Lecture 4: Solving crossword-riddles (CWRs).

The first assignment of this class is to write a python code which can find at least one solution of a 3x3 CWR, in which one word is given and the other 3 words fit in a specific way to another, as described in the picture bellow.

T	Ü	R
	■	

This is an image I used at my thesis so it includes a German word. I´ll change it possibly in the next version of this text.

According to my previously created code, most German 3x3 CWRs have more than several thousand solutions. Since the English vocabulary is a bit bigger, I expect English CWRs to have way more solutions.

The rest of this text is a guideline on how to do that. You may want to do that all by yourself, or read parts (or all) of it. If you choose the former, remember Lectures 2 and 3, especially that we want a depth-first, brute-force tree-approach. A list of all available 3-letter words is located at http://geo.sibralien.de/reddit

General description of the implementation

As stated in lecture 3, to solve the problem we want to generate all possible legal states to a given CWR. We want to find all possible solutions. This is how we will proceed:

1) Create a vocabulary of all English words with exactly 3 letters, which we can iterate through.
2) Create a vertex class, with the attributes CWR (which is a List of strings which are the words in the representing CWR in a specific order), father (which is the object which has the father-CWR as a father), children (which is a list of objects representing all children of the vertex) and depth (which is an integer representing the depth of the tree we are in).
3) Create a recursive algorithm that creates a search tree. In this search tree, all vertices represent legal CWRs. A legal CWR is a CWR where all words belong to the English language. The root at depth 0 is a vertex that represents the CWR with only word 0 in it (the original word of the CWR). Children of vertices represent CWRs that contain all words the father-CWR contained, plus another fitting word. Vertices of the depth 3 thus represent completely filled legal CWRs, aka solutions.

Note: complete example codes are given at the end of this lecture.

Detailed description

1) Getting the vocabulary

To solve this, we first need a list of available 3-letter words. We will use the text file I uploaded to this website. An approach to get the vocabulary is to create a python list with all these words as separate string-type elements. Python supports the command *open*, which lets us read the txt file and copy them to a list. We have to read the lines, and split the text. Find the appropriate functions here (strings) and here (file reading, see chapter 5.9).

2) **Representing the CWR**

Now we have to find a way to represent a crossword in python. We can choose a list like [word0, word1, word2, word3], where each word is positioned like in the image below: from (1,3) downwards for word 1, from (3,1)to the right for word 2 and from (1,1) downwards for word 3.

	1	2	3
1	3 T	Ü	1 R
2	O		A
3	2 R	O	T

The above CWR would be represented as [‘TÜR’,’RAT’,’ROT’,’TOR’].

We can use this because python has the nice capability of accessing a string through the letters like lists:

```
>>> my_string = "TAB"
>>> my_string[0]
'T'
```

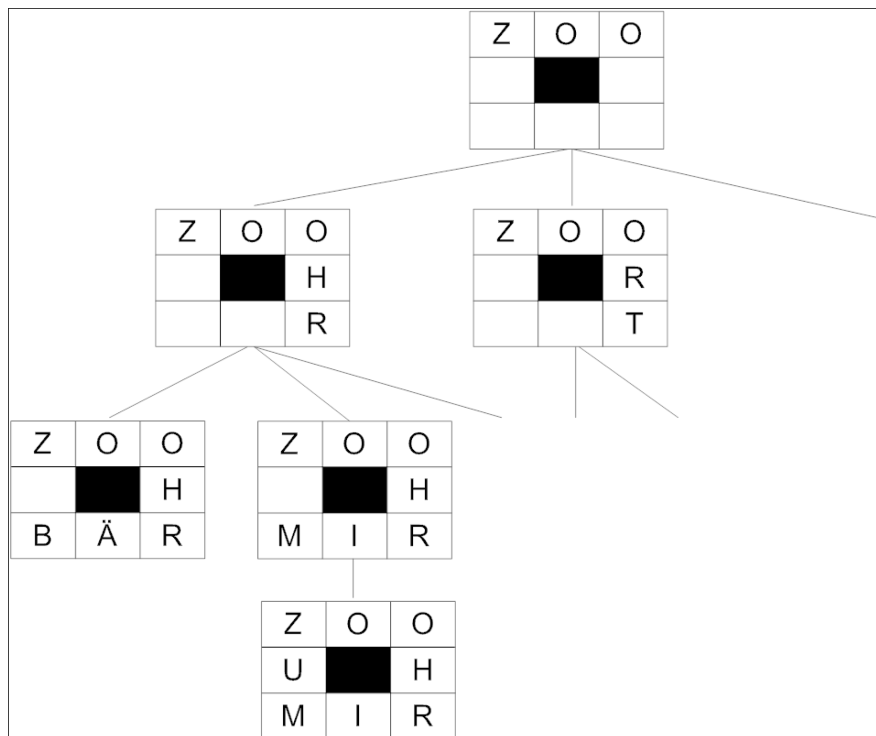
With this representation, each word has to fulfill certain formalized requirements, for example

word1[0]=word0[2].

3) **Creating the search algorithm**

So this is the toughest part. We first need a concept for the search tree. The concept is to create a tree where all vertices represent legal CWRs. An example of this can be

found at the picture below, which is again German.



I have to note that there is no German word "Z_B" and no German word "Z_M" other than "ZUM", therefore the CWR [ZOO, OHR, BÄR] has no children, and [ZOO, OHR, MOR] has got only one child, as shown. This is an example of how the tree vertices should look; once again you should note that only legal CWRs actually become vertices of the tree.

A pseudo code for the algorithm could go like this:

1) If vertex depth is 3, we found a solution, so break the recursion for this vertex
2) For every word in the vocabulary, if the word fits to the current CWR, add it to it and create a vertex with the new CWR.
3) Redo the procedure with the newly created vertex

So, you hopefully can implement the code now. If not, you can find some example code in the next pages.

```
# -*- coding: iso-8859-1 -*-
def create_vocabulary():
    global vocabulary
    text_file = open("c://words.txt", "r")
    text_list = text_file.readlines()
    text_file.close()
    vocabulary = text_list[0].split()
```

```
# Vertex class
class Vertex(object):

def __init__(self, father, depth, CWR):
    self.text = text
    self.depth = depth
    self.children = []
    self.father = father
```

```
#solution! Dont read that yet!
def create_solution(word):
    global solutions
    root = Vertex(None, 0, [word])
    solutions = []
    create_tree(root)
    return solutions

def create_tree(word_vertex):
    global solutions
    if word_vertex.depth == 3:
        solutions.append(word_vertex)
        return
    for word_new_text in vocabulary:
        if check_legality(word_vertex, word_new_text):
            new_text = word_vertex.text + [word_new_text]
            word_new = Vertex(word_vertex, word_vertex.depth+1, new_text)
            create_tree(word_new)

def check_legality(word_vertex, word_new_text):
    return (word_vertex.depth == 0 and word_vertex.text[0][2] == word_new_text[0]) or \
        (word_vertex.depth == 1 and word_vertex.text[1][2] == word_new_text[2]) or\
        (word_vertex.depth == 2 and word_vertex.text[2][0] == word_new_text[2] and \
       word_vertex.text[0][0] == word_new_text[0])
```