A Lisp Connect-Four Player
Final Project for CSCI 4510

Jesse Rusak

April 20, 2005

Contents

1 Introduction

This project is an implementation of an alpha-beta searching connect-four player, written in common Lisp. An interface is supplied to allow a human to play against the computer program. This program uses a simple heuristic based on examining every possible line on the board, and is surprisingly effective. A discussion of the search depth used and the resulting playing ability follows. The validity of the heuristic function was also investigated by having the computer play against itself. The manual for this project can be found in Appendix A.

2 Outline of Data Structures and Algorithms

Although written in Lisp, the game of connect-four is intrinsically imperative. The board is most easily represented as an array that is modified, and any evaluation of a line's value is similarly awkward to represent in a recursive fashion. Nevertheless, every effort was made to hide this nature behind a more Lisp-like code interface.

States are stored as a list of a two-dimensional array of symbols representing the board and an indication of whose turn is next. The `alpha-beta` routine is the driving function for the search that takes place for each move. It takes in lambda forms to generate and evaluate future states. The search itself is a fairly straight-forward recursive implementation of a depth-first-search with alpha-beta pruning. As such, each invocation of `alpha-beta` keeps a cutoff value that is updated based on the values found and passed to subsequent invocations of the routine.

The heuristic evaluation function is based on assigning a score to every possible line of length four in the grid. The score given is determined by the current pieces in that line. To facilitate this, the function `map-each-line` was created, which applies some function to every line in the board and returns a list of all the results. The state evaluation function calls `map-each-line`, passing a line evaluation function, and sums the resulting list.

The function used to generate future states is similarly simple. It attempts to place a piece in each column, returning a list of the states generated by all successful moves. There is no detection of already-seen states.

3 Search Depth

Initially, a depth of three was chosen. This was quickly found to be insufficient. The simple opening move of placing two pieces on the bottom level side-by-side defeats a depth-three search. The computer must be able to see that if it does not place its own piece on one side of these two, the human will make a line of length three that can be expanded to four on either side, resulting in a human win. With only three moves of foresight, the computer cannot see this eventuality. There are many other similar situations. With four moves of look-ahead, though, it seems much more competent.

4 Heuristic Details

The heuristic used is quite simple. Every possible line of length four is considered. Lines that have pieces of both colour in them are ignored, as they are of no use to either side. Empty lines are also ignored, since they are neither beneficial or detrimental. The remaining lines are evaluated based on the number of pieces in them. Points are added (and subtracted for and opponent's pieces) based on these counts. A full line is worth some very large number of points, and lines with one, two, or three pieces in them are worth a smaller amount. The question that naturally arises from such a heuristic is how much the incomplete lines are worth.

4.1 Heuristic Parameter Investigation

When originally creating this heuristic, values of one point for a single piece, two points for two pieces and three points for three pieces was chosen. In practice, this has turned out to work quite well. This program has only lost a couple of times to a human out of dozens of games.[1] To be fair, only myself, my friends, and my roomates have played against it. All of us have expressed frustration at the seemingly omniscient opponent, and several times I have disbelieved it when it declared it was guaranteed a win; of course, it was correct. Despite this success, I felt some validation of these arbitrary point values was needed.

In order to do so, a round-robin style tournament was arranged between eleven versions of this algorithm with differing score values for incomplete lines.[2] The code for this can be found in `contest.lisp`. This tournament showed that the original selection of values did well against its peers. Curious at this somewhat surprising result, the top three winners of this tournament were taken and their values were adjusted slightly, resulting in twelve evaluation functions. These were then played against each-other. This second tournament showed that another group, those with around four points for two pieces and nine points for three pieces, was better. A final tournament of the winners from each of the previous two rounds suggested that, in fact, the winners of each tournament were of similar strength. Full results can be found in Appendix B.

While this gives some confidence that the numbers chosen are not a bad choice, I think that the strange inconsistencies in the results of these tournaments show that it is hard to evaluate these values by playing this heuristic against itself. With a deterministic algorithm such as this one, the results of such a tournament are highly suspect, especially with only 6 to 12 players. Unfortunately, due to time constraints, it was not possible to further examine these values. Further work with larger numbers of computer players and

[1] These wins were achieved by carefully considering the weaknesses of the algorithm used and exploiting it.

[2] It was sufficient to test differing values of incomplete lines with 2 and 3 pieces, leaving the points for a single piece set at 1, and the points for 4 being effectively infinite. The numbers chosen for these parameters are the possible combinations of typical functions such as $sqrt(x)$ and x^2.

different heuristics would be beneficial in determining the effectiveness of this heuristic.

