Vortänzer Challenge

Lösungsidee:

Der Zustand eines Roboters R ist definiert durch seine Position auf dem Raster und seine Blickrichtung.

Die Blickrichtung kann dabei 8 verschiedene Zustände einnehmen, da der Roboter nur Drehungen um 45° ausführen kann.

Das Programm P_R eines Roberts R definiert eine Befehlsfolge, die der Roboter ausführt. Pro Frame wird dabei eine Aktion ausgeführt. Eine Aktion ist entweder eine Drehung des Roboters nach links oder rechts, eine Pause oder eine Bewegung um eine Einheit in Blickrichtung. Auch schräge Bewegungen sind erlaubt. Aktionen können in Schleifen zusammenfasst werden. Eine Schleife kann dabei aus maximal 10 Iterationen bestehen.

Da jeder Roboter nur maximal 255 Instruktionen speichern kann, wird das Programm nach der 255. Instruktion abgebrochen. Die Instruktionsanzahl bezieht sich nicht auf die Programmlänge, sondern die Anzahl der Aktionen, die der Roboter ausführt.

An einer Runde einer Vortänzer Challenge nehmen immer drei Roboter teil. Ein Runde besteht aus neun Matches, wobei jeder Roboter dreimal Vortanzt und sechsmal nachtanzt. Der Sieger einer Runde ist dabei der Roboter, der die wenigsten Strafpunkte hat, usw. Je nach Platzierung innerhalb einer Runde erhalten die Roboter unterschiedlich viele Tatas. Der Gesamtsieger eines Turniers ist dabei der Roboter mit den meisten Tatas.

Der Vortänzer V hat für sein Programm 10 Freizeichen und bekommt für jedes weitere Zeichen 9 Strafpunkte.

Um die Strafpunkte eines imitierenden Roboters I zu berechnen, sind zwei Schritte nötig:

1. Die Anzahl aller Zeichen des imitierenden Programms wird mal drei genommen
2. In jedem Frame wird die Entfernung zwischen dem imitierenden Roboter und dem Originalroboter berechnet. Diese wird dann den Strafpunkten des imitierenden Roboters hinzugefügt.

Das Programm eines imitierenden Roboters I ist dabei umso besser, je weniger Strafpunkte es verursacht.

Die Entfernung zwischen zwei Robotern ist als die Anzahl der Instruktionen ohne Drehungen definiert, die ein Roboter benötigt um zum anderen zu gelangen.

Um die Entfernung zwischen V und I zu berechnen geht man wie folgt vor:

$$d = \max(|x_V - x_I|, |y_V - y_I|)$$

wobei d die zwischen den beiden Robotern liegende Entfernung ist, x_R die x-Position eines Roboters R ist und y_R die y-Position eines Roboters R ist.

Wenn im Folgenden von P_S gesprochen wird, ist das Programm gemeint, das der Server an die Nachtänzer zurückgibt. Dieses Programm muss nicht identisch zu P_V sein, da der Turnierserver sämtliche Schleifen in P_V auflöst und das Programm beim 255. Zeichen abschneidet.

Um P_I zu ermitteln muss man ein gegebenes P_S so verändern, dass es möglichst kurz ist. Aufgrund der Taktik, mit der die meisten Vortänzer tanzen, lohnt es sich nicht zu versuchen, durch eine veränderte Bewegungsfolge Strafpunkte einzusparen. Die meisten Vortänzer geben Programme mit Schleifen aus, die die Instruktionszahl von 255 überschreiten und dennoch nur 10 Zeichen lang sind. Das heißt, dass es in vielen Fällen möglich ist, P_S auf 10 Zeichen zu verkürzen. Die Wahrscheinlichkeit hier mit Wegfindung ein kürzeres Programm zu erzeugen ist extrem gering, so dass diese Möglichkeit ausgeschlossen werden kann. Die eigentliche Herausforderung liegt, darin ein gegebenes P_S zu verkürzen.

