ICS 46 Spring 2018
Project #3: *Black and White*

Due date and time: *Friday, May 18, 11:59pm*

Introduction

Many of you have played video games, and while it can be more compelling to play them against other people, we've all become quite accustomed to the idea that you can often play them without anyone else being involved, with the game making its own decisions about how to play against you. This project explores one way to write programs that can make these kinds of decisions, that provide a kind of *artificial intelligence* (AI). In our case, we'll focus our efforts on techniques based around *search trees*, which will also give you more practice with the kinds of algorithms that can be used to traverse tree-based structures and with implementing a more complex recursive algorithm than the ones you built in the previous project, which will be a necessary skill for our later work.

While you work, you will have an opportunity to play your AI against the AIs of other students, or to play against them yourself. A server will be running throughout the project cycle, which will act as a central location to connect to one another and launch games. Simply log into that server with the provided GUI and you'll be able to engage in network-based battle.

Finally, after everyone has submitted this project, I'll run a tournament in which your AI will compete against those written by others. While the tournament has no course credit associated with it — you don't get a higher grade for finishing the tournament rated higher than others, though we do pay attention to who finishes it without being disqualified — you are competing for the notoriety of putting together a better algorithm than anyone else's. May the best algorithm win!

Getting started

Before you begin work on this project, there are a couple of chores you'll need to complete on your ICS 46 VM to get it set up to proceed.

Refreshing your ICS 46 VM environment

Even if you previously downloaded your ICS 46 VM, you will probably need to refresh its environment before proceeding with this project. Log into your VM and issue the command **ics46 version** to see what version of the ICS 46 environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version with a timestamp older than the one listed below, you'll want to refresh your environment by running the command **ics46 refresh** to download the latest one before you proceed with this project.

```
2018-04-29 17:55:50
Project #3 template added
```

If you're unable to get outgoing network access to work on the ICS 46 VM — something that afflicts a handful of students each quarter — then the **ics46 refresh** command won't work, but an alternative approach is to download the latest environment from the link below, then to upload the file on to your ICS 46 VM using SCP. (See the Project #0 write-up for more details on using SCP.) Once the file is on your VM, you can run the command **ics46 refresh_local NAME_OF_ENVIRONMENT_FILE**, replacing **NAME_OF_ENVIRONMENT_FILE** with the name of the file you uploaded; note that you'd need to be in the same directory where the file is when you run the command.

- ics46-2018spring-environment.tar.gz

Creating your project directory on your ICS 46 VM

A project template has been created specifically for this project, containing a similar structure to the **basic** template you saw in Project #0, but including a fair amount of code (both source code and compiled libraries) that is being provided as a starting point. So you'll absolutely need to use the **project3** template for this project, as opposed to the **basic** one.

Decide on a name for your project directory, then issue the command **ics46 start *YOUR_CHOSEN_PROJECT_NAME* project3** to create your new project directory using the **project3** template. (For example, if you wanted to call your project directory **p3**, you would issue the command **ics46 start p3 project3** to create it.) Now you're ready to proceed!

The project directory

Change into your project directory and take a look around. Having already completed Project #1, what you will see will look very familiar. Once again, your project directory is capable of building (any or all of) three separate programs that you can run by issuing the commands **./run app**, **./run exp**, or **./run gtest**. As before, a **lib** directory contains precompiled libraries that make up the part of the project that you won't be implementing yourself, an **include** directory contains declarations of things (only some of which you'll need to use directly), and the usual **app**, **core**, **exp**, and **gtest** directories for writing your code.

The game of Othello

This project asks you to implement an artificial intelligence for a game called Othello. Othello (also known as Reversi) is a well-known two-player strategy game. The game is played on a square board divided into a grid — usually 8x8, though the size of the grid can vary. Players alternately place *tiles* on the game board; one player's tiles are black and the other player's are white. When tiles are placed on the game board, other tiles already on the board are *flipped* (i.e., a black tile becomes a white tile or vice versa). The game concludes when every cell in the grid contains a tile, or when neither player is able to make a legal move; the winning player is the one who has more tiles on the board at the end of the game.

