Othello Project

October 12, 2020

1 General Idea

You will be making a bot to play the game Othello.

This "project" is envisioned as a series of milestones, outlined in Section 5. In keeping with our professional development focus, each milestone has *suggested* deadlines and will build on the previous milestones. The term *suggested* is used, because we already have evidence that during COVID-19, some people just need more time and are finding themselves a bit overwhelmed in general. Contrasted with the first two assignments, however, progress will be checked weekly, which means there is no longer any need to worry about being late with your work. We will simply grade your work based on what progress we observe in your repository.

In the last of the milestones, there are number of possibilities for creative explorations. This means you will be able to incorporate elements into your version of Othello that may or may not have much to do with operating systems but allow you to grow in areas of personal interest. There are even opportunities to be creative that have been sprinkled throughout the milestones.

2 How Playing a Board game is similar to an Operating System

There are several pieces to an operating system: memory management, process/thread management, scheduling, input/output, and usability. When designing a bot to play a game, the bot must be able to process each possibility based on the player per turn. There could be millions of possible moves to make. A well made bot will condense that million down into a data structure and evaluate the best method applying all of the concepts that go into an operating system.

When a user uses an operating system the system must be able to adapt to anything that the user throws at it. For example, security within the operating system something widely studied because permission systems, even today, have many flaws. Like a operating system, code for a game closely mimics the jobs that an operating system must preform. Games and operating systems must also run continually until the user of the said system/game stops. Games must manage their components effectively such that the game is fast and playable. The same ideas apply to an operating system.

3 Designing Your Code

The ultimate goal of your code is to do all of the following,

- Develop a board abstract data type and use modular thinking for all aspects of game play.
- Learn to work with recursion and management of game state.
- Ensure game logic is 100% reliable by first getting human-human play working and well-tested.
- Manage the game such that the user and bot take turns based on a timer. Timers will require you to learn and use signal handling.

- The bot must process it's turn within the timer the same way the player does. (This means you and the computer both can lose terms and control should pass back and forth. Get it working with human-human first so you know the timer logic works.)
- The bot must make intelligent decisions based on the move the player makes.
- The bot must manage multiple different processes/threads to improve performance of its decision making process.
- Your code must manage all of the memory for the bot and the player

4 Grading Progress Reports

This series of homework assignments will last the rest of the semester. You may work at your own pace to incorporate as many elements as possible, and it is entirely possible you won't finish. However, the goal is to make progress, and if you are making progress each week, you can get a passing grade or better. We'll even go so far as to say, if you only finished 2-3 of the milestones (at least through the lookahead tree and human-computer basic play) you would be in a position to get a good grade based on your weekly progress reports. What we are seeking is actual effort that shows progress with the C language, operating systems concepts, and engagement with the material in general.

Weekly progress reports will be done each week. Each report will be worth 10 points. The report should be named `report-<DATE>.md` file placed in a subdirectory of your repository named `reports`.

When Grading, we will look for all of the criteria below,

- 2 point for *evidence of meaningful commit activity*. People who do all the work in one day or wait until the last minute are not showing respect of software process.
- 2 point for a **improvement report**. Specifically, what did you accomplish since the last report? You must be truthful about this. It is actually possible to get at least 1 point here if we perceive you to be honest.
- 2 point for a **goals list**. Specifically, what do you plan to accomplish in the following week? Careful, we will look at your previous week report when grading the current week.
- 2 points for *code inspection*. This includes whether you are making an attempt to write good C code, use modular structure, do unit tests, and put effort into making your code look nice. Anyone with sloppy code should expect a zero in this category with minimal justification.
- 2 point for a 5 minute demo of your current working solution. Most should expect to get 0 or 1 the first couple of weeks. You are strongly encouraged to learn to use OBS Studio or similar software, which makes it easy to make nice videos of your work. You aren ot required to appear in the video but must speak or use captions.

We will allocate points as follows:

- needs improvement (0)
- satisfactory (1)
- well-developed (2)

Earning well-developed in any category will be a challenge. Most should expect well-developed in 1-2 categories at most. The key to succeeding in the project is going to be solid effort *every week*. Effort is not just about lines of code. It also is how we *perceive* your effort. For example, if you make a video of your work that seems uninspired, please expect a 0 (for needs improvement). If your work in any category is exceptional, expect a 2 (well-developed).

