Symbolic AI's for Generalised Connect Four

Bram Pulles – S1015194

October 6, 2022

Contents

1 Introduction

Connect Four is a widely known game, most of us probably played this as a kid for fun. What most people do not know is that Connect Four is actually strongly solved, i.e. for every state the best move is known [4]. In this assignment we are tasked with creating various search programs for Generalised Connect Four. In this version the board is of arbitrary width and height, and the number, N, of consecutive stones needed for a win is unspecified.

I have implemented a negamax AI, which is equivalent to minmax, an alpha-beta pruning AI and an insane AI which contains a lot of extra's such as move ordening, a transposition table, iterative deepening and MTD(f).

2 Evaluation Space

Before we get into how all of the different algorithms work and how they are implemented we have to talk about evaluations. I have made a generalised evaluation space which is suited for the algorithms that I implemented.

```
 <- MIN                                          0                                               MAX ->
 <----------------------------------------------------------------------------------------------------->
    |   LOSS     | |   HEURISTIC    | |         DRAW          | |    HEURISTIC    | |     WIN    |
    |           / \                / \           |           / \                 / \           |
  L in 0   L in ML Hl at HL   Hl at 0 D in ML   D in 0   D in ML  Hw at 0   Hw at HL W in ML   W in 0
    |        |         |          |      |       |         |         |          |        |       |
-2ML-HL-2 -ML-HL-2 -ML-HL-1    -ML-1   -ML       0         ML       ML+1     ML+HL+1  ML+HL+2   2ML+HL+2
```

The evaluation space is shown above. Every evaluation of a board state is represented by a single integer. This makes it extremely fast to compare different evaluations.

In the evaluation space we can see on the left side negative integers going to $-\infty$, while on the right side we have positive integers going to ∞. There are a lot of abbreviations in the diagram: L stands for Loss, D for Draw, W for Win, Hl for Heuristic loss and Hw for Heuristic win. We also have HL for Heuristic Limit, a heuristic value v always satisfies $v \in [0, \text{HL}]$, and ML for Movecount Limit which is the maximum number of moves that can be made, i.e. the width of the board times the height of the board.

2.1 Win, Loss, Draw

The evaluation space is carefully constructed such that desired outcomes have a higher value than less desired outcomes. The best possible situation is a win in zero moves, while the worst possible situation is a loss in zero moves.

Within the winning space we prefer evaluations which lead us to a win faster than other winning evaluations. The worst winning evaluation would be a win in ML moves.

Within the losing space we prefer evaluations which lead us to a loss slower than other losing evaluations. The best losing evaluation would be a loss in ML moves. This makes intuitive sense when the opponent might not play perfectly. By trying to extent the game the chances of an imperfect player making a mistake increases which could change the game from losing to either a draw or winning situation.

From the information given so far about the evaluation space we observe the most important property of all the evaluations, namely that all evaluations are zero-sum. This means that an advantage for one player is an equivalent loss for the other player [5]. In other words, we have that any evaluation e_{player_1} for player 1 and e_{player_2} for player 2 at a certain node adds up to zero, i.e.

$$e_{\text{player}_1} + e_{\text{player}_2} = 0$$

$$e_{\text{player}_1} = -e_{\text{player}_2}$$

This is extremely useful as we can now just negate an evaluation to change the perspective of the evaluation to the other player. This property is on the basis of negamax and often given as a requirement for minmax [1].

Now that we have taken a look at losses and wins and that we have discussed the zero-sum property it is time to take a look at draws. Draws are usually considered to be neutral in search programs and an equal outcome for both players. However, this does not reflect differences in player strength. For example a better player would rather not play a draw against a worse player, while for the worse player this would constitute a good result.

I have implemented negamax and various variants of negamax. Since negamax, when no heuristic is used, gives a perfect result it is safe to assume that the opponent will not be stronger. Because of this we can prefer later draws for the negamax player and earlier draws for the opponent. Considering this we give the positive draw space to the negamax player and the negative draw space to the opponent.