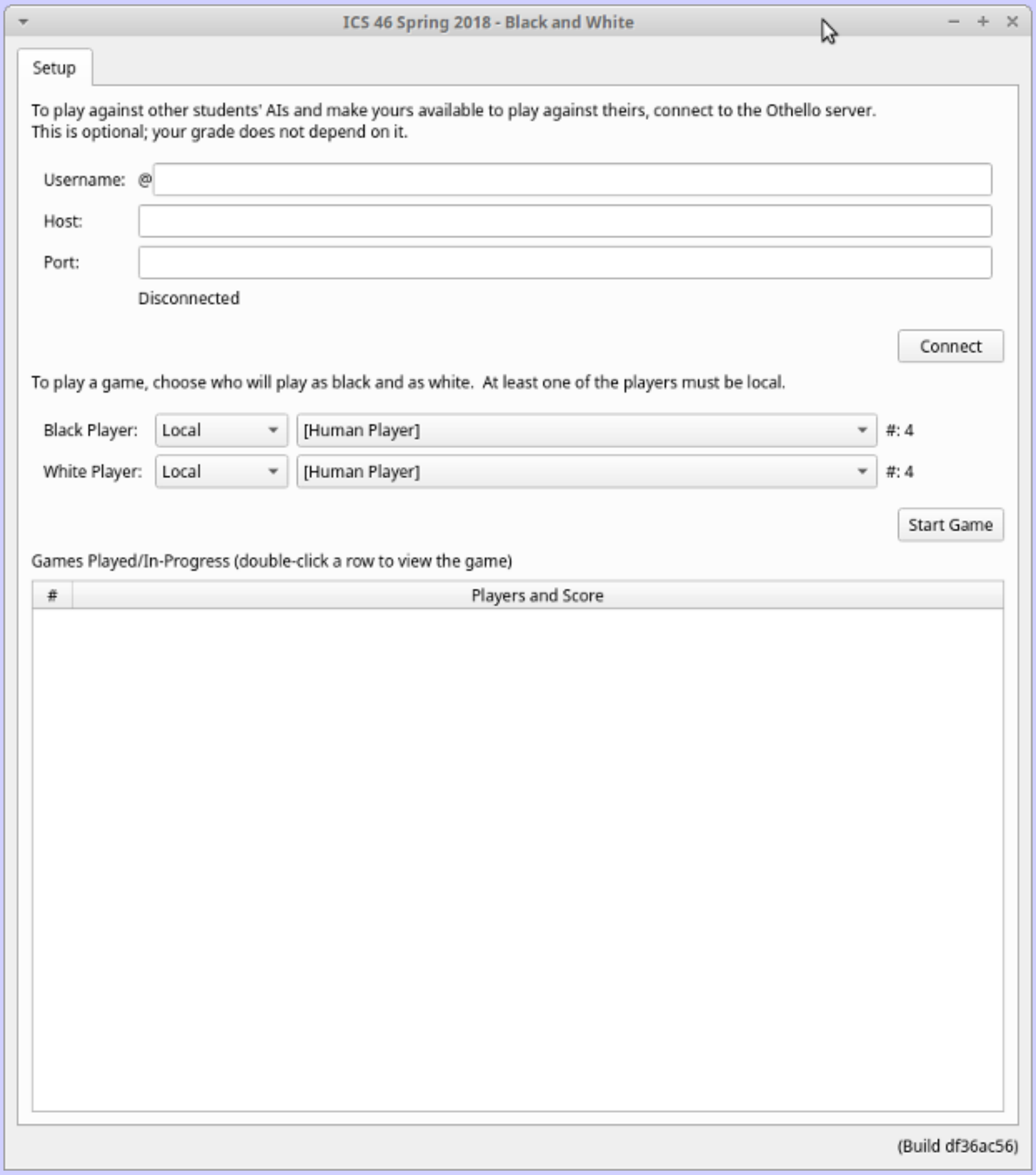
The rules of the game, along with some notion of strategy, are described in the [Wikipedia entry on Reversi](). If you haven't played Othello before, or have seen it previously but don't remember how it works, you should at least read the sections of the Wikipedia entry that covers the rules of the game; knowing how the game is played is crucial to implementing code capable of playing the game well.

If you want to try playing the game, you'll find that a complete, working version of Othello is included as part of this project already.

The application

Your work on this project begins with an already-working Othello game with a graphical user interface (GUI), allowing two human players to play the game against each other, as well as play involving AIs and across a network against other AIs. Also included are two rudimentary artificial intelligence implementations — neither is exceptionally intelligent, actually, but they are at least capable of selecting moves automatically so you can play against them.

When you start the application, a window arranged like the one below will appear:

