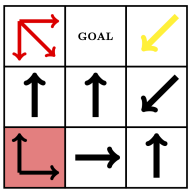
CSC263 Problem Set 3

Question 1: Alice Mazes

Alice Mazes (`http://www.logicmazes.com/alice.html`) are puzzles created by Robert Abbott. They are based on Lewis Carroll's *Alice In Wonderland* where Alice grows and shrinks during the story. Read the puzzle instructions from Abbott's site. Unfortunately, Abbott passed away a few years ago and the applet for executing the mazes has not been maintained. This means that you can't run the mazes and you will need to read the text below the first maze to learn how they work. Sadly the only example maze visible on Abbott's site does not have red or yellow arrows. In that example maze, there is no way to change the step size.

Below is another example of a simple maze where the solution requires a varying step size. Notice that in this one it is possible to increase the step size and then step right off the maze. That is effectively a dead end.



(a) Your first task is to design an efficient algorithm and associated data structures to solve an arbitrary Alice maze. Your algorithm must be guaranteed to find a shortest solution if one exists. You may assume that every maze is a square, has a single start location, and has a single goal location. You may also assume that for a given maze cell, all the arrows are the same colour. For example, in the maze pictured above, the top left square can not have a red arrow going to the right and black arrows going in the other directions. You may also assume that the arrows in the start cell are black. Beyond this, you shouldn't make assumptions about the design.[1] In particular, you should not assume that there is a solution to the maze. If one does not exist, your program should stop running and report this.

Submit a written description of your algorithm. It should be based on a graph algorithm we learned in class and you should be explaining **how that algorithm is adapted and applied to this problem**. Specifically, describe the data structures you are using to support your algorithm. The audience is someone who already understands the algorithms we learned and the specific terminology we used in lectures so you can refer to those terms.

(b) Design your own concise text (i.e. human readable) representation of a maze. Write clear specifications for this representation. Make sure that your design can represent any maze that meets the assumptions specified above. As an example in your explanation,

[1] In all of Abbott's mazes, the outer cells never have arrows pointing off the maze, but you don't need to add this constraint or make this assumption.

include the representation for the small maze above and then also submit the input file for this maze to MarkUs using the filename `example_maze.txt`.

(c) Implement **your algorithm** in Python. You must have a single file `Alice.py` and it must not import any Python modules other than sys (which you need for command-line argument processing.) The output of your program should be a shortest path through the maze as well as the length of this shortest path. The exact format of this output is up to you and may depend on your maze representation. Your program must run on teach.cs with the following command `python3 Alice.py inputfilename`. You can assume that the inputfilename passed to your program in the command-line argument corresponds to a **valid** maze file according to your representation in part b. It is fine to fail spectacularly if this assumption is violated. Don't spend time on error checking the input but do use excellent coding style including variable names and comments that will help the marker see immediately how your code connects to the algorithm you described in part a.

We will separately post some extra advice about writing your program including some sample code for reading the command-line argument and opening and reading from a file.

(d) Design a set of tests for your program and use them to demonstrate to the marker that your program works correctly. You should focus on **algorithmically important and interesting tests** not boundary cases that are uninteresting algorithmically. Each test case is a maze representation, **NOT a unittest**, and should be in its own .txt file. Suppose your first test was in the maze file called test1.in.txt. Then the TA should be able to run your code by saying `python3 Alice.py test1.in.txt` in your A3 repository on teach.cs.

In Q1.pdf (where you provide the English solution to this question), expand on each test case you designed, explaining what it is that you are testing, the specific input, the expected output and the actual output when you ran the test. You should reference the exact filenames of your test files which you will submit to MarkUs separately as part of your submission but a reader should be able to understand your testing **without opening any other files**. You may wish to provide a visual picture of the mazes (inside the discussion in Q1.pdf) if that makes it clearer for the reader.[2]. A word of advice, keep the input files small. We will post the LaTeX source for the maze picture above, but it is also fine to hand-draw maze examples or to not have pictures at all.

