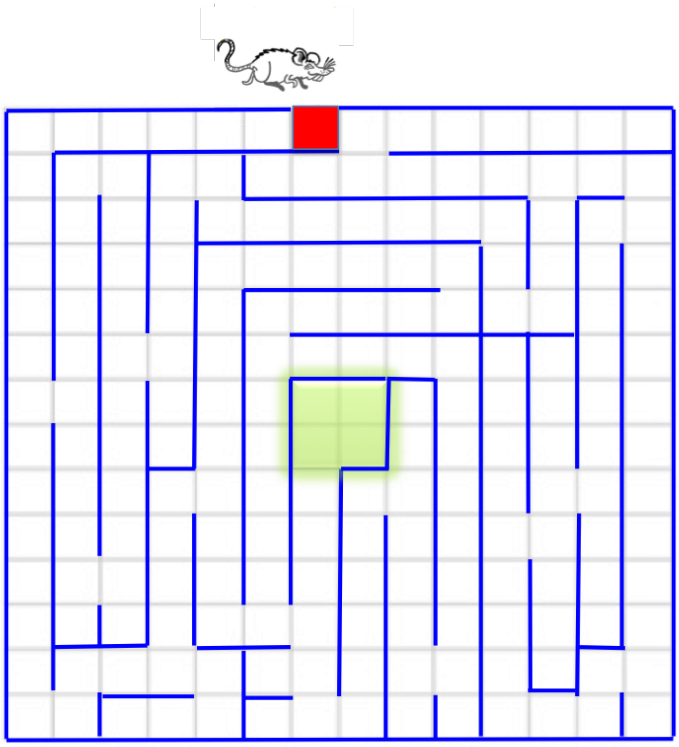
CS 364 – Assignment 2 – due 10/3

For this assignment you may again work with a partner.

Python platform For this assignment the Python code containing all of the searches and classes have been rearranged into a single file, with 2 versions. `search.py` contains the original version in which arc costs are uniformly 1. A new version, `search1.py`, extends the original version to include non-uniform arc costs. The file `romania1.py` demonstrates this new version by applying the five algorithms of problem 1 to the Romanian cities problem.

1. (25) Consider the maze built on a 14×14 grid shown below. Roger Rat wishes to find the shortest path from the entrance (at $(0, 7)$, if we use matrix coordinates for the grid) to any cell in the goal where there is plenty of limberger cheese to eat (i.e., any cell in the square $(6, 6), (6, 7), (7, 6), (7, 7)$). Can you help Roger find his dinner?



   (a) Design a state-space representation using the python search platform, including an admissible heuristic where needed, and run your model using each of the following search methods. (Note: all search algorithms are now contained in `search.py`).

      i. Depth-first search (`depth`)
      ii. Hill-climbing (`hill`)
      iii. Steepest-ascent hill climbing (`steep`)
      iv. Best-first search (`best`)
      v. Algorithm A* (`algA` or `algAM`)

   (b) Did the algorithms perform as expected? Explain.

Hint: The most difficult part is determining how to use the walls of the maze to restrict motion on the grid. One way to do this is to record a value for each cell that expresses the directions Roger

may travel if he's in that cell. This can be done using a 4 bit binary string, where 0 means wall and 1 means path.

Suppose we record these data starting with the top of the square ("North") and rotate clockwise (to "East", "South" and "West"). For example, the cell shown below, which is at $(0, 0)$ would be associated with the string `0110`.

Recording this for every cell results in a matrix of wall configurations of the following form:

```
maze0 = [
    ['0110', '0101', ...],
    ['1010', '0110', ...],
    ...
    ['1100', '1001', ...]
]
```

