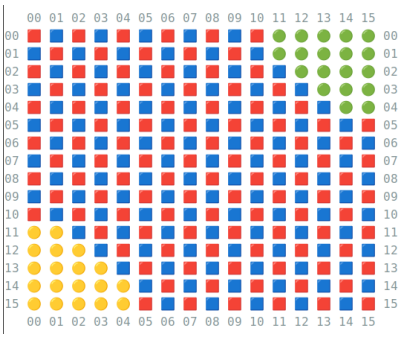
Minimax - Halma

Jakub Ner

Abstract

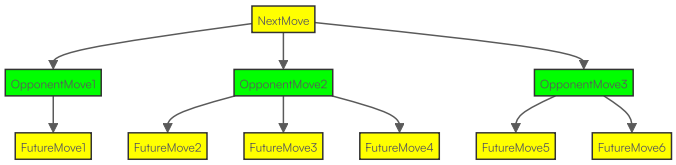
The aim of the paper is to compare different implementations of minimax algorithm (recursive, concurrent, accompanied by alpha-beta pruning) and analyze different heuristics and parameters. The algorithm is showcased on example of Halma - a Chinese checkers where the objective is to get to the opponent base, before him.



Board at the beginning of a game, wins who first gets to opposite corner

Minimax algorithm

Minimax is an algorithm that searches through all possible moves and its consequences (until specified depth) in order to find best solution. It rates solutions according to specified heuristic.



The speed of calculations can be increased by replacing recursion with concurrence. In the presented implementation a node starts a new goroutine (Golang lightweight coroutine) for every child.

The amount of calculations can be significantly reduced by utilization of alpha-beta pruning. A technique that eliminates edges that are not promising.

	recursive	**concurrent**	**with alpha-beta pruning**
Time [ms]	43 002	20 514	12 626
Visited nodes	3 725 728	3 725 728	244 331
Pruned edges			28 563

Statistics for the same game but with different implementations, for depth=2

Heuristics

To find moves that may be considered as good I used different heuristics. All of them return positive values. The bigger value suggest better move. In this section I focus on them.

Euclidean Distance

For every pawn of the current player power of Euclidean distance is calculated between a pawn position and the map corner ((15, 0) or (0, 15); which corner, depends on the player). The smaller the distance, the higher score. Also player gets additional points for a pawns that reach opponent base.

Manhattan Distance

For every pawn of the current player Manhattan distance is calculated between a pawn position and the map corner ((15, 0) or (0, 15); which corner, depends on the player). The smaller the distance, the higher score. Also player gets additional points for a pawns that reach opponent base.

Number of Moves (with Euclidean distance)

Intuition behind that heuristic is that the more pawns have potential moves, the the easier it is for them to navigate through the map and reach the goal. That is why the score is based on number of possible moves and Euclidean distance heuristic. To measure the best influence of moves number I compared 9 games where bot used the heuristic but the number of moves was scaled

```
func MoveNumScore(board *Board, currentPlayer utils.Player) int {
    moveScore := moveNumScoreHelper(board, currentPlayer)
    distScore := DistanceScore(board, currentPlayer)
    return int(float64(moveScore)*_influence) + distScore
}
```

For all tested combinations wins a player with **influence=1**, what indicate that this is the best value. In a table are all results:

	0.5	**1.0**	**1.5**
0.5	yellow	*green*	yellow
1.0	*yellow*	*yellow*	*yellow*
1.5	draw	*green*	draw

Names of columns 2-4 are influence values for green player. Whereas the first column contains influence values for yellow player.

Number of Moves (with Manhattan distance)

Similar to the above, but Manhattan distance heuristic was applied. Also 9 test games were run to determine best scalar.

- A player who had 0.5 won 2 times,
- who had 1.0 won 3 times
- who had 1.5 won 4 times, that indicates that **influence=1.5** is the best choice

	0.5	1.0	1.5
0.5	green	green	green
1.0	green	green	yellow
1.5	yellow	yellow	green

Names of columns 2-4 are influence values for green player. Whereas the first column contains influence values for yellow player.