2.2 Heuristic

Apart from exact evaluations we can also have a heuristic evaluation. This represents an approximation of the outcome when to exact evaluation is unknown. In my program implementation I have an interface from which various heuristics can be defined. For every heuristic the zero-sum property must hold and the value must be between zero and a given limit. This limit is used in the evaluation space to create enough width for all the possible heuristic values.

In the evaluation space we have two gaps for heuristic values. One on the positive side and one on the negative side. This way we can negate the heuristic value to change to the other player's perspective.

Note that the heuristic space is outside of the draw space. This means that heuristic values which seem positive but can turn out into a loss are preferred over definite draws. If we have high confidence in our heuristic this is fine. If not we can also swap the heuristic and draw spaces. This is just a design choice that I have made.

3 Heuristics

My program contains two heuristics the given heuristic and a new heuristic.

3.1 Given Heuristic

Let me start by stating that the given heuristic is does not satisfy the zero-sum property, and this is quite the problem. So the given heuristic provides the number of consecutive stones the current player on turn has. This however, does not say anything about the relative position compared to the other player.

Imagine that player 1 has three consecutive stones, then we have $e_{\text{player}_1} = 3$. Lets also say that we need four consecutive stones for a win, so $\text{HL}+1 = N = 4$. There is no way that we can use this value to say anything about e_{player_2}. For example, we could say $e_{\text{player}_2} = -e_{\text{player}_1} = -3$. This would constitute the worst heuristic evaluation for player 2, however for all we know player 2 also has 3 in a row, therefore this is not a good translation. Another example could be to say $e_{\text{player}_2} = \text{HL} - e_{\text{player}_1} = 3 - 3 = 0$. This would be the best heuristic value for player 2, but maybe player 2 does not have any consecutive stones anywhere!

The problem with the given heuristic is that it only gives information about one of the two players. This makes it that we could never change this value in a meaningful way to represent a heuristic evaluation for the other player.

How would this be solved? In chess engines for example, a heuristic value often constitutes the *difference* in points given by the pieces on the board, where every piece is assigned a static value[1]. If we would do this in Connect Four, we could take the maximum number of consecutive for both players and take the difference. This does satisfy the zero-sum property. For example player 1 has 3 consecutive stones and player 2 only 1, then the heuristic value for player 1 is $e_{\text{player}_1} = 3 - 1 = 2$ and for player 2 is $e_{\text{player}_2} = -e_{\text{player}_1} = -2$. Now if player 2 gets more consecutive stones the heuristic value reflects the advantage and the loss for both player 1 and player 2.

I have implemented the heuristic as given in the assignment. This means that when this heuristic is used the outcome is incorrect. However, a fix would be trivial, namely by calling the heuristic twice and taking the difference. Albeit this would be extremely inefficient.

3.2 Nop Heuristic

I have also created a simple heuristic of my own so I can work with a depth bounded search. This heuristic always returns 0[2]. This way we prefer to go through a path which has an undefined outcome over a definite loss, but we prefer a definite draw[3] and definite win over an undefined outcome. This heuristic is meant as a filler to make depth bounded search work.

4 Negamax AI

4.1 Algorithm

The first algorithm that I have implemented is negamax, this is equivalent to minmax but more elegant. The negamax algorithm makes use of the fact that minmax applies to zero-sum games and the fact that $max(a, b) = -min(-a, -b)$. Making use of these properties we can get rid of the case distinction in the minmax algorithm. The negamax algorithm is shown in 1.

[1] I am oversimplifying a lot here, see `https://www.chessprogramming.org/Score`.

[2] In the code this is `Eval(const = Eval.UNDEFINED)`.

[3] Remember that the algorithm always has the positive draw space.

