42. Bundeswettbewerb Informatik

Runde 1

Aufgabe 5: Stadtführung

Team-ID: 00380

Team-Name: CTRL-C / CTRL-V

Bearbeiter dieser Aufgabe:

Nicolas Beninde

20. November 2023

Inhaltsverzeichnis

1. Lösungsidee

Bei Aufgabe 5 handelt es sich um ein Optimierungsproblem, bei dem aus einer festgelegten Route aus Punkten, zwischen denen jeweils eine bestimmte Distanz liegt und die selbst essenziell oder nicht sein können, geschlossene Teilrouten aus nicht essenziellen Punkten entfernt werden sollen. Die Optimierung gilt der Länge. Eine geschlossene Teilroute beginnt am selben Punkt, an dem sie endet. Zusätzlich kann ein neuer Startpunkt gewählt werden. Wichtig hierbei ist, dass die Gesamtroute am selben Punkt enden muss, an dem sie beginnt. Die Reihenfolge der Punkte darf nicht verändert werden, da die Route chronologisch ist.
Um das Problem zu lösen bietet es sich an die Route in einzelne Unterabschnitte einzuteilen, die an den essenziellen Punkten getrennt werden, diese aber als Grenzen enthalten. Für jeden Unterabschnitt werden dann alle „Schleifen“ zusammen mit den Überschneidungen dieser Schleifen gespeichert. Nun besteht das Problem, dass zwei oder mehr sich überschneidende Schleifen nicht alle entfernt werden können. Stattdessen muss die Kombination aus verschiedenen Schleifen jedes Unterabschnitts entfernt werden, die die meiste Strecke einspart.
Dazu wird sich die Menge an Schleifen als Knoten und deren Überschneidungen als Kanten eines Graphen vorgestellt. Dieses Problem ist in der Form als „maximum-weight independent set problem“ bekannt. Da die Anzahl der Schleifen in einem Abschnitt so gering ist, dass ein Brute-Force-Algorithmus keine Laufzeitprobleme erzeugt, wird dieses Problem mit diesem Ansatz gelöst.
Der neue Startpunkt wird ähnlich bestimmt. Erst werden alle Start-/Endkombinationen ermittelt und dann für jede Kombination die möglichen Schleifen vor dem ersten und nach dem letzten essenziellen Punkt gespeichert. Auf diese Menge an Schleifen wird wieder der Brute-Force-Algorithmus angewendet. Die Start-/Endkombination, die in Summe mit den für sie möglichen Schleifen in dem genannten Bereich, die meiste Distanz einspart wird die neue Kombination. Dies kann auch die alte Kombination sein.
Die Laufzeit für das Finden aller Schleifen beträgt linear O(n). Das Lösen des „maximum-weight independent set problem“ mit Brute-Force ist nur in nicht polynomieller Zeit $O(n_w * 2^s)$ (s = Anzahl Schleifen) machbar. Für dieses Problem ist eine Optimierung auf Laufzeit, erkennbar an der benötigten Zeit für die Beispiele, nicht nötig.

2. Umsetzung

Das Programm wird in Python implementiert.