5 Resulting Behaviour

When playing against this opponent, it quickly becomes clear that it has a predisposition to build tall towers. This can be traced easily to the heuristic function's inner-workings. By building on top of a tower, the computer creates many lines with an additional piece that were previously empty. Consider a single tower in the middle of the board. The piece on the top of this column contributes to the score of four previously-unoccupied horizontal lines, as well as up to eight diagonal and three vertical lines. This is a huge sum of points, and significantly larger than the benefit from, say, building near the base of the tower to make a single line of length two or three.

Taking into account the simple explanation for much of this program's behaviour, and the earlier discussion about the inconsistent effect of the heuristic's parameters on the effectiveness of the program, one must ask why it seems to be able to win against humans. I would like to hypothesize that almost any moves would be suitable until the search detects a possibility of winning or losing. Simply being able to look four moves in advance allows the computer player to see most of the traps that typically end a game of connect-four. Future investigation could try a random-placement strategy and compare its effectiveness to the current program.

6 Complexity Analysis

Evaluating a single board position involves looking at each possible line. There are $O((w-l)*(h-l))$ lines in a given board, where w and h are the width and height of the board, and l is the length of a winning line (four, in this case, but it could be changed). The pieces in each line are each considered separately, giving $O(l*(w-l)*(h-l))$ spots to consider. Finally, there are $O(w)$ possible moves generated for each state at each level, d, of search depth. This gives $O(w^d)$ total states, resulting in a worst-case search complexity of $O(w^d*l*(w-l)*(h-l))$. Alpha-beta pruning improves this for the average-case, but it does not change the worst-case behaviour of evaluating every line for every possible move.

7 Conclusion

Although the selected heuristic is quite simple, this alpha-beta searching computer opponent seems enjoyable to play and difficult to beat. There is some question as to how much difference the heuristic itself makes, so long as it correctly identifies end-game situations. Future work could include adding non-determinism to the algorithm to enhance re-playability, and further investigation into the effectiveness of the chosen heuristic.

8 Appendix A — Manual

8.1 Files

There are two main source files, `ab.lisp`, which implements a generic alpha-beta search and `connect.lisp`, which implements the connect-four specific functions. There is also a file `contest.lisp`, which implements the round-robin testing mode for different heuristics and `run-game.lisp`, which acts as a wrapper for running a single game.

8.2 Running

To run this, you will need common lisp. Unfortunately, the version on `torch` does not support long argument lists, and so is not usable since this code applies functions like `+` to large lists. To run a game, simply load `run-game.lisp`. Alternatively, you may follow the steps in that file of loading `ab.lisp` and `connect.lisp` and executing the `run-game` function.

8.2.1 Playing Remotely

At the moment, you can play this game remotely by logging into my home desktop with `ssh cf@jrusa6077.resnet.dal.ca`. The password is `lisprocks`. This is only available when the computer is on, which is much, but not all, of the time. This only works from within the Dalhousie network (eg. from `torch`).

8.3 Automated Games

You can also run a single computer-on-computer game with the function `auto-game`. This function takes in two heuristic evaluation functions: one for each player in the game. These functions can be created with `get-eval-state`, which returns a heuristic evaluation function when given a line evaluation function. The function `get-eval-line` returns a line evaluation function given scores for incomplete lines of length 2 and 3, the parameters varied in the round-robin tests discussed above.

8.4 Writing New Heuristics

A heuristic function simply takes in a state (a list with the board and the next player) and returns a numeric score for that state. Any new heuristic functions can take advantage of the `map-each-*-line` functions. The functions allow the programmer to apply some function to every horizontal, vertical, or diagonal line. The function used by the supplied heuristic, `map-each-line`, performs the given operation on all possible lines in the board. All of these functions collect their results into a list, which is returned.

A new heuristic can then be passed to the `auto-game` function to run a game between it and another computer player, or it can be inserted into the `run-game` function and played against directly.

8.5 Modifying Game Parameters

The board size and goal line length can be easily modified by changing or overriding the values at the beginning of `connect.lisp`. Please note that the built-in heuristic assumes a line length of 4 is the goal, but will work with any board size. The search depth can also be adjusted here.

8.6 Further Information

Please see the in-code comments for further details.

