keine Route gefunden wird), sollte dies der Fall sein so wird im zweiten Durchlauf die 10 am besten gelegenen Hotels genommen.

Diese Methode findet immer einen Weg und in den allermeisten Fälle den besten Weg, Ausnahme ist, wenn zum Beispiel die Hotels so positioniert sind dass man am Anfang ein Hotel wählen muss mit sehr niedriger Bewertung und sehr schlechter Positionierung. Dieses Problem kann man jedoch beheben mit der Erhöhung der Iterationen.

Umsetzung

Das Programm basiert auf drei Klassen, eine Klasse erstellt jeweils Objekte des Types Hotel, eine Zweite erstellt die Wege und berechnet die Durchschnittsbewertung und in der letzten Klasse

befinden sich 4 Methoden, welche den Namen Day1, Day2,Day3,Day4 tragen. Die Methode 1 iteriert dabei durch seine 30 Möglichkeiten und gibt diese an Day 2. Dies Wiederholt sich solange bis Day 4. In Day4 wird zum einen Überprüft ob die Familie überhaupt ankommen würde und zum anderen wird Mithilfe der 2. Klasse ein Weg erstellt. Dieser Weg wird wieder an Day3 zurückgegeben, welcher wieder den besten Weg aus Day 3 an Day2 weiter gibt, usw. An Day1 wird dann schlussendlich der beste Weg zurückgegeben, sollte es keinen besten Weg geben, so wird der zweite Durchgang gestartet

Beispiele

Für die Reise von "HOTELS1.TXT" sind folgende Hotels am besten geeignet, mit einer Durchschnittsbewertung von 3.17/5:
Am 1.Tag: Das 4.Hotel, Distanz: 347, Bewertung 2.7/5
Am 2.Tag: Das 8.Hotel, Distanz: 687, Bewertung 4.4/5
Am 3.Tag: Das 9.Hotel, Distanz: 1007, Bewertung 2.8/5
Am 4.Tag: Das 12.Hotel, Distanz: 1360, Bewertung 2.8/5

Für die Reise von "HOTELS2.TXT" sind folgende Hotels am besten geeignet, mit einer Durchschnittsbewertung von 3.77/5:
Am 1.Tag: Das 4.Hotel, Distanz: 341, Bewertung 2.3/5
Am 2.Tag: Das 11.Hotel, Distanz: 700, Bewertung 3.0/5
Am 3.Tag: Das 16.Hotel, Distanz: 1053, Bewertung 4.8/5
Am 4.Tag: Das 26.Hotel, Distanz: 1380, Bewertung 5.0/5

Für die Reise von "HOTELS3.TXT" sind folgende Hotels am besten geeignet, mit einer Durchschnittsbewertung von 2.6/5:
Am 1.Tag: Das 99.Hotel, Distanz: 359, Bewertung 4.6/5
Am 2.Tag: Das 197.Hotel, Distanz: 717, Bewertung 0.3/5
Am 3.Tag: Das 299.Hotel, Distanz: 1076, Bewertung 3.8/5
Am 4.Tag: Das 402.Hotel, Distanz: 1433, Bewertung 1.7/5

Für die Reise von "HOTELS4.TXT" sind folgende Hotels am besten geeignet, mit einer Durchschnittsbewertung von 4.65/5:
Am 1.Tag: Das 70.Hotel, Distanz: 249, Bewertung 4.8/5
Am 2.Tag: Das 190.Hotel, Distanz: 605, Bewertung 4.9/5
Am 3.Tag: Das 287.Hotel, Distanz: 918, Bewertung 4.6/5
Am 4.Tag: Das 412.Hotel, Distanz: 1260, Bewertung 4.3/5

Für die Reise von "HOTELS5.TXT" sind folgende Hotels am besten geeignet, mit einer Durchschnittsbewertung von 5.0/5:
Am 1.Tag: Das 243.Hotel, Distanz: 280, Bewertung 5.0/5
Am 2.Tag: Das 582.Hotel, Distanz: 636, Bewertung 5.0/5
Am 3.Tag: Das 914.Hotel, Distanz: 987, Bewertung 5.0/5
Am 4.Tag: Das 1169.Hotel, Distanz: 1271, Bewertung 5.0/5