Die Route wird als Liste aus Unterlisten umgesetzt. Jede Unterliste enthält Bezeichnung, Jahr, Distanz, Status und einen Index. Eine weitere Liste realisiert die Speicherung der Indexe aller essenziellen Punkte. Schleifen werden im Programm in Listen realisiert, zu verschiedenen Punkten jedoch auf unterschiedliche Weisen. Beim Finden der Schleifen und ihrer Überschneidungen setzt eine Liste graph, die den Graphen simuliert, das Speichern der Schleifen um. Jede Schleife ist hier eine Unterliste die aus einem Index, der Distanz, Start- und Endpunkt und einer weiteren Liste, die die Indexe der überschneidenden Schleifen enthält. Wird die beste Kombination aus Schleifen ermittelt, werden die Schleifen in einer Liste removableLoops gespeichert. Hier wird nur noch Distanz, Start- und Endpunkt jeder Schleife gespeichert. Zusätzlich enthält die Liste jedoch noch die Gesamtdistanz aller enthaltenden Schleifen. Dies stellt jedoch nur eine Zugriffserleichterung dar. Vor dem Löschen der einzelnen Punkte bestehen die Schleifen in der Endliste nur noch aus dem Triplett aus Distanz, Start- und Endpunkt. Aus diesen wird dann abgelesen wie die Distanz und Indexe der übrigen Punkte verändert wird.

```
#returns every possible combination of elements from a list
#works by recursivly getting all combinations of smaller parts of the list
starting with length 0 and incrementing by 1 till length n
#for a list of length n there are 2^n combinations
def getAllCombinationsFromList(list):

    #if the input list is empty an empty list is returned
    if len(list) == 0:
        return [[]]

    #for every combination of a list, that is the current input list without
    the first entry,
    #the combination itself and the combination + the first entry of the
    current input list are added to the return list
    combinations = []
    for combination in getAllCombinationsFromList(list[1:]):
        combinations += [combination]
        combinations += [combination+[list[0]]]

    #the list containing the combinations is returned
    return combinations
```

Diese Methode erstellt aus einer Liste eine Liste aus allen Kombinationsmöglichkeiten der Elemente dieser Liste. Dies dient dazu aus der Liste die den Graphen simuliert, alle Möglichkeiten aus den verschiedenen Schleifen zu extrahieren. Die Methode erstellt rekursiv eine Liste aller Kombination einer Unterliste der eigentlichen Liste und erweitert diese stetig.

Die weiteren Methoden werden aufgrund ihrer Länge nicht mehr abgedruckt. Der Code ist hinten in der Dokumentation oder in der angehängten Datei abgedruckt.

Die Methode findLoopsAndIntersectionsInSegment() ist eine Mehrzweckfunktion. In einem gegebenen Bereich werden Schleifen anhand der Bezeichnungen einzelner Punkte erkannt und gespeichert. Der Bereich für den Beginn von Schleifen kann dabei getrennt von dem Bereich für das Ende von Schleifen angegeben werden. Dies hat den Zweck, dass auch Start-/Endkombinationen mit der Funktion ermittelt werden können, indem den Parametern, die für Start und Ziel in Frage kommenden Intervalle zugewiesen werden. Sollten die beiden Bereiche identisch sein, also dem Zweck der normalen Schleifenerkennung dienen, werden zusätzlich Überschneidungen durch Vergleiche von Start und Ende der einzelnen Schleifen ermittelt und an die Rückgabe angehängt.

Die Methode findBestIndependentSetInGraph() verwendet getAllCombinationsFromList() um eine Liste aller Möglichkeiten von Schleifenkombinationen zu erhalten um dann jede Kombination auf ihre Gültigkeit (keine Überschneidungen zwischen Schleifen) zu prüfen. Ist eine Kombination gültig wird die eingesparte Distanz mit dem aktuellen Maximum verglichen und wenn größer, wird die beste Kombination ersetzt. So kann nach der Iteration die beste Kombination zurückgegeben werden.

Die Methode findBestStart() findet erst alle möglichen Start-/Endkombinationen mit der Methode findLoopsAndIntersectionsInSegment() um dann für jede Kombination die beste Kombination aus Schleifen in dem noch übrigen Bereich vor dem ersten und nach dem letzten essenziellen Punkt mit den Methoden findLoopsAndIntersectionsInSegment() und findBestIndependentSetInGraph() zu finden. Die Summe der gesparten Distanzen wird addiert und mit dem aktuellen Maximum verglichen. Nach der Iteration wird die beste Kombination aus Start-/Endkombination und zusätzlichen Schleifen zurückgegeben. Die Start-/Endkombination wird dabei im selben Format wie eine Schleife dargestellt mit der Ausnahme, dass die „Startschleife“ einen Index vor dem Array, also -1, startet und die

„Endschleife“ einen Index nach dem Array, also last index + 1, endet, um das Löschen des ersten bzw. letzten Punktes zu berücksichtigen. Denn die Löschfunktion entfernt nur die Indexe zwischen Start und Ende der Schleife.

