B551 Assignment 2: Heuristic search and games

Spring 2015
Due: Sunday October 4, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

As we've seen in class, many AI problems can be posed as search problems, but actually solving the search problems using a naïve (uninformed) technique is computationally intractible (often NP-hard). We've seen various techniques in class for dealing with this problem. While none of these techniques can find exact solutions to all problems efficiently (if they did, the underlying problems could not be NP-hard!), they often work quite well in practice, allowing applications to trade-off between speed and optimality of the solution. This assignment will give you an opportunity to explore different heuristic search techniques that allow different trade-offs between speed and quality of solution.

As in Assignment 1, we've assigned you to a team based on your teamwork preferences. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. Both people on the team will receive the same grade on the assignment, except in very unusual circumstances. Please read the instructions below carefully; we cannot accept any submissions that do follow the instructions given here. Most importantly: please **start early,** and ask questions on the OnCourse Forum or in office hours.

Academic integrity. You and your partner may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about Python syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials (e.g. in source code comments). However, the work and code that you and your partner must be your group's own work, which the two of you personally designed and wrote. You may not share written answers or code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

You can find your assigned teammate by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should a pull-down menu. Select `cs-b551`. Then in the yellow box to the right, you should see a repository called *userid1-userid2*-p2, where the other user ID corresponds to your teammate.

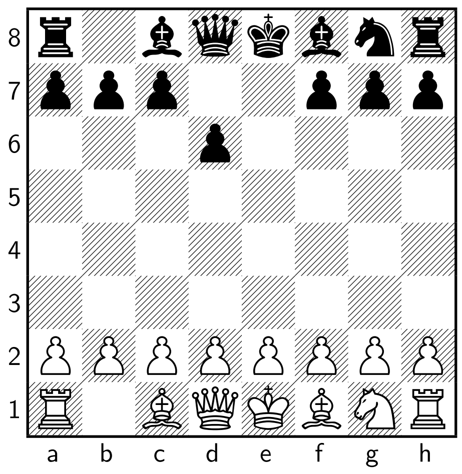
To get started, clone the github repository:

```
git clone https://github.iu.edu/cs-b551/your-repo-name-p2
```

where *your-repo-name* is the one you found on the GitHub website above. (If this command doesn't work, you probably need to set up IU GitHub ssh keys. See OnCourse wiki for help.)

Part 1: Written problem

One Sunday afternoon you tune in to ESPN to watch a chess championship. Unfortunately, the game is already in progress so you don't know the moves that have happened so far, but you do know that there have been 4 plies (i.e. White has moved 4 times and so has Black). You can also see the current state of the board:



Determine the previous moves performed by each player, with a brief explanation of your answer. Submit this problem as a PDF file, consisting either of a typed document or a scan of a neatly-written handwritten document.

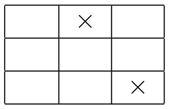
Part 2: Programming problems

The following problems require you to write programs in Python. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and basic data structures like queues. You must write all of the rest of the code yourself. If you have any questions about this policy, please ask us. We recommend using the CS Linux machines (e.g. `burrow.soic.indiana.edu`). You may use another development platform (e.g. Windows), but make sure that your code works on the CS Linux machines before submission.

For each programming problem, please include a detailed comments section at the top of your code that describes: (1) a description of how you formulated the search problem, including precisely defining the state space, the successor function, the edge weights, and any heuristics you designed; (2) a brief description of how your search algorithm works; (3) and discussion of any problems you faced, any assumptions, simplifications, and/or design decisions you made.

1. *Rameses* is a two-player game that requires a board having an $n \times n$ grid and n^2 pebbles. Initially the board starts empty and all pebbles are in a pile beside the board. Player 1 picks up a pebble and places it in any square of the grid. Player 2 then picks up a pebble from the pile, and places it in any open square (i.e. any square except the one selected by Player 1). Play continues back and forth, with each player picking up a pebble from the pile and placing it in any open square. A player *loses* the game as soon as they place a pebble that completes a row, a column, or one of the two diagonals of the board, and then the other player wins.

   Your task is to write a Python program that plays Rameses well. Your program should accept a command line argument that gives the current state of the board as a string of `x`'s and `.`'s, where `x` indicates a pebble and `.` indicates an empty square, respectively, in row-major order. For example, if $n = 3$ and the state of the board is:

   then the encoding of the state would be:

```
.x......x
```

More precisely, your program will be called with three command line parameters: (1) the value of n, (2) the state of the board, encoded as above, and (3) a time limit in seconds. Your program should then decide a recommended move given the current board state, and display the new state of the board after making that move, *within the number of seconds specified.* Displaying multiple lines of output is fine as long as the last line has the recommended board state. For example, a sample run of your program might look like:

```
[djcran@macbook]$ python rameses.py 3 .x......x 5
Thinking! Please wait...

Hmm, I'd recommend putting your pebble at row 2, column 1.
New board:
.x.x....x
```

The tournament. To make things more interesting, we will hold a competition among all submitted solutions. We will not reveal ahead of time the values of n and the time limit that we will enforce, but we plan to hold multiple tournaments with relatively low and high values of n and the time limit. While the majority of your grade will be on correctness, programming style, quality of answers given in comments, etc., a small portion may be based on how well your code performs in the tournaments, with particularly well-performing programs eligible for prizes including extra credit points.

2. A *Totogram* is a puzzle played on a board that has a graph structure, with edges connecting some pairs of vertices. The edges of the graph form a rooted tree with a particular structure: the root has exactly three children, every other non-leaf node has exactly two children, and the leaves are all at the same depth k in the tree. This graph has the property that every vertex has exactly 1 or 3 neighbors. A sample board with $k = 3$ is shown in Figure 1. (Note that a Totogram is almost a balanced binary tree, except that the root has three subtrees instead of two.) To play the Totogram, the player arranges a set of N tiles numbered from 1 to N, where N is the number of vertices in the graph (which, in turn, is a function of k), on the vertices of the board. Their score is equal to the *maximum* absolute value of the difference between any pair of adjacent vertices, and the goal is to find an arrangement that makes the score as *low* as possible. For example, the Totogram in Figure 1 shows an arrangement with score 4 (since tiles 7 and 3 are adjacent to one another).

   Your goal is to write a program that finds the best (lowest-cost) solution that it can for a given k. The program should run on the command line like:

   ```
   python totogram.py k
   ```

   The final two lines of output should be (1) the space-delimited tile arrangement, in breadth-first search order, and (2) the score for this arrangement. For example, for Figure 1, the output would be:

   ```
   4
   5 4 7 8 1 2 3 6 9 10
   ```

   In addition to describing in detail how your approach works, the comments section at the top of your code should give a brief analysis of your results, including (1) the best solution your code was able to

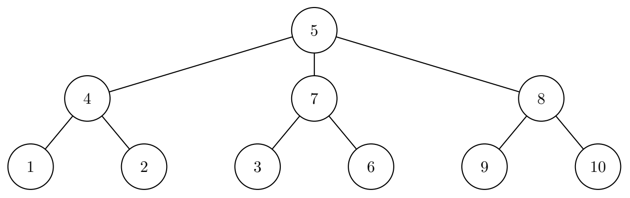


Figure 1: A sample Totogram.

find for $k = 3$, 4, 5, 6, and 7, including both the score and the actual tile arrangement, and (2) the amount of time it took to find each solution.

What to turn in

Turn in the files required above by simply putting the finished version (of the code with comments and PDF file for the first question) on GitHub (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.