APS 105 — Computer Fundamentals
Lab #8: Reversi Game-Playing Program
Winter 2022

The goal of this lab is to build upon your work in Lab #7 to create a program that actually plays the Reversi game against a human opponent. There are two parts to this lab. In the first part, the exact algorithm to use for the computer moves is completely specified. In the second part, you are free to make your own algorithm.

You must use examify.ca to electronically submit your program by 11:59 pm on **Saturday, April 2, 2022**.

Objective: Complete the Implementation for a Reversi Game

In Lab #7, you developed a program that represents the board of a Reversi game, and that computes, for a given board state, the possible moves for the Black and White player. Your solution also accepts a move as input and flips the tiles accordingly. In this lab, you will build upon your work to create a Reversi game where a human can play against the computer, with the computer making intelligent decisions about where to place its tiles.

Recap: The Rules of Reversi

The rules of the game were presented in the Lab #7 handout, and are briefly repeated here. Reversi is played on a board that has dimensions $n \times n$, where n is even and ranges from 4 to 26. In the picture below $n = 4$. The game uses tiles that are white on one side, and black on the other side (they can be "flipped" over to change their colour). One player plays white; the other player plays black. The picture below shows the initial board configuration, which has two white and two black tiles pre-placed in the centre. Observe that rows and columns are labelled with letters.

A "turn" consists of a player laying a tile of his/her own colour on a candidate empty board position, subject to the following two rules:

1. There must be a continuous straight line of tile(s) of the opponent's colour in at least one of the eight directions from the candidate empty position (North, South, East, West, and diagonals).

2. In the position immediately following the continuous straight line mentioned in #1 above, a tile of the player's colour must already be placed.

After playing a tile at a position that meets the criteria above, all of the lines of the opponent's tiles that meet the criteria above are flipped to the player's colour.

The turns alternate between the players, unless one player has no available move, in which case the only player with an available move is allowed to continue to make moves until a move becomes available for the opponent. At this point, the opponent is allowed to take a turn and the alternating turns between the players resume. The game ends when either: (1) the entire board is full, or (2) neither player has an available move. The winner of the game is the one with more pieces on the board.

The Game Flow and Prescribed Algorithm

Your program must first ask the user for the board dimensions. Then, the program asks the user if the computer is to be the Black or White player. For this lab, we will assume that the Black player *always* gets the first move. So, if the computer is black, then the computer should make the first move; otherwise, the program prompts the human player to make the first move. The board is printed after every move.

Once the first move is out of the way, the turns proceed as described above, alternating between Black and White unless one of the players has no move to make, which case your program should print a message "W player has no valid move." (i.e. for the case of the White player) and should prompt the Black player for another move. After each turn, your program must print the board, and must detect whether the game has been won, or whether there is a draw. If your program detects the game is over (i.e. a win or a draw), a message is printed and the program terminates. The specific messages to print are: "W player wins.", "B player wins." or "Draw!". If the human player makes an illegal move, your program must detect this, print an error message, and end the game, declaring the winner (with the corresponding message above).

Part 1: Implementing the Prescribed Algorithm

How Should the Computer Make Moves?

The method for the computer to make a move is as follows: for each possible place it could make a move, it should compute a number, called a "*score*" that is larger when a square is a better place for the computer to lay a tile. As these scores are computed for each square, it should keep track of which one is the highest, and use that square as the move. (See below on what to do in the event of a tied score).

The score for each candidate position is defined as the total number of the human player's tiles that would be flipped if the computer were to lay a tile at that position. The figure below shows the scores if the computer were the White player.

A few important notes:

1. It is likely that two or more squares may end up having the same score. In that case your program **must** choose the position with the lower row. If there are two positions with the same score in the same row, your program **must** choose the solution with the lower column. You can ensure this happens easily by the order in which you compute the score and test to see if it is the best.

2. Your program must be able to work with the computer being either the Black or the White player.

Here is a sample execution of the program (your program must conform to this output). Notice that, close to the end, the computer (playing White) has no valid move, and the turn returns to the human player (playing Black).

```
Enter the board dimension: 4
Computer plays (B/W): W
  abcd
a UUUU
b UWBU
c UBWU
d UUUU
Enter move for colour B (RowCol): ba
  abcd
a UUUU
b BBBU
c UBWU
d UUUU
Computer places W at aa.
  abcd
a WUUU
b BWBU
c UBWU
d UUUU
Enter move for colour B (RowCol): ab
  abcd
a WBUU
b BBBU
c UBWU
d UUUU
Computer places W at ac.
  abcd
a WWWU
b BBWU
c UBWU
d UUUU
Enter move for colour B (RowCol): bd
  abcd
a WWWU
b BBBB
c UBWU
```

```
d UUUU
Computer places W at ca.
  abcd
a WWWU
b WWBB
c WWWU
d UUUU
Enter move for colour B (RowCol): da
  abcd
a WWWU
b WWBB
c WBWU
d BUUU
Computer places W at cd.
  abcd
a WWWU
b WWWB
c WBWW
d BUUU
Enter move for colour B (RowCol): dd
  abcd
a WWWU
b WWWB
c WBWB
d BUUB
Computer places W at db.
  abcd
a WWWU
b WWWB
c WWWB
d BWUB
Enter move for colour B (RowCol): dc
  abcd
a WWWU
b WWWB
c WWWB
d BBBB
W player has no valid move.
Enter move for colour B (RowCol): ad
  abcd
a WWWB
b WWBB
c WBWB
d BBBB
B player wins.
```

