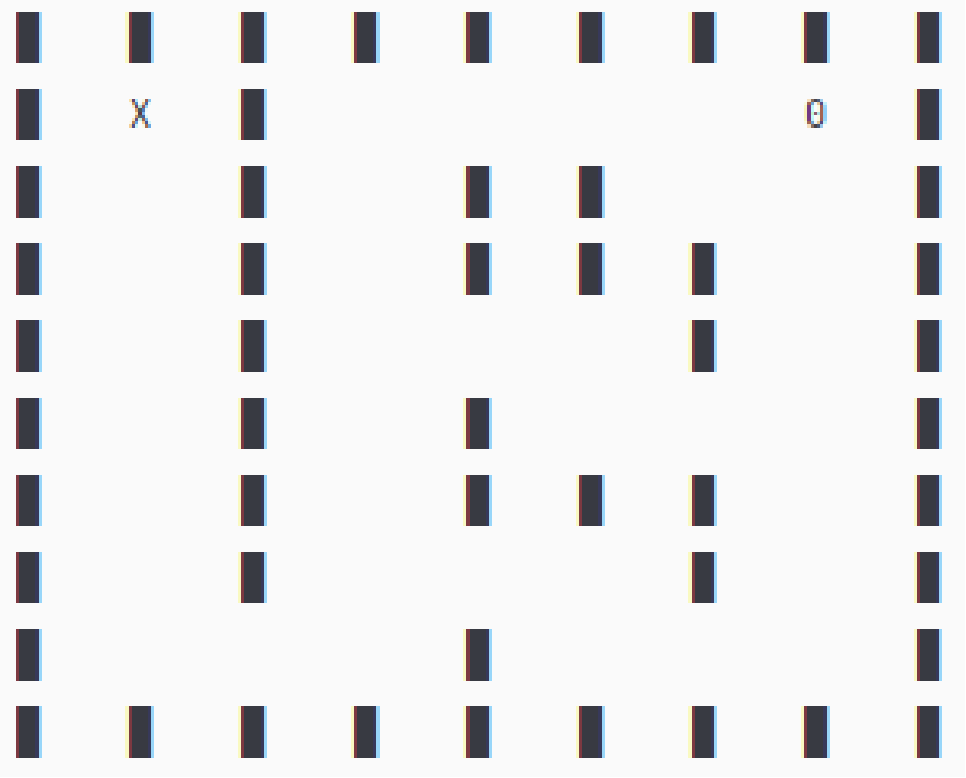
Klassieke Artificial Intelligence
Opdracht 1:

Zoek algoritmes in een doolhof



De doelstelling van deze opdracht is om breadth first en depth first zoek algoritmes toe te kunnen passen op een simpele situatie, en om daarna de gevonden route weer te geven.

In dit pdf bestand volgt een korte verklaring van alle files en methodes die je tegen gaat komen. Maak je echter geen zorgen als het veel is, in de files zelf staat ook uitleg. Dus je kunt tijdens het werken gemakkelijk opzoeken wat alles doet, en je kunt altijd vragen stellen.

Om te beginnen worden een paar project files aangeleverd, ook wel Classes.

Candidate.py

De class Candidate.py zorgt er voor dat wij voor ieder uniek vak in het doolhof een set coördinaten kunnen toewijzen, en een zogenaamde parentCandidate zodat wij terug kunnen zien vanuit welk vak wij oorspronkelijk langs zijn gekomen.

```
class Candidate:
    def __init__(self, row, col, parentCandidate):
        self.row = row
        self.col = col
        self.parentCandidate = parentCandidate
```

Deze Class creëert een object van het type Candidate.

Een Candidate object heeft een aantal attributen: Een row, een col, en een parentCandidate.

De row en col komen overeen met een rij en een kolom in het doolhof. Dit is de positie van het vak, de startpositie op het nulletje bevind zich op row 1, col 7.

Tijdens deze opdracht moet er een lijst van alle candidates bijgehouden worden, zodat het algoritme weet welke posities al bekend zijn, en hoe ze bereikt zijn.

Deze lijst is in het bestand MazeEx.py al gecreëerd, en als je de bijgeleverde helper methodes gebruikt worden nieuwe items automatisch toegevoegd, mits je de al gemaakte lijst meegeeft als parameter. Als je je eigen helper functies besluit te schrijven moet je zelf zorgen dat deze lijst bijgehouden wordt.

Tip: Maak het jezelf niet te moeilijk.

Het parentCandidate attribuut verwijst naar het indexnummer in de candidates lijst van het naastliggende vak waar het algoritme vandaan is gekomen.

Omdat de startpositie het geen parentCandidate heeft krijgt het Candidate object voor deze positie een -1 als indexnummer. Om aan te geven dat er in realiteit geen parentCandidate is. Alle aanliggende vakken aan de startpositie zouden dan dus voor parentCandidate een nul hebben, omdat de index van de startpositie in de lijst nul is.

Maze.py

Deze Class creëert een object van het type Maze

Dit object heeft een set attributen en methodes.

Het eerste attribuut van de Maze class is self.maze.