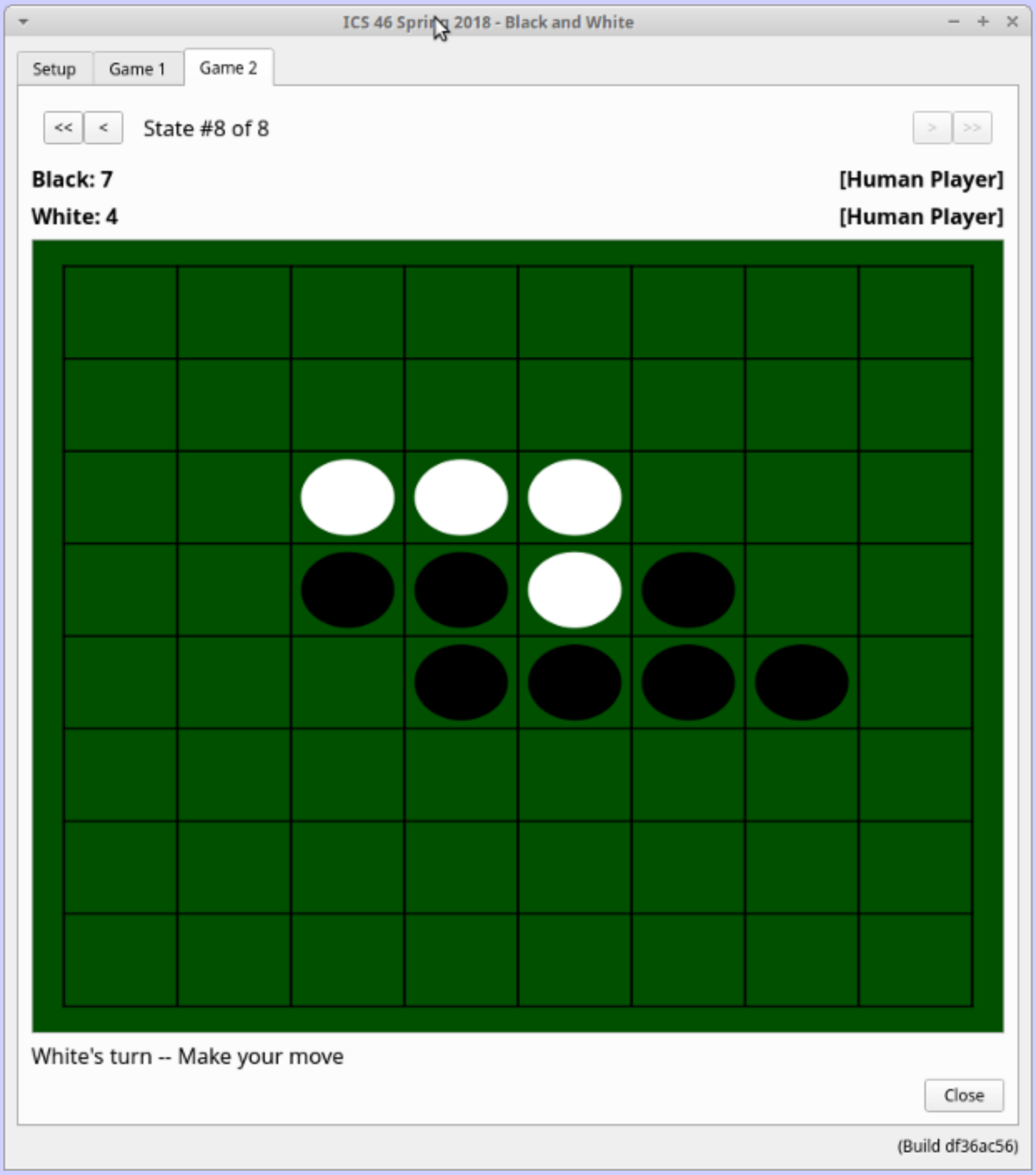


In the **Setup** tab, you can set up a game to be played, as well as see the results of previously-played games. It's arranged into three sections.

- At the top, you can select a username, then specify the host and the port where the Othello server is running — you'll receive an email specifying this soon, if you haven't already. That will connect you to the Othello server, which you'd want to do if you want to either make your AI available to other players or to play against other students' AIs. More about that later.
- In the middle area, you can select who will play as the black player and the white player. **Local** means either a human player (i.e., you) or one of the AIs you have on your own VM; if you want to play against other students' AIs, you can change that to **Network** (if you're connected to the server). Once you've chosen the players, click the **Start Game** button and a new game will start.

- The bottom section displays the results of previously-played games, as well as games currently in progress. These results are cleared out whenever the program ends — they're not saved to disk or anything like that — but can still be handy if you run multiple tests and want to see the games played out again.

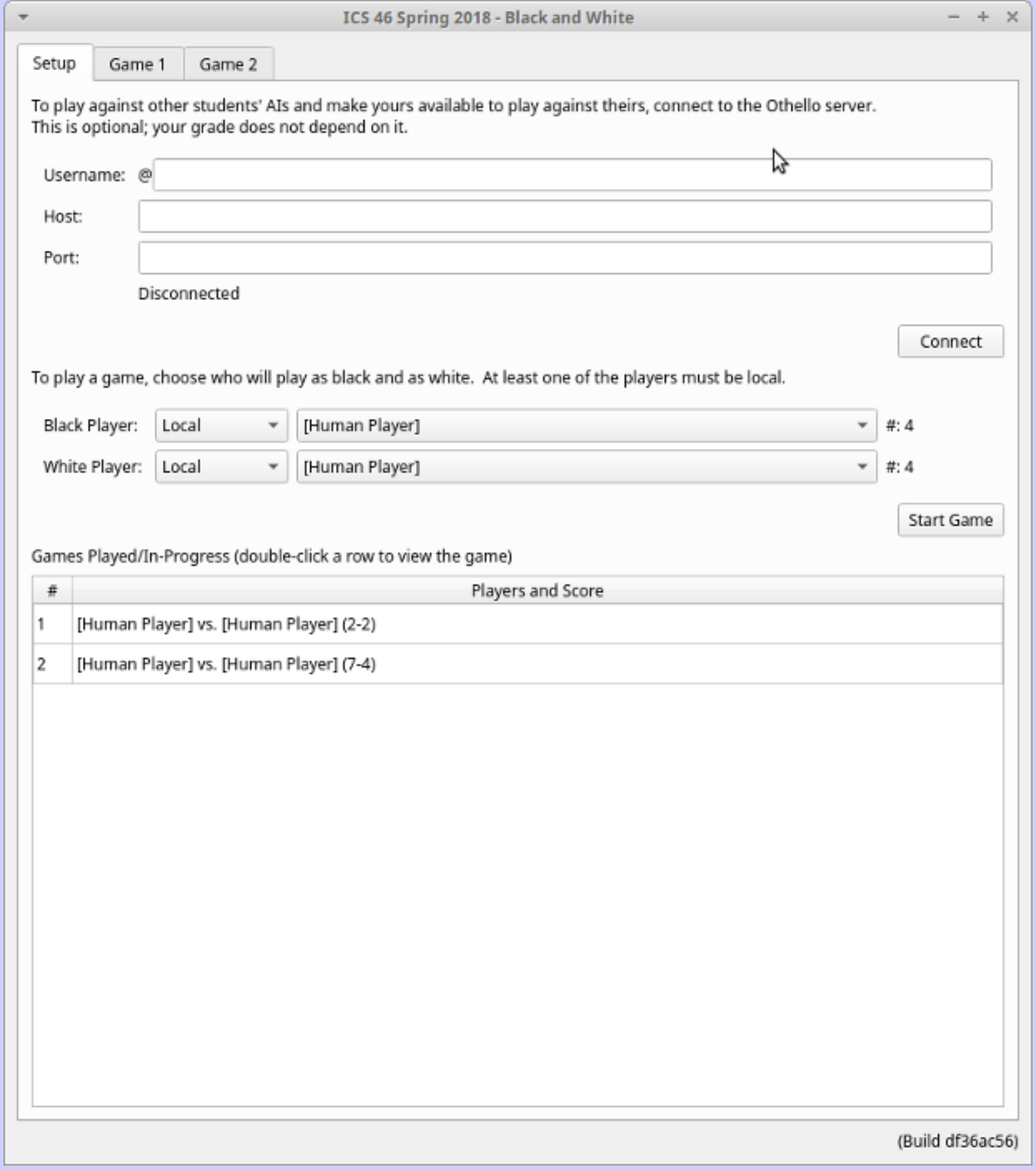
When you start a new game, an additional tab will show up in your user interface, which will display the status of that game.