Number of neighbors (with Euclidean distance)

If pawns are grouped together, then it is hard for a opponent to jump through them. The opponent pawns have to circle them. That's why in this heuristic I reward for that. Also to follow the main objective of the game, I calculate the distance to the board corner. I run 9 tests but all of them led to draws, what shows that the heuristic is not self-sufficient. Player should care about number of neighbors only for a part of the game, that is why below I introduced adaptive heuristic based on this one.

Number of neighbors (with Manhattan distance)

Similar to above but with Manhattan distance. For tests results where also similar - all led to draws.

Manhattan distance with adaptive number of moves

At the beginning of the game pawns are rewarded by having more possible moves. That leads to pawns disperse at the middle of the game until a specified moment. Then they only Manhattan distance matters.
I prepared Tests in which I compared this heuristic (yellow player) with non-adaptive number of moves (green player). The results suggest that the interval should not start to early and finish to late. My hypothesis is that at the beginning pawns should focus on dispersing in order to increase number of possible jumps (that is achieved by number of moves heuristic). In the middle those should use the advantage and jump to the board corner.

```go
func ManhattanAdaptiveMoveNumScore(board *Board, currentPlayer utils.Player) int {
    distScore := 0
    if _turnCounter > _activationMoment1 {
        distScore += DistanceManhattanScore(board, currentPlayer)
    }
    moveScore := 0
    if _turnCounter < _activationMoment2 {
        moveScore = moveNumScoreHelper(board, currentPlayer)
    }
    return int(float64(moveScore)*1.5) + distScore
}
```

interval:	0 - 20	0 - 25	0 - 30	5 - 25	5 - 30	5 - 35	10 - 30	15 -35	10 - 35
won:	green	green	green	**yellow**	**yellow**	**yellow**	draw	green	green

Three intervals led to winning after the same number of rounds.

Manhattan distance with adaptive number of neighbors

During the specified interval, number of neighbors is added to the Manhattan distance. That groups pawns in the middle of the game and makes it difficult for the opponent to pass.
I determined the best interval through games where yellow player use this heuristic, and green one used non-adaptive with number of neighbors. The best result gave interval that starts in 15th round and finishes in 30.

```go
func ManhattanAdaptiveNeighbourScore(board *Board, currentPlayer utils.Player) int {
    distScore := DistanceManhattanScore(board, currentPlayer)
    neighScore := 0
    if _turnCounter > _activationMoment1 && _turnCounter < _activationMoment2 {
        neighScore = neighbourScoreHelper(board, currentPlayer)
    }
    return neighScore + distScore
}
```

interval:	10 - 15	10 - 20	10 - 25	15 - 20	15 - 25	15 - 30	20 - 25	20 -30	20 - 35
won:	draw	draw	draw	draw	draw	**yellow**	draw	**yellow**	draw
number of rounds						105		117	

Heuristics Comparison

I run 64 games to test how heuristics deal with each other. Every heuristic was used 14 times in total. The depth was set to 3 due to time constraints. Results are followings:

	1	2	3	4	5	6	7	8
1. DistanceEuclideanScore	**yellow**	**yellow**	**yellow**	green	green	**yellow**	**yellow**	**yellow**
2. DistanceManhattanScore	green	green	green	**yellow**	draw	draw	draw	green
3. MoveNumScore	green	green	green	draw	green	**yellow**	yellow	green
4. MoveNumManhattanScore	draw	draw	draw	**yellow**	green	green	green	draw
5. NeighbourScore	draw	**yellow**	**yellow**	draw	**yellow**	**yellow**	green	**yellow**
6. NeighbourManhattanScore	**yellow**	green	green	**yellow**	green	green	green	green
7. ManhattanAdaptiveMoveNumScore	green	**yellow**	green	green	draw	green	green	**yellow**
8. ManhattanAdaptiveNeighbourScore	green	green	green	**yellow**	draw	draw	draw	green

Names of columns 2-9 are heuristics used by the green player. Whereas the first column contains heuristics used by the yellow player.

The results depend on a used heuristic. DistanceEuclideanScore won the most number of times - 10, but loses when plays with MoveNumManhattanScore and NeighbourScore. Even though the most effective is NeighbourScore - the heuristic that makes it difficult for the opponent to pass by grouping its pawns, because It won 9 times and led to draw if is not able to win.

