CSC 236 Data Structures

L02: Backtracking and Caves

Objectives

- Gain practice in using stacks
- Use a stack made of a composite data type
- Learn about backtracking
- More practice in creating and using nested lists

Backtracking

A very important use of the ***stack*** data structure is in backtracking, a process by which one investigates an option for solving a problem and then abandons it for another option when it does not lead to a solution. An intuitive to explain this process is with hiking.

Suppose you are hiking in the woods where there are many paths to follow and are trying to reach a specific campsite to meet with some friends. There are many paths to take where one or two can lead to the campsite, whereas others will take to around the mountain to some other part of the woods. At each intersection, therefore, you must make a choice which path to follow. For example, if the path forks, do you go to the left or right? If there is a four-way intersection, do you go left, right or straight?

Suppose you picked a sequence of turns and end up at a cliff, where there is a great view, but it is not the campsite.

What do you do? Well, a typical and intuitive response would be to go back to where you most recently had to choose a direction, and select a different path with the hopes that this time, it will work. If none of the alternatives led to the campsite, then you have to back even further to the second to last intersection and try those alternatives. If none of those alternatives work either, then you have to back one more intersection. Now, if you end up back at your starting point and have not found the campsite, you will probably need to call your friends to check to make sure you are at the correct park (assuming you and your friends have cell phone reception).

Note that as you go back and try all the paths, you need to keep track of what you had done before so you do not repeat mistakes (hence called "backtracking"). The intuitive way to store these alternatives is in a LIFO manner because when a choice did not work, the alternatives to test are at the intersection you had most recently encountered.

Another example of using backtracking can be demonstrated by the problem of finding treasure in a cave. If there was only one way through the cave, that would not be a challenge (people who place treasures would not make finding them that easy). Instead, there are various intersections of paths, and a treasure hunter must choose one to follow. It is entirely possible that the chosen path will lead to a "dead end", meaning that (1) there is no way to go any further, and (2) there is no treasure found yet. In these situations, one can return back to the last point at which a choice was made, and select an alternative that will hopefully lead to the treasure.

What do caves, maps, and backtracking have to with this lab?

You are work on this lab with a partner or individually. If you decide to work with a partner, be sure to follow good **"pair-programming"** practices as you did for CSC226.

Your task is to write a program that starts with an initial starting point and a map of a cave. It then finds as much loot in the cave as possible by exploring all the accessible locations. Every time you encounter treasure, your program will output the path you took from the starting point to the loot. Every time you reach a dead-end, you will retrace your steps and try another of the untried paths until you return back to your starting point having explored all the paths accessible to you. Once the program has explored all accessible spots on the map, it stops.

In this lab, choosing an appropriate algorithm and set of data structures will prove crucial to how difficult the lab will be.

Details

1. No matter where you are in the cave, you can only travel in one of the four cardinal directions, north, south, east and west:

```
  N
W me E
  S
```

2. Because the position on the map is a location (row, column), you should use a compound data structure to represent this position. For example you might consider using a tuple (row, col) or you might consider using a small class:

```
3. def pos:
4.     self.rol # Which "ROW" is this position?
       self.col # Which "COLUMN" is this position?
```

   You can choose the data structure which makes sense.

5. When you have multiple path options, it could simplify your algorithm if you always try the available paths in exactly the same order (such as always first look to the north, then to the west, then to the south, and finally to the east).

6. The maps have paths marked by a '.', walls marked by a 'W', treasure marked by a 'T', and your position is marked by a 'M'.
   For example, a layout of a cave map at a particular point in time while your program is exploring it may look like the following:

```
W W W W W W W W W W W W
W W W . . . W W W . W W
W W . . . . . W W . W W
W W . W . W W W . . . W
W W . . W . . W W W . W
W W W . W W W . W W . W
W T . . . . W . W W . W
W W W W M W W W W . . W
W W W W . . W W W . . W
W . T W . W W W W . W W
W . W W . W W W W . . W
W . W W . W W W . W . W
```

```
W.  WWWWW. . . . W
W. .  WW. .  W.  WWW
WW. . . . . . . WWW
WWWWWWWWWWWWW
```

7. The outside edges of the map will have only walls; i.e. it will not have paths. Note that it will be particularly important for you to think about what compound data structure you will use for this rectangular set of data. Be sure to think about whether you will want it to be mutable or immutable.

