1 Description

In the first part of this programming project you designed and implemented an Othello playing program. In this second part, you will add a learning component to your player, so that it can learn to improve its performance from its own experience.

As in the first part, after you have completed your programs a tournament will be held and the four finalists will receive extra credit. For this tournament, the constraints are:

- 2 minutes total time for each player, i.e. 4 minutes total per game.
- 20,000 nodes expansion limit per move.

Again, in this second part of the assignment you will work individually. The preferable implementation language is C, but other languages can be used as well as long as you can establish socket communication with the game server that we provided. The due date for this part of the project is Friday, April 27th.

2 Tasks to Do

Here is list of tasks that you are expected to accomplish for this assignment:

- Design and implement an Othello player with learning abilities. You can build upon your program from Part I of the project or start from scratch. You may apply any learning method you prefer. Some ideas and details are given in this document.

- Create an executable that will participate in the tournament. Use the arch7?.cs.duke.edu and arch8?.cs.duke.edu machines for testing; we will use these machines during the tournament. Make sure that your player completes any game within the time limits.

- Turn your Othello player in by Friday, April 27th. Your program should be able to play any number of games without crashing.

- Turn in a writeup explaining your learning algorithm and any clever extensions you have devised.

3 Learning to Play Othello

The Othello player you designed in the first part of this project is probably a sophisticated Othello player and you may have trouble beating it. However, you have probably realized that it is quite deterministic, static, and somewhat predictable. There is no variation in its playing and it does not
improve performance with experience. The goal of this part of the project is to implement at least one learning method and show that it significantly improves performance over your original player.

There are several ways to incorporate learning methods in a game playing agent. For example, you can learn directly how to take actions in any state of the game, or you can use learning to fine tune parameters of your evaluation function in minimax search. This document outlines some ideas with more emphasis on the use of reinforcement learning for linear evaluation functions.

4 Evaluation Functions

In the first part of this assignment you designed an evaluation function to complement your minimax and alpha-beta search procedure. By now, it should be clear that the evaluation function plays an important role in creating a good computer program for Othello.

One important aspect of choosing a good evaluation function is to consider what features of a board configuration make it a good or a bad configuration. One thing you can look at is the number of pieces each side has. Another possible feature to look at is mobility. This is a measure of the ability to make moves. Clearly we would like to be able to make moves and we would like our opponent to be able to make as few moves as possible. One other aspect of Othello is that corner squares are valuable since placing your disc on a corner means that it can never be flipped (our modified board has eight corners). Edges are fairly valuable since they have less opportunity to be flipped. Many Othello players have noted that exact edge configurations are also important, i.e., looking at specific edge configurations is more helpful than just counting pieces on the edges. There are other features that you can take into account. Some are more useful in determining how good a board configuration is than others.

One way to combine some or all of these features into one evaluation function is taking a weighted linear combination of them:

\[ V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s), \]

where \( s \) is a board configuration, the \( f_i \)'s are features of the board, and the \( w_i \)'s are the weights.

The \( f_i \)'s are functions from board configurations to real numbers. Each \( w_i \) is a weight that determines how much importance \( f_i \) has.

Alternatively, you can combine these features into any arbitrary (linear or non-linear) parametric function \( g \) with parameters \( w \):

\[ V(s) = g(f_1, f_2, \ldots, f_n, w) \]

For example, \( g \) can be a polynomial of some degree, a multi-layer perceptron, or some other kind of a neural network. There are many choices for \( g \) but the points to keep in mind is that your approximation architecture should be expressive enough to represent the value function and simple enough to facilitate learning of the parameters.

Note that you will have to experiment to find which features are useful. Think about other features and which features to use in the evaluation function. Note that it is not always a good idea to use as many features as possible since this means that there are more weights to learn. The more weights we have the more data we will generally require to learn them well, so it may be better to have a few excellent features rather than a lot of reasonable ones.

Notice that the game board is symmetrical. Can we take advantage of this to reduce the number of features? For example, does this mean that every corner needs a different weight or can we combine them into one feature? If you like, you could consider creating combined features by multiplying features together, e.g. you might like to look at \( f_1(s) f_2(s) \) as one feature instead of (or together with) \( f_1 \) and \( f_2 \). In general, the problem of feature selection is a very important (and open) problem in Artificial Intelligence and other areas.
5 Learning an Evaluation Function

5.1 Temporal Difference Learning

For an evaluation function which is a parametric combination of features, learning boils down to finding the “best” set of weights or parameters (for a fixed set of features). “Best” weights/parameters are the ones that yield an evaluation function that achieves best performance in playing the game.

