Overview

There's nothing intrinsically different or magical about AI programming; it's all just software…only it just happens to be specialized towards certain analytic orientations, algorithms, and problem-solving goals. Well heck, you're all rock-star programmers, right? So let's get that programming mojo warmed up with a little basic problem-solving challenge!

I'm calling this "Fred Flintstone problem-solving" because it's just that: we know nothing at this early point in the course, so we can offer only completely off-the-cuff, uninformed, "naive" solving of a problem. As we move forward, we'll soon develop a broader understanding of the intellectual "terrain" surrounding problems like these: what we are really doing, what the alternative solution approaches in this terrain are, and how to think about what will work best. But for now, we're just going to get out our caveman club and flail away… and then grunt happily when we get a solution!

The Problem:

In this first small programming exercise, we will consider how we can solve Boggle. If you haven't played in awhile, here's the gist of the game: There are 16 cubes with letters on the faces. These cubes are randomly arranged in a 4x4 matrix by shaking the boggle game. The goal of the game is to make words out of these letters by traversing adjacent (horizontal, vertical or diagonal) tiles. This "chain" of letters may snake all over the board, but you can only use each tile once, i.e., no fair using the same letter twice in a word. In a fixed amount of time players must make as many words as possible. Words are then scored as follows: 1 point for each 3-4 letter word, 2 points for a 5-letter word, 3 points for a 6-letter word, 5 points for a 7-letter word, 11 points for a 8 (or more)-letter word.

The Assignment: Overview

In this problem, you are asked to solve Boggle boards exhaustively in whatever way you can (go Fred Flintstone!): given a particular boggle board as input, your algorithm should enumerate all possible words that can be found in that Boggle board.

The dictionary we will use for our game of Boggle is the Tournament Scrabble Wordlist which includes 178,691 words. I've cleaned up and provided a file of dictionary words for you here.

Your program should be given a dictionary and an NxN boggle board (either in command line or program params, up to you). It should then run and discover all possible words existing in the given board and print out a summary of its findings.

Details:

- You are permitted to use only basic “standard” Python data structures, i.e., lists, dictionaries, and sets. Nothing fancy like Trie, Queue, etc. which are complexity overkill that will just slow you down, confuse you, and impede your understanding of what you’re really doing. In short, the only package you may import for this program is the time package to do the timing. Everything else is standard core Python.
- Any dictionary used will have the same format as the twl06 dictionary, i.e., one word per line in a text file. This means that, given the appropriate dictionary file, your program can Boggle in French just as easily as in English!
- A boggle board file will consist of N lines of N letters separated by spaces. You should ignore extra space at the end of the line or extra newlines at the end of a file. (Hint: check out the python strip() function)
- Your solver should be able to take in whatever NxN board size you pass it, simply deducing the board size from the given board input file. A solid algorithm will work just as well on 2x2 boards as on 10x10 boards!
- Efficiency matters in time-based scenarios (like games). Keep efficiency close in mind as you design your code.

This is not a particularly hard problem…provided you think it through! (Hint: elegance, recursion). Just as a reference point: my solution has one function of about 15 lines, plus three smaller helpers to load/print out boards and stuff. Without comments, the whole thing fits on a page.

How to think about this problem:

The main point of this exercise is to just get everybody warmed back up on Python, but of course we also want to get our problem-solving knives honed up. Although we don’t know much about it in any formal sense (but we will soon!), this problem involves exploration of a large space of possible states to find solutions…which is what a lot of AI is based on. In this case, a “state” is a place I’ve ended up after taking some path across the board, starting from some starting position…the letters in the path you’ve followed form a string…which might or might not be a “solution”, i.e., depending on whether this string you have is in the dictionary. Ok, so it’s really pretty simple: you have to write a program that (a) starts in each of the possible starting points on the boggle board; and (b) explores all possible paths from that point, recording/scoring any correct words it finds along the way. Put this way, it’s simple: your key needs are a function that, given a current position, generates all possible adjacent positions (gotta stay on the board!). Then you have to consider that not all adjacent positions are possible: you have to subtract away any tiles that are already on the path you’ve already explored (can’t use a tile twice!). Then you put it together:

start at some position, see if the letter it contains is a legal word (if so, score it), then generate all possible next positions to jump to from there, jump to each one (adding its letter to the path)...and then repeat until none of your paths can go any further. If your brain is thinking “massive recursion!” then you’re on the right track...

Part 1 Deliverables

- A “loadBoard” function that takes the filename of a board file, and loads that in to work on — returns a new board data structure (NxN matrix). Obviously you’ll call this right at the start.
- A “printBoard” function. Takes in a reference to a loaded board data structure (an NxN matrix) and prints it out. Simple.
- A “possibleMoves” function. Takes in a current position (just an x-y pair) and a boggle board and generates all possible next positions (x-y pairs in a list, set, or whatever you decide).
- A “legalMoves” function. Takes in a list of possible moves (i.e. generated by PossibleMoves) as well as a path (list of x-y pairs) of places you’ve already been, and essentially subtracts the latter from the former: the only legal moves are possible moves minus any places that you’ve already been.
- An examineState function that takes in a boggle board, a current position, and a path up to that position. It adds the current position’s tile to the path, computes the word now formed by that path, and returns a tuple of (<current word generated>, <yes/no depending on whether that word is in dictionary>).
- These functions should be defined in a file named project1_funs.py.
- Download project1_test1.py and put it in the same directory. Running it should give output like this.
- You must use this board.txt and twl06.txt as your input files.
- Submit a PDF with (1) Cover sheet: Name, course, assignment title, date, (2) output of running project1_test1.py including prompt, input/output for each command, and a newline between commands, (3) your project1_funs.py source code with comments.

FAQ

- Where should I put call to loadBoard()? At the top of test.py
- What should I do if interpreter.py does not work? Please type “python” to start the REPL, then copy and paste the test code into the python interpreter so we can see the prompt, the input, and the output for each command.

- Can my functions use different inputs and/or return different outputs than shown in the test/example code? Your functions should take the same inputs/outputs as shown in the test/example code.
- For functions which accept mutable data structures like lists as arguments, should my function modify or copy? your choice as long as the output is correct.
- How should I organize the board data structure and indexing? You are free to organize your data structure / indexing as you like as long as the output is correct.