APS 105 — Computer Fundamentals
Project: Reversi Game (Part 2)
Playing Program
Winter 2021

The goal of this lab is to build upon your work in part 1 of the project 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.

This lab will be due in the week of March 29th.

Objective: Complete the Implementation for a Reversi Game

In part 1 of this project, 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 part 1 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.
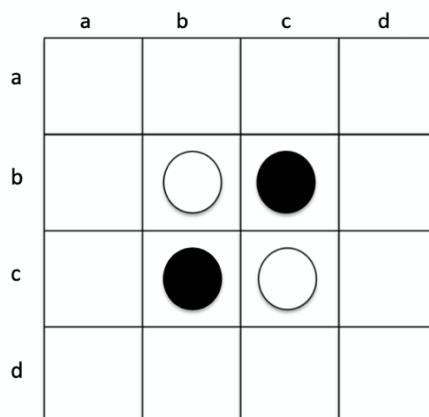
A "turn" consists of a player laying a tile of his/her own colour on a candidate empty board position, subject to the following two rules:

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 above criteria, 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 resumes. 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.

3. **Continue to implement your program from the first part of the project. Still, you need to use the given header file.**

4. **You need to implement function makeMove() which gives you the next step.**

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.
```

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.

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

Other requirements:

1. Your program must conform to the identical output format as in Part 1.

2. For a 26×26 board, and indeed all sizes of the board, your program must make a move (lay a tile) within reasonable time **(one second).** Otherwise, your program for part 2 will not be marked. See below for 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 above).

4. **Continue to implement your program from the first part of the lab. Still, you need to use the given header file. <u>You should modify makeMove() to implement your own AI that decides the next move.</u> Note: In the Online Interactive Mode, the backend will only call your makeMove() function, so please make sure that all the AI logics are completed within the makeMove function.**

5. **All the requirement from lab7 still holds. That mean you can't use global variables and etc.**

Testing Your Work in Part 2

In Part 2 of this lab, your solution will be marked by playing your game-playing program against two game-playing Reversi programs, developed by the APS105S 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. Your program will be played against the two staff-developed programs twice: 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 in "aps105_smarter" instance, you will receive two marks. If your program is able to win in "aps105_smartest" instance, you will receive two BONUS marks.

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 code 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 code 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 CodeLite project. Both files can be found with the other files (such as the handout) in Lab 8 in the course website. To link the provided library called `liblab8part2.a` in your CodeLite project, right click on your project within the workspace, select `Settings` (at the bottom) and then select `Linker` from the left hand side in the window that pops up. Then, in `Linker Options`, enter `-llab8part2` (that is "dash" "lower-case L" "lab8part2").
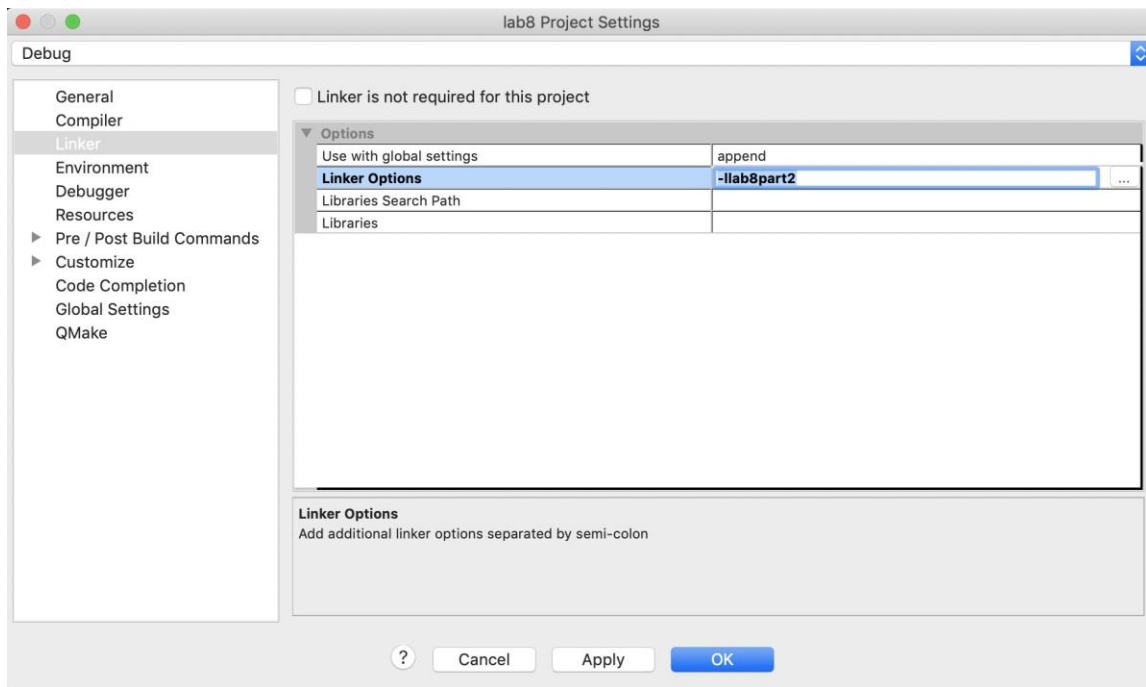
Click `Apply` and then `OK` to close this window.

You are now ready to compile your solution with the provided static library, and test your solution to your heart's content.

Note 1: In the provided archive file `liblab8part2.zip`, we have provided pre-compiled header and library files for three different operating systems: Linux (64-bit), MacOS (Intel base), and Windows (64-bit). Use Linux (64-bit) version for the ECF computers.

Note 2: If you prefer to compile your code using command-line tools, use the following command line:
```
gcc lab8part2.c -L. -llab8part2 -std=c11
```
where `lab8part2.c` is assumed to be the name of the file containing your Part 2 solution code. Before running this command to compile your solution, you should make sure that both `lab8part2lib.h` and `liblab8part2.a` have been copied to the current working directory. These files can be found with the other files in Lab 8 in the course website.

Note 3: This is only for testing! When you submit your final solution, you should not include aps105 library.

Leaderboard and New this year!

You can compete against aps105_smarter and aps105_smartest online through the platform: http://142.150.239.187:8090/. There was a leaderboard for this project from a few years ago and this year we will continue having a leaderboard for it! Once your AI compete against aps105_smarter, your AI will enter the pool of all smart AIs. You can check your ranking on this page: http://142.150.239.187:8090/leaderboard. We will keep the top students on the leaderboard. Later this week or next week, we will also send a private token to you through Quercus so that you can use it to check your own ranking. Cooler than that, once there are students submit their code for part 2 and enter the smart AI pool, you can start competing with them as well!

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 time. 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 a 26 x 26 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
```