9 Appendix B — Test Results

Round 1

Value for 2	Value for 3	Wins
1.4	1.7	3
1.4	3	7
1.4	9	6
1.4	27	4
2	3	10
2	9	5
2	27	4
4	9	8
4	27	8
8	9	6
8	27	7

Round 2

Value for 2	Value for 3	Wins
1.5	2.5	5
1.5	3.5	2
2.5	2.5	8
2.5	3.5	7
3.5	8.5	14
3.5	9.5	14
4.5	8.5	13
4.5	9.5	15
3.5	26.5	8
3.5	27.5	8
4.5	26.5	4
4.5	27.5	4

Round 3

Value for 2	Value for 3	Wins
2	3	6
4	9	5
4	27	3
4.5	8.5	5
4.5	9.5	6
3.5	9.5	5
3.5	8.5	3

10 Appendix C — Sample Session

```
[1]> (load "ab.lisp")
T
[2]> (load "connect.lisp")
T
[3]> (run-game)

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 4

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 4."
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | | |X| | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 3

| | | | | | | |
| | | | | | | |
| | | | | | | |
```

```
| | | | | | | |
| | | | |O| | |
| | | |X|X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 2."
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 4

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |X| | |
| | | | |O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 4."
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | | |X| | |
| | | | |O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 2

| | | | | | | |
| | | | | | | |
| | | | |O| | |
```

```
| | | | |X| | |
| | |X| |O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 2."
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | |O| |X| | |
| | |X| |O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 3

| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | |O| |X| | |
| | |X|X|O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 2."
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| | |O| |X| | |
| | |X|X|O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 2

| | | | | | | |
| | |X| | | | |
| | |O| |O| | |
```

```
| | |O| |X| | |
| | |X|X|O| | |
| | |O|X|X| | |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 5."
| | | | | | | |
| | |X| | | | |
| | |O| |O| | |
| | |O| |X| | |
| | |X|X|O| | |
| | |O|X|X|O| |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 3

| | | | | | | |
| | |X| | | | |
| | |O| |O| | |
| | |O|X|X| | |
| | |X|X|O| | |
| | |O|X|X|O| |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 3."
| | | | | | | |
| | |X| | | | |
| | |O|O|O| | |
| | |O|X|X| | |
| | |X|X|O| | |
| | |O|X|X|O| |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 0

| | | | | | | |
| | |X| | | | |
| | |O|O|O| | |
```

```
| | |O|X|X| | |
| | |X|X|O| | |
|X| |O|X|X|O| |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 3."
| | | | | | | |
| | |X|O| | | |
| | |O|O|O| | |
| | |O|X|X| | |
| | |X|X|O| | |
|X| |O|X|X|O| |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 0

| | | | | | | |
| | |X|O| | | |
| | |O|O|O| | |
| | |O|X|X| | |
|X| |X|X|O| | |
|X| |O|X|X|O| |
|0|1|2|3|4|5|6|

"My Turn"

"I went in 0."
| | | | | | | |
| | |X|O| | | |
| | |O|O|O| | |
|O| |O|X|X| | |
|X| |X|X|O| | |
|X| |O|X|X|O| |
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 6

| | | | | | | |
| | |X|O| | | |
| | |O|O|O| | |
```

```
|O| |O|X|X| | |
|X| |X|X|O| | |
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 4."
"Looks like I'm ahead."
| | | | | | | |
| | |X|O|O| | |
| | |O|O|O| | |
|O| |O|X|X| | |
|X| |X|X|O| | |
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 6

| | | | | | | |
| | |X|O|O| | |
| | |O|O|O| | |
|O| |O|X|X| | |
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 2."
"Looks like I'm ahead."
| | |O| | | | |
| | |X|O|O| | |
| | |O|O|O| | |
|O| |O|X|X| | |
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 6

| | |O| | | | |
```

```
| | |X|O|O| | |
| | |O|O|O| | |
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 6."
"Looks like I'm ahead."
| | |O| | | | |
| | |X|O|O| | |
| | |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 6

| | |O| | | | |
| | |X|O|O| |X|
| | |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 4."
"Looks like I'm ahead."
| | |O| |O| | |
| | |X|O|O| |X|
| | |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 6
```

```
| | |O| |O| |X|
| | |X|O|O| |X|
| | |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 0."
"Looks like I'm ahead."
| | |O| |O| |X|
| | |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 0

| | |O| |O| |X|
|X| |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 3."
"Oh, no. That's not good!"
| | |O|O|O| |X|
|X| |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"
```

```
"Where do you want to go?" 0

|X| |O|O|O| |X|
|X| |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X| |O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"I went in 1."
"Oh, no. That's not good!"
|X| |O|O|O| |X|
|X| |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X| |X|X|O| |X|
|X|O|O|X|X|O|X|
|0|1|2|3|4|5|6|

"Your Turn"

"Where do you want to go?" 1

|X| |O|O|O| |X|
|X| |X|O|O| |X|
|O| |O|O|O| |O|
|O| |O|X|X| |X|
|X|X|X|X|O| |X|
|X|O|O|X|X|O|X|
|0|1|2|3|4|5|6|

"My Turn"

"You win!"
```