Die Methode findShortestRoute() ist die Hauptfunktion des Programms und verbindet die einzelnen Unterfunktionen. Erst werden iterativ alle besten Schleifenkombinationen in den jeweiligen Schleifenabschnitten zwischen den essenziellen Punkten bestimmt und gespeichert. Darauf wird die Methode findBestStart() ausgeführt. Die so ermittelten Schleifen werden dann nach ihrem Startindex absteigend sortiert und iterativ gelöscht. Bei der Löschung werden die nötigen Änderungen in den Distanzen und Indexen der anderen Punkte durchgeführt. Die Indexe werden absteigend gelöscht, um das Löschen falscher Punkte zu vermeiden. Nach der Löschung werden die optimierte Route und die eingesparte Distanz zurückgegeben.

3. Beispiele

Das Programm gibt zu einer gültigen Route die optimale, verkürzte Route in der gleichen Syntax aus. Zusätzlich werden die entfernten Schleifen sowie die gesparte Distanz ausgegeben. Des Weiteren berechnet das Programm mit der Methode perf_counter() der time-Bibliothek die Zeit, die das Programm benötigt hat. Entfernte Schleifen, die einen neuen Start- oder Endpunkt zur Folge haben, beginnen bei -1 bzw. enden 1 hinter der Länge der Liste (s. Umsetzung).

3.1 tour1.txt

['Brauerei', '1613', 'X', 0, 0]

['Karzer', '1665', 'X', 80, 1]

['Rathaus', '1678', 'X', 150, 2]

['Rathaus', '1739', 'X', 150, 3]

['Euler-Brücke', '1768', ' ', 330, 4]

['Fibonacci-Gaststätte', '1820', 'X', 360, 5]

['Schiefes Haus', '1823', ' ', 480, 6]

['Theater', '1880', ' ', 610, 7]

['Emmy-Noether-Campus', '1912', 'X', 740, 8]

['Emmy-Noether-Campus', '1998', 'X', 740, 9]

['Euler-Brücke', '1999', ' ', 870, 10]

['Brauerei', '2012', ' ', 1020, 11]

Entfernte Schleifen: [[690, 10, 15], [350, 2, 5]]

Gesparte Distanz: 1040 Laufzeit: 0.005s

3.2 tour2.txt

['Brauerei', '1613', ' ', 0, 0]

['Karzer', '1665', 'X', 80, 1]

['Rathaus', '1678', ' ', 150, 2]

['Rathaus', '1739', ' ', 150, 3]

['Euler-Brücke', '1768', ' ', 330, 4]

['Fibonacci-Gaststätte', '1820', 'X', 360, 5]

['Schiefes Haus', '1823', ' ', 480, 6]

['Theater', '1880', ' ', 610, 7]

['Emmy-Noether-Campus', '1912', 'X', 740, 8]

['Emmy-Noether-Campus', '1998', 'X', 740, 9]

['Euler-Brücke', '1999', ' ', 870, 10]

['Brauerei', '2012', ' ', 1020, 11]

Entfernte Schleifen: [[690, 10, 15], [350, 2, 5]] Gesparte Distanz: 1040 Laufzeit: 0.011

3.3 tour3.txt

['Talstation', '1768', ' ', 0, 0]

['Wäldle', '1805', ' ', 520, 1]

['Mittlere Alp', '1823', ' ', 1160, 2]

['Observatorium', '1833', ' ', 1450, 3]

['Observatorium', '1874', 'X', 1450, 4]

['Piz Spitz', '1898', ' ', 1920, 5]

['Panoramasteg', '1912', 'X', 2140, 6]

['Panoramasteg', '1952', ' ', 2140, 7]

['Ziegenbrücke', '1979', 'X', 2390, 8]

['Talstation', '2005', ' ', 2670, 9]

Entfernte Schleifen: [[600, 8, 11], [1290, 3, 6]]

Gesparte Distanz: 1890 Laufzeit: 0.011

3.4 tour4.txt

['Marktplatz', '1549', ' ', 0, 0]

['Marktplatz', '1562', ' ', 0, 1]

['Springbrunnen', '1571', ' ', 80, 2]