Algorithm 1 Negamax

```
Require: node, depth, color, timeout
Ensure: node evaluation
function NEGAMAX(node, depth, color, timeout)
    if timeout is reached then
        return undefined
    end if
    if node is over then
        return color × evaluation of node
    end if
    if depth = 0 then
        return color × heuristic evaluation of node
    end if
    value := −∞
    for child of node do
        value := max(value, −NEGAMAX(child, depth − 1, −color, timeout))
    end for
    return value
end function
```

The algorithm works pretty straightforward. The color represents which person is on the move[4]. Note that I have added a timeout which when reached stops the execution of the algorithm, this is very useful when benchmarking or live interacting with the algorithm. The algorithm which calls the negamax function is a bit different and is shown in 2.

4.2 Complexity

5 Alpha-beta AI

5.1 Algorithm

The second algorithm that I have implemented is alpha-beta pruning in the negamax algorithm. This is very similar to negamax, we just add an alpha-beta window. The alpha value represents the value that the current player is assured of, the beta value represents the value that the opponent is assured of. Whenever alpha > beta we know that the opponent would never let us go down this path as they already have a better move somewhere else, so we can prune that branch. With alpha-beta pruning we still get the same result as with pure negamax. Note that $[\text{alpha}, \text{beta}]$ for player 1 is the same as $[-\text{beta}, -\text{alpha}]$ for player 2, this is repeatedly used in negamax. Negamax with alpha-beta pruning is shown in 3, the move generation is shown in 4.

One peculiarity in my alpha-beta pruning implementation is that we do not want to have a cutoff when either the alpha or the beta value is undefined. This would lead to too early cutoffs and incorrect results.

[4] In my implementation this is called `rootplayer`.

Algorithm 2 Negamax move

Require: node, depth, timeout
Ensure: best moves, node evaluation

```
function MOVE(node, depth, timeout)
    bestvalue := −∞
    bestmoves := ∅
    for move at node do
        child := play move at node
        value := −NEGAMAX(child, depth − 1, −1, timeout)
        if timeout is reached then
            return undefined
        end if
        if value > bestvalue then
            bestvalue := value
            bestmoves := move
        else if value = bestvalue then
            bestmoves := bestmoves + move
        end if
    end for
    return bestmoves, bestvalue
end function
```

Algorithm 3 Negamax with alpha-beta pruning

Require: node, depth, color, timeout
Ensure: node evaluation

```
function NEGAMAX(node, depth, color, alpha, beta, timeout)
    if timeout is reached then
        return undefined
    end if
    if node is over then
        return color × evaluation of node
    end if
    if depth = 0 then
        return color × heuristic evaluation of node
    end if
    value := −∞
    for child of node do
        value := max(value,
                −NEGAMAX(child, depth−1, −color, −beta, −alpha, timeout))
        if alpha >= beta and alpha ≠ undefined and beta ≠ undefined then
            break
        end if
    end for
    return value
end function
```

Algorithm 4 Negamax with alpha-beta pruning move

Require: node, depth, timeout
Ensure: best moves, node evaluation

```
function MOVE(node, depth, timeout)
    alpha := −∞
    beta := ∞
    bestvalue := −∞
    bestmoves := ∅
    for move at node do
        child := play move at node
        value := −NEGAMAX(child, depth − 1, −1, −beta, −alpha, timeout)
        if timeout is reached then
            return undefined
        end if
        if value > bestvalue then
            bestvalue := value
            bestmoves := move
        else if value = bestvalue then
            bestmoves := bestmoves + move
        end if
    end for
    return bestmoves, bestvalue
end function
```

5.2 Complexity

6 Insane AI

6.1 Algorithm

The third algorithm that I have implemented is an extension of the alpha-beta pruning negamax algorithm with move ordening, a transposition table, iterative deepening and MTD(f).

Move ordening

The first change is move ordening. Optimally, the alpha-beta pruning algorithm is run with the best move first every time, this would cause the maximum number of cutoffs. Ofcourse, we do not know the best move at first, but an approximation can already greatly increase the speed of the algorithm. To this end I use a simply move ordening algorithm which prefers columns in the middle of the board over columns towards the edges of the board. This can be easily achieved by sorted the moves according to their value given by the formula $w/2 - c$, where w is the width of the board and $c \in [0, w)$ the number of a column representing a move.