Dazu müssen die Muster in P_S ermittelt und erneut in Schleifen zusammengefasst werden. Ein Ansatz, der sich für diesen Einsatz gut eignet sind Regular Expressions kurz RegEx. Mit der RegEx `(.+?)\1+` lassen sich wiederholende Zeichenketten erkennen. Dabei steht `(.+?)` für einen beliebige Zeichenkette im Text und `\1+` für eine mindestens einmalige Wiederholung dieser Zeichenkette. Die Wiederholung muss dabei unmittelbar nach der beliebigen Zeichenkette gefunden werden. `(.+?)` ist dabei „lazy“, das heißt, dass immer die kürzeste Zeichenkette genommen wird. Dieser sehr simple Ansatz hat aber noch einige Schwächen. So werden z.B. drei Punkte als Wiederholung erkannt und durch eine Schleife ersetzt, wodurch die Syntax des Programmes ungültig wird.

Die obige Definition muss deshalb erweitert werden. Anstatt nur nach der kürzesten sich wiederholenden Zeichenkette zu suchen, werden noch weitere Einschränkungen hinzugefügt:

1. Die Zeichenkette darf nicht mit einem Punkt starten
2. Die Zeichenkette darf nicht mit einer Zahl enden
3. Die Zeichenkette darf nicht ausschließlich aus Punkten oder Zahlen bestehen

Diese Bedingungen verhindern, dass Wiederholungen gefunden werden, die nach dem Zusammenfassen zu Schleifen dafür sorgen, dass die Syntax des Programms ungültig wird. Wenn man das RegEx dementsprechend erweitert, kommt man zu folgendem Ergebnis: `(?!\.)(.+?)\1+(?<=\D)`. Das `(?!\.)` verhindert dabei, dass das erste Zeichen einer Wiederholung ein Punkt ist, indem alle Zeichenketten, die mit einem Punkt anfangen übersprungen werden. `(?<=\D)` überprüft, sobald ein Match gefunden wurde, ob der letzte Buchstabe dieses Matches eine Zahl ist, und wenn ja, wird das Match verworfen. Aber auch diese erweiterte Version hat noch Nachteile. Beispielsweise lässt sie sich leicht durch ein FF in einer größeren Wiederholung ablenken. Dadurch dass FF nicht durch 2F. ersetzt werden kann, ohne zusätzliche Strafpunkte zu erzeugen, liegt hier die Schwäche dieser Lösungsstrategie. Dieses Problem wird durch zwei verschiedene Ansätze behoben.

Einerseits wird FF zu 2F. zusammengefasst und nach Erzeugen des verkürzten Programmes wird mit der RegEx `([2][FB-lr][\.])` jede solche Schleife erkannt und wieder aufgelöst. Dadurch, dass diese

Schleifen bei der Komprimierung erzeugt werden, wird die Wiederholungsfindung nicht durch eine kleine Wiederholung gestört und kann größere Wiederholungen erkennen.

Andererseits wird zusätzlich eine Variation der RegEx zur Wiederholungsfindung eingesetzt: `(?!\.)(.+)\1+(?<=\D)`. Durch das fehlende Fragezeichen ist diese RegEx „greedy“ und versucht eine möglichst lange, sich wiederholende Zeichenkette zu finden. Dabei wird aber in manchen Fällen eine zu lange Zeichenkette gefunden, sodass die Syntax des Programms ungültig wird.

Auch die Tatsache, dass der Turnierserver Programme nach 255 Instruktionen abschneidet, kann man ausnutzen. Deshalb wird sowohl mit der „lazy“ als auch mit der „greedy“ RegEx getestet, ob der Anfang der Instruktionszeichenkette eine Wiederholung darstellt. Der Anfang kann dabei auch die komplette Zeichenkette beinhalten. Wenn das der Fall sein sollte, wird überprüft, ob die restliche sich nicht wiederholende Zeichenkette ein Teil der 1.Wiederholung ist. Wenn ja, dann wird dieser Teil der Zeichenkette durch die 1. Wiederholung ersetzt. Dadurch kann ein Programm mit über 255 Instruktionen auch als eine Wiederholung erkannt werden. Dieses Programm wird dann mit der „lazy“ oder der „greedy“ RegEx weiter zusammengefasst. Auch die Ergebnisse dieser Vervollständigung müssen nicht unbedingt gültig sein.

