# Hoi zäme , Willkommen zum finalen Projekt in Algorithmen

# Abgabe über github, erwartet wird eine ausführbare Python Datei

# Convoluted Kernel Maze

In this project, use depth first search and breadth first search to construct a corn maze littered with goodies. After building the maze, reflect on the algorithms available in your toolbox to find a path through the maze.

**Tasks**

### Initialize the Grid

**1.**

The first step in this project is building the maze to be solved. We are working towards a function build\_maze which returns a two dimensional array. The size of the array is defined by input parameters m and n.

Each cell of the maze will contain a wall, item from the input swag, or be empty.

Begin by defining the build\_maze function with these three arguments - m, n, and swag.

Hint

def build\_maze(m, n, swag):

**2.**

Within build\_maze, create a grid variable and set it to an empty list.

Hint

grid = []

**3.**

The grid is our representation of the m by n array. In Python, this will be a list of lists.

Use two nested for loops or a list comprehension to create and initialize every location in grid to the string "wall".

We are going to start our maze as a solid corn field and "mow" down locations for our path.

Hint

As a for loop:

for i in range(m):

row = []

for j in range(n):

row.append("wall")

grid.append(row)

**4.**

Now that we have our cornfield, or grid, let's return it to make sure it looks right.

return the grid at the end of build\_maze

Test the function by calling it with an m of 5, n of 10, and swag set to None. Print the result.

Save your code and run it by typing python3 script.py in the terminal.

Hint

def build\_maze(m, n, swag):

grid = []

for i in range(m):

row = []

for j in range(n):

row.append("wall")

grid.append(row)

return grid

print(build\_maze(5,10,None))

**5.**

That looks like.... something. Let's see if we can clean up the way we are printing this data.

Make a new function print\_maze that takes a grid as an argument.

Hint

def print\_maze(grid):

**6.**

Similar to constructing the grid, we want to use for loops to iterate over every rowand column of the maze.

Begin by iterating over every row in the grid. For each row, create a printable\_rowand set it to an empty string.

Hint

for row in grid:

printable\_row = ''

**7.**

Nested within this for loop, iterate over each cell in this row.

Depending on the contents, we will add a character (char) to the printable\_row.

For now, set this char to | (the vertical bar character) if the cell is "wall" and a space () otherwise.

Be sure to add this char to the printable\_row.

Hint

def print\_maze(grid):

for row in grid:

printable\_row = ''

for cell in row:

if cell == 'wall':

char = '|'

else:

char = ' '

printable\_row += char

**8.**

Now we need to print something to the terminal. Add a print statement outside of the inner for loop to print the printable\_row.

Test the build\_maze function again, but instead of passing its output to print, pass it to print\_maze.

You should see your maze in the terminal.

Hint

def print\_maze(grid):

for row in grid:

printable\_row = ''

for cell in row:

if cell == 'wall':

char = '|'

elif cell == 'start':

char = "s"

else:

char = ' '

printable\_row += char

print(printable\_row)

print\_maze(build\_maze(5,10,None))

**9.**

Okay, now you have some scaffolding to visualize what is going on as we develop the build\_maze function.

The first thing we need to do is select a starting location. Let's randomly select one from within the range of m and n.

Make sure to import randint from random. Within build\_maze before returning the grid, we will use randint to randomly index into the grid.

Calculate the first index, start\_i, which should be between 0 and m - 1.

Calculate the second index, start\_j, which should be between 0 and n - 1.

Use these indices to change the corresponding cell value within grid from "wall" to "start".

Run your script a couple of times. Is the starting location changing?

Hint

grid[randint(0, m - 1)][randint(0, n -1)] = 'empty'

### Mow the maze

**10.**

We now need to define a mow function. Here's how the algorithm will work:

From a starting location, this function randomly picks a direction to explore. If the grid location 2 moves in that direction is still within the grid and is currently a "wall", we set the cell to empty. We also set the cell 1 move in that direction to empty so it is connected to our path.

The algorithm continues exploring down this path until no directions work for exploring. We then backtrack to the previous location. This may remind you of Depth First Search.

We explore 2 moves in each direction because if we only explored 1 move, every single wall would be removed and it wouldn't be so much of a maze as an empty field. 3 moves would make for a more sparse maze and so on.

We will implement the mow function as a recursive algorithm.

We're going to begin by defining the function mow that takes the maze grid as well as location indices i and j as parameters to explore from.

Hint

def mow(grid, i, j):

**11.**

We need a list of possible directions to explore. Define directions to contain "U", "D", "L", and "R". These will represent up, down, left and right.

Hint

directions = ['U', 'D', 'L', 'R']

**12.**

We will randomly be exploring directionsas long as there is at least 1 left. Create a loop that continues as long as directionscontains at least a single element.