Suppose that we have a current guess for the weights in our evaluation function. We now want to update the parameters $w_i$ so that the evaluation function gives us a better indication of the value of a board position. More precisely, suppose that it is our turn to move, and that the current state is $s$. We find another state $s'$, such that $v' = V(s')$ is a better estimate for the value of $s$ than the current value of $V(s)$. We then adapt $V(s)$ to get it closer to $v'$. The main question then is which state $s'$ we use to define $v'$. There are two basic ways of doing this:

- **Method 1**: We can construct a game tree with root node $s$, and then expand the game tree to some depth to figure out the “optimal” move $a$. We then define $s'$ to be the state reached after we take action $a$ in state $s$ and then the opposing player takes the move that minimax predicted that the player would take. In other words, $s'$ is the state which our analysis predicted would be reached after two moves (one for each player). In this case, we would take $v'$ to be the value of our heuristic function $V$ at $s'$. Note that, since players move in alternation, $s'$ is a position where it is again our turn. This property is useful, because the heuristic function might depend heavily on whose turn it is, so it makes sense to make $V(s)$ closer to $V(s')$ only if it is the same player’s turn in both.

- **Method 2**: An alternative approach is, at node $s$, to actually take the optimal move $a$, as predicted by the minimax search. We then wait for our opponent to move. The opponent’s move leads us to a new board position $s''$, which may be different from the $s'$ that our minimax search predicted. We then take $v'$ to be the heuristic value of $s''$. Notice that this approach will only work effectively if each player chooses to take the move that they think is best: otherwise, we are training $V(s)$ to be closer to the $V$ value not of the best next state, but of a very suboptimal next state.

Given $v'$, which can be selected by either of the above two methods, we wish to update the parameters $w_i$ to make $V(s)$ closer to $v'$. In the case of a linear evaluation function, this is straightforward. For each weight $w_i$, we use the update rule:

$$w_i^{(t+1)} = w_i^{(t)} + \alpha f_i(s) (v' - V(s))$$

For arbitrary functions, the update rule becomes:

$$w_i^{(t+1)} = w_i^{(t)} + \alpha (v' - V(s)) \frac{\partial g(f_1, f_2, ..., f_n, w)}{\partial w_i}$$

For specialized functions $g$ there exist specialized update rules. For example, if $g$ is a multi-layer perceptron then the update rule is the back-propagation rule.

Notice that these update rules are simply instances of gradient descent; they make small updates in the direction of the gradient. Here $\alpha$ is called the learning rate and determines the effect of each update. You will have to experiment with its value; it should be a small positive real number.

Notice also that these rules suggest updates based on the difference between temporally distant estimates of the target function $V$. For that reason, they are known as Temporal Difference learning methods.

After adapting the weights/parameters, we continue playing with the new heuristic function. You may use either of the methods above in your training. All have been used in practice. You may also wish to experiment to see which one is more suitable to Othello.
5.1.1 Further Issues

When we try to learn the weights of the evaluation function there are several issues to consider. Here we give a few suggestions of things to think about.

- What is a good training strategy? To train the program against itself? Perhaps to train against another program? Do you want your program to keep learning when it is playing in the tournament, or do you want to first train the program (i.e., find good weights for the evaluation function) and then just use the learned weights?

- While we generally wish to take the best estimated move, in the training phase we often want to explore the space, so we may want to occasionally take suboptimal moves. In method 1, the move we actually take doesn’t affect $v'$, so that there is no problem taking random moves if you use this method. In method 2, however, if we take a bad move then $V(s'^0)$ will not really reflect the best value reachable from $s$. You may like to try and think of a way of getting around this limitation by using method 2 when we take the best estimated move and using method 1 if we decide to take a random or estimated bad move.

- Generally, we can initialize the weights to random numbers and then use learning to adjust the weights to better values. For different initializations of the weights we may get different learned weights, and some may produce a better evaluation function than others. Perhaps you should try many different random initializations of the weights. What about trying to guess some good weights and use these as a starting point for the learning algorithm?

5.1.2 Advanced Issues

Here are a few more suggestions that you may like to consider. These are purely optional, but may improve your program’s performance in the tournament.

There are other ways of adjusting the weights/parameters. Methods 1 and 2 above just look at the value of a node’s best successor and compare the difference. The more we expand the game tree the better idea we generally have of how good the state at the root is since we have looked at more moves ahead. Thus you could try expanding the game tree as much as possible, use minimax search to find the value of the state at the root and use that value instead of $v'$. There are some relative advantages of training the value of $s$ using states $s'$ that are immediate successors of it, versus using states that are closer to the end of the game. The former are more accurate because the state at $s'$ is fairly close to the state at $s$, so that there is less room for accidental disparity. The latter are better because the actual value of a state only becomes clear at the end of the game.