Um ein Programm auf Korrektheit zu prüfen, wird zuerst die Syntax getestet. Dabei wird überprüft, ob es für jeden Schleifenanfang auch ein Ende gibt. Danach wird das Programm zusammen mit dem Originalprogramm simuliert um zu prüfen, ob das Imitationsprogramm nicht vom Pfad des Originalprogramms abweicht. Wenn dieser Test auch bestanden ist, gilt das Programm als gültig, sonst als ungültig.

Die verschiedenen Ansätze können dabei auch kombiniert werden, so dass es zum Beispiel möglich ist zu versuchen, mit einer „greedy“ RegEx das Programm zu vervollständigen, das vervollständigte Programm dann aber mit einer „lazy“ RegEx zu komprimieren.

Auch die korrekte Generierung der Schleifen ist nicht ganz trivial. Solange die Iterationszahl unter 10 bleibt, ist die Erzeugung offensichtlich. Doch sonst muss die Iterationszahl so zerteilt werden, dass möglichst wenige geschachtelte Schleifen nötig sind. Dazu wird folgender Algorithmus eingesetzt:

```
Eingabe: Anzahl an Iterationen der Schleife i (i > 9), wiederholte Zeichenkette Z
Ausgabe: Eine Zeichenkette E

E:= ""
solange wahr #Eingedeutschtes while true
     iAlt := i
     ETeil := ""
     für jedes x in {10, 9, ..., 2}
          wenn i < 10
               Abbruch #Verlässt nur aktuelle Schleife
          wenn i % a == 0 #i ist durch a teilbar
               wenn ETeil == ""
                    ETeil := Z
               ETeil := x + ETeil + "."
               i := i / x
               Abbruch
     wenn i == iAlt #Geht nicht mehr weiter
          wenn i < 10 #Fertig
               E + i + ETeil + "." zurückgeben
          sonst #Weiterschachteln nicht möglich
               E += ETeil
               E += "9" + Z + "."
               i -= 9
```

Um für den Fall vorzusorgen, dass das ganze Originalprogramm eine einzige sich wiederholende Sequenz ist, wird beim Versuch das Programm zu vervollständigen automatisch getestet, ob die Wiederholung sich über das komplette Programm erstreckt. Wenn das der Fall sein sollte und die sich wiederholende Sequenz nicht länger als vier ist, wird ein Programm mit 9*9*5 Wiederholungen dieser Sequenz generiert.

Am Ende des Imitierens wird dann ermittelt, welches der so erzeugten Programme am kürzesten ist und dieses wird dann an den Turnierserver zurückgegeben.

Trotz der verschiedenen Optimierungen ist das Programm nicht in der Lage immer das kürzeste Programm zu ermitteln. Deshalb ist der hier gezeigte Lösungsansatz nicht der bestmögliche, da er zwar in fast allen Fällen gute bis perfekte Lösungen liefert, es aber auch vorkommen kann, dass die zurückgegebene Lösung unnötig lang ist.

Um einen Tanz zu erzeugen, wird ein zufälliger Tanz aus einem Pool von acht Tänzen ausgewählt. Diese Tänze sind jeweils zehn Zeichen lang und verfügen über eine Instruktionszahl größer als 255. Außerdem wurde darauf geachtet, dass die Tänze sich nicht auf unter 10 Zeichen zusammenfassen lassen.

Die Taktik die Tänze genau 10 Zeichen lang zu halten, ist besonders vorteilhaft, weil dadurch der Vortänzer keine Strafpunkte bekommt, der Nachtänzer aber mindestens 30. Da der Vortänzer für jedes weitere Zeichen 9, der Nachtänzer aber nur 3 Strafpunkte bekommt, ist so gewährleistet, dass Punktedifferenz bei optimaler Wiederholungserkennung durch den Nachtänzer größtmöglich ist.