['Dom', '1596', 'X', 150, 3]

['Bogenschütze', '1610', ' ', 270, 4]

['Bogenschütze', '1683', ' ', 270, 5]

['Schnecke', '1698', 'X', 420, 6]

['Fischweiher', '1710', ' ', 600, 7]

['Reiterhof', '1728', 'X', 720, 8]

['Schnecke', '1742', ' ', 860, 9]

['Schmiede', '1765', ' ', 1030, 10]

['Große Gabel', '1794', ' ', 1140, 11]

['Große Gabel', '1874', ' ', 1140, 12]

['Fingerhut', '1917', 'X', 1210, 13]

['Stadion', '1934', ' ', 1330, 14]

['Marktplatz', '1962', ' ', 1420, 15]

Entfernte Schleifen: [[370, 25, 29], [610, 19, 22], [390, 7, 13], [200, 2, 4], [210, -1, 2]]

Gesparte Distanz: 1780 Laufzeit: 0.012

3.5 tour5.txt

['Gabelhaus', '1638', ' ', 0, 0]

['Gabelhaus', '1699', ' ', 0, 1]

['Hexentanzplatz', '1703', 'X', 160, 2]

['Eselsbrücke', '1711', ' ', 280, 3]

['Dreibannstein', '1724', ' ', 390, 4]

['Dreibannstein', '1752', ' ', 390, 5]

['Schmetterling', '1760', 'X', 540, 6]

['Dreibannstein', '1781', ' ', 620, 7]

['Märchenwald', '1793', 'X', 700, 8]

['Märchenwald', '1840', ' ', 700, 9]

['Eselsbrücke', '1855', ' ', 780, 10]

['Eselsbrücke', '1877', ' ', 780, 11]

['Reiterdenkmal', '1880', ' ', 920, 12]

['Riesenrad', '1881', ' ', 1100, 13]

['Riesenrad', '1902', ' ', 1100, 14]

['Dreibannstein', '1911', 'X', 1230, 15]

['Olympisches Dorf', '1924', ' ', 1390, 16]

['Haus der Zukunft', '1927', 'X', 1520, 17]

['Stellwerk', '1931', ' ', 1640, 18]

['Stellwerk', '1942', ' ', 1640, 19]

['Labyrinth', '1955', ' ', 1850, 20]

['Gauklerstadl', '1961', ' ', 1930, 21]

['Planetarium', '1971', 'X', 2010, 22]

['Känguruhfarm', '1976', ' ', 2060, 23]

['Balzplatz', '1978', ' ', 2140, 24]

['Dreibannstein', '1998', 'X', 2230, 25]

['Labyrinth', '2013', ' ', 2360, 26]

['CO2-Speicher', '2022', ' ', 2550, 27]

['Gabelhaus', '2023', ' ', 2620, 28]

Entfernte Schleifen: [[300, 30, 32], [270, 23, 26], [170, 19, 21], [410, 14, 18], [410, 8, 11], [820, 0, 5]]