Quellcode

```
class Hotel:

    def __init__(self,distance,rating,index):

        self.distance = distance

        self.rating = rating

        self.index = index

        self.deltaDistance = distance #fEhler anfällig nicht verwenden



    def get_deltaDistance(self, lastDistance):

        self.deltaDistance = self.distance - lastDistance
```

```
        return self.deltaDistance


class GetPathRating: #Diese Klasse dient dem Speichern der Wege
    def __init__(self,HotelsList):
        self.HotelsList = HotelsList
        self.HotelsNum = len(self.HotelsList)
        self.averageRating = GetPathRating.get_AverageRating(self)


    def get_AverageRating(self):
        Sum = 0
        for i in range(0,self.HotelsNum):
            Sum += self.HotelsList[i].rating
        return Sum / self.HotelsNum


class BestPathRating: #Dies ist die Hauptklasse
    def __init__(self, HotelsList, entireDistance,maxIterations=30):
        self.hList = HotelsList
        self.entireDistance = entireDistance
        self.maxIterations = maxIterations
        self.mode = 0
        self.running_Variable = [0,BestPathRating.adjust_maxIterations(self)]
        self.CatchErrorsPath = GetPathRating([Hotel(0,0,0)]) #dieser Path soll Errors verhindern, welche auftreten könnten, falls eine abzweigung keine mögliche Lösung zurück gibt.
        self.bestRatedPath = BestPathRating.Day1(self)
        BestPathRating.Control_bestRatedPath(self)



    def Control_bestRatedPath(self):
        if self.bestRatedPath == self.CatchErrorsPath:
            self.mode = 1
            self.running_Variable[0] = 0
            PathRating = BestPathRating.Day1(self)
            self.bestRatedPath = PathRating
            if PathRating == self.CatchErrorsPath:
                print("\nes existiert keine Reiseroute mit ihren Eingaben\n")



    def Day1(self):
```

```
    Day1HotelsList = BestPathRating.getList(self,None) #erstellt eine Liste mit allen Hotels, welche für Tag 1 infrage kommen und speichert diese in Day1HotelsList

    Day1HotelsList = BestPathRating.optimizeList(self, Day1HotelsList)

    PathRatingsFromDay1 = []

    for h in Day1HotelsList:

      print("\n",30*"-",f"\n {(Day1HotelsList.index(h)+1)} / {len(Day1HotelsList)}","\n",30*"-","\n")

      previousHotels = [] #erstellt eine Liste mit den Vorherigen Hotels, diese soll im Laufe der Schleife immer wieder angepasst und erweitert werden

      previousHotels.append(h) #fügt als erstes das Ursprungs Hotel h hinzu

      if BestPathRating.TomorrowAtDestination(self,None) == False: #überprüft ob wir morgen bereits am Ziel ankommen, sollte das nicht der Fall sein:

        PathRating = BestPathRating.Day2(self,previousHotels)

        PathRatingsFromDay1.append(PathRating) #so werden die besten Hotels für den nächsten Tag gesucht, und die jeweils besten in die Liste PathRatingsFromDay1 hinzugefügt

      else:

        PathRating = GetPathRating(previousHotels)

        PathRatingsFromDay1.append(PathRating)

    return  BestPathRating.Find_BestRatingInList(self,PathRatingsFromDay1)



  def Day2(self,previousHotels):

    Day2HotelsList = BestPathRating.getList(self,previousHotels[-1]) #erstellt eine Liste mit allen Hotels, welche für Tag 2 infrage kommen previousHotels[-1] ist äquivalent mit dem letzten Hotel

    Day2HotelsList = BestPathRating.optimizeList(self, Day2HotelsList)

    PathRatingsFromDay2 = [self.CatchErrorsPath]

    for h in Day2HotelsList: #diese Funktion soll die besten Hotels für PathRatingsFromDay2 finden

      preHotels = []

      preHotels.extend(tuple(previousHotels))

      preHotels.append(h) #fügt zusätzlich das Hotel h hinzu

      if BestPathRating.TomorrowAtDestination(self,h) == False: #überprüft ob wir morgen bereits am Ziel ankommen, sollte das nicht der Fall sein: #lastHotel ist in diesem Fall h

        PathRatingsFromDay2.append(BestPathRating.Day3(self,preHotels)) #so werden die besten Hotels für den nächsten Tag gesucht und zur Liste der PathRatingsFromDay2 hinzugefügt

      else:

        PathRating = GetPathRating(preHotels)

        PathRatingsFromDay2.append(PathRating)

      print(f"{round(self.running_Variable[0] / self.running_Variable[1] *100,2)}% | {self.running_Variable[0]} / {self.running_Variable[1]}")

      self.running_Variable[0] += 1


    return BestPathRating.Find_BestRatingInList(self,PathRatingsFromDay2) #gibt den Weg mit dem besten Rating an die Funktion Day1 zurück
```

```
    def Day3(self,previousHotels):

        Day3HotelsList = BestPathRating.getList(self,previousHotels[-1]) #erstellt eine Liste mit allen Hotels, welche für Tag 3 infrage kommen previousHotels[-1] ist äquivalent mit dem letzten Hotel

        Day3HotelsList = BestPathRating.optimizeList(self, Day3HotelsList)

        PathRatingsFromDay3 = [self.CatchErrorsPath]

        for h in Day3HotelsList: #diese Funktion soll die besten Hotels für PathRatingsFromDay3 finden

            preHotels = []

            preHotels.extend(tuple(previousHotels))

            preHotels.append(h) #fügt zusätzlich das Hotel h hinzu

