Artificial Intelligence	Programming Assignment 2
Comp440 Fall 2007	18 September 2007

This assignment is due October 9 in class.

1 Game playing programs that learn from experience [120]

In this problem, we will explore strategies for an agent that decides what to do by systematically considering the outcomes of various sequences of actions that it might take. Our agent is in a stochastic environment, where there is a probability distribution over a set of outcomes resulting from an action in a state. We study this abstract problem in a concrete setting — that of playing backgammon. We choose this setting because unlike real-world problems that have vague specifications, the rules of backgammon are simple, well-defined, and have a concise description. For a nice description of rules, look at `http://www.bkgm.com/rules.html`.

Backgammon is an interesting game with a very large state space (estimated to be about 10^{20}), and an imposing branching factor (there are 21 unique dice rolls, and about 20 moves per roll, on average).

Your assignment consists of two creating two backgammon players:

1. a baseline expectiminimax player for as described in Section 6.5 of your textbook. The most critical component that you will create is the static evaluation function which estimates the merit of boards generated during search.
2. a hill-climbing learner that tunes a parametric evaluation function of your own design.

You will then test both players against a random player and (for extra credit) against a championship level player `gnubg`. This is just an overview description of the assignment. Additional details are provided in subsequent sections of this document.

All the support code for this assignment is available at

`http://www.owlnet.rice.edu/∼comp440/handouts/bkg.tar.gz`.

A round-robin tournament among all student submitted programs will be held. The top three teams in the tournament will receive extra credit of 30 points.

1.1 Background

There is a long history of work on backgammon players. Much of this work has been focused on hand-crafted evaluation functions. There is a much smaller body of work exploring smart ways of exploring fewer lines of play, and performing deeper searches. One of the earliest programs was BKG, built by Hans Berliner of CMU. The best known player today is TD-Gammon which plays at the world-championship level. TD-Gammon was built by Gerry Tesauro using reinforcement learning to tune a parametric value function by self-play. Another impressive program, which is better than most commercial programs, is `gnubg`. Gnubg is available at http://www.bkgm.com/gnu/AllAboutGNU.html.

As you develop both your expectiminimax and hill-climbing players, you can consult papers, books or online references on these and other systems for ideas you want to incorporate into your backgammon player. *You need to clearly cite your sources both in your code as well as in your report.*

2 Your Tasks

Below are the tasks you should complete for this programming assignment. Although backgammon normally involves betting using a "doubling cube", we have removed the doubling aspect in order to simplify the assignment.

2.1 Static evaluation functions

A good evaluation function should correctly estimate the probability of a win from different board positions. An important aspect of designing a good static evaluation function is to choose features of a board affect that the probability of winning from that position. In backgammon, one obvious feature is the pip count for a player. Another feature is the number of blots. A blot is a vulnerability for a player, because if an opposing piece lands on a blot, the piece will be moved to the bar. Additionally, certain locations on the board are more valuable to be in than others.

- (20 points) First you will build the expectiminimax algorithm for selecting moves. Like minimax, expectiminimax is a full-width search up to a particular look-ahead depth. Since the game tree for backgammon has a large branching factor, deep look-ahead searches are not feasible in tournament play (unless you distribute the computation over a cluster of machines, which is possible because search expansions corresponding to particular dice-rolls can be done in parallel). For this assignment, you will experiment with 1-ply and 2-ply searches. Here 1-ply refers to a roll of the dice by one of the players and the play he makes for that roll.

  To test your player construct a simple static evaluation function utilizing at most 10 hand-picked features of the backgammon board (such as the ones described above). Get your move selector to return the best move according to this evaluation function. Tabulate the scores over 100 games for 1-ply and 2-ply look-ahead search. Select our random player as your opponent for this part. Comment on your results. Does your evaluation function perform better with deeper look-ahead? Why or why not? For extra credit of 10 points do this performance comparison against `gnubg`.

2.2 Learning an evaluation function

TD-Gammon uses a very simple but large feature set to estimate the probability of a win from a board. For each point on the board, there is a feature vector (x_1, x_2, x_3, x_4), where all the elements are 0, when there are no white pieces on that point. If there is exactly one white piece, it is represented as the vector (1,0,0,0). If there are exactly two white pieces, the vector is (1,1,0,0). If there are exactly three white pieces, the vector is (1,1,1,0). If there are $n > 3$ white pieces, the vector $(1, 1, 1, (n-3)/2)$ is used to represent it. Thus, just for white, there are $24 \times 4 = 96$ features, for both black and white, there are a total of 192 features. There are four additional features: the number of white and black pieces on the bar, and the number of black and white pieces already removed from the board. TD-Gammon scales each feature into the [0,1] range. These features are designed for use in a multilayer feedforward neural network.