Gesparte Distanz: 2380 Laufzeit: 0.016

4. Quellcode

```
#returns every possible combination of elements from a list
#works by recursivly getting all combinations of smaller parts of the list starting with length 0 and
incrementing by 1 till length n
def getAllCombinationsFromList(list):

    if len(list) == 0:
        return [[]]

    #for every combination of a list, that is the current input list without the first entry,
    #the combination itself and the combination + the first entry of the current input list are added
    combinations = []
    for combination in getAllCombinationsFromList(list[1:]):
        combinations += [combination]
        combinations += [combination+[list[0]]]

    #the list containing the combinations is returned
    return combinations


#returns loops and their individual intersections with other loops in certain segment
#can be used for loop finding between important points as well as for finding new start/end
#possibilities
def findLoopsAndIntersectionsInSegment(segmentForStartOfLoops, segmentForEndOfLoops):

    #can be understood as a graph with the vertices being loops and edges being the intersections
    graph = []

    #adds loop / new start/end to return list when the names of the start and end possibility are equal
    #and the end possibility is located behind the start possibility
    for startPossibilty in segmentForStartOfLoops:
        for endPossibility in segmentForEndOfLoops:
            nameOfStartOption, nameOfEndOption, indexOfStartOption, indexOfEndOption =
            startPossibilty[0], endPossibility[0], startPossibilty[4], endPossibility[4]
            if nameOfStartOption == nameOfEndOption and indexOfStartOption < indexOfEndOption:

                #for normal loop finding between two important points // start segment and end segment
                #are identical
                #loop contains index placeholder, length and start index as well as end index
                routeLengthToStartOption, routeLengthToEndOption, routeLength =
                startPossibilty[3], endPossibility[3], segmentForEndOfLoops[-1][3]
                if segmentForStartOfLoops == segmentForEndOfLoops:
                    graph.append([None, routeLengthToEndOption - routeLengthToStartOption,
                    indexOfStartOption, indexOfEndOption])

                #for finding new start/end possibilities // start segment and end segment are different
                #new start/end contains index placeholder, length of route till new start, length of
                #route from new end till end and new start index as well as new end index
```

```
                if segmentForStartOfLoops != segmentForEndOfLoops: graph.append([None,
                routeLengthToStartOption, routeLength - routeLengthToEndOption, indexOfStartOption,
                indexOfEndOption])

    #gets element as well as index, that is directly placed at the placeholder, from the created graph
    for i, loop1stDegree in enumerate(graph):
        loop1stDegree[0] = i

        #if the function is used for loop finding the indices of loops that intersect the current loop
        #are saved in the loop
        if segmentForStartOfLoops == segmentForEndOfLoops:
            loop1stDegree.append([])
            for j, loop2ndDegree in enumerate(graph):

                #it is necessary to check all four option as the main and checked loop start/end index
                #can be identical
                mainLoopStartIndex, mainLoopEndIndex, checkedLoopStartIndex, checkedLoopEndIndex =
                loop1stDegree[2], loop1stDegree[3], loop2ndDegree[2], loop2ndDegree[3]
                if checkedLoopStartIndex < mainLoopStartIndex < checkedLoopEndIndex or
                checkedLoopStartIndex < mainLoopEndIndex < checkedLoopEndIndex or mainLoopStartIndex <
                checkedLoopStartIndex < mainLoopEndIndex or mainLoopStartIndex < checkedLoopEndIndex <
                mainLoopEndIndex:
                    loop1stDegree[4].append(j)

    #the list containing all loops or start/end options and their intersections is returned
    return graph



#returns best combination of non-intersecting loops that are saved in the input list
def findBestIndependentSetInGraph(graph):

    removableLoops = [[], 0]
    sets = getAllCombinationsFromList(graph)

    #brute-force iteration through all possibile combinations
    for set in sets:

        #checks for every loop, contained in the current combination, if every other loop in the
        #combination is valid / not saved as intersecting
        for i, loop1stDegree in enumerate(set):
            for loop2ndDegree in set[i + 1:]:

                #checks if the index of the checked loop is contained in the main loops intersections
                listOfIndecesOfIntersectingLoops, checkedLoopIndex = loop1stDegree[4], loop2ndDegree[0]
                if any(checkedLoopIndex == loopIndex
                for loopIndex in listOfIndecesOfIntersectingLoops):
                    break
```

```
            else:
                continue
            break

        #if all loops in the combination are non-intersecting, the combination is tested on the total
        #saved length
        else:
            savedDistance = sum([loop[1] for loop in set])