When it is a human player's turn to move, all you need to do is click on a square to make your move. When it an AI's turn to move, the AI will make its own decision and the GUI will be updated automatically once the move has been chosen. If you'd like to quit the game, or if you'd like to dismiss it after it's ended, you can just click the **Close** button. (And note that you can bring this tab back, since the game will still be listed in the **Setup** tab.) Any time an AI makes an invalid move or throws an exception, an error message will be displayed along the bottom of the window.

Note, also, the buttons marked <<, <, >, and >>. These allow you to "rewind" and "fast forward", so that you can see previous states of the game — either while the game is playing or after it's over, it can be handy to back up and see previous states. You can only make a move when it's your turn *and* when you're displaying the most recent state; this doesn't "undo" a move, but instead lets you see previous states.

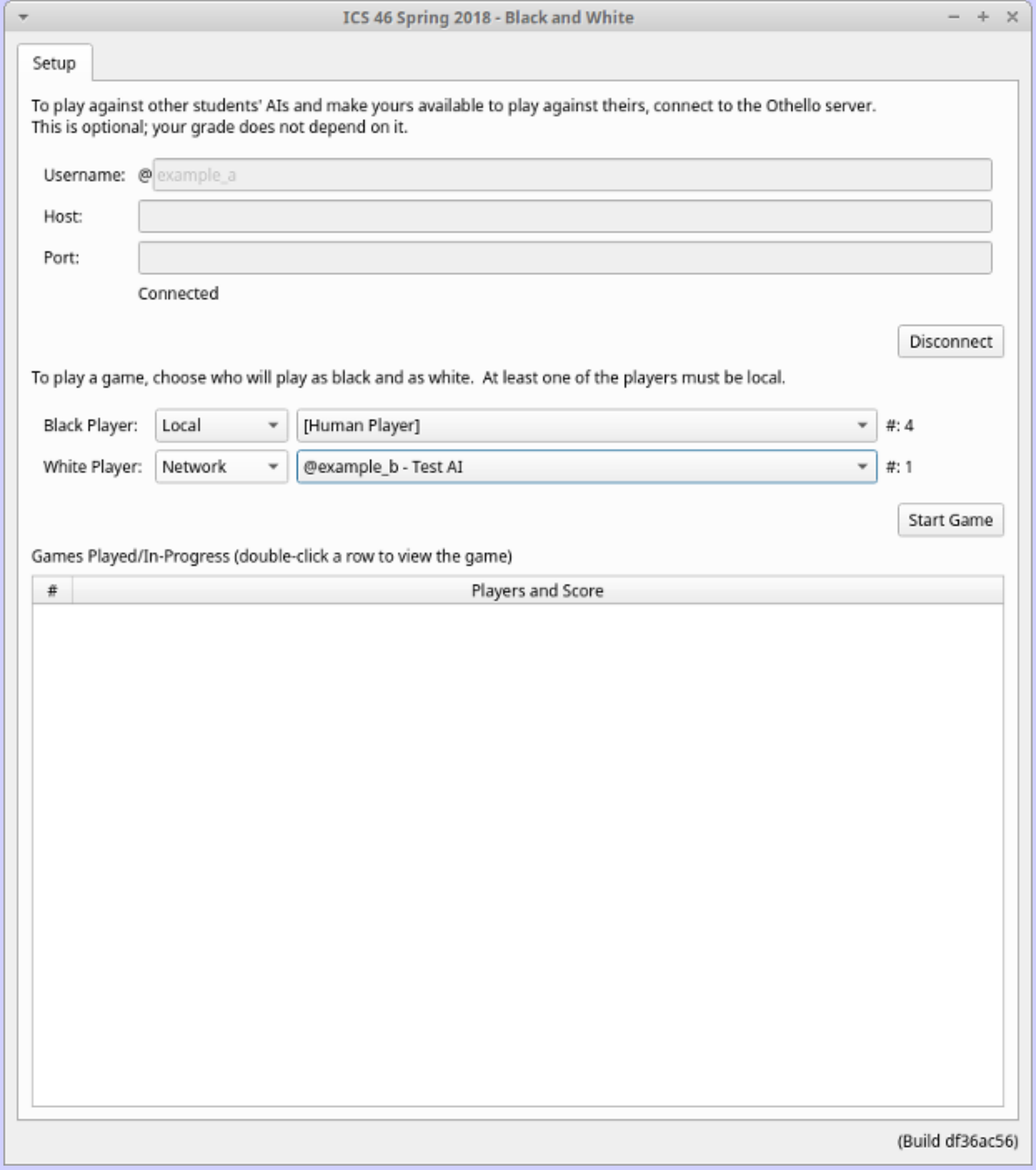
Once games have been started, you'll see them listed in the bottom area of the **Setup** tab.



Double-clicking any of those games will display the game's details in the GUI.

The Othello server

Once you log into the Othello server, you'll be able to play against other students' AIs. Logging in is straightforward: Choose a username (which, notably, cannot contain spaces), then specify the host and port (which I'll send to you in an email separately), then click **Connect**. Once you're connected, you can switch one of the two **Local** selections to **Network** where you configure the players for a game.



Next to **Network**, you'll see a list of all of the available AIs belonging to other students. Note that you will probably see these come and go — as students disconnect and reconnect — but as long as the other student is connected, you should be able to launch a game against an AI.

Please note that the Othello server is running on a machine on the ICS network that is not exposed to the open Internet. In order to connect to it, you will need to be connected to the campus network, which means you'll either need to be on campus or you'll need to be connected to something called the campus VPN, which allows you to access the campus' network from off-campus. Note, also, that certain residential areas are not connected to a part of the campus network that will allow you direct access to the Othello server, so you'll need to use the campus VPN in those cases. In general, if you're not able to connect to the Othello server, the first thing you should try is using the campus VPN.

