Rocket Monkey Deathmobile

Jon McAlister and Daniel Wright

December 7, 2000

Overview

The overall design of our client did not deviate far from the standard client as expected for the milestone. We used a normal minmax seach with alhpa-beta pruning (with sorted node-expasion orderings on higher nodes but not lower), with a weighted linear evaluation function. We did deviate significantly in our training algorithm.

The Search

For the most part, we used a standard minmax search with alpha-beta pruning and sorted node-expansions for higher level nodes. At each non-leaf node, ExploreTree produces an array of children and iterates over the array recursively calling itself on each child (stoping for alpha-beta pruning). This makes it easy to sort the children - we simply call the evaluation function (the same one used for evaluating leaves) on each child and sort the results before iterating over the children.

An important consideration is when to sort the children. Calling the evaluation function on each child is expensive, and in many cases it is more expensive than the gain n improving alpha-beta pruning. Here, we noticed that there was far more gain higher up in the tree - since much larger branches are pruned. Thus, we only sort the children if the lookahead parameter passed to ExploreTree is above a certain threshold. Choosing the threshold was a simple matter of testing: we found that at 0 (i.e. always sort the children), of course the most nodes were pruned, but overall performance was actually barely faster than without sorting at all. We found the best performance (3 times faster!) with a threshold of 2.

Evaluation Features

CurrentSide this was a simple 1 or -1. We assumed that it was an advantage for it to be your turn with a given board position. Surprisingly, our training showed otherwise - around the middle of the game there was a substantial weight given to it not being your turn.

Number of pieces (greedy) interestingly, near the begining and middle of the game , this was weighted strongly negatively, due to the increased mobility implied by fewer pieces. Towards the end of the game, it was weighted positively again.

Position Counts we assigned a weight to each position adjacent to a corner (and the corners themselves), taking symmetry into acount. We only counted the adjacent squares if the corner was not already taked. Then we counted how many of each type each side had. Not surprisingly, corners weighted positively, and adjacent squares weighted negatively near the bigining of the game. Somewhat surprisingly, adjacent edge squares weighted positively from fairly early on - far earlier than adjacent diagonal squares. These all cobined to produce fairly powerful corner-play. Initially, we had a seperate weight for each position on the board (taking symmetry into account), but we decided that having so many weights was effecting the training and was slow to calculate (iterating over the whole board), so we reduced it to these combined with greedy (above).

Mobility initially, we calculated this by calling IsLegalMove on each square in the board. This is extremely slow (we were spending over 80% of our time computing it), so we sought a quick heuristic. In our finaly submission, we used a 'perimeter' function - we looked at the top and bottom pieces in each column, and the left and right pieces in each row. This was extremely quick to calculate, and gave a very useful heuristic - perhaps more useful than that given by the IsLegalMove calculation.

Avaliable Corners this was simply IsLegalMove on each corner

Stable Pieces this was the last feature we added - for each side in each direction, we counted how many consecutive pieces the player had, and how many they had in the adjacent row/column. This is a fairly inaccurate estimate of the stable pieces, but is quick to culculate and performed well. This feature improved edge play significantly - without it, the program would fight for a corner, even if all other pieces on the side were already lost. With it, it fought for the corners that mattered.

Weights and Game Phases

Given that we have a lot of cumputing resources available (there is a lot of unused computing power in Sweet Hall), we decided to be generous on the number of game phases. Initially, we had a different set of weights for each move, but decided that that was overkill, so we moved to one set of weights for every four moves. This performed very well - our training showed that the ideal weights do vary significantly through the game (for example, the weight for number of pieces is substantially negative in the begining, but becomes substantial and positive in the end.

Training

This is where we diverged substantially from the assignment. We did implement reinforcement learning method #2 as decribed in the handout, but found that it provided little additional benefiet, so we turned it off for our final submission. Instead, we used a form of batch supervised learning. The algorith was as follows:

- For each game phase i in descending order
  - generate sample boards in phase i
  - for each sample board, caclulate a reference score = the value obtained by searching n moves ahead (using the same ExploreTree function as in the game). Note that this is calculated using the previously caclulated weights for phases after i. (That is why we do each phase in descending order). For this to work, n must be larger than or equal to the number of moves in each phase. We used $n = 6$ with 4 moves per phase.
  - initialize the weights for pahse i randomly
  - repeatedly iterate over each sample board and train the weights towards the reference score, using stocastic gradient descent. Continue until convergence (or just for a long time, as we did. This was very quick computationally)

It is very important generate a good set of sample boards. We started by moving randomly until we reached the correct move in the game, but this generated weights that were not representative of games against skilled opponents. For example, in these boards the corners were usually taken early in the game, while in a realy game the corners usually remain empty long into the game. So, we generated a set of weights using random boards. Then, we re-ran the procedure, except this time, when generating each board we made a random move 10% of the time, and the rest of the time made the best move as determined by a 2-move look-ahead using our previous weights. After two or three iterations, this generated a set of boards all characteristic of normal game positions, and generated excellent weights.

