COMP 560 A2: 3D Tic-Tac-Toe (Qubic)

https://github.com/tylery8/comp560-tyler_youngberg/tree/master/src/qubic

Group Members:

- Tyler Youngberg
- Nikhil Vytla

How to Run Code:

Note: This requires jdk
Open a command prompt and navigate to an appropriate directory

- > git clone https://github.com/tylery8/comp560-tyler_youngberg.git
- > cd comp560-tyler_youngberg
- > cd src
- > javac -cp . qubic/*.java
- > java -cp . qubic/Trials

Type the input into the standard input stream.

The standard output will first print out the generated utility values of the center, corner, edge, and face pieces of a generic board after each of the three specified numbers of trials (ex. 1000 2000 4000), and then print out each iteration of the 3D Tic-Tac-Toe board based on the user's move input in the format [plane line index] (ex. 1 1 1 or 2 0 3) until the game ends in either a win ("X won!"), a loss ("O won!"), or a tie ("Draw!").

In addition to allowing a human to play the AI, the program can also play itself or allow for 2-player interaction by adding the proper input as specified by the program. Currently we are working on an implementation where our program can automatically interact with and competitively play another program (in potentially another language/interface) by parsing the opponent's outputted board/turn sequence and returning a similar board/turn sequence.

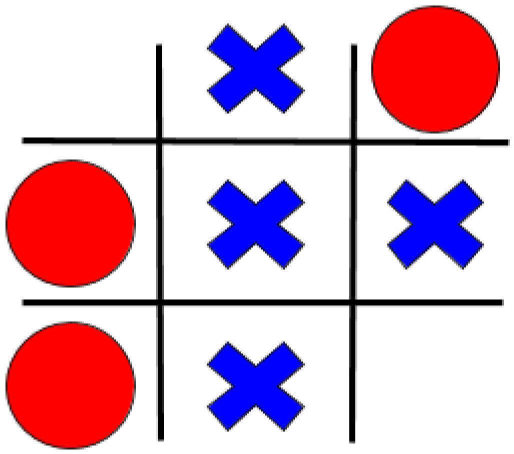
Updating the Utility for each Square:

We update the utility function in the aptly named `updateUtilityFunction` method in the `QubicImpl` class. Essentially, each trial consists of one game played to an end, where the utility function isn't updated until each game finishes.
A little bit of background. All 64 squares are split into four (4) categories: `CENTER`, `CORNER`, `EDGE`, and `FACE` (all designated in the `SquareType.java` enum). A `CENTER`

square is one of the 8 squares in the very center of the cube. `CORNER`s are one of the 8 squares on the outside tips of the cube, `EDGE`s are one of the 24 squares that run along the shortest path between two corners, and `FACE`s are one of the 24 squares directly adjacent to the middle squares. We chose this breakdown in order to better order and value certain squares over others. A small note: our implementation effectively considers `EDGE` and `FACE` squares equivalent in the scope of the utility function.

A simple explanation. Our value for each type of square is based on a *proportional update*. Imagine a regular Tic-Tac-Toe game, similar to the scenario below. Here, we make the following assumptions and conclusions:

- Total value of all X's: **+1**
- Total value of all O's: **-1**
    - Corner value: **-2/3**
        - 2 of 3 total O's are in corner positions; ⅔ of -1 is -2/3.
    - Center value : **+1/4**
        - 1 of 4 total X's are in the center position; ¼ of +1 is +1/4.
    - Edge value: **+5/12**
        - 3 of 4 total X's; +¾. 1 of 3 total O's; -⅓. +¾ - ⅓ = +5/12.
- Net sum: -2/3 + 1/4 + 5/12 = **0**

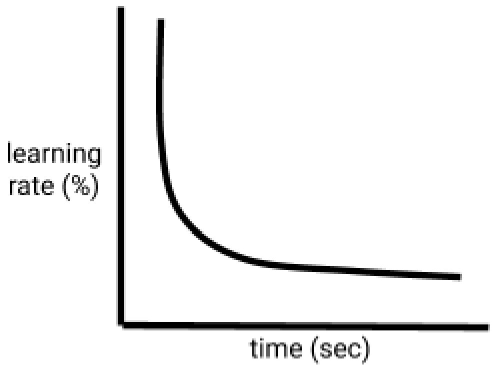


By ensuring that the net sum of all values equates to **0** for each trial, our utility function update effectively reassigns a distribution of value among each of the square types/categories.

Setting the value. Within the `updateUtilityFunction` method, the actual `setValue` function is called at the bottom, wherein the new value assigned to each square is the *weighted average* of the current value and the new value, and the *weight* is the learning rate (specified as a parameter to the method). Our current implementation decreases our learning rate in proportion with the generic inverse function over time (see method `runTrial` in `Trials.java`).

Choosing Moves During the Learning Process:

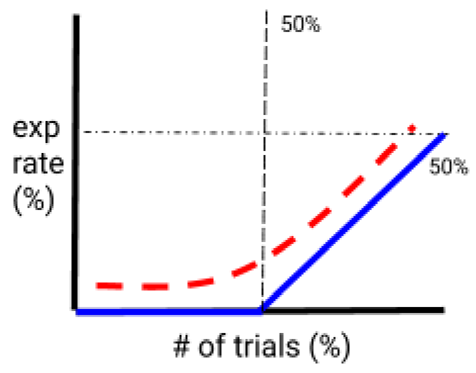
As mentioned before, the learning rate of our program decreases according to the generic



inverse function over time (1/n). In `Trials.java`'s `runTrial` method, we define the following variables:

- **`learning_rate`** `= 1/current_trial`
- **`exploitation_rate`** `= current_trial/total_trials - 0.5`

Exploitation consists of initially playing against random opponents for the **first half** of our trials and slowly increasing our exploitation rate from **0%** to **50%** beginning in the **latter half** of all trials run. For the purpose of our algorithm, we've found that this standard formula works well, and closely models more complex functions with similarly optimal results.



An important note. In the exploitation phase, each game's iteration only look at a single ply (half-move ahead), in order to save runtime and best optimize the speed of move choices further on. Moves chosen during the learning process are dependent on our evaluation function (probability distribution given to each of the 4 types of squares) as well as win patterns.

Handling Special Cases:

For handling special cases, we opted to implement **minimax** and its additional features for future prediction, including **alpha-beta pruning** (see `QubicImpl.java`'s `alphabeta` method). Our algorithm also uses **iterative deepening** combined with a **transposition table** that is used to implement better **move ordering** based on the value generated from the previous iteration. Additionally, rather than using a constant depth search, the algorithm uses **quiescence search** to search at a larger depth only when the game is in an "unstable" position. This helps to circumvent the **horizon effect** by searching deeper when the game is in an *almost-win* scenario (3 X's in a row, etc.). By combining all this, we can efficiently handle up to **4 ply in advance**.

Individual Contribution:

Tyler worked on writing and commenting the majority of the code, and worked on ways to improve the program to handle the "special cases" such as minimax with alpha-beta pruning and iterative deepening with transposition tables.

Nikhil worked on designing and implementing ways to efficiently handle “special cases” such as move ordering and quiescence search, and on writing and illustrating the majority of the report.