Umsetzung:

Das Programm ist in Python umgesetzt. Auf die Implementierungsdetails der Simulation möchte ich hier nicht näher eingehen, da sich diese im Vergleich zur Implementierung der 1. Runde fast nicht geändert haben. Die einzigen Anpassungen die vorgenommen wurden, betreffen das Übersetzten der entsprechenden Klassen von C# nach Python. Außerdem wurde die Berechnung der Strafpunkte so angepasst, dass die Länge des Programmes nicht mehr automatisch während der Simulation berechnet wird.

Der für die Aufgabenlösung relevante Teil des Programmes befindet sich in der Datei `Dancechallenge.py`. Diese basiert auf der vom Turnierserverteam veröffentlichen Vorlage. In der Methode `nach` wird die Methode `imitateDance` aufgerufen, die sich um das Imitieren eines Programmes kümmer. Die Methode `vor` ruft `createDance` auf. `CreateDance` gibt dabei ein beliebiges Element aus einer Liste mit acht Tänzen zurück.

Für die Auswertung der RegEx ist das mit Python mitgelieferte Modul `re` zuständig. Für die Anwendung der einzelnen Prozeduren sind die Methoden `tryCompress`,`tryCompressWithAutoComplete` sowie `tryCompressWithGreedyAutoComplete` zuständig. Um eine bestmögliche Performance beim Anwenden der RegEx zu erreichen, werden die RegEx vor dem Anwenden kompiliert. Zum Ersetzen der Wiederholungen wird die `sub` Methode verwendet, die für jedes Match, das vom RegEx gefunden wird die Methode `createLoop` aufruft. Diese ersetz dann eine sich wiederholende Sequenz. Um Iterationswerte über 9 zu verarbeiten, wird die Methode `getRepeatedProgramString` eingesetzt.

Der Name der KI auf dem Turnierserver lautet „FinalAI“ und mein Nutzername „l.k.1234“.

Beispiele:

1.Beispiel

Komplette Schleifenerkennung für „FrFl-FrFl-FrFl-BFrFl-FrFl-FrFl-BFrFl-FrFl-FrFl-B“ mit „greedy“ RegEx:

1. Wiederholung „FrFl-FrFl-FrFl-B“ in „FrFl-FrFl-FrFl-BFrFl-FrFl-FrFl-BFrFl-FrFl-FrFl-B“ gefunden
2. Durch „3FrFl-FrFl-FrFl-B.“ ersetzt
3. Wiederholung „FrFl-“ in „FrFl- FrFl- FrFl-“ gefunden
4. Durch „3FrFl-.“ ersetzt
5. Keine Wiederholung mehr gefunden

Ergebnis: „33FrFl-.B.“

2. Beispiel

Erkennung einer unvollständigen Wiederholung am Ende des Programmes „FrFlBFrFlBFrFl“ (dieses Programm wurde gekürzt, um das Beispiel übersichtlich zu halten)

1. Wiederholung „FrFlB“ in „FrFlBFrFlB“ gefunden
2. Unvollständige Wiederholung gefunden: „FrFl“, fehlender Teil „B“
3. Schleife erzeugt: „3FrFlB.“

Laufzeiten zum 1. Beispiel:

1. 75 Schritte der RegEx-Engine zum Finden des Matches; 5 Schritte um festzustellen, dass es kein weiteres Match gibt
2. –
3. 75 Schritte der RegEx-Engine zum Finden des Matches; 16 Schritte um festzustellen, dass es kein weiters Match gibt
4. –
5. 147 Schritte um festzustellen, dass es kein weiteres Match mehr gibt

Anhand dieser Laufzeiten kann man auch leicht die Schwäche einer RegEx-Engine erkennen: Wenn es kein Match mehr gibt, dann sind die meisten Schritte nötig. In diesem Anwendungsfall ist dieses Problem aber zu vernachlässigen, da die Eingabedaten mit extrem hoher Wahrscheinlichkeit eine Wiederholung enthalten und wenn keine Wiederholung mehr gefunden wird, ist die Eingabe schon verkürzt.