It is interesting to note the similarities that this algorithm has to normal reinforcement learning - the primary difference is that the sample boards were re-used after iterating through all of them, where re-inforcement learning generates new sample boards each time.

Implementation Issues

Performance

We made some modifications to the stub code to improve preformance. Specifically, we changed CopyIntoBoard to use memcpy (much faster and safe since the board struct is fairly simple). We didn't use AllocateBoard at all. In fact, we used no dynamic memory allocation at all - all our memory use was on the stack which is much faster. Also, we turned on gcc's optimizations for a huge performance improvement.

Training

As described in the above section, we used a batch-like supervised learning system. It turned out that the majority of the computation time here was done generating sample boards and generating reference scores for them. Since these boards are relatively independant, we decided to parallelize the task amoung multiple machines - giving each machine a batch of boards to produce, and then having a central controlling machine do the gradient descent on all their combined boards.

So, we wrote a perl script that woulde iterate over each phase, and for that phase would spawn processes on each of the twelve firebird machines in the basement of sweet hall, having each generate 250 sample boards with reference scores. Thus, each phase had 3 000 sample boards with reference scores to train against. The entire training process took apporximately 2 hours.

The Final Weights

The columns in the following table are: (see *Evaluation Features* earlier for details)

Phase the game phase to which the weights apply. Multiply by 4 for the range on move numbers

Side 1 for white's turn, -1 for black's turn (this was the other way around in the code, the the weights were negated)

Pieces the number of $\frac{White\,Pieces - Black\,Pieces}{100}$

Diag., Corners, Adj. cnr, Diag cnr these are the different pieces around the corner - for each one, the feature is $\frac{White\,Pieces - Black\,Pieces}{8}$ except diag. which is divided by 4 (4-way symmetry of the board). The colums correspond to the possitions symmetric to: Diag: (1,1); Corners: (0,1); Adj. cnr: (0,2); Diag cnr: (1,2).

Mobility this is a cheap approximation. It is $\frac{White\,Pieces - Black\,Pieces}{40}$, where the pieces counted are top and bottom in each column, and left and right in each row. If a piece is the only onw in a column or row, it is double-counted.

Avl.cnr $\frac{Num\,corners\,white\,can\,take - Num\,corners\,red\,can\,take}{4}$

Stable cheap approximation. Number or consecutive pieces from each corner along the side, and in the adjacent row/col. $\frac{White\,Pieces - Black\,Pieces}{60}$

Phase	Side	Pieces	Diag.	Corners	Adj. cnr	Diag cnr	Mobility	Avl. cnr	Stable
1	0.03	17.66	1.00	1.00	1.00	1.00	-17.91	1.00	1.00
2	-2.19	-2.01	-0.80	1.00	1.00	-42.42	-16.58	4.81	1.00
3	-5.41	81.75	-10.83	13.31	-41.23	-97.94	60.37	17.91	4.28
4	-5.02	34.36	-9.66	5.61	-48.45	-95.82	61.83	20.71	2.29
5	-5.50	6.69	-10.52	36.72	-32.78	-101.66	57.35	16.89	11.66
6	-5.78	-9.93	-10.42	93.04	-20.61	-91.93	57.08	21.25	36.36
7	-5.36	-16.55	-12.61	106.95	-3.41	-81.94	55.67	28.22	50.73
8	-5.53	-13.77	-10.18	118.10	0.32	-70.66	56.81	32.71	60.12
9	-5.38	-19.16	-14.00	96.58	7.02	-77.10	56.39	28.68	76.88
10	-5.21	-17.51	-12.17	83.88	12.10	-76.48	57.11	29.29	90.31
11	-4.95	-15.61	-14.27	78.09	10.36	-58.66	56.15	34.19	84.90
12	-4.48	-14.33	-12.73	65.75	10.77	-68.22	54.35	29.62	89.06
13	-4.21	-14.85	-9.78	68.51	10.85	-44.65	49.11	37.52	74.10
14	-3.36	-10.95	-10.05	75.67	9.54	-46.19	46.72	29.24	43.56
15	-2.49	-11.33	-7.58	67.84	7.64	-28.49	42.48	32.53	43.09
16	-2.01	-7.34	-5.07	60.63	7.43	-28.57	41.60	30.45	37.52
17	-1.24	-4.10	-3.18	58.58	7.37	-23.50	39.09	25.05	30.09
18	-0.59	1.05	-1.58	46.80	8.07	-15.77	38.74	23.37	27.02
19	0.18	9.49	-0.45	32.90	7.31	-14.81	37.59	20.00	34.12
20	0.88	13.11	1.01	21.72	10.68	-7.51	38.74	16.23	30.84
21	1.36	29.47	2.74	11.16	12.97	-8.28	41.66	16.00	31.58
22	2.50	41.36	3.99	3.91	14.24	-12.42	44.93	11.12	19.13
23	3.68	81.31	5.82	5.86	11.98	-6.38	44.71	10.84	8.26