Softwarepraktikum SS 2019

Assignment 3

Group - 6

Roman Vuskov	376809	roman.vuskov@rwth-aachen.de
Songran Shi	357791	songran.shi@rwth-aachen.de
Marlene Damm	379945	marlene.damm@rwth-aachen.de
Marvin Gazibarić	378154	marvin.gazibaric@rwth-aachen.de

Task 1 & 2

Three different search algorithms were considered for this task: Max^n search, paranoid search and Best Reply search. In the following sections we will explore the strengths and weaknesses of each algorithm and explain our ultimate decision in favor of Best Reply search. We will also discuss the implementation details of our algorithm as well as alpha-beta pruning.

1.1 Max^n Search

The most straightforward extension of the standard 2-player Minimax algorithm to multi-player[1] games is the Max^n algorithm: Instead of assigning each node a single (scalar) value, the valuation of a game state is now represented by a vector with one component for each player. A layer in the game tree still corresponds to a particular player's turn, but he only pays attention to his component of the value vectors when making a move decision.

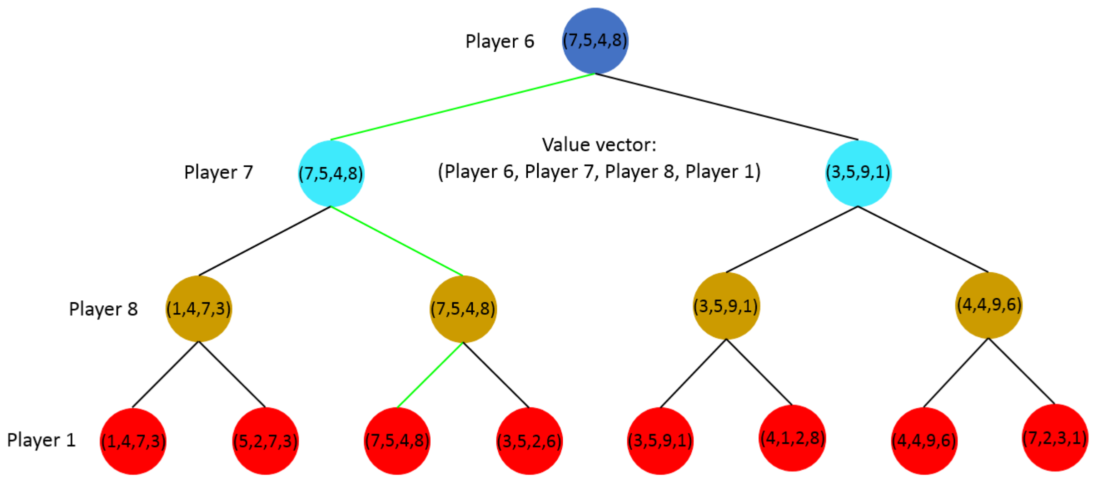

Figure 1: A 4-player Max^n game tree

The obvious advantage of Max^n is that it accurately represents the true interest of each player - winning the game - and thus mimics real player behavior the closest.

[1]Multi-player refers to >2 players

This is an essential property for any game tree search algorithm since looking ahead and gaining a more accurate picture of the future board state evolution is the reason we're utilizing tree searches in the first place. If the results diverge too strongly, we might as well not bother.

However, there is a fatal flaw to this algorithm: It has been mathematically proven [1] that there exists no pruning scheme for Max^n comparable in computational efficiency with alpha-beta pruning in the two-player case, where the branching factor is reduced to $\sqrt{b}$ (b being the branching factor without pruning) in the best case and $b^{3/4}$ realistically. Alternative pruning schemes for Max^n such as Hypermax [2] only yield a branching factor of $b^{(N-1)/N}$ (where N is the total player count) even in the best case. In the 8-player case, this amounts to an improvement by a factor of only $b^{d/8}$ in search time (where d is the search depth).

1.2 Paranoid Search

An alternative to Max^n for multi-player games is paranoid search: Here we simply assume the worst case possible where every other player is teaming up against us, each choosing the move that harms our position the most. This assumption allows us to – just like in the 2-player case – assign each board state a single value, namely the valuation from our point of view. The game tree then consists of one Max-layer followed by $N-1$ Min-layers:

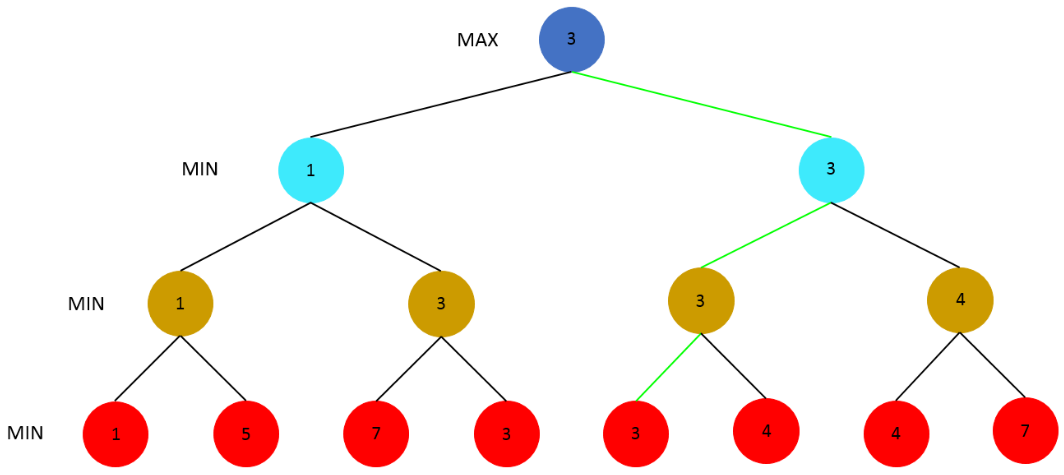

Figure 2: A paranoid search game tree. The situation is the same as in Figure 1

Since the evaluation of board states return single scalar values again, alpha-beta pruning becomes applicable here in a similar fashion to standard 2-player Minimax. Surprisingly, paranoid search actually performs quite well against Max^n in experiments [3][4] with other games. However, there are several reasons to doubt the applicability of these results to Reversi++:

1. In [3], it is apparent that paranoid search's win-rate against Max^n decreases with growing total player count. This is expected as the paranoid assumption deviates more and more from reality with increasing player count: Harming another player's position increasingