Team Domination Association

Michael Cline (Project Leader)

Kyle Cibak(Stats/Scapegoat)

Philip Relic (Training Advisor)

Francis Mutuc (PseudoFrank)

March 21, 2011

Contents

Overview

The initial reinforcement learning agent was designed in such a way that the actions of our algorithm will be determined based on the resulting sum of a weighted feature vector. Since these features determine our move directly, they need to be comprehensive of as much of the information of the board that would dictate the "optimal" move as possible. As such, a feature vector was created for each column using the elements of the priority list used within the baseline. Next, the optimal weights were determined for the feature vectors. Initially, the learning program was run against our baseline until it was consistently winning. Then, the learning program was run against itself so that the opponent will continually be learning along with the agent. In this case, only the first player's final weights were recorded as the updated weights, but since both using the same player agent, presumably the first player will win the game half of the time. The value for each column was then calculated using the column's feature vector along with the most current weight vector. Choosing to drop the piece in a specific column was then as simple as choosing the column with the maximum value. Furthermore, if multiple columns shared this maximum value, a random number from an approximate Normal(5, 25/9) distribution was obtained and rounded to the nearest integer until an index corresponding to a column with the maximum value is obtained. This manner of randomization in our column choices will presumably prevent repeatedly losing due to deterministically selecting columns.

Data Structure

The main data structures that we used were for representing the board as well as for the feature vectors. For the board, a Vector<Vector<Player>>, which is an 11 x 10 matrix of Player objects, was used where the agent will place himself as ONE in the board, his opponent as TWO in the board, and all other spaces as EMPTY in the board. For the feature vectors, a Vector<Vector<Integer>>, which is an 11 x 17 matrix of Integer objects, was used where the player will place the number of instances of each of the specific 17 features for the first empty row in each of the 11 specific columns. The features that were checked include:

- the agent having a three threat horizontally
- the agent having a three threat vertically
- the agent having a three threat diagonally with an increasing slope
- the agent having a three threat diagonally with a decreasing slope
- the opponent having a three threat horizontally
- the opponent having a three threat vertically
- the opponent having a three threat diagonally with an increasing slope
- the opponent having a three threat diagonally with a decreasing slope
- the agent having a two threat horizontally
- the agent having a two threat vertically
- the agent having a two threat diagonally with an increasing slope
- the agent having a two threat diagonally with a decreasing slope
- the opponent having a two threat horizontally
- the opponent having a two threat vertically
- the opponent having a two threat diagonally with an increasing slope
- the opponent having a two threat diagonally with a decreasing slope
- the row above is a three threat for either the agent or the opponent

Using the getRowFromColumn() function, the first available row for the first column was determined, which was that column's location of interest. Windows were then created, if possible, within 3 spaces in each direction of the location of interest, and the numbers of occurrences of each of the above features were recorded. This process was then repeated for each of the eleven columns.

Training

First, the way in which the agent updates the weights for its feature vectors was determined. This process was done by taking the weights used in the just finished game, and adding or subtracting a fraction of the absolute value of the weight back to itself based on whether the agent won or lost the game. The fraction that was added was based on a discounting factor times the weight where the discounting factor was 1/100 times the learning rate raised to the power of the number or moves minus the current move ($lamda^{n-m}/100$). To determine whether to alter the weight, an indicator function was used to show which features were present for the chosen column of each move. Only the weights for the features that were present in determining to drop in that specific column were then altered using this function. Thus, the corresponding feature vectors for the columns for each move made by the agent had to be stored in an ArrayList. At the end of the game, these moves were then taken out of the ArrayList backwards so that the moves towards the end of the game have a larger influence on altering the weights then the moves at the beginning of the game.

Before any training was done, there were some factors of the weights that had to be addressed. Due to the large initial weights for the agent's three threats, columns containing these features will always be chosen, and since this will immediately lead to a win the weights for these features will always grow when they are chosen. Thus, a cap of 10 was placed on these values in order to avoid having these weights blow out of proportion. Also, there was a concern that the negative weight representing the feature for the three threat in the row above to outweigh the positive weight representing the features for the three threats in the current row. In order to solve this problem, the absolute value for the weight representing the features for the three threat in the row above was capped to be .1 less than the three threat so that the three threat will still be chosen most of the time even if it is also a column where the row above the first empty row also has a three threat.