We can combine some or all of these features into one evaluation function by taking a weighted linear combination of them:

$$F(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

The f_i's are functions from board positions to real numbers (the function f_i corresponds to the i^{th} feature). The magnitude of w_i determines how big a contribution parameter i makes to the final evaluation. The sign of w_i determines whether parameter i has a positive or negative impact. In general, the more weights you have, the more data you will generally need to learn them well.

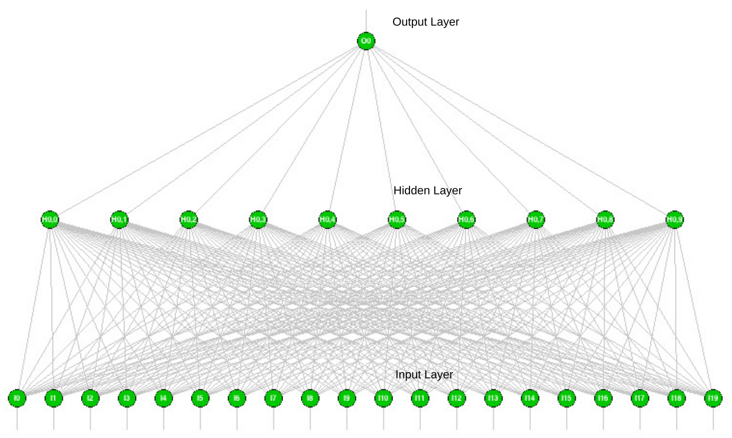
The correct weight for each parameter often depends on other aspects of the game. The value of hitting, for example, is usually greater when you are behind. So there is a relationship between the number of checkers your opponent has on the bar and, the relative pip count. We can account for such relationships by adding more terms to the static evaluator. To the linear terms we have already described, we add second-degree terms consisting of pairs of parameters multiplied together. Each second-degree term also has its own weight.

$$F(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) + w_{n+1} f_1(s) f_2(s) + \ldots +$$

For backgammon, the value produced by the static evaluator is most useful to us if we can interpret it as the probability of the on-roll player winning from that position. In other words, we want the evaluator to give us a number between zero and one. This allows the expectiminimax player to work correctly (see the discussion on page 177 of your text for the need for scaling evaluation functions appropriately). To generate probability estimates, which are in the $[0, 1]$ range, we use a squashing function to scale the sum of the weighted parameters. The recommended squashing function is the sigmoid function $\sigma(a) = \frac{1}{1+e^{-a}}$. This is a smooth and increasing function which takes any real value as input and produces a result between 0 and 1. Large negative numbers give values close to zero and large positive numbers give values close to one. Thus, the final evaluation function $V(s)$ is:

$$V(s) = \sigma(F(s)) = \frac{1}{1 + e^{-F(s)}}$$

The neural network which TD-Gammon uses consists of an input layer of the above-described 196 features, a "hidden layer" of 20 units, and a layer of 4 output units. The four output units return the probability of a white win, white gammon, black win, and black gammon, respectively. A sample neural network is shown below:



As you can see, there are a very large number of weights (almost 4000 in the network you will build), so hand-tuning such a network is clearly not feasible.

- (100 points) Your second task is to have your player learn a good static evaluator for backgammon by repeated self-play. We will use a neural network with the same structure as Tesauro's network for TD-Gammon with one variation: instead of 4 output units, you will have a single output unit representing the probability of the black player winning from the current position.

  Instead of using Tesauro's method of learning, you will use simple hill-climbing algorithm to learn the network weights. Suppose we have a the current weights w_{i_A} for our evaluation function A. If we wish to improve A by incorporating the weights from another evaluation function B we will use the update rule

  $$w_{i_A} \leftarrow (1-\alpha) * w_{i_A} + \alpha * w_{i_B} \tag{1}$$

  which will nudge the weights in A towards the weights in network B. The parameter α is a small positive real number. You will have to experiment to find a good value for α – we recommend that you start with a value like 0.05.

  In order to build an effective network, one approach is try and build a champion by playing other players and continually improving your champion by incorporating better players. Doing this yields the following algorithm:

  1. Initialize your champion network by choosing random weights or setting all of the weights to some fixed constant.
  2. Do the following N times:
     (a) Pick a challenger for your champion. You need to make this choice carefully because it directly impacts the evolution of your champion. A good option is to use a slightly mutated version of your current champion. You can add a random number in the $[-0.5, 0.5]$ range to each of the weights in the champion network.

(b) Determine which of the two networks is better. This is an important decision; simply playing the two networks together once is not sufficient, since backgammon involves a good deal of luck. A good choice is a best-of-x tournament. If the challenger beats your current champion, incorporate the challenger using equation 1 from above. Otherwise, ignore the challenger.