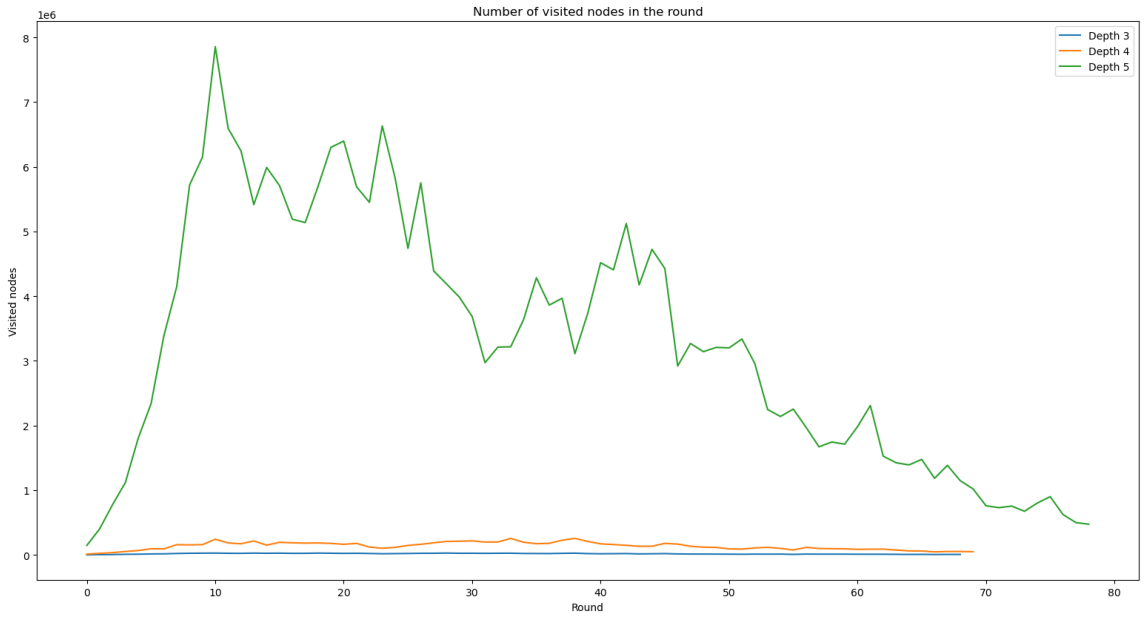
	yellow player pick heuristic and wins	green player pick heuristic and wins	total wins	draws
1. DistanceEuclideanScore	6	4	**10**	2
2. DistanceManhattanScore	1	4	5	4
3. MoveNumScore	1	5	6	2
4. MoveNumManhattanScore	1	2	3	**6**
5. NeighbourScore	5	4	**9**	**5**
6. NeighbourManhattanScore	2	3	5	2
7. ManhattanAdaptiveMoveNumScore	2	4	6	3
8. ManhattanAdaptiveNeighbourScore	1	4	5	4