Grading by TA and Submitting Your Program for Auto-Marking

There are a total of **10 marks + 2 bonus marks** available in this lab, marked in two different ways:

- **By your TA, for 4 marks out of 10.** Once you are ready, show your program to your TA so that it can be marked for style (1 mark), and to ask you a few questions to test your understanding of what is happening (3 marks). Programs with good style have been described in previous labs, so that will not be repeated here. Be ready to explain your approach in Part 2 to your TA.

- **By an auto-marking program for 6 marks (+ 2 extra bonus marks) out of 10**. You must submit both of your program files through the ECF computers for marking. We will use a software pro- gram to compile and run your programs, and test them with different inputs. Part 1 is worth 4 marks; Part 2 is worth 2 marks + 2 bonus marks. The two parts will be marked differently from one another.

8

1. Part 1 will be marked using the same procedure applied in prior 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 one instance your program will play Black and in the second instance your program will play White. If your program is able to win in either instance, you will receive two marks.

- Long before you submit your program for marking, you should run the exercise program that compiles and runs your program (Part 1) and gives it sample inputs, and checks that the outputs are correct. Note that for this lab, the exercise program will only test Part 1. Ensure your Part 2 conforms to the identical output style as Part 1. Similar to the previous labs, you should submit your code through Examify. If there is anything wrong, Examify will report this to you, so read its output carefully, and fix the errors that it reports.

- Once you have determined that your programs are as correct as you can make them, then you must submit your program for auto-marking. This must be done by the end of your lab period as that is the due time. To do so, you must submit your work through Examify. Note that you may submit your work as many times as you want prior to the deadline; only the most recent submission is marked.

  For Part 1, Examify will be looking for the exact letters as described in the output in this handout, including the capitalization. When you test your program using the exercise program, you will see that it is expecting the output to be exactly this, so you will have to use it to see if you have this output correct.

  Important Note: You must submit your lab by the end of your assigned lab period. Late submissions will not be accepted, and you will receive a grade of zero.