Hint

while(len(directions) > 0):

**13.**

We now want to randomly pick a direction to explore. Use randint to select a number between 0 and the current length of directions minus 1. Assign this number to directions\_index.

Hint

directions\_index = randint(0, len(directions) - 1)

**14.**

Now we can remove this direction from directions and explore. Assign this direction to a variable, direction and remove it from the array.

Hint

This can be done in one step by passing the index as an argument to .pop():

direction = directions.pop(directions\_index)

**15.**

Now we need to consider what happens for each of the possible 4 directions. Remember how we are printing the gridand that we are currently exploring grid[i][j].

If we are moving in the "U", or up, direction, i is decreasing. What do the other directions mean?

Hint

* "U" - i decreasing
* "D" - i increasing
* "L" - j decreasing
* "R" - j increasing

**16.**

Create a set of conditionals to handle the 4 directional cases.

Hint

if direction == 'U':

elif direction == 'D':

elif direction == 'L':

else:

**17.**

For each of these directions, remember that we are moving two spaces in that direction. Check to make sure that the resulting location is still within the grid. If not, use continue to move to the next iteration of the loop.

Hint

if direction == 'U':

if i - 2 < 0:

continue

elif direction == 'D':

if i + 2 >= len(grid):

continue

elif direction == 'L':

if j - 2 < 0:

continue

else:

if j + 2 >= len(grid[0]):

continue

**18.**

Remember that we only want to mow in a particular direction if the resulting space is currently a wall. If it is, set it to empty as well as the position one space in this direction.

Hint

if direction == 'U':

if i - 2 < 0:

continue

elif grid[i - 2][j] == 'wall':

grid[i - 1][j] = 'empty'

grid[i - 2][j] = 'empty'

elif direction == 'D':

if i + 2 >= len(grid):

continue

elif grid[i + 2][j] == 'wall':

grid[i + 1][j] = 'empty'

grid[i + 2][j] = 'empty'

elif direction == 'L':

if j - 2 < 0:

continue

elif grid[i][j-2] == 'wall':

grid[i][j-1] = 'empty'

grid[i][j-2] = 'empty'

else:

if j + 2 >= len(grid[0]):

continue

elif grid[i][j+2] == 'wall':

grid[i][j+1] = 'empty'

grid[i][j+2] = 'empty'

**19.**

If we have found a wall to remove, continue to mow from this new location recursively.

Hint

if direction == 'U':

if i - 2 < 0:

continue

elif grid[i - 2][j] == 'wall':

grid[i - 1][j] = 'empty'

grid[i - 2][j] = 'empty'

mow(grid, i - 2, j)

elif direction == 'D':

if i + 2 >= len(grid):

continue

elif grid[i + 2][j] == 'wall':

grid[i + 1][j] = 'empty'

grid[i + 2][j] = 'empty'

mow(grid, i + 2, j)

elif direction == 'L':

if j - 2 < 0:

continue

elif grid[i][j-2] == 'wall':

grid[i][j-1] = 'empty'

grid[i][j-2] = 'empty'

mow(grid, i, j - 2)

else:

if j + 2 >= len(grid[0]):

continue

elif grid[i][j+2] == 'wall':

grid[i][j+1] = 'empty'

grid[i][j+2] = 'empty'

mow(grid, i, j + 2)

**20.**

Back within the build\_maze function, before the grid is returned, call the mow function with the grid and the maze starting indices.

If the maze doesn't seem too exciting, try increasing the size.

Hint

mow(grid, start\_i, start\_j)

### Dropping swag

**21.**

As we build the maze, we want to drop goodies for the patrons to pick up as they explore the maze.

We also need to find the end of the maze. We want the end to be as many moves away from the start as possible.

To achieve both of these goals, we'll use Breadth First Search. As we search, we will occasionally drop an item in the maze.

When we have exhausted the visitable locations in the maze through BFS, we know we are the farthest number of moves from the start location.

Begin by defining a function explore\_mazethat takes the grid, start\_i, start\_j, and swag as arguments.

Hint

def explore\_maze(grid, start\_i, start\_j, swag):

**22.**

The first thing we'll need is a copy of gridto mark up with the visited state of each cell. Create grid\_copy using for loops or a list comprehension.

Hint

Using a list comprehension:

grid\_copy = [row[:] for row in grid]

grid\_copy = []

for row in grid:

grid\_copy.append(row)

**23.**

For our BFS queue, we will maintain a list of coordinates. The bfs\_queue should be initialized to contain the start coordinates [start\_i, start\_j].

It will also be helpful to have a list of all the directions we consider exploring ('U', 'D', 'L', 'R'). Assign this to the variable directions.

Hint

bfs\_queue = [[start\_i, start\_j]]

directions = ['U', 'D', 'L', 'R']

**24.**

While there is still a queue, we will pop a location from it to explore around.