            currentSavedDistance = removableLoops[1]
            if(currentSavedDistance < savedDistance):
                removableLoops[1] = savedDistance

                #only distance, start and end of each loop is saved as the rest of the information is
                #not needed anymore
                newRemovableLoops = []
                for newRemovableLoop in set:
                    newRemovableLoops += [newRemovableLoop[1:4]]
                removableLoops[0] = newRemovableLoops

    return removableLoops



#returns the best combination of start/end and removable loops before the first and after the last
#important stop
def findBestStart(route, posOfX):

    # all possible start/end combinations are found using findLoopsAndIntersectionsInSegment()
    removableLoops = [[], 0]
    startEndOptions = findLoopsAndIntersectionsInSegment(route[:posOfX[0] + 1], route[posOfX[-1]:])

    for option in startEndOptions:

        #the best combination of loops in front of the first and after the last important stop, for
        #the current start option, is saved
        possibleStart, possibleEnd, firstImportantStop, lastImportantStop =
        option[3], option[4], posOfX[0], posOfX[-1]
        graph = findLoopsAndIntersectionsInSegment(route[possibleStart:firstImportantStop + 1],
        route[possibleStart:firstImportantStop + 1])
        graph += findLoopsAndIntersectionsInSegment(route[lastImportantStop:possibleEnd + 1],
        route[lastImportantStop:possibleEnd + 1])
        bestSet = findBestIndependentSetInGraph(graph)

        #the saved distance of the current start/end is calculated
        savedDistanceFromLoops, savedDistanceFromNewStart, savedDistanceFromNewEnd =
        bestSet[1], option[1], option[2]
        currentSavedDistance = savedDistanceFromLoops + savedDistanceFromNewStart +
        savedDistanceFromNewEnd
```

```
        if removableLoops[1] < currentSavedDistance:
            removableLoops[1] = currentSavedDistance
            newRemovableLoops = bestSet[0]
            removableLoops[0] = newRemovableLoops

            #if there is a new start, a loop containing all points before the new start is created /
            #start is set to -1 as previous start also has to be removed
            #if there is a new end, a loop containing all points after the new end is created / end is
            #set to index of last stop + 1 as previous end also has to be removed
            newStartIndex, newEndIndex, indexOfLastElement = option[3], option[4], len(route) - 1
            if newStartIndex != 0: removableLoops[0] +=
            [[savedDistanceFromNewStart, -1, newStartIndex]]
            if newEndIndex != indexOfLastElement: removableLoops[0] +=
            [[savedDistanceFromNewEnd, newEndIndex, len(route)]]

    return removableLoops



#main function that combines the individual functions to find the shortest route
def findShortestRoute(route):

    removableLoops, posOfX = [], []
    removedPoints, totalSavedDistance = [], 0

    #fills the list containing the indeces of important stops and prints the information to the console
    for i in range(len(route)):
        if route[i][2] == "X": posOfX.append(i)
    print(posOfX), print("")

    #if there is no important point in the route all stops between the start and end are deleted and
    #the necessary information is returned
    if posOfX == []:
        totalSavedDistance = route[-1][3]
        route[-1][3], route[-1][4] = 0, 1
        removedPoints = route[1:-1]
        return [route[0], route[-1]], removedPoints, totalSavedDistance

    #for every interval between two important points, all loops are found and the best set of them is
    #added to the removable loops list
    for i in range(len(posOfX) - 1):
        startOfInterval, endOfInterval = posOfX[i], posOfX[i + 1]
        graph = findLoopsAndIntersectionsInSegment(route[startOfInterval:endOfInterval + 1],
        route[startOfInterval:endOfInterval + 1])

        #as the total saved distance of the best set of loops is not required in the deletion process,
        #only the loops are saved
        removableLoops += findBestIndependentSetInGraph(graph)[0]
```

```
    #as the total saved distance of the best set of start/end/loops is not required in the deletion
    #process, only the loops are saved
    removableLoops += findBestStart(route, posOfX)[0]

    #the removable loops are sorted in reverse order by their starting index and printed to the console
    removableLoops = sorted(removableLoops, key = lambda x: x[1], reverse = True)
    print(removableLoops)

    for loop in removableLoops:
        savedDistanceFromLoop, loopStart, loopEnd = loop[0], loop[1], loop[2]

        totalSavedDistance += savedDistanceFromLoop

        #for every route entry from the end of the loop on, the index is adjusted by the number of
        removable stops in that loop
        for entry in route[loopEnd:]:
            entry[3] -= loop[0]
            entry[4] -= loopEnd - (loopStart + 1)

        #the removable stops are removed in reversed order
        for index in range(loopEnd - 1, loopStart, -1):
            removedPoints.append(route[index])
            route.pop(index)

    return route, removedPoints, totalSavedDistance
```