In order to earn full marks for correctness for your algorithm and its implementation, you need to convey to the marker (in the time that they have available to read and understand your Q1.pdf) that your code has been tested properly. Conveying technical information clearly, concisely and in a way that can be understood **quickly** is the skill we are practicing here. It isn't easy. If you say too much, the reader won't have time to dig through all your tests. If you say too little, they won't be convinced that the code was tested thoroughly. Finding the correct balance and presenting the information in a way that can be grasped quickly is the goal.

[2]You might need to compress the images to keep Q1.pdf within the Markus submission size limits

Expectations for Question 1

There are multiple learning goals for this question. One is for you to solidify your knowledge of graph algorithms by doing some actual implementation and designing test cases. The second is for you to practice concisely communicating to a technical audience. While it is important that your program works correctly, the majority of the marks will not come from part (c). Do not leave the testing or the written explanations as after thoughts or assign either of these tasks to a single person. Both partners should be involved in the designing, coding, testing and writing.

Question 2: Detecting a Sun

An undirection graph on $2n$ vertices is a *sun* if it consists of a single cycle of length n, each node in the cycle is connected to exactly one node of degree one, and there are no other cycles in the graph. For example, the following graph is a sun on 12 vertices.

(a) Suppose that you are given the adjacency list representation of an undirected graph with $2n$ nodes. Describe an algorithm that determines whether the graph is a sun. Your algorithm must run in $\mathcal{O}(n)$ time.

(b) Explain why your algorithm is correct.

(c) Justify that it runs in the required time.

Question 3: Amortized Analysis

The Vector class in Java is essentially implemented as a dynamic array. When you instantiate a Vector, you can specify an *initial capacity* (of the array) and a *capacity increment* (a fixed amount by which to grow the array each time it expands). Note that the capacity increment is additive, not multiplicative (i.e., new capacity = old capacity + increment). In this question, we will redo the dynamic array analysis we did in lecture using a capacity increment instead of a multiplicative expansion factor. We will still assume (as we did in lecture) that the only operation that can be performed on the data structure is INSERT.

(a) Suppose the initial capacity and the capacity increment are both equal to 10, and we use the accounting method with the same charge and cost as we used in lectures (i.e., each INSERT is charged \$5 and costs \$1 per array access.) Determine how many INSERT operations it will take before the data structure runs out of money, or prove that it never runs out.

(b) Let b be the initial capacity as well as the capacity increment, as above, where b is a positive integer constant. Starting with \$3 per INSERT, suppose we increase the charge

per INSERT by \$2 each time the array grows. So after b calls to INSERT, the charge becomes \$5, and after $2b$ INSERTs, the charge becomes 7, etc. Determine how many INSERT operations it will take before the data structure runs out of money, or prove that it never runs out.

(c) Using the aggregate approach, compute the amortized complexity of INSERT on this data structure, using some positive integer b for the initial capacity as well as the capacity increment, for a sequence of m INSERT operations performed on an initially empty array. Express your answer in terms of big Θ.

Question 4: Disjoint Set Representations

Let $T1$, $T2$, and $T3$ be three tree implementations of the disjoint set ADT. $T1$ uses union-by-weight without path compression, $T2$ uses union-by-weight with path compression, $T3$ uses union-by-rank with path compression all as discussed in lectures this term.

Consider a sequence of MAKE-SET, FIND-SET, and UNION operations $P = P_1, P_2, ..., P_n$, and suppose we perform P on each of $T1$, $T2$, and $T3$ (in each case starting with an empty ADT.) For a UNION(x, y) operation, when there is a choice, assume that the representative of the set that contains x becomes the new representative.

Two collections of trees are considered *equivalent* if they store the same items and can be drawn to look the same.

(a) What is the smallest number n such that $T1$ and $T2$ are not equivalent after executing P on each one? Justify your answer.

(b) What is the smallest number n such that $T2$ and $T3$ are not equivalent after executing P on each one? Justify your answer.