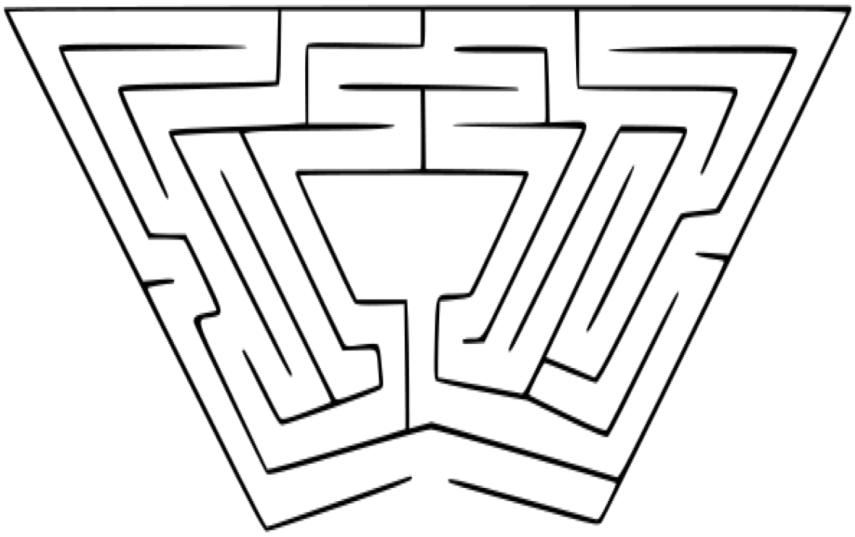
To save you time the complete matrix is contained in the file `maze.py` distributed with the assignment.

Deliverables: In the folder `maze` include the following:

`maze.py` The Python model for the maze, including definitions for the variables `root`, `goal` and `h`. I should be able to run using these variables using any of the 5 methods in a(i)-(v) above. Include a `main` that runs all 5 methods (in silent mode) and prints the result.

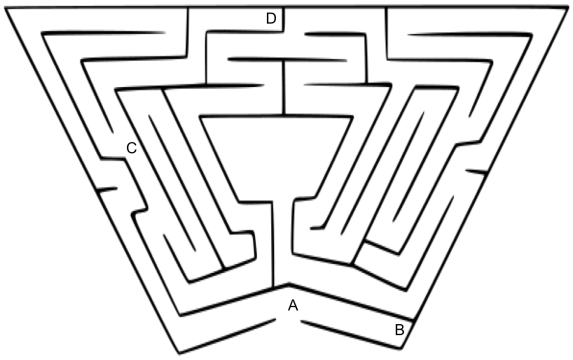
`maze.txt` Give your answer to part (b).

2. (25) The figure below shows the plan of the hedge maze at Hampton Court Palace in England, reputed to be the oldest hedge maze in the world.



The lack of a grid makes this more difficult to solve using our search techniques. Here's an approach that should work.

(a)
- Label the entrance 'A', and then label every location in the maze that is either a dead end or a choice point (i.e., the location of more than one path), with 'B', 'C', ...
- Your last label should be at the goal. I found I needed 13 labels. Here's a picture of the maze with the first few labels:



- Build a graph with a node corresponding to each label, in which labeled points connected by a passageway in the maze have their corresponding nodes in the graph connected by an arc.
- Run the model using using the same five methods as in Problem 1. Similarly discuss the results.

(b) Print the maze, and refine your model by attaching actual maze distances measured with a ruler to each of the arcs in your graph. Use the "air distance" as your heuristic for the goal. Run this with the modified versions in `search1.py` that include arc weights. Compare your result with the results of part (a).

Deliverables: In the folder `hampton` include the following:

`hamptonA.py` Solution to part (a), including values for variables `root`, `goal` and `h`. Include a main that runs the program using Algorithm A* in silent mode. Use the distributed file `hampton.py` as a template.

`hamptonB.py` Same for part (b).

`hamptonA.txt, hamptonB.txt` Use these files for your discussions of the results.

3. (50) Write a program to play a variation of the African game *Mancala* which we shall call *Kalah*.

The rules of Kalah are as follows:

- The Kalah board, above, is made up of two rows of six *pots* each.
- Four *stones* are placed in each of the 12 pots.
- Each player has a store, or *kalah* to the right side of the Kalah board.
- The game begins with one player picking up all of the stones in any one of the pots on his side.
- Moving counter-clockwise, the player deposits one of the stones in each pot until the stones run out.
- If you run into your own kalah, deposit one stone in it. If you run into your opponent's kalah, skip it.
- If the last stone you drop is in your own kalah, you get a free turn.
- If the last stone you drop is in an empty pot on your side, you capture that stone and any stones in the pot directly opposite.
- Always place all captured stones in your kalah.
- The game ends when all six spaces on one side of the Kalah board are empty.
- The player who still has stones on his side of the board when the game ends captures all of those stones.
- Count all the stones in each kalah. The winner is the player with the most stones.