Transposition table

The second change is the addition of a transposition table, or TT for short. A TT is a memory construct which saves evaluations of states that we have visited.

Whenever we visit a state a second time, we can take the evaluation from the TT and prune the branch.

The implementation of such a table has to be fast. To this end I made a primitive hashtable without collision resolution. This table is implemented as an array with a prime number sized length. It has two functions, one for storing a value and a flag and one for retrieving the value and flag, the pseudocode can be seen in 5.

Algorithm 5 Table put and get function

```
function PUT(key, value, flag)
    index := key % table size
    table[index] = (key, value, flag)
end function
function GET(key)
    index := key % table size
    entry := table[index]
    if entry ≠ null then
        if entry.key = key then
            return entry
        end if
    end if
    return null
end function
```

The key generation for the transposition table should also be incredibly fast. In order to achieve this I have made a representation of the board in a ternary number. This ternary number uniquely identifies the board. The number functions as a flattened two dimensional array. The digits in the ternary number represent the state of every cell, a 0 for empty, 1 for a stone of player 1 and 2 for a stone of player 2. For a move made in a colomn $c \in [0, w)$ I can now update the key with $key = key + (x + w \cdot y)^3 * (onturn + 1)$ with $onturn \in [0, 1]$ and w the width of the board. This achieves a constant time key generation.

When we add the transposition table to the alpha-beta pruning negamax algorithm we have another complication. Some values may not be an exact evaluation because of alpha-beta cutoffs. Luckily for us these values can still provide a lowerbound or an upperbound. For example, when the best value we find in a negamax iteration is bigger than beta, we know a cutoff has occured, and we know that the value that we found is a lowerbound on the actual value of the node. Similary when the best value we find is smaller than the original alpha value we know that this is an upperbound for the actual value of the node.

Using all of the information we have about the transposition table we can now add this to our algorithm. The new algorithm is shown in 6 [3]. Note that the move generation algorithm is the same as in 4.

Iterative deepening

The third addition is iterative deepening. This combines the best of depth first search, low space complexity, with the best of breadth first search, optimality. Iterative deepening is relatively simple, the algorithm can be seen in 7. Using the iterative deepening algorithm is now as simple as replacing the call in algorithm

Algorithm 6 Negamax with alpha-beta pruning, move ordening and a TT

Require: node, depth, color, timeout, table

Ensure: node evaluation

```
function NEGAMAX(node, depth, color, alpha, beta, timeout)
    if timeout is reached then
        return undefined
    end if
    alpha_original := alpha
    entry := table.get(node.key)
    if entry ≠ null then
        if entry.flag = lowerbound then
            alpha := max(alpha, entry.value)
        else if entry.flag = upperbound then
            beta := min(beta, entry.value)
        else
            return entry.value
        end if
    end if
    if node is over then
        return color × evaluation of node
    end if
    if depth = 0 then
        return color × heuristic evaluation of node
    end if
    value := −∞
    moves := sorted moves from node
    for move in moves do
        child := move at node
        value := max(value,
                −NEGAMAX(child, depth−1, −color, −beta, −alpha, timeout))
        if alpha >= beta and alpha ≠ undefined and beta ≠ undefined then
            break
        end if
    end for
    if value ≠ undefined then
        if value ≤ alpha_original then
            flag := upperbound
        else if value ≥ beta then
            flag := lowerbound
        else
            flag := exact
        end if
        table.put(node.key, value, flag)
    end if
    return value
end function
```

4 at line 8 with the iterative deepening function. Note that instead of stopping when we do not have an undefined value anymore, we can also check whether this is a heuristic value and continue until we find an exact value.