Connecting to the campus VPN requires that you install some software *on your host operating system*, which you can obtain from UCI's Office of Information Technology at the following link:

- Campus VPN software

Open the link above, find the section titled VPN Software (as opposed to WebVPN, which won't work for this purpose), and click the link titled Download the VPN Software. This will require you to log in with your UCInetID and password, at which point you'll be offered links to Windows, Mac OS X, or Linux versions of the VPN software. Download the one that's appropriate for your host operating system and install it; instructions for setting it up are available from that page, as well.

A brief word of warning

When you start the application on your ICS 46 VM, you may notice an error message like this one in your Terminal window:

```
QApplication: invalid style override passed, ignoring it
```

This is something you can safely ignore, since it has no effect on our work here.

The requirements

This project actually requires you to complete only a single task: Write an AI that is capable of choosing moves in an Othello game, by using the recursive, search-tree-based algorithm described below. Optionally, you can implement as many additional AIs as you'd like, but one of them constitutes your "official" submission that will be graded; the others are strictly for use in the tournament (described below).

Each of your Othello AI implementations is subject to a few restrictions:

- You must derive the outermost class that implements your AI from the abstract base class **OthelloAI**, which is declared in a file **OthelloAI.hpp** in **include/othellogame**. (You can include this file by simply saying **#include "OthelloAI.hpp"**, since the compiler has already been configured to look in the **include/othellogame** directory for header files.)
- Your class must provide an implementation of this member function, which is pure virtual in the **OthelloAI** class:

  ```
  virtual std::pair<int, int> chooseMove(const OthelloGameState& state);
  ```

  the goal of which is to intelligently choose a move that the current player would make in the given game state, by returning a **std::pair** that contains an **(x, y)** coordinate for the move.
- Your class must be registered using the **ICS46_DYNAMIC_FACTORY_REGISTER** macro, similarly to how you registered your maze generator and maze solver in Project #1.
- Your class must be declared in a namespace whose name is your UCInetID (all lowercase). For example, my UCInetID is **thornton**, so I would declare and define my class this way:

  ```
  // MyOthelloAI.hpp

  #include "OthelloAI.hpp"


  namespace thornton
  {
      class MyOthelloAI : public OthelloAI
      {
      public:
          virtual std::pair<int, int> chooseMove(const OthelloGameState& state);
      };
  }



  // MyOthelloAI.cpp

  #include <ics46/factory/DynamicFactory.hpp>
  #include "MyOthelloAI.hpp"


  ICS46_DYNAMIC_FACTORY_REGISTER(OthelloAI, thornton::MyOthelloAI, "a human-readable name for your class");


  std::pair<int, int> thornton::MyOthelloAI::chooseMove(const OthelloGameState& state)
  {
      // implementation of the AI goes here
  }
  ```

  You would do something similar, though, of course, you wouldn't use my UCInetID of **thornton**; you'd need to use your own. You might also want to choose a better name for your class than **MyOthelloAI**; any name is fine, as long as you put your class into a correctly-named namespace. This requirement ensures that everyone's classes have unique full names, necessary so that they can all be linked together for me to run the tournament.
- All of the other functions you write *must* either be in the namespace whose name is your UCInetID, *or* in the unnamed namespace. *Do not* write any code that is in any other namespace (or in the default, "global" namespace). This, too, is to ensure that everyone's code can be linked together for me to run the tournament.

Naming your Othello AIs

Each of your Othello AIs is registered with a *display name* using the **ICS46_DYNAMIC_FACTORY_REGISTER** macro. The display name serves three purposes:

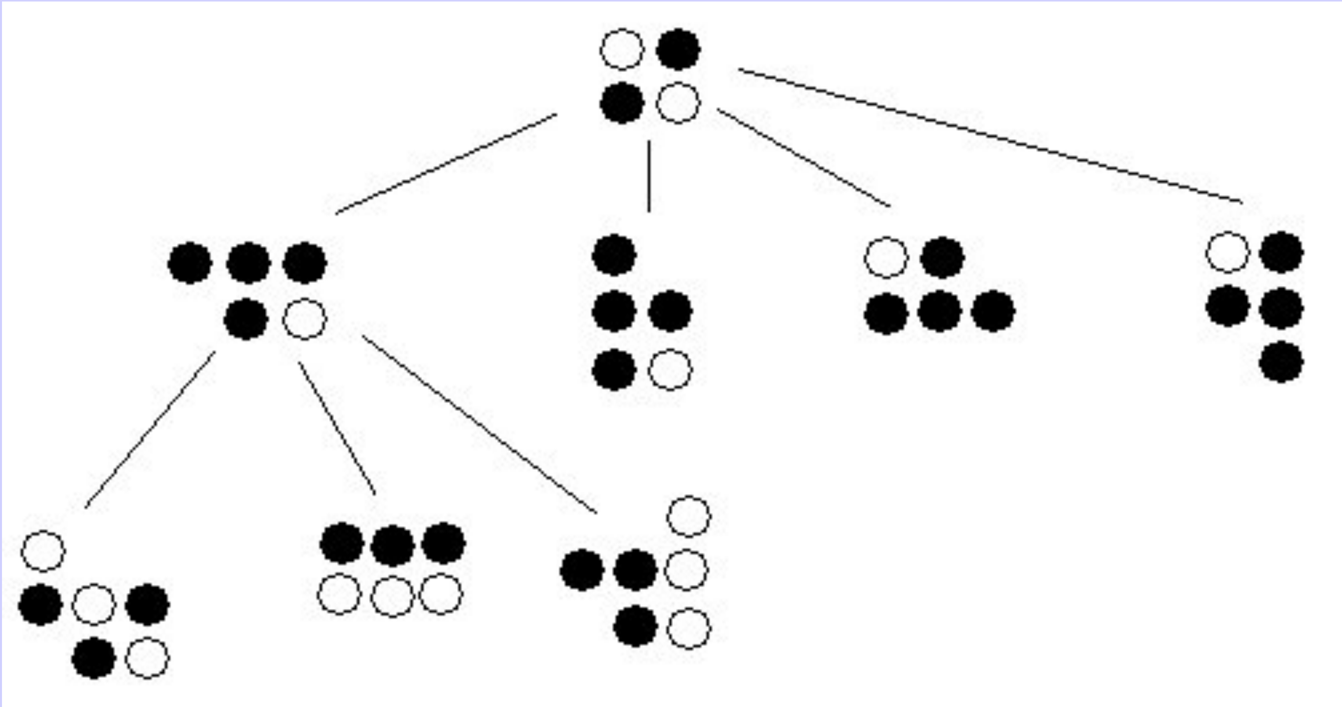
- It is displayed in the GUI, so you select your AIs as one or both of the players in a game.
- It uniquely identifies which of your AIs is intended to meet this project's requirements. The one you want graded *must* have a name that has **(Required)** at the end of it; capitalization and the parentheses are important here. Any other AIs you submit should not have names that end in **(Required)**, and, of course, *none* of your AIs should have a name that has **(Provided)** at the end of it, because none of your AIs were provided by us.
- It will show up in the final results of the Othello tournament, so be sure you choose something that is (a) unique enough that no one else will have chosen it, and (b) is inoffensive (I reserve the right to change names that I think are potentially offensive).