These progress reports will make up a majority of your grade. Completing each one per week is very important. Although there might be 1-2 more tests, you should not assume that tests will determine or affect the final outcome. The syllabus makes it abundantly clear that everything is worth the same.

Some people will be excited about this: If you work hard, you really could get a good grade, but it will depend upon you. There are no grade quotas, and there can be any number of any letter grade as a final result. Do your best work, and your grade will probably reflect that.

5 Milestones

You should therefore be thinking about this assignment as a *series* of assignments. You actually could work on the entire project at once, but it will be necessary to demonstrate that all of the expectations for each *milestone* are completed to receive full credit for any given milestone.

The following is a suggested schedule for what to complete by when:

5.1 Milestone 1: Develop Board Abstract Data Type, by 10/19/2020

Earlier assignments have shown that students need practice with abstract data types. Coming into 310/410, most seem to have experience *using* an ADT but not in the actual *creation* of one. For this phase, you will develop a data type to represent the board `othello_board_t`. You can derive some inspiration from `point_t` in our `systems-code-examples` repository.

What should your board ADT be able to do? Following the `point_t` example:

- Is the board full? If it is, the game is over
- Can a piece with *color* move into a given position? (If so, this means it can also flip pieces in horizontal, vertical, or diagonal positions.)
- Which direction can pieces be flipped without actually doing so?
- initialize the board to a starting configuration (i.e. the 4 pieces at center of the board)
- copy the board into a new board. This will be helpful when we start working with threads, where each configuration should be evaluated separately without destroying the existing board.
- compare two boards to see what changed?
- etc.

Unlike previous assignments, we're looking for some creativity. The key idea, however, is to make sure there is a proper board representation. It is entirely possible we will update the list of available functions in the board ADT. Game design tends to be iterative, and sometimes additional methods will be needed to help build a better interface.

5.2 Milestone 2: Two-person play with a text-based interface, by 10/23/2020

This is not a course about UI/UX design. But we're going to need a basic interface that supports gameplay. Based on the board ADT created in Milestone 1, you will basically be creating an interface that allows *human* players to take turns playing. This is also going to be a great way to make sure the logic of your board game is completely working before introducing the human-computer (or even computer-computer) aspects.

As a creative option, you may want to consider learning and using the `ncurses` programming library. See https://www.linuxjournal.com/content/getting-started-ncurses and https://www.linuxjournal.com/article/1124 for articles on how to get started with it. This C-based library is a great way of learning to work with terminal graphics.

5.3 Milestone 3: Develop Search Tree / Human-Computer Play, by 11/2/2020

To play Othello (or just about any game), you need to develop a tree representation of the search space for the game. Of course, with the latest advances in AI and ML, there are ways to look at game transcripts to see how players play the game and "predict" the next good move to make, based on a given board configuration. Nevertheless, learning about this classic way of computing the game tree ties directly into the history of AI, and it will help you to understand many things about machine learning methods in general.

In this milestone, you will start by trying to compute the lookahead tree. It's interesting to ponder: How big would this tree get? Every game is different. In Othello's case, pieces are basically being *added* to the board. There are 64 squares on the game board, 4 of which are occupied initially. So there are only 2^60 possible combinations to fill the entire board. An interesting question (answered by reading the Othello docs) is whether a game can end before the board is entirely filled?

So in this phase, your job is to build a tree of moves, where the next board configuration(s) are computed by considering *what your opponent might do*. Of course, you are also considering yourself as an opponent, since you have to simulate your moves as part of the tree lookahead computation.

What are the challenges in creating a lookahead tree? First, it's recursive, so you'll have to brush up on that. Second, It's not going to be long before you're using a substantial amount of memory. Today's computers, of course, have plenty of memory. Third, however, as game play progresses, there are certain configurations that are no longer needed. This suggests a possible strategy: Keep only as much of the tree as you need. Of course, it also suggests that we could use threads to compute only so much lookahead but also to discard configurations that are no longer needed.