Algorithm 7 Iterative deepening

Require: node, maxdepth, color, timeout, table
Ensure: node evaluation

```
function ITERDEEP(node, maxdepth, color, alpha, beta, timeout)
    value := undefined
    for depth in [0, maxdepth] do
        if timeout is reached then
            return value
        end if
        value := NEGAMAX(child, depth, −1, alpha, beta, timeout)
        if value ≠ undefined then
            return value
        end if
    end for
end function
```

MTD(f)

At last I have added MTD(f) a type of zero-window search. This search iteratevely executes negamax with a window of just one big, causing a lot of cutoffs. Depending on the return value we can conclude whether the actual value is below or above our initial guess. This way we narrow down our guess until we find the correct one. The algorithm for MTD(f) can be seen in 8 [2].

Algorithm 8 MTD(f)

Require: node, depth, color, timeout, table
Ensure: node evaluation

```
function MTDF(node, depth, color, alpha, beta, timeout)
    lowerbound := alpha
    upperbound := beta
    guess := 0
    while lowerbound < upperbound do
        beta := max(g, lowerbound + 1)
        guess := NEGAMAX(child, depth, −1, beta - 1, beta, timeout)
        if guess < beta then
            upperbound := guess
        else
            lowerbound := guess
        end if
    end while
end function
```

6.2 Complexity

7 Benchmarks

7.1 Negamax AI

test set	total	timeouts	mean time	max time	mean count	max count	count per sec
30-07	30	2	0.253901	9.589689	28040	1051029	110008
20-07	30	30	0	0	0	0	0
30-14	30	1	0.719904	8.746326	67918	832406	96602
20-14	30	30	0	0	0	0	0

Table 1: Negamax / minimax

7.2 Alpha-beta AI

test set	total	timeouts	mean time	max time	mean count	max count	count per sec
30-07	30	0	0.031056	0.809639	2909	73283	90247
20-07	30	27	6.843755	9.205496	560175	824355	86068
30-14	30	0	0.058018	0.635897	4228	46229	79448
20-14	30	29	2.546612	2.546612	202736	202736	79610

Table 2: Alpha beta pruning

7.3 Insane AI

test set	total	timeouts	mean time	max time	mean count	max count	count per sec
30-07	30	1	0.095417	2.011071	197	146619	38598
20-07	30	7	0.108103	2.660797	278	216104	52110
30-14	30	1	0.222108	5.05804	6886	323909	52655
20-14	30	4	0.921285	6.190709	61932	468116	67406

Table 3: Insane AI with iterative deepening

7.4 Comparision

8 Conclusion

9 Discussion

10 References

[1] Wikipedia contributors. Minimax — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Minimax&oldid=`

test set	total	timeouts	mean time	max time	mean count	max count	count per sec
30-07	30	0	0.13946	3.717605	1519	246017	45682
20-07	30	30	0	0	0	0	0
30-14	30	0	0.183881	0.819843	4072	50014	39368
20-14	30	28	8.018537	8.018537	670090	670090	81651

Table 4: Insane AI with MTD(f)

test set	total	timeouts	mean time	max time	mean count	max count	count per sec
30-07	30	1	0.095708	4.261424	377	339069	55753
20-07	30	8	0.118574	4.366139	660	335230	59456
30-14	30	1	0.415391	9.557776	20226	664495	61041
20-14	30	6	2.968389	9.57551	209221	643107	71115

Table 5: Insane AI with iterative deepening and MTD(f)

1113619084, 2022. [Online; accessed 6 October 2022].

[2] Wikipedia contributors. Mtd(f) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MTD(f)&oldid=1112174230, 2022. [Online; accessed 6 October 2022].

[3] Wikipedia contributors. Negamax — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Negamax&oldid=1067237200, 2022. [Online; accessed 6 October 2022].

[4] Wikipedia contributors. Solved game — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Solved_game&oldid=1112712449, 2022. [Online; accessed 6 October 2022].

[5] Wikipedia contributors. Zero-sum game — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Zero-sum_game&oldid=1105258284, 2022. [Online; accessed 6 October 2022].