Game trees

You can think of the possible game states as being arranged, conceptually, in a kind of search tree called a *game tree*. Each node of the tree contains a particular game state *g*. Its children are the game states that can result from making each valid move from the state *g*.

The root of the tree is the initial game state — that is, the Othello game before the first move is made. The children of this initial state are all of the possible states that can arise from the black player (who moves first) making a valid opening move. There are four such states, corresponding to the four possible moves that the black player is permitted to make at the opening. (All other moves are illegal and, as such, are not to be considered.)

Here is a partial look at an Othello game tree:



In the picture, from the initial state, there are four possibilities from which the black player can choose its initial move. From the first of those, we see that there are three possible moves that the white player can make in response. Other moves aren't pictured, but the tree continues to grow in this fashion. (Not surprisingly, the game tree can grow large rather quickly, so you'll find that it's difficult to draw very much of it on paper.)

We'll call the leaves in such a game tree the *final states*. These leaves indicate the states in which one player or the other has won the game.

Exhaustively searching all possibilities

Each time a player wants to pick a move, he or she wants to pick the one that will lead to a winning game state. One algorithm for doing that would determine the best move in three steps:

1. We apply an *evaluation function* to each final game state. An evaluation function typically returns a number, where higher numbers are considered better. We then identify the final state with the highest value — that is the "end game" that we would like to occur, as it is the best win for us.
2. We determine the path from the current game state to the final state that we chose above.
3. We make the move that takes us from the current game state down the path toward the chosen final state.

Assuming that you had a complete game tree at your disposal, this is a simple approach to implement. However, practical limitations make this approach impossible. First of all, the number of game states on each level of the tree grows exponentially as you work your way down the tree, since there are a number of possible moves that can be taken from any particular game state. There simply won't be enough memory to store the entire game tree. (You can imagine that, if you build the game tree 20 levels deep, and there are four possible moves that can be made from any particular state, the number of nodes in the tree would be greater than 4^{20}, which is more than one quadrillion nodes!) Besides, even if there was enough memory available to store the tree, the processing time to create the entire game tree would be prohibitive.

So we'll need to find a compromise — an approach that perhaps doesn't always find the best possible outcome, but that makes a decision in a reasonable amount of time and uses a reasonable amount of memory.

Also, it's important to realize that just because you've found a path to the end game you want doesn't mean that you can force the events to take place that will get you there. Just as your goal is to make moves that are in your best interest, your opponent's goal is the opposite. So your algorithm will need to account for the fact that your opponent wants to beat you, just as you want to beat your opponent.

Heuristic search

The study of artificial intelligence has much to say about good ways to search toward a goal when it's impractical to check all possible paths toward it.

We can first make use of the following observation: Suppose the black player has made a move in the game, and the white player wants to figure out the best move to make, using the search tree approach we've been discussing. Then the white player need only be concerned with the subtree that has the current game state as its root. Once a move is made, all the other moves that could have been made can be ignored, as it is now not possible to take those paths down the tree. Thus, when analyzing the next move to make, we need only generate the part of the search tree that originates from the current game state. That's a good step toward reducing our storage needs significantly, though it's only the first step; especially early in the game, there might still be huge numbers of states that can arise from the state we're currently in.

To reduce our workload even more, we can employ a technique called *heuristic search*. In a heuristic search, we generate as much of the relevant subtree as is practical, using the resulting game states to guide us in selecting a move that we hope will be the best, even though we don't have time to get full information about how the move might turn out.

There are several strategies that we could use. At the heart of the strategy that we'll use is the notion of an *evaluation function* that we discussed earlier.

We'll need to rate each particular game state in some way, so that we can decide which of a large number of game states is the best outcome for us. A simple approach — though one that ignores some important aspects of the game — is the following:

eval(state) = number of tiles belonging to me − number of tiles belonging to my opponent

It's also important to note here that ***you do not need to actually build a game tree in memory***. Our algorithm will perform a sort of *depth-first search* on the game tree, meaning that we can use parameters in a recursive method (stored on the run-time stack) to perform the search, negating the need to actually build and store a game tree. This will dramatically reduce the amount of memory needed to choose a move, since only one path in the tree will ever need to be stored on the run-time stack at a time. So, in an eight-level-deep search, we'll store as many as eight nodes, rather than all of the nodes that can be reached in eight moves (which might be huge).