Der Ausdruck zur Erkennung von Schleifen mit zwei Iterationen ist von diesem Effekt ebenfalls nicht betroffen, da bei derart simplen RegEx die automatische Optimierung der Engine dafür sorg, dass die Überprüfung in den meisten Fällen in denen es kein Match gibt, übersprungen wird.

Quellcode:

Dancechallenge.py

```
import re
import math
import random

def zug(id, zustand, zug):
 zug.ausgabe("Neuer Zug")
 zug.ausgabe("Meine Strafpunkte:")
 zug.ausgabe(getMe(id, zustand).strafpunkte())
 if ichBinVortaenzer(id, zustand):
   vor(id, zustand, zug)
 else:
   nach(id, zustand, zug, tanzZumNachtanzen(id, zustand))

def vor(id, zustand, zug):
 zug.ausgabe("Ich bin Vortaenzer")
 zug.tanzen(createDance())
 pass

def nach(id, zustand, zug, tanz):
 zug.ausgabe("Ich bin Nachtaenzer")
 zug.ausgabe(tanz)
 zug.tanzen(imitateDance(tanz, zug))
 pass

def ichBinVortaenzer(id, zustand):
 return getMe(id, zustand).istVortaenzer()

def tanzZumNachtanzen(id, zustand):
 for i in range(0, len(zustand.listeDanceRobot())):
   if zustand.listeDanceRobot()[i].istVortaenzer():
     return zustand.listeDanceRobot()[i].letzterTanz()

def getMe(id, zustand):
 for i in range(0, len(zustand.listeDanceRobot())):
   if zustand.listeDanceRobot()[i].identifikation() == id:
     return zustand.listeDanceRobot()[i]

def createDance():
    programPool = []
    programPool.append("464F-.B-..")
    programPool.append("Fl9FFr9B..")
    programPool.append("566F..2l..")
    programPool.append("499B..ll-.")
    programPool.append("99913B....")
    programPool.append("999lBrB...")
    programPool.append("99B-.9-F..")
    programPool.append("789rFlF...")

    return random.choice(programPool)

def checkDance(dance, newDance):
    danceProgram = DanceProgram(newDance)
    if danceProgram.IsValid():
```

```
        simulation = DanceSimulation()
        simulation.SetOriginalProgram(dance)
        simulation.SetImitatingProgram(newDance)
        if simulation.GetPenaltyPoints() == 0:
            return True
    return False

def imitateDance(dance, zug):
    robot = DanceRobot(dance)
    zug.ausgabe("Imitieren startet")
    regex = re.compile(r"(?!\.)(.+?)\1+(?<=\D)")
    greedyRegex = re.compile(r"(?!\.)(.+)\1+(?<=\D)")
    currentBestDance = ""
    newDance = ""

    newDance = tryCompressWithAutoComplete(dance, zug, regex)
    zug.ausgabe(newDance)
    if newDance != "" and checkDance(dance, newDance):
        currentBestDance = newDance

    newDance = tryCompress(dance, zug, regex)
    zug.ausgabe(newDance)
    if checkDance(dance, newDance) and (len(newDance) < len(currentBestDance) or
currentBestDance == ""):
        currentBestDance = newDance

    newDance = tryCompressWithAutoComplete(dance, zug, greedyRegex)
    zug.ausgabe(newDance)
    if len(newDance) > 0 and len(currentBestDance) > len(newDance): #Gueltigkeit pruefen,
ist bei greedyRegex nicht garantiert
        if checkDance(dance, newDance):
            currentBestDance = newDance

    newDance = tryCompress(dance, zug, greedyRegex)
    zug.ausgabe(newDance)
    if len(currentBestDance) > len(newDance): #Gueltigkeit pruefen, ist bei greedyRegex
nicht garantiert
        if checkDance(dance, newDance):
            currentBestDance = newDance

    newDance = tryCompressWithGreedyAutoComplete(dance, zug, regex, greedyRegex)
    zug.ausgabe(newDance)
    if len(newDance) > 0 and len(currentBestDance) > len(newDance): #Gueltigkeit pruefen,
ist bei greedyRegex nicht garantiert
        if checkDance(dance, newDance):
            currentBestDance = newDance

    zug.ausgabe(currentBestDance)
    return currentBestDance

def getRepeatedProgramString(orderChar, iterations):
    if iterations > 1: #Eine Wiederholung macht keinen Sinn
        if iterations < 10:
            return str(iterations) + orderChar + "."
        else:
            returnString = ""
            newSection = ""
            while(True):
```

```
                oldIterations = iterations
                for i in reversed(range(2,10)):
                    if iterations < 10: #Nichts mehr zu tun
                        break
                    if iterations % i == 0:
                        if newSection == "":
                            newSection = orderChar
                        newSection = str(i) + newSection + "."
                        iterations = iterations / i
                        break

                if iterations == oldIterations:
                    if iterations < 10:
                        return returnString + str(iterations) + newSection + "."
                    else:
                        returnString += newSection
                        returnString += "9" + orderChar + "."
                        iterations = iterations - 9
    else:
        return orderChar

def createLoop(match):
    return getRepeatedProgramString(match.group(1), len(match.group()) /
len(match.group(1)))

def tryCompress(dance, zug, regex):
    zug.ausgabe("Schleifenfindung gestartet")
    oldDance = dance + "."
    while(len(oldDance) > len(dance)):
        oldDance = dance
        dance = regex.sub(createLoop, dance)
    zug.ausgabe("Komprimiertes Ergebnis")
    zug.ausgabe(dance)
    badLoopRegex = re.compile("([2][FB-lr][\.])")
    dance = badLoopRegex.sub(replaceBadLoop, dance)
    return dance

def replaceBadLoop(match):
    return match.group()[1] + match.group()[1]

def tryCompressWithAutoComplete(dance, zug, regex):
    return tryCompressWithGreedyAutoComplete(dance, zug, regex, regex)

def tryCompressWithGreedyAutoComplete(dance, zug, regex, greedyRegex):
    zug.ausgabe("AutoComplete gestartet")
    if len(dance) != 255:
        return ""

    match = greedyRegex.match(dance)

    if match:
        if len(match.group()) == 255 and len(match.group(1)) == 1:
            return "994" + match.group(1) + "..."
        zug.ausgabe("Kein Match")
        endString = dance[len(match.group()) - 255:]
        zug.ausgabe(endString)
        if match.group(1).startswith(endString):
            zug.ausgabe(dance[len(endString):len(match.group(1))])
```

```
            dance += dance[len(endString):len(match.group(1))]
            dance = getRepeatedProgramString(match.group(1), len(dance) /
len(match.group(1)))
            zug.ausgabe(dance)
            return tryCompress(dance, zug, regex)
    return ""
```