8. The cave map file:
   1. Your program will ask from the user the name of the file that contains the cave map it will explore, but that is the only input the program will ask from the user.
   2. The dimensions will be on the first line of the file as a pair of numbers, the first for the number of rows and the second the number of columns.
   3. The layout of the map read from the file is rectangular, where there is one row per line where each position has no spaces between other positions. The map above will look like this:
      ```
      16 12
      WWWWWWWWWWWW
      WWW...WWW.WW
      WW.....WW.WW
      WW.W.WWW...W
      WW..W..WWW.W
      WWW.WWW.WW.W
      WT....W.WW.W
      WWWWMWWWW..W
      WWWW..WWW..W
      W.TW.WWWW.WW
      W.WW.WWWW..W
      W.WW.WWW.W.W
      W.WWWWW....W
      W..WW..W.WWW
      WW.......WWW
      WWWWWWWWWWWW
      ```
   4. The starting position is indicated by the 'M'---you may assume that there is only one 'M' in the cave.
   5. Here is a sample test file: **<u>cave_sample.txt</u>** You are expected to create at least three additional map files which test boundary conditions.

Suggestions and Assumptions

- It is much more fun to watch your program run if it displays the map on the screen after each move.
- It is not a good idea to assume that there is only one position where there is treasure, or that if there is treasure, it is accessible. In other words, maps can be configured in which you return empty handed or loaded with treasure found in multiple spots in the cave map.

- You must use backtracking and you must effectively use a stack for the backtracking.
- For each treasure you find, you must display the path to that location, which can be implemented as a stack.
  Give some thought to how to do this display well.

Design to Implementation

It is recommended that you first start with a clear idea of what the problem you are trying to solve is and an outline of your solution in enough detail so that someone else can implement it for you in any language. Note that you should use English and not programming language specific statements (like Python code).

Do NOT try to implement anything yet.

In the reflection document (explained below), you will include this design as a response to the first prompt.

Once the general solution design is done, you can start to fill out some more details. By breaking the problem into pieces, you can design functions (that encapsulate specific operations) with details clearly explaining how they will solve each of the smaller pieces. You should specify the input each function gets, what it produces or outputs, and a rough outline (NOT CODE!) of what you want it to do.

Once you have finished with the design, you can start to implement each function, test it with known inputs and outputs, and start to build your complete program piece by piece.

Reflection

You are work with a partner for this lab or individually.
If you decide to work with a partner, be sure to follow good "**pair-programming**" practices as you did for CSC226.
Answer the following questions in *yourusername(s)-L01-reflection.doc*x:

- AUTHORSHIP: Describe who did the work on this lab. If you worked as a pair, did you use good pair-programming practices? Explain.
- INITIAL DESIGN PLAN: What is a pseudocode design plan which meets the computational requirements of this lab?
- SUMMARY: A brief summary description of the design and implementation, including how much your initial design plan evolved, the final result you achieved and the amount of time you spent as a programmer in accomplishing these results. This should be no more than two paragraphs.

- IMPLEMENTATION: A list in bullet form of specifically what was accomplished including any challenges overcome and innovations that were not specifically required by the assignment.
- TESTING: A list in bulleted form of all input values used for testing. Here you should be careful to select representative input cases, including both representative typical cases as well as extreme cases.
- FILES: A list in bulleted form of the names of all files submitted (source code and input, etc.)
- ERRORS: A list in bulleted form of all known errors and deficiencies.
- COMMENTS: A paragraph or so of your own comments on and reactions to the Lab.
- BIG-O: What is the big-O analysis of the method (or methods) which do the backtracking?

On L02: Documenting, saving, and submitting your files

To submit:

1. Create a folder called *yourusername-csc236L02*
2. Copy your program files into it. (all classes in addition to the driver file and three or more test maps you have created)
3. Complete your reflection *yourusername-reflectionL02.docx* and copy into this folder.
4. Zip this directory and submit your zipfile, *yourusername-L02.zip*, onto Moodle when you are done.