Here is another sample execution where the human player makes an illegal move and the computer wins.

```
Enter the board dimension: 6
Computer plays (B/W): B
```

```
  abcdef
a UUUUUU
b UUUUUU
c UUWBUU
d UUBWUU
e UUUUUU
f UUUUUU
Computer places B at bc.
  abcdef
a UUUUUU
b UUBUUU
c UUBBUU
d UUBWUU
e UUUUUU
f UUUUUU
Enter move for colour W (RowCol): fa
Invalid move.
B player wins.
```

Your solution to Part 1 should be submitted as your answer to **Question 1** on examify.ca.

Part 2: Implementing Your Own Algorithm

In this part of the lab, your objective is to apply your wits, creativity and resourcefulness to produce the best computer player possible, using *any* approach you are aware or can think of. For example, you may wish to consider using an approach that looks ahead some number of moves. There is, however, a time limit on how much computer time will be allowed per move.

Here are a few requirements for your program in Part 2:

1. In order for your program to be tested using the test cases on examify.ca, your own algorithm should start from calling the function makeMove(), the prototype of which is shown below. You can, of course, call other functions that you prefer to define within the makeMove() function.

   ```
   int makeMove(const char board[][26], int n, char turn, int *row, int *col);
   ```

   This function takes five parameters — the game board, the dimension of the board, the colour of the next move (who should take the turn, Black or White?), and two pointers to two integer values — and computes the best move for that colour. You should not update the board within this function — the parameter has a const keyword indicating that you should not modify anything stored in the board array. The location of the best move should be returned to the calling program using the two pointer variables, row and col, to two integer values. The function returns an int type value as well, and it is up to you on what it would return. If you have no useful information that needs to be returned, you may simply return 0.

2. For an 8×8 board, your program must make a move (lay a tile) within **one second** on remote.ecf.utoronto.ca. If your program does not make a move within one second, it will be considered an invalid move. Please refer to Appendix C in this handout for some additional information on how to measure the execution time of your moves.

3. If your program makes an invalid move, it will immediately lose the game (as in Part 1 of this handout).

Note that the input and output formats of this part are identical to Part 1. The only difference is that your program may use any method to choose the moves made by the computer.

Your solution to Part 2 should be submitted as your answer to **Question 2** on examify.ca. However, you only need to submit your code of the makeMove() function and all the functions that are called within the makeMove() function. On examify.ca, we will prepend and append some code in order to mark your program based on our marking criteria, as outlined in the next section.

Marking

You must submit both of your program files through examify.ca for marking. This lab will be marked out of a total of **10 marks**, along with **4 bonus marks**. 6 marks for Part 1 and 4 marks + 4 bonus marks for Part 2.

1. Your marks for Part 1 will be assigned by examify.ca using a collection of public and private test cases just like the previous labs, to ensure your program's output exactly matches the prescribed output.

2. Part 2 will be marked by playing your game-playing program against two game-playing Reversi programs, developed by the APS105 course staff: a first which is just slightly "smarter" than the Part 1 solution, and a second that is a little bit smarter than the first. Both programs will be identical to the library we provided to you for testing your work. Your program will be played against the two staff-developed programs twice in an 8×8 board on examify.ca: in one instance your program will play Black, and in the second instance your program will play White. If your program is able to win *one* of the instances against our "smarter" program, you will receive two marks. If your program is able to win *one* of the instances against our "smartest" program, you will receive two additional marks. You will get a total of 8 points possible in this part, 4 of which are bonus marks in this lab.

Stay Safe and Good Luck!

Appendix A: A Tutorial for Testing Your Work in Part 2

If you have implemented a computer AI strategy to play the Reversi game in Part 2, and would like to test the strength of your AI, we provide you with both the "smarter" AI and the "even smarter" AI strategies in a static library file called liblab8part2.a. This tutorial helps you to use this library file to test the strength of your AI before submitting your solution.

Your solution in Lab 8 should have two lines of code that uses printf() and scanf() to prompt a human player to enter his/her move. To set up your program for testing using liblab8part2.a, replace these two lines of code with something similar to the following:

```
findSmarterMove(board, n, colour, &row, &col);
printf("Testing AI move (row, col): %c%c\n", row + 'a', col + 'a');
```

findSmarterMove will use the "smarter" AI strategy to find the best move. If you wish to use the "even smarter" AI, call the findSmartestMove function with the same parameters.
You will also need to add:

```
#include "lab8part2lib.h"
```

to the top of your program.