DanceSimulation.py

```
class DanceSimulation(object):
    penaltyPoints = 0
    updateCount = 0

    def SetOriginalProgram(self, originalProgram):
        self.originalProgram = originalProgram;
        self.originalRobot = DanceRobot(originalProgram)
        self.originalRobotFinished = False
        self.finished = False
        self.updateCount = 0

    def SetImitatingProgram(self, imitatingProgram):
        self.imitatingProgram = imitatingProgram;
        self.imitatingRobot = DanceRobot(imitatingProgram)
        self.finished = False
        self.updateCount = 0

    def GetPenaltyPoints(self):
        if(self.finished):
            return self.penaltyPoints
        else:
            while(not self.Update()):
                pass
            self.finished = True
            return self.penaltyPoints

    def Update(self):
        if self.updateCount == 254:
            return True

        self.updateCount += 1

        self.originalRobotFinished = originalRobotFinished = self.originalRobot.Update()
        imitatingRobotFinished = self.imitatingRobot.Update()
        robotsFinished = originalRobotFinished and imitatingRobotFinished

        if(not robotsFinished):
            self.penaltyPoints += max(abs(self.originalRobot.x - self.imitatingRobot.x),
abs(self.originalRobot.y - self.imitatingRobot.y))

        return robotsFinished
```