Create a while loop that continues as long as there are items in the bfs\_queue. Within the loop, use multi-assignment to assign the current coordinates, i and j.

Hint

while bfs\_queue:

i, j = bfs\_queue.pop(0)

**25.** AM HERE NOW !

We want to place swag 10% of the time and only if we are not at the start. Use randint to create an if statement to determine if we are placing swag at a location.

Hint

if grid[i][j] != 'start' and randint(1,10) == 1:

**26.**

Select an item from the swag list using randint and save it to the current location within grid.

Hint

grid[i][j] = swag[randint(0, len(swag) - 1)]

**27.**

Within grid\_copy, mark this location as visited.

Hint

grid\_copy[i][j] = 'visited'

**28.**

Now we can explore around our current location. We must first consider each direction we can move in. Use a for loop to iterate over each direction in directions.

Create variables explore\_i and explore\_jset to i and j to consider new locations.

Hint

for direction in directions:

explore\_i = i

explore\_j = j

**29.**

Use conditionals based on the direction to update the explore\_i and explore\_jcoordinates. Unlike before, we are now moving 1 cell in a direction.

Again, the directions are:

* "U" - i decreasing
* "D" - i increasing
* "L" - j decreasing
* "R" - j increasing

Hint

if direction == 'U':

explore\_i = i - 1

elif direction == 'D':

explore\_i = i + 1

elif direction == 'L':

explore\_j = j - 1

else:

explore\_j = j + 1

**30.**

Similar to before, if the new location defined by explore\_i and explore\_j is outside of the grid, just continue. We can't explore here.

Use a conditional to continue if the location is outside of the grid.

Hint

if explore\_i < 0 or explore\_j < 0 or explore\_i >= len(grid) or explore\_j >= len(grid[0]):

continue

**31.**

If the new location is not a wall and not visited, add it to the bfs\_queue.

Hint

elif grid\_copy[explore\_i][explore\_j] != 'visited' and grid\_copy[explore\_i][explore\_j] != 'wall':

bfs\_queue.append([explore\_i, explore\_j])

**32.**

After exploring all the unvisited nodes, we should be at the end of our BFS. Since we want a long path through the maze, the final values of i and j are the indices of the "end" of the maze.

Update grid with the end of the maze just before the end of explore\_maze.

When you're done, modify build\_maze by adding a call to the explore\_maze function, just before the return statement. Pass in the grid, start\_i, start\_j, and swag.

Hint

grid[i][j] = 'end'

Then, in build\_maze, modify the last few lines so it looks like:

# ...

mow(grid, start\_i, start\_j)

explore\_maze(grid, start\_i, start\_j, swag)

return grid

**33.**

To test this function, we need to update our print function. Modify print\_maze to keep printing | for "wall", but update it to explicitly print for "empty, and print the first character of anything else.

Hint

for cell in row:

if cell == 'wall':

char = '|'

elif cell == 'empty':

char = ' '

else:

char = cell[0]

**34.**

Lastly, we need some swag! Change the swag list from None in the call to build\_maze to something like ['candy corn', 'werewolf', 'pumpkin'].

Avoid using words that start with "e" or "s", so you can see the start and end of the maze.

How does the maze look now?

Play with changing the size of the maze.

Hint

print\_maze(build\_maze(10,30,['candy corn', 'werewolf', 'pumpkin']))

### Presentation

**35.**

Now that you've built a maze, spend some time considering how the maze was built and how you would solve it as a patron.

For this part of the project, create a presentation addressing the following:

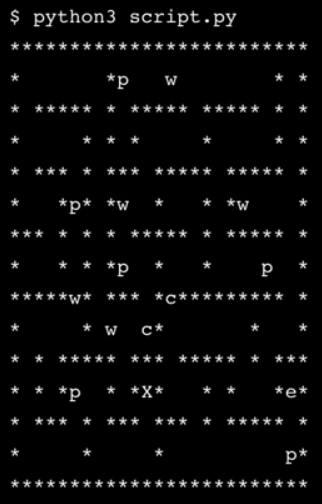
* While building the maze, we attempted moving 2 cells at a time.
  + What would the maze look like when moving a larger number of cells?
  + What would the maze look like if this number was not constant?
* What algorithms could you use to find a path through this maze? Compare and contrast at least 2.
* How does knowing the algorithm used to generate the maze influence the best algorithm to solve it with?
* As a patron picking up swag along the way, how might you best store the list of items you've collected?
* If the farmer asked you to sort the items you collected before leaving the maze, what sorting algorithms would you consider using (assume a much larger list of possible swag)?
* How does the quantity and variety of swag influence your answer?

Feel free to implement or tinker with the code to help think about these questions. When you are ready, submit the code and presentation using the link in the previous item in the syllabus.

Bsp Ergebnis :



Oder So:



Abgabetermin:

20.02.2019 18:00