Also, as your network gets better over time, you will want to make sure that your network is not brought astray by inferior players who get lucky. You will want to make sure that a "lucky random" player does not cause you to worsen your champion. This is done by monitoring the progress of your champion by periodically testing it against the random player we provide. Do not use our random player as the challenger. You use the random player to make sure that your champion improves over time. You may see that it take $N = 10,000$ iterations of the above algorithm in order to see improvement in your champion. Get started early on this assignment!!

(c) In previous years, many students found this algorithm to be insufficient to get sufficient stablization for a good player. A better alternative, that you are welcome to try, is to train your hill-climbing player against the expectiminimax player. Once your hill-climber gets reasonably better than expectiminimax, you can switch to training against a champion and just use the expectiminimax to make sure that your climber never gets off track.

(d) Note that since you are only training your network to play for the black player, using the same network naively to play as the white player will not result in good performance. In practice, we have found that simply subtracting the probability that black will win from one does not provide a white player of equal strength. Therefore, the easiest approach to using the same network to play for white is to permute the inputs when white plays so as to interpret the white player as black and vice versa.

2.3 What to turn in

You need to turn in a working, documented program that implements one `Player` for backgammon that implements expectiminimax and one `Player` that learns from its experience. You should also turn in a copy of your final neural network with the weights you learned. This should be done by using the `NeuralNetwork.writeTo(filename)` method which serializes the neural network object into a file.

The final evaluation function learned by your hill-climbing player will be used in a round robin tournament. Results of the tournament will be posted on the newsgroup. The top three teams will get extra credit points for this assignment.

You need to submit a written report providing

1. the results of your expectiminimax player against a random player on 100 games with a simple evaluation function

2. your experience and approach building a learning player. For the learning player, you should include a learning curve showing improvement in performance with training, as well as expla-

nations of your experimental results. You should also play against your learning player and describe some of the behaviors which it has learned.

For extra credit of 20 points, play both of your players against `gnubg` and report the results for 100 games.

3 The code

In this section, we will give you an overview of the provided code for this assignment. We have provided the engine which simulates a backgammon game and allows you to program your player. The `Backgammon` class implements this functionality - you can create a `Backgammon` game with two provided players and then run the `run()` method to play the game and return the winner of the game.

3.1 Drivers

We have provided two drivers for the game, which allows you to either play against your player or watch as two different players play each other. The first of these drivers is `TextDriver` which allows you to use a shell to interact with the backgammon game. You can simply run the `main` method in `TextDriver`, and then type `help` in order to get a list of all of the available commands.

The second driver we provide is a Swing-based driver which allows you to use a GUI to play backgammon. You can invoke this by running the `main` method of the class `SwingDriver`. Most of the play is straight-forward clicking and dragging. Once you are done with a move, simply click on the dice to end your turn.

We recommend that you write your own driver (likely without and human interface) to perform your experiments. Neither of the drivers we provide are optimal for running a large number of simulations.

3.2 Board

The `edu.rice.comp440.board` package provides all of the necessary support code to represent the board as well as the dice. There are many utility functions on the `Board` which allow you to get the number of pieces on the board, bar, as well as the number of pips left for each player.

3.3 Moves

The `Move` class represents a move in backgammon. We have provided a utility class `MovementFactory` which will generate all of the possible `Move`s from a given board setup. Thus, you can simply ask for the list of all valid moves and return the once which your player this is the best. In order to get the state of the board after a given `Move`, you can use the `Move.getCurrentBoard()` function.

3.4 Neural Network

All of the code to implement the neural network are provided in the `edu.rice.comp440.net` package. The neural network consists of a layer of `InputUnits`, followed by a number of layers of `HiddenUnits`, and finally a layer of `OutputUnits`. The most interesting one of these is the `HiddenUnit`, as `OutputUnit` simply extends it.

The `HiddenUnit` contains all of its weights in a `double[]` called `weights`. This is the set of weights which you will be learning. You can create a copy of a given network by using the `public NeuralNetwork(NeuralNetwork source)` method. Additionally, you can save a copy of your neural network to disk by using the `writeTo()` method, and recover it by using the `readFrom()` method.

In order to get the neural network to evaluate a given input, you should use the `getValue()` method, which returns a `double[]`, containing an entry for each output unit. In your case, there will only be one element in this array since you only have one output unit.

Lastly, we have provided a visualization tool for your neural network in the class `NeuralNetworkVisualizer`. This class will show you the weights in your neural network, the red weights being negative and green being positive. The intensity of the color indicates the magnitude of the weight.

3.5 Javadoc

All of the current Javadoc for this project is available online at

`http://www.owlnet.rice.edu/~comp440/pa2-javadoc/`.