Putting these ideas together, we can develop a search algorithm that will help us to evaluate each of the possible moves we might make. That algorithm, sketched in a very rough pseudo-code, looks something like this:

```
int search(OthelloGameState s, int depth):
    if depth == 0:
        return evaluation of s
    else:
        if it's my turn to move:
            for each valid move that I can make from s:
                make that move on s yielding a state s'
                search(s', depth - 1)
            return the maximum value returned from recursive search calls
        else:
            for each valid move that my opponent can make from s:
                make that move on s yielding a state s'
                search(s', depth - 1)
            return the minimum value returned from recursive search calls
```

There are a few things we need to discuss about the algorithm above. First, notice that there are two cases of recursion: either it is your algorithm's turn (who is currently making the decision) or its opponent's turn. In each case, the algorithm is almost the same, except:

- ...when it is your algorithm's turn, the *maximum* value is returned. In other words, the algorithm wants to make the best possible move it can on its own behalf.
- ...when it is the opponent's turn, the *minimum* value is returned. This is because it is assumed that the opponent will also make the move that's in *its* best interest (which, in turn, is in our worst interest).

You *may not* assume that your algorithm will always be the black or the white player. Either the black or the white player (or both!) might be played by your algorithm. When deciding whether it's "my turn" or "my opponent's turn," you'll have to exercise some caution to ensure that you're making the right decision. There's more than one way to solve that problem, but you'll need to choose one. One thing to be aware of, though, is that you won't be able to use a global variable to solve this problem, as there might be two instances of your AI playing the game against each other, or there might be multiple simultaneous games being played at the same time.

Second, notice the **depth** parameter. This will be used to limit the depth of our search, to make sure that our search does a manageable amount of work. Each time we recurse one level deeper, the depth is reduced by one, and we stop recursing when it reaches zero. You'll need to experiment a bit to decide what depth can be handled in a reasonable amount of time, but without limiting the depth, you'll find that moves will take orders of magnitude longer than you'll be willing to wait.

Third, observe that when one player makes a move, it isn't necessarily the case that the other player will be making the next move; occasionally, in Othello, the same player gets to move twice in a row. So, care must be taken in deciding whose turn it is. The easiest way to deal with this problem is to count on the current game state to keep track of this for you; it can always tell you reliably whose turn it is.

Lastly, note that this algorithm returns the *evaluation* of the best state, not the best state itself. In short, calling search(*s*, 6) for some state *s* asks the following question: "Looking six moves into the future, and assuming I do the best I can do and so does my opponent, how well will the state *s* turn out for me?" You'll need to exercise some care in actually implementing this algorithm so that chooseMove() will be able to call search() and use the result to help it choose the right move.

Evaluation functions

The core of your AI — probably the most important factor that will set it apart from others — is the evaluation function that it uses to decide how "good" each board configuration is. I'm leaving this as an open problem and you're welcome to implement your evaluation function however you'd like. You might want to poke around the web looking for strategy guides or other information, taking into account, for example, that some squares on the Othello board are considered more important than others.

It's intended to be fun to play against your own program to see if you and your friends can beat it, and I also hope you enjoy fine-tuning your program until you have trouble beating it.

Why namespaces are important in this project

What are namespaces?

One problem that arises in large-scale software is that of *naming collisions*, which is to say that you end up wanting to have two functions or classes that have the same name. There are lots of reasons why this kind of collision can happen, such as using separate libraries that both happen to declare the same name but want it to mean different things (like having something called **vector** in both the C++ Standard Library and also in a linear algebra library, but you want to use both libraries in the same program). But because C++ linkers can't link a single program together if it has multiple, separate definitions for the same name, we need something to help us work around the problem. In most programming languages intended for large-scale software, some mechanism exists from arranging names in some way to prevent them from conflicting with each other. Java provides *packages*, for example, while Python provides *modules*. In C++, the mechanism for solving this kind of problem is called a *namespace*.

A namespace in C++ is a simple idea: It's simply a name that is added as a kind of prefix to the names of the things declared inside of it. You may have noticed, for example, that this code in C++ is illegal:

```
#include <string>

int main()
{
    string s;    // WILL NOT COMPILE!

    // ...

    return 0;
}
```

The reason it's illegal to declare **v** to have type **string** is that *there is no such type as **string** in the C++ Standard Library*. There is, however, a type called **string** *declared in the **std** namespace* in the C++ Standard Library, so we need to use its full name (or introduce a shorthand with a **using** declaration) if we want to use the type.

```
#include <string>

int main()
{
    std::string s;    // THIS WILL COMPILE!

    // ...

    return 0;
}
```

Declaring things in a namespace

Declaring names in a namespace in C++ is a simple matter of surrounding the declarations with a **namespace** block. For example, if you were implementing a class called **Person** and you wanted it to be declared in a namespace called **boo**, you might declare it this way:

```
namespace boo
{
    class Person
    {
        // Inside the class declaration, everything is as you've
        // already learned it.
    };
}
```

What you've actually done is declared a class called **boo::Person**, which is a class **Person** in the namespace **boo**. When you implement the member functions of your class in a source file, you'd use a slightly different technique: Include **boo::** in the names of what you're defining. For example, you might define your constructor this way in a source file:

```
boo::Person::Person()
{
}
```

Note that this is no different from what you've been doing all along, *except* for the use of **boo::** in the name of the class. We're telling the compiler that we'd like to define a function called **Person** (i.e., a constructor) in a class called **Person** that can be found in a namespace called **boo**. Because the compiler will have already seen the declaration, then it will be able to figure out what your intentions were.

The "unnamed namespace" for hiding names inside of source files

When you write a source file in C++, there are usually two things you're defining:

- The classes and/or functions that were declared in the header file. You can think of these as the *public interface* of your source file (i.e., the things that other source files are intended to use).

- Classes and/or functions that are utilities that you're using within that source file, but that are not part of its public interface. Code outside of the source file shouldn't be able to see or interact with these.