When you write your code, you need to think about what problem you are trying to solve. In the end, you have a given board position and want to know what *next move* gets yourself to a *winning* outcome. Because the computer can look far ahead, it would be able to see all the way to the end of the board. To make it more possible for the human to do well, we can make this a tunable parameter, *difficulty*, which only allows so many levels of lookahead by the computer. This also makes the code a little more complicated, since you may have to design your code to *cache* or *recompute* new levels of the tree, based on the difficult level.

Speaking of human-computer play, Othello also allows for the game to be played with a time limit. If the human takes too long to make its move, it should be possible to "cancel" the players turn and make the next move. (This can also create some complications for the lookahead tree, since you now need to assume that the current board representation is the next configuration. A clever trick is just to "flip" the interpretation of which color goes next.

5.4 Milestone 4: Use threads and a thread pool to improve performance of the lookahead scheme, by 11/16/2020

Details will be forthcoming on this, but the idea is that you can use threads (a thread pool) and a queue to keep track of configurations to be evaluated. When a board configuration is added to the queue, we will compute additional lookahead to see what is the best move to make based on that board configuration. By using a thread pool, we will only use the number of threads as as available cores on the system.

5.5 Milestone 5: Your Creativity, by end of semester

Once you have all Milestones 1-4 completed, you are essentially done. So what would you do next? Here are some possibilities:

- Learn a framework such as GTK or Qt to create a *graphical* desktop interface. These are C/C++ libraries that will play nicely with your C code.
- Build a socket-based version, where the server runs on one computer and the client on your desktop. This would allow (in theory, at least) the server to run on a more powerful computer on your network or the internet that has more cores or RAM than your own computer does.

- Learn and use a HTTP and JSON libraries to develop a web service for your game. This would allow you to develop a non-C client for the system, perhaps in nodejs/ECMAScript. Or in a favorite programming language (since you had to suffer so much by writing C all semester) such as Python or Java.

- Use embedded databases (sqlite3) to storage all of the possible game configurations so a board can be looked up in the database and tell what is the best move for either player to make. This method is known as precomputation and means that you would swap out the lookahead tree for a database-based method of game play.

We need to add a list of features/capabilities and dates. What we'd like to see each week. I really like the idea of this project being for the rest of the semester. No more write-ups after this!! Yay!!

6 Grading Final Product

For the final project criteria we are looking for the below objectives.

[This will be due in Finals week. There will not be a final test or exam in this class. However, there will probably be another quiz two weeks before the end of semester.]

I'm almost thinking we can have a checklist:

- are abstract data types used properly to represent key abstractions and concepts of the game? You should be following the conventions that I use in many examples in my systems examples.

- is there good modular structure. More .c and .h means yes. Fewer means no.

- Is the code parameteric? Can I tune its behavior based on command line parameters (e.g. `--timeout seconds` or `--difficulty 4`, etc.

- Is multiprocesing used to assist in lookahead. Everyone's computer has at least 2-4 cores. So writing your code such that lookahead can be handled with processes (and eventually threads) is a plus.

- Once the principles of basic processes and threads have been learned, can the performance be improved by making use of fixed process or thread pools? Or does it matter? A really good solution will do timings to see how long it takes to compute the next move and show improvement over time.

- Are unit tests employed to ensure correctness. Games are complex. Othello has several things to do on boards that require proper testing.

- Does the solution make use of data structures concepts? This one is going to require array and some form of a list. It is entirely possible hashtables and trees could greatly simplify the implemetnation, too. We're going to be looking for some imagination and creativity, since everyone *knows* data structures coming into this course. (Ok, just kidding, but we gave you some examples!)

- Does the solution do anything to distinguish itself. For example, you could build a web service to "expose" the C code, which would allow you to build a client in, say, JavaScript, that interfaces to the service. Another possibility is to build a nice "terminal" interface using curses/ncurses. Another possibility is to develop a native desktop interface using something like LibQt (which would require some C++) or Python (via SWIG).

Grading

- 4 points project is able to play a full game of Othello.

- 3 points the bot is able to play a full game.

- 1 point proper github usage throughout the project.

- 1 point for well made code and comments.
- 1 point for tests for each method

7 Extra Credit

- 1 point for adjustable difficulty for the bot.
- 1 point for a analysis of your code (a write up of the data structures and layers)
- 1 point for continuous integration on github with travis.