One extension that may be interesting to explore is to see which of these is more appropriate in Othello. For example, an extension to method 1 is to do minimax search to some (even) depth $n$, and use the $V$ value of the state $s^*$ that the minimax search predicts will be reached in this tree. An even more elaborate extension would be to use some exponentially decayed average of the $V$ values of the various (even ply) nodes along the path from $s$ to $s^*$ (with further nodes being weighted less). Similarly, for method 2, one can use the exponentially decayed average of the evaluations of the nodes actually encountered in the game starting at $s$. Again, you should only use the evaluations for nodes where the same player moves. This variation of temporal difference learning is known as TD($\lambda$).

There are also other reinforcement learning algorithms described in the textbook, in other textbooks (for example, R. Sutton and A. Barto, Reinforcement Learning: An Introduction, MIT Press, 1998, available online at http://www-anw.cs.umass.edu/ rich/book/the-book.html), and in various research papers (check out http://www.cse.msu.edu/rlr/). You may like to try using one of them instead.
5.1.3 Potential Problems

Here are some things to keep in mind if you find that your weights are not converging.

- You may want to use your evaluation function when evaluating non-terminal nodes, and use something like piece differential (or -1 for lose, 0 for tie, 1 for win) to evaluate terminal game states.

- If you notice that your weights stabilize for one part of the game then diverge and then stabilize to something else at later parts of the game then try splitting the game up into phases and train different sets of weights for the different phases. If so, it is better to train the end game phase first, then the middle, and then the start since this is effectively propagating the true value of the states (obtained from the terminal state evaluations) up the tree.

- The choice of initial weights is critical and so is the learning rate. The learning rate should be very small (on the order of 0.05 or smaller). You might also consider reducing the learning rate gradually iteration by iteration.

- Normalize the features to vary between -1 and 1 by default (you can experiment with other ranges). The choice of features is important. You need features that will capture the goodness of the board at the beginning game, middle game and end game. As the game progresses different features might become the important features. All these types of features need to be included to aid convergence.

- Don’t re-normalize the weights after each iteration!

- Don’t disallow negative weights!

5.2 Confidence Intervals

Another way to use learning methods for your evaluation function is to estimate confidence intervals (or variance) of your heuristic evaluation function. The variance in the evaluation function can be used in many ways; it can be used as a crude measure of quiescence, but it can also be used in a more sophisticated variant of alpha-beta called MGSS*. Details for this method as well as a description of the MGSS* algorithm can be found in the book Do the Right Thing: Studies in Limited Rationality by Stuart Russell and Eric H. Wefald (Cambridge, MA: MIT Press, 1991).

6 Supervised Learning

The methods above are geared toward improving your heuristic evaluation function. Alternatively, you can use supervised learning to train a neural network to pick the “right” action in each state (with or without search). The training set may consist of examples obtained from expert (either human, or deep search) rankings of moves. You can use back-propagation to adjust the weights of your network. Neuro-Gammon, a program that plays Backgammon has been built on this idea (see G. Tesauro, “Neurogammon: A Neural-Network Backgammon Program,” Intl Joint Conf. on Neural Networks Proceedings III, 33–39, 1990).

Alternatively, you can use supervised learning to learn how to order node expansions during alpha-beta search. This technique can be combined with reinforcement learning for the evaluation function as described above to yield better results.
7 Remembering What You’ve Learned

Your learning player should be able to improve its performance with experience; the more games you play, the better it becomes. That implies that there should be some way of transferring whatever was learned during previous games to all future games. In other words, you probably need to store some information (e.g. the learned values of the weights/parameters) between different executions of your code. You may want to have a file that you update at the end of each execution and you read it off at the beginning of a new execution.

8 Other Bells and Whistles

We have given an outline of the project and a few suggestions on how to improve it. Note that this is by no means a complete list of things you can do and you are encouraged to explore other methods and ideas. This is an open-ended project and all ideas are welcome.

9 Implementation Issues

9.1 Executables from Part I

All players from Part I of this project have been compiled for both Sun and i86 architectures and are available in the directory /usr/project/courses/spring01/cps271/OthelloPlayersPartI/, which is accessible from all CS machines. You will find two subdirectories there, Sun/ and i86/. Choose one of the two depending on the type of machine you are using. The filenames are the first names of the authors. You can use these players for training and testing your new player.

The script play can be used to start a game between two programs. The command:

```
play <program1> <program2>
```

will start the server and the two programs you specified. <program1> will play white and <program2> will play red. Once the server window appears, you can select the program controllers for each player and the game will begin.

9.2 Othello Server

You are encouraged to use the GUI server for testing your programs. It offers a much better interface and allows switching between players at any time. You can start it by running the script othello.tcl available in the executables directory.

9.3 Time

The Tcl/Tk server passes the remaining time to each of the players. In the main procedure, it can be accessed as &msg.time_left; this is the remaining time in seconds. You may want to use this extra information to adjust your player. Note that this applies only to the GUI server, not to the ASCII server.