To compile your program successfully with our provided static library, you will need to first copy the following two files:

```
lab8part2lib.h
liblab8part2.a
```

to the directory that contains your Lab 8 Part 2 solution in your Visual Studio Code project. Both files can be found with the other files (such as this handout) in Lab 8 in the course website. **Note**: Your will see several folders in the provided library, choose just one library that matches your operating system and computer architecture.

Your file structure will be similar to the following:

```
/
├── .vscode
│   ├── c_cpp_properties.json
│   ├── launch.json
│   └── settings.json
├── liblab8part2.a
├── liblab8part2.h
├── reversi.c
└── reversi.h
```

In order to build your project successfully, you need to modify some compiler and linker settings in the setting.json file under the .vscode folder.

```
{
  "C_Cpp_Runner.cCompilerPath": "...",
  ...
  "C_Cpp_Runner.compilerArgs": [],
  "C_Cpp_Runner.linkerArgs": [],
  ...
}
```

Update the field in "C_Cpp_Runner.compilerArgs" and "C_Cpp_Runner.linkerArgs" to reflect the absolute path to the liblab8part2.a library.

Here is an example for Windows:

```
{
  ...
  "C_Cpp_Runner.compilerArgs": [
    "\"${workspaceFolder}\\liblab8part2.a\""
  ],
  "C_Cpp_Runner.linkerArgs": [
    "\"${workspaceFolder}\\liblab8part2.a\""
  ],
  ...
}
```

Here is an example for macOS and Linux:

```
{
  ...
  "C_Cpp_Runner.compilerArgs": [
    "\"${workspaceFolder}/liblab8part2.a\""
  ],
```

```
  "C_Cpp_Runner.linkerArgs": [
    "\"${workspaceFolder}/liblab8part2.a\""
  ],
  ...
}
```

Alternatively, you can compile and run your program directly in the terminal (for example, PowerShell in Windows or Terminal in macOS and Linux) without using Visual Studio Code's built-in C/C++ Runner extension. Change to the working directory to where the project is stored at, and run one of the following command depending on your operating system —

For Windows:

```
gcc -L. .\reversi.c -llab8part2 -o reversi.exe
```

For macOS and Linux:

```
gcc -L. ./reversi.c -llab8part2 -o reversi
```

You can then run the executable that you have just built —

For Windows:

```
.\reversi.exe
```

For macOS and Linux:

```
./reversi
```

You are now ready to test your solution to your heart's content.

Appendix B: Participating in the Online Interactive Leaderboard

As an **optional** added challenge, the APS 105 teaching staff developed an online interactive leaderboard website located at `http://aps105.ece.utoronto.ca:8090/leaderboard`.
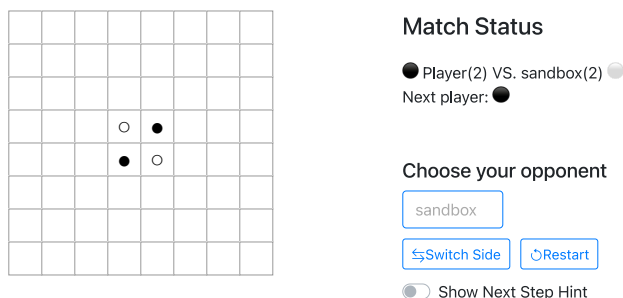

Figure 1: The APS105 Reversi Interactive Leaderboard Website

After you submit your program for Part 2 on `examify.ca`, your algorithm will be considered in the online interactive leaderboard automatically if it can pass all the sanity checks from test case 5 to test case 10. All your moves should be completed within one second on `remote.ecf.utoronto.ca`.

Appendix C: Measuring the Passage of Time in your Program

If you create an algorithm that searches through a sufficiently large number of choices for moves, it may use more than the allowed one second time limit. There are at least two ways to deal with this:

- Write an algorithm that you are certain will never use more than a second on an 8×8 board, by testing it and measuring it on the slowest test cases.

- Have your program check the elapsed time while it is running, and stop searching for the next move, when it gets close to the time limit. Here it should use the best choice found to that point. To do this, you will need to make use of the following code, which shows how to measure time:

At the top of your C program, add these two lines:

```
#include <sys/time.h>
#include <sys/resource.h>
```

Then, to time part of your program, use the code as below. It consists of two calls to functions that returns the current time in seconds and microseconds. The code below produces a value, in the variable `total_time` in seconds.

```
struct rusage usage; // a structure to hold "resource usage" (including time)
struct timeval start, end; // will hold the start and end times

getrusage(RUSAGE_SELF, &usage);
start = usage.ru_utime;
double timeStart = start.tv_sec +
                   start.tv_usec / 1000000.0; // in seconds

// PLACE THE CODE YOU WISH TO TIME HERE

getrusage(RUSAGE_SELF, &usage);
end = usage.ru_utime;
double timeEnd = end.tv_sec +
                 end.tv_usec / 1000000.0; // in seconds

double totalTime = timeEnd - timeStart;

// totalTime now holds the time (in seconds) it takes to run your code
```