In that latter case, C++ provides a solution to allow you to "hide" something in a source file. If you surround a declaration and/or definition in a **namespace** block without a name associated with it, you're placing that name into what's called the *unnamed namespace*.

```
namespace
{
    bool isUppercaseChar(char c)
    {
        return c >= 'A' && c <= 'Z';
    }
}
```

If you wrote that function in a source file and placed in into the unnamed namespace, as I've done above, then you've specified that the function **isUppercaseChar** is available within this source file — without even having to qualify its name — but is invisible to the linker, meaning that it will not be available in any source files. This is a great way to hide functions or even whole classes that have no use outside of a particular source file, keeping internal details internal. (And part of what I mean by *hiding* here is that its name won't conflict with the names of other things defined in other source files. The fewer details of each source file "leak out" to other source files, the better chance you can keep a large project moving forward without mysterious bugs and negotiation about names.)

Why this matters in this project

The Othello tournament is one large program compiled together, including all of the AIs submitted by every student. Unfortunately, there's a pretty good chance that at least two of you will have an AI class with the same name, or that at least two of you will have some utility function with the same name. Unfortunately, that means I'll be unable to compile and link the tournament — and will have to disquality those students' code from it. But there is a way to ensure that everyone's code will link together properly: *All of your names have to be isolated from everyone else's.*

To accomplish that goal, you'll need to do two things:

- Your AI class will need to be declared in a namespace whose name is your UCInetID.
- Any additional functions and/or classes that are part of your AI will need to either be declared in that same namespace (named with your UCInetID) or will need to be in the unnamed namespace.

If everyone follows these rules, then everyone's code can be compiled and linked together into one Othello tournament.

A tournament!

After the project's due date has passed, I'll be gathering all of your AIs together and running a tournament to determine who has the best AI. In fairness, I'll explain here how the tournament will be organized; you'll need to follow these rules in order to win (or possibly even to participate).

- You can submit as many AIs as you'd like, but do be sure that there's something interesting and different about them; don't submit ten AIs that are virtually identical, for example, because I'll need to run $\Theta(n^2)$ games to complete a tournament with *n* AIs. That grows fast!
- When you register your AIs with a name using the **ICS46_DYNAMIC_FACTORY_REGISTER** macro, the name you choose will be displayed in the final results posted to the web, so (a) make sure that you don't include your name unless you want your name posted online, and (b) choose names that are somehow unique to you — be clever (but inoffensive).
- Each AI will play two games against every other AI, one each as black and as white.
- The primary factor in determining the "best" AI is the total percentage of games won. (Draws will count as 1/2 of a win and 1/2 of a loss.) So, first and foremost, your goal is to win games.
- A secondary factor, to be used in the case of a tie, is the total number of tiles accumulated in all games. This means that winning games big, as opposed to squeaking out close wins, is important if there's a tie, but that winning small more often still trumps winning big less often.
- Your AI will be given three seconds of CPU time to choose each of its moves. (Note that *CPU time* is not a measurement of actual time passed, but only of time consumed by the CPU. On the other hand, note that threads won't buy you any additional time, since four threads running simultaneously consume four times as much CPU as one thread does.) The three seconds is a hard limit; as soon as your AI reaches this limit, it will be aborted immediately.
    - I'll run the tournament on a machine with these specs: Intel Xeon E3-1505M CPU @ 3.0 GHz, 32 GB RAM, Windows 10 64-bit as the host operating system, with the ICS 46 VM running in a VirtualBox instance (with 8 GB of memory allocated to the VM). I can't guarantee I won't be doing anything else with the machine simultaneously, but since only CPU time consumed by your algorithm will be measured, this will only affect how long it will take me to run the tournament, not how much work your algorithm can do per move.
- Your AI needs to be *stateless*. It is entirely possible that different objects of your AI class will be called to choose different moves during the course of the same game, so you won't be able to save values in member variables and expect them to still be available the next time your AI chooses a move.
- Your AI is not permitted to launch additional processes, though you can launch threads if you'd like. (Note, though, that every thread consumes CPU time.)
- If your AI takes too long to make a move, returns an invalid move, throws an exception, crashes, isn't implemented in a class *within a namespace named by your UCInetID*, or violates any of the other rules laid out in the project write-up, it will be disqualified from the tournament.
    - Ultimately, only games between AIs that completed the tournament without being disqualified are counted. As soon as an AI is disqualified, all of its results are removed from the overall totals.

How well you do in the tournament will have no bearing on your grade, but it will hopefully motivate you to think a bit about how you might tune up your evaluation function — or explore alternative ways of helping your AI to see farther into the future. (You are required, fundamentally, to use the algorithm shown in this write-up to implement your "required" AI, though you can additionally do anything you'd like, and you can apply any optimizations to your required AI, provided that you're still basically implementing a recursive search tree based algorithm.)

Being disqualified from the tournament is an indicator we'll use in the grading, however, particularly because it will demonstrate to us that your AI has some kind of flaw (e.g., it chooses invalid moves, or crashes in certain circumstances).

Good luck!

Deliverables

After using the **gather** script in your project directory to gather up your C++ source and header files into a single **project3.tar.gz** file (as you did in Project #0, submit that file (and only that file) to Checkmate. Refer back to Project #0 if you need instructions on how to do that.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're reponsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally. (It's not a bad idea to look at the contents of your tarball before submitting it; see Project #0 for instructions on how to do that.)

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

Description and examples of new GUI implementation added by Alex Thornton, Spring 2018.
Additional clarification about namespaces, plus additional tweaks by Alex Thornton, Winter 2018.
New version of UI, along with some tweaks and clarifications made by Alex Thornton, Spring 2016.
Requirements tuned, and implementation beefed up (including an online component) by Alex Thornton, Spring 2015.
Originally written by Alex Thornton, Spring 2014, with heavy influence from a similarly-named project from ICS 23.