DanceRobot.py

```
"""
Directions:
0 = West,
1 = Northwest
2 = North
3 = Northeast
4 = East
5 = Southeast
6 = South
7 = Southwest
"""

class DanceRobot(object):
    viewDirection = 2

    def __init__(self, instructionSequence):
        self.x = 0
        self.y = 0
        self.program = DanceProgram(instructionSequence)

    def ResetPosition(self):
        self.x = 0
        self.y = 0
        self.viewDirection = 2

    def Update(self):
        nextInstruction = self.program.NextInstruction()

        if(nextInstruction == 0):
            self.x += self.GetXOffset()
            self.y += self.GetYOffset()
        elif(nextInstruction == 1):
            self.x -= self.GetXOffset()
            self.y -= self.GetYOffset()
        elif(nextInstruction == 2):
            if(self.viewDirection == 0):
                self.viewDirection = 7
            else:
                self.viewDirection -= 1
        elif(nextInstruction == 3):
            if(self.viewDirection == 7):
                self.viewDirection = 0
            else:
                self.viewDirection += 1
        elif(nextInstruction == 5):
            return True
        return False

    def GetXOffset(self):
        if(self.viewDirection == 4 or self.viewDirection == 3 or self.viewDirection ==
5):
            return 1
        elif(self.viewDirection == 0 or self.viewDirection == 7 or self.viewDirection ==
1):
            return -1
```

```
        return 0

    def GetYOffset(self):
        if(self.viewDirection == 6 or self.viewDirection == 7 or self.viewDirection ==
5):
            return 1
        elif(self.viewDirection == 2 or self.viewDirection == 1 or self.viewDirection ==
3):
            return -1
        return 0
```

DanceProgarm.py

```
"""
Instruction:
0 = Forwards
1 = Backwards
2 = left
3 = right
4 = pause
5 = finished
"""

class DanceProgram(object):
    instructionSequence = ""
    instructionPointer = 0

    loopIterationCounters = []
    loopLimits = []
    loopJumpBackIndices = []

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

    def NextInstruction(self):
        finished = False
        while not finished:
            finished = True
            if self.instructionPointer >= len(self.instructionSequence):
                return 5
            instructionChar = self.instructionSequence[self.instructionPointer]
            self.instructionPointer += 1

            if instructionChar == 'F':
                return 0
            elif instructionChar == 'B':
                return 1
            elif instructionChar == 'l':
                return 2
            elif instructionChar == 'r':
                return 3
            elif instructionChar == '-':
                return 4
            elif instructionChar == '.':
                if self.loopIterationCounters[-1] >= self.loopLimits[-1] - 1:
```

```
                    self.loopIterationCounters.pop()
                    self.loopLimits.pop()
                    self.loopJumpBackIndices.pop()
                else:
                    self.loopIterationCounters.append(self.loopIterationCounters.pop() +
1)
                    self.instructionPointer = self.loopJumpBackIndices[-1]
                finished = False
            else:
                if instructionChar.isdigit():
                    loopLimit = int(instructionChar)
                    self.loopLimits.append(loopLimit)
                    self.loopIterationCounters.append(0)
                    self.loopJumpBackIndices.append(self.instructionPointer)
                    finished = False
                else:
                    raise ValueError(instructionChar + " is no valid instruction char.
Please check the input string.")

    def IsValid(self):
        loopCounter = 0
        for c in self.instructionSequence:
            if c == 'F' or c == 'B' or c == 'l' or c == 'r' or c == '-':
                continue
            elif c == '.':
                loopCounter -= 1
                if loopCounter < 0:
                    return False
            elif c.isdigit():
                loopCounter += 1
            else: return False
        return loopCounter == 0
```