How to do it. This game is readily modeled with the alpha-beta minimax procedure using the following static evaluation function:

$$h(b) = \{\#kalah\}_{MAX} - \{\#kalah\}_{MIN}$$

i.e., the difference in the sizes of the board's kalahs. Because of the complexity of the moves we'll use Java rather than Python as our language of choice. You should break the problem into 2 files:

Board.java Contains the class that defines the board and all methods required to manipulate the board according to the rules. Instances of this class represent all of the different board configurations.

Kalah.java Contains the `main` method; and static methods for the alpha-beta minimax procedure, and to provide the main loop for interaction with the human player. A sample of the latter is shown here:

```
CS364 Assignment 2.3
Sample Solution
Using Depth Bound 5

Let's play Kalah!
Who goes first (0) me; (1) you ? 0
OK, I go first
Initial Board

     4  4  4  4  4  4
  0                     0
     4  4  4  4  4  4

My turn ... hmm ...
I choose 5

     0  4  4  4  4  4
  1                     0
     5  5  5  4  4  4

Your turn ... Move (0-5)? 1

     0  4  4  4  4  4
  1                     1
     5  0  6  5  5  5

Your turn ... Move (0-5)? 0

     0  4  4  4  4  4
  1                     1
     0  1  7  6  6  6

My turn ... hmm ...
I choose 1

     0  5  5  5  0  4
  2                     1
     0  1  7  6  6  6

Your turn ... Move (0-5)?
```

Our two players are MAX and MIN, as usual. Assume the computer is MAX and the human player is MIN. MAX's side is the upper set of 6 pots, and MIN's side is the lower set of 6. MIN's pots are numbered from 0 to 5 running from left to right. MAX's pots are also numbered from 0 to 5, but from right to left (so that from MAX's point of view his/her pots are numbered from left to right).

The rule that states that when a player drops the last stone into his/her kalah, s/he gets a free turn complicates the minimax framework, since we assume that the levels of the game tree alternate players. This is solved by substituting a list of board positions at each node for the single game postion used in conventional minimax. Initially, the list contains only a single board position (and most of the time there will only be one); however, if a move results in a free turn, the board position of that free turn is added to the list of the current level and considered as part of that level, rather than passed to the next level. Consequently, the descendants passed to the next level consist of all board positions resulting from the original board position and all free turn board positions produced

along the way. Be careful, however, that in any chain of free turn boards produced at a given level, only the pot that began the chain is used in making an alpha-beta choice, since no free turn is a direct descendant of the opponent's move.

Your program should be robust enough to cope with error input, which could include typing a non-numerical character, a character outside the range 0-5, or choosing an empty pot.

Your program should use a default depth of 5 for the alpha-beta search. However, the user should be able to override this by passing a command line parameter.

The mechanics for checking move legality and performing moves, checking for a winning position and properly maintaining board state are all part of the Board class. The Board class is also responsible for printing out a display like the one shown above. The simplest way to manage the game is to create a new board each time a move is executed on a given board. (The minimax process will generate many boards during its analysis). This can be done by creating the new board as a copy of the current board and then executing the move on the copy.

The files `Board.java` and `Kalah.java` provide an outline of a possible design structure, but you are free to ignore these if you like. I've also included a running solution, `Kalah.jar`. Launch this in the shell by typing:

```
java -jar Kalah.jar [depth]
```

The optional depth parameter defaults to 5, as described above. [Note: it is a violation of the honor code to use this file other than to run it as shown.]

Deliverables: In the folder `kalah` include all `.java` and `.class` files. Your output should follow the format of the sample solution. When run, the second line of output (where *Sample Solution* appears above) should print your name(s). You should also submit a runnable jar file called `Kalah.jar`. Runnable jar files are easy to create using Eclipse or ant, or they can be built directly using the `jar` command.