Initially, in order to train the weights, the learning agent was trained against the baseline for one epoch where one epoch was 50 consecutive games. The learning agent beat the baseline consistently, with only a few losses due to the baseline obtaining "luckier" random starts than the learning agent. In order to avoid over fitting the weights to only the baseline, the learning agent was then trained against itself, an opponent who was also able to learn and adapt its strategy. This process of running the learning agent against itself was done for 40 epochs where one epoch was again 50 consecutive games. The decision to stop here was made as we saw small oscillations in the range of numbers that continued to get smaller and smaller giving us the indication of semi convergence.

Improving the Baseline

In order to improve on the baseline, the major change to the code was the implementation of reinforcement learning. This includes adding the feature vectors as discussed in the Data Structures section above as well as training the weight vectors that determine the optimal column as discussed in the Training section above. Other changes that were made include minimizing the locations that were being examined as well as optimizing the checking for columns in which the agent should not place the piece. The locations that were observed were minimized because now instead of viewing every location on the board, only windows around the first empty row in every column were examined. Also, the checking for the above row was now easier to implement because the agent was able to use the function for checking three threats for the current row, but instead pass it the location of the row directly above.

Weaknesses

Though we have implemented a useful 'Do-Not-Drop' feature, our foresight is limited. We only take into account the opponent's next move, and further foresight will improve our learner even more. We plan to implement a variable-length foresight in our next learner.

Another possible weakness of our learner is that our board checking often leads to dropping on the right side of possible limited-sized threats. When we find threats of length 2 and attempt to build upon them or block them, we take the right-most position that would fulfill that build or block. In a future learner, we will be more flexible in the way we examine the board to allow for more options when building or blocking threats.

Lastly, we plan to re-examine our starting strategy to determine whether it may be more beneficial to use our first several moves to attempt to set up an explicit structure. We were defeated a few times by teams that used their first few moves to setup a quick structure in an attempt to end the game quickly. We won most of the games that lasted longer than a few rounds, so we will look into the possibility of developing an early-game strategy to make our learner more effective in the short-run.

Strengths

Our learner has a few strengths that are worth mentioning. One of the strengths of our learner is our 'Do-Not-Drop' feature. We had a simple version of this in our baseline, but we improved upon this and made it a member of our feature vector. This feature looks ahead to determine possible moves that will either impede our progress or enable a better move for an opponent. This works by examining possible moves and determining which other features the hypothetical move will affect. The result of this is that we will avoid making moves that will give the opponent an easier win, or enable the blocking of one of our structures by the opponent.

Another strength of our learner is our random drop choice. Using the feature vector and our learned weights, we have determined which features often lead to victories. In our baseline, we chose this move randomly. In our learner, however, we choose the optimal move based on our feature vector and our learned weights. Choosing the optimal possible move to take as opposed to arbitrarily choosing a random move was a significant addition over our baseline. When there are no optimal moves to take, we still don't arbitrarily choose a random move, but take a random decision from a normal curve, as we have determined that taking moves positioned near the center of the board tend to result in better outcomes.

Algorithms

```
initFeatures()          // creates F, a vector of our feature vectors, having a total of
                        // numFeatures, where numFeatures is the number of feature vectors
                        // we have and initialize all of these vectors to 0
  1 for i = 1 to numFeatures
  2   Add a column vector of 0 to F
```

```
initWeights()           // initializes the weights of each feature vector to 0
  1 for i = 1 to numFeatures
  2   Weight of F[i] = 0
```

```
setFeatures()           // individually sets each vector as the specific scenario we are checking
  1 for i = 0 to 16
  2   if i = 0
  3     F[i] = threeThreatVertical(Us, i, row)
  4   if i = 1
  5     F[i] = threeThreatHorizontal(Us, i, row)
  6   if i = 2
  7     F[i] = threeThreatRightDiagonal(Us, i, row)
  8   if i = 3
  9     F[i] = threeThreatLeftDiagonal(Us, i, row)
 10   if i = 4
```

```
11        F[i] = threeThreatVertical(Them, i, row)
10    if i = 5
11        F[i] = threeThreatHorizontal(Them, i, row)
12    if i = 6
13        F[i] = threeThreatRightDiagonal(Them, i, row)
14    if i = 7
15        F[i] = threeThreatLeftDiagonal(Them, i, row)
16    if i = 8
17        F[i] = twoThreatVertical(Us, i, row)
18    if i = 9
19        F[i] = twoThreatHorizontal(Us, i, row)
20    if i = 10
21        F[i] = twoThreatRightDiagonal(Us, i, row)
22    if i = 11
23        F[i] = twoThreatLeftDiagonal(Us, i, row)
24    if i = 12
25        F[i] = twoThreatVertical(Them, i, row)
26    if i = 13
27        F[i] = twoThreatHorizontal(Them, i, row)
28    if i = 14
29        F[i] = twoThreatRightDiagonal(Them, i, row)
30    if i = 15
31        F[i] = twoThreatLeftDiagonal(Them, i, row)
32    if i = 16
```