Dit attribuut is een tweedimensionale lijst waarin het doolhof weergeven wordt (Zie foto). Tweedimensionaal houd in dat het een grote lijst is met daarin kleinere lijsten waar de daadwerkelijke informatie on opgeslagen staat. Stel je een tabel voor met rijen en kolommen. Geef het nummer op van een rij (de hoogte), en dan heb je een lijst waarvan je een nummer op kan geven voor een kolom (de breedte). Om specifieke items uit een tweedimensionale lijst op te halen moet je dus ook twee index nummers op geven, een beetje zoals coördinaten, maar dan zijn de x-waarden en de y-waarde omgewisseld.

Het declareren en opvragen van items uit een tweedimensionale lijst ziet er dus ongeveer zo uit:

```
list = [[1, 2, 3],
        [2, 5, 6],
        [7, 8, 9]]

print(list[1][1])
resultaat:
              5
```

Vergeet niet dat veel programmeertalen beginnen met tellen bij nul, dus list[1][1], betekent eigenlijk tweede rij, tweede kolom.

Het tweede attribuut van Maze is ook een tweedimensionale lijst.

Hier zijn alleen alle plaatsen opgevuld met False of True. Hiervan wordt gebruik gemaakt om bij te houden waar het programma al is geweest

Verder zijn er verschillende methodes die je kunt gebruiken:

```
    def __str__(self):
        . . .
        for i in self.maze:
        . . .
        return string
```

De methode __str__() geeft een tekst object (string) waar het doolhof in staat als output.

```
    def drawRoute(self, candidates, i):
        . . .
        return self.__str__()
```

De methode drawRoute() stippelt een route terug naar het startpunt vanaf de positie op index ‘i’ in de lijst candidates.

```
    def isEndPoint(self, row, col):
        return self.maze [row][col] == 'X'
```

De methode isEndPoint() beantwoord True als er op de gegeven rij en kolom het teken van het eindpunt staat, en anders False.

```
    def hasWall(self, row, col):
        return self.maze [row][col] == '█'
```

De methode hasWall() beantwoord True als er op de gegeven rij en kolom een muur staat, en anders False

```
    def visited(self, row, col):
        self.beenHere[row][col] = True
```

De methode visited() geeft voor de gegeven rij en colom aan dat het algoritme daar geweest is in het attribuut beenHere

```
    def hasVisited(self, row, col):
        return self.beenHere[row][col]
```

De methode hasVisited() beantwoord True als het algoritme de gegeven rij en kolom al bezocht heeft, en anders False.

```
def checkUp/checkDown/checkLeft/checkRight (self,candidates, c):
    row = . . .
    col = . . .

    return self.verifyCandidate(candidates,c, row, col)
```

De methodes checkUp(), checkDown(), checkLeft() en checkRight() geven de rij en kolom van het vak in de corresponderende richting vanaf het vak op index c in de lijst candidates door aan de methode verifyCandidates().
In andere woorden ze geven de coördinaten van mogelijke volgende stappen door, en geeft het antwoord van verifyCandidates() ook weer terug.

```
def verifyCandidate(self, candidates, c, row, col):
    if (not self.hasWall(row, col) and not self.hasVisited(row, col)):
        self.visited(row, col)
        candidates.append(Candidate(row, col, c))

    return self.isEndPoint(row, col)
```

De methode verifyCandidate() kijkt of het vak op een gegeven rij en colom begaanbaar is, en voegt het toe aan de lijst bezochte candidates als dit het geval is. Voordat de methode afrond kijkt het ook of de gegeven positie het eindpunt is, in dit geval antwoord de functie True, en anders False.

MazeEx.py

Dit is het bestand waarin je de opdracht zult gaan maken.

Al aanwezig zijn de volgende stukken code:

```
if __name__ == '__main__':
```

- Dit is waar het programma begint met uitvoeren van code, dit wordt ‘de main’ genoemd
- Onder dit stuk zijn alle nodige objecten al gedeclareerd. Je hoeft hier dus niets aan te passen.

```
def depthFirst(candidates):
    while True:
    #   Your code here
        break

    return ""


def breadthFirst(candidates):
    while True:
    #   Your code here
        break

    return ""
```

- Dit is waar je de functies gaat toepassen
- Beide functies vragen om de parameter ‘candidates’
  Dit is dezelfde lijst die eerder in de opdracht genoemd is.

Om de functies goed te schrijven moet je je bezig houden met hoe je correct de first in first out en last in first out principes toe past in een while loop. Denk hierbij aan welke volgorde en richting je

door de lijst moet lopen, maar ook in welke volgorde je in verschillende richtingen zoekt, zodat je algoritme niet begint door in de verkeerde richting te zoeken.

Als de algoritmes hun route naar het einde hebben gevonden moeten ze de while loop eindigen en hun route uitstippelen. Hier kun je de drawRoute() methode voor gebruiken, maar je kunt net zoals bij de algoritmes zelf ook je eigen methodes schrijven.

Als je alles dan goed hebt gedaan zou je van beide algoritmes de kortste route naar het einde moeten krijgen.