Table with summarized statistics from the previous table

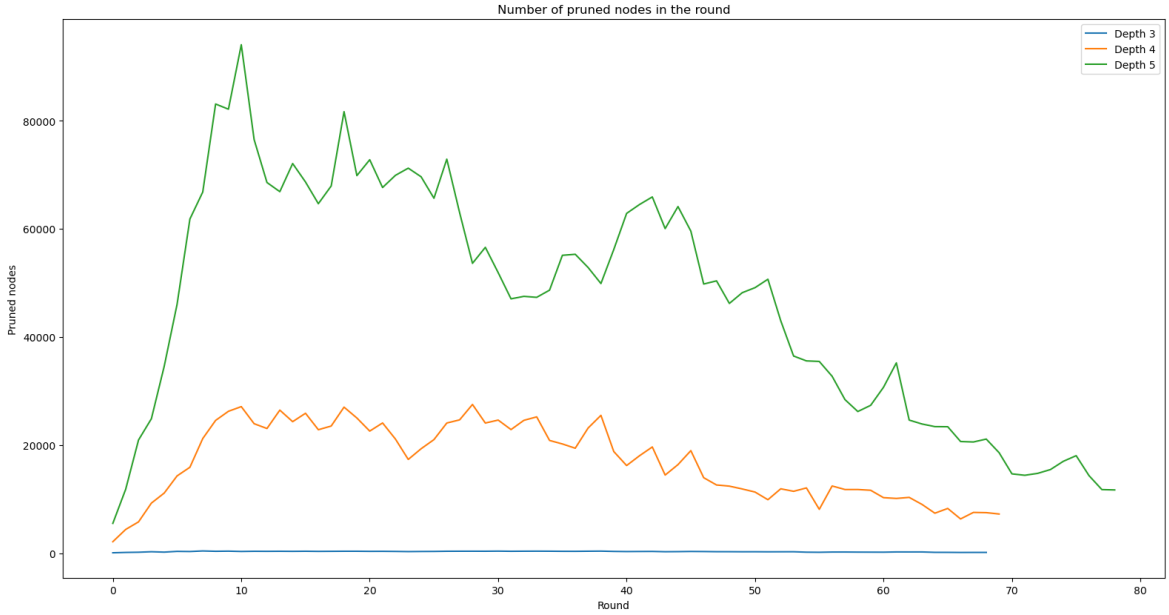
Depths comparison

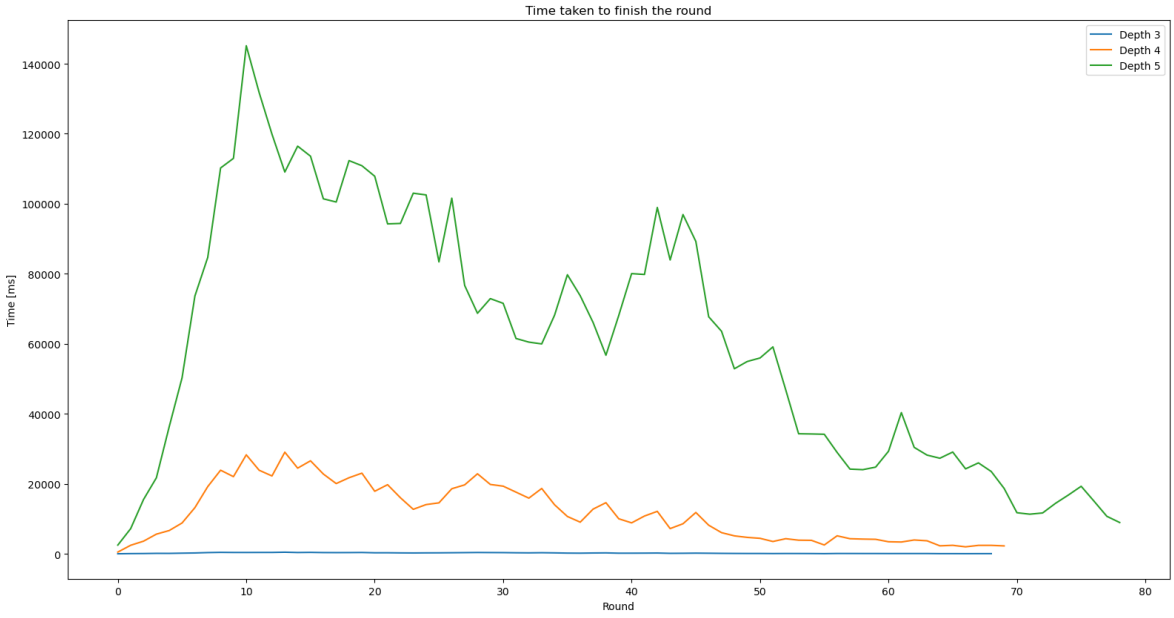
In Halma a player has 19 pawns and each of them can have over 20 possible moves in just one round. That makes deep searching computationally expensive and time consuming. For example a game where two minimax algorithms play with themselves last:

- ~30 seconds, if a searching depth is equal to 3
- ~27.3 minutes, if a searching depth is 4
- ~2.5 hours, if a searching depth is 5



With each level, a number of nodes grows exponentially.


Number of pruned nodes in the round
Depth 3
Depth 4
Depth 5
Pruned nodes
Round
0
20000
40000
60000
80000
0
10
20
30
40
50
60
70
80


Time taken to finish the round
Depth 3
Depth 4
Depth 5
Time [ms]
Round
0
20000
40000
60000
80000
100000
120000
140000
0
10
20
30
40
50
60
70
80