33 F[i] = doNotPlace(i, row)

threeHorizontal(Player, col, row)

1 Check all available horizontal windows of four spaces for all possible combinations of location row, col

2 **If** a space and three of Player's our pieces is found in any of these windows, then return found. Otherwise, return not found

threeVertical(Player, col, row)

1 **if** row = 3

2 return not found

3 **for** each possible combination of creating three in a row vertically in col and row

4 Check all available vertical windows of four spaces for each column

5 If a space and three of Player's pieces is found in any of these windows, then return found. Otherwise, return not found

threeDiagRight(Player, col, row)

1 **for** each possible combination of creating three in a row diagonally to the right in col, row

2 Check if there exist three of Player's pieces in col and row

3 **If** three of a Player's piece is found, return found

Otherwise, return not found

threeDiagLeft(Player, col, row)

1 **for** each possible combination of creating three in a row diagonally to the left in col, row

2 Check if there exist three of Player's pieces in col and row

3 **If** three of a Player's piece is found, return found

Otherwise, return not found

twoHorizontal(Player, col, row)

1 **for** each possible combination of placing two of Player's pieces in a window spanning four horizontal spaces up to row

2 Check if there exist two of Player's pieces within the window

3 **If** two of a Player's piece is found, return found

Otherwise, return not found

twoVertical(Player, col, row)

1 **for** each possible combination of placing two of Player's pieces in a window spanning four vertical spaces up to row

2 Check if there exist two of Player's pieces within the window

3 **If** two of a Player's piece is found, return found

Otherwise, return not found

twoDiagRight(Player, col, row)

1 **for** each possible combination of placing two of Player's pieces in a window spanning four spaces diagonally to the right up to row

2 Check if there exist two of Player's pieces within the window

3 **If** two of a Player's piece is found, return found

Otherwise, return not found

twoDiagLeft(Player, col, row)

1 **for** each possible combination of placing two of Player's pieces in a window spanning four spaces diagonally to the left up to row

2 Check if there exist two of Player's pieces within the window

3 **If** two of a Player's piece is found, return found

Otherwise, return not found

doNotPlace(col, row) // determines if a given col should be taken away as a possible move

1 **for** i = 0 **to** 8

2 Check if there exists a threat of each kind for each Player in col and row+1

3 If a threat is found, then put that column in the feature vector. Otherwise, put column 0 in the feature vector.

weightUpdate // iterate through feature vectors log and update the weights based on the // results of the game played

1 **for** i = 0 **to** numFeatures

2 Weight for feature i += learning rate^{n-m} * indicator function * |weight for feature i| / 100

```
        // our learning rate is raised to n-m, where n is total num of moves in game, m is current
        // move of game because we want to have our weights change more for moves later in the
        // game
        // our indicator function is a flag that denotes if a feature is present in the current vector

readWeights()           // reads in the weights from our weights textfile
                        // no voodoo magic in here; it is as simple as it sounds

getMove()               // gets the sums of each column and makes a move in the column with the
                        // largest sum
                        // If the largest sum is less than 1, then randomly choose a move
    1   for i = 0 to numberOfCols
    2   for j = 0 to numFeatures
    3   Sum += F[i] * Weight[i]
    4   if Sum < 1
    5   DropOurPiece(colSumsArray, Sum) // random move
    6   return maxCol

DropOurPiece(colSumsArray, max)  // drops our piece in a random column with proper bounds
                                 // checking
```