            if BestPathRating.TomorrowAtDestination(self,h) == False: #überprüft ob wir morgen bereits am Ziel ankommen, sollte das nicht der Fall sein: #lastHotel ist in diesem Fall h

                PathRatingsFromDay3.append(BestPathRating.Day4(self,preHotels)) #so werden die besten Hotels für den nächsten Tag gesucht und zur Liste der PathRatingsFromDay3 hinzugefügt

            else:

                PathRating = GetPathRating(preHotels)

                PathRatingsFromDay3.append(PathRating)


        return BestPathRating.Find_BestRatingInList(self,PathRatingsFromDay3) #gibt den Weg mit dem besten Rating an die Funktion Day2 zurück



    def Day4(self,previousHotels):

        Day4HotelsList = BestPathRating.getList(self,previousHotels[-1]) #erstellt eine Liste mit allen Hotels, welche für Tag 4 infrage kommen previousHotels[-1] ist äquivalent mit dem letzten Hotel

        Day4HotelsList = BestPathRating.optimizeList(self, Day4HotelsList)

        PathRatingsFromDay4 = [self.CatchErrorsPath]

        for h in Day4HotelsList: #diese Funktion soll die besten Hotels für PathRatingsFromDay4 finden

            preHotels = []

            preHotels.extend(tuple(previousHotels))

            preHotels.append(h) #fügt zusätzlich das Hotel h hinzu

            if BestPathRating.TomorrowAtDestination(self,h): #überprüft ob wir morgen am Ziel ankommen, sollte das  der Fall sein: #lastHotel ist in diesem Fall h

                PathRating = GetPathRating(preHotels)

                PathRatingsFromDay4.append(PathRating) #so wird der Weg zur Liste PathRatingsFromDay4 hinzugefügt

            else: #Falls wir am nächsten Tag nich da wären, so würden wir die Deadline sprengen

                continue
```

```
    return BestPathRating.Find_BestRatingInList(self,PathRatingsFromDay4) #gibt den Weg mit dem besten Rating an die Funktion Day3 zurück



  def getList(self,lastHotel): #gibt eine Liste zurück mit allen infrage kommenden Hotels
    returnList = []
    if lastHotel == None: #Falls lastHotel == None ist, so wird lastHotelDistance auf 0 gesetzt, dies wird dazu gebraucht, da es an Tag1 nach kein letztesHotel gibt
      lastHotelDistance=0
    else:
      lastHotelDistance = lastHotel.distance
    for hotel in self.hList: #iteriert durch alle Hotels aus der Liste hList durch
      if hotel.distance > lastHotelDistance and hotel.get_deltaDistance(lastHotelDistance) in range(1,361): #falls das Hotel weiter entfernt ist als das letzte Hotel und wenn das Hotel zwischen 1 und 360 Minuten vom letzten entfernt ist dann:
        returnList.append(hotel) #wird es zur Liste hinzugefügt
    return returnList


  def optimizeList(self, hotelsList):
    if self.mode == 0: #mode 0 ist der Normale Modus welcher beim ersten Durchlauf verwendet wird
      inputList = []
      for hotel in hotelsList:
        inputList.append(hotel.rating) #erstellt eine Liste mit den Ratings
      indexList = SortIndexes_MaxToMin(inputList) #Sortiere Liste nach ihren ratings. Gibt nur die Indexe sortiert zurück
      returnList = []
      for index in indexList: #Wandelt die Indexe wieder in Objekte des Types Hotel um
        returnList.append(hotelsList[index])
      if self.maxIterations != None:
        returnList = returnList[:self.maxIterations] #falls maxIterations nicht None ist so gebe nur eine Liste mit der länge maxIterations zurück
        return returnList
      return returnList #wenn maxIterations == None ist so gebe die vollständige Liste zurück
    elif self.mode == 1 and self.maxIterations != None:
      inputList = []
      for hotel in hotelsList:
        inputList.append(hotel.distance) #erstellt eine Liste mit den Ratings
      indexList = SortIndexes_MaxToMin(inputList) #Sortiere Liste nach ihren ratings. Gibt nur die Indexe sortiert zurück
      returnList = []
      for index in indexList: #Wandelt die Indexe wieder in Objekte des Types Hotel um
        returnList.append(hotelsList[index])
```

```
        returnList = returnList[:self.maxIterations] #falls maxIterations nicht None ist so gebe nur eine Liste mit der länge maxIterations zurück
        return returnList


    elif self.mode == 1 and self.maxIterations == None: #falls maxIterations == None ist und trotzdem in den Modus 1 kommt, so existiert keine Lösung


  def TomorrowAtDestination(self,lastHotel): #diese Methode überprüft ob die Familie morgen ankommen würde
    if lastHotel == None: #Falls lastHotel == None ist, so wird lastHotelDistance auf 0 gesetzt, dies wird dazu gebraucht, da es an Tag1 nach kein letztesHotel gibt
      lastHotelDistance=0
    else:
      lastHotelDistance = lastHotel.distance


    if lastHotelDistance + 360 >= self.entireDistance:
      return True
    else:
      return False



  def Find_BestRatingInList(self,RatedHotelList):
    PathList_ratings = []
    for i in RatedHotelList: #erstellt die Liste PathList_ratings in dem es nur die averageRatings einfügt
      PathList_ratings.append(i.averageRating)
    bestPath_rating = Find_Max(PathList_ratings) #gibt einmal das höchste Rating wieder und zweitens den Index davon wieder von der Liste RatedHotelList [HighestRating, IndexOfHighest Rating]
    return RatedHotelList[bestPath_rating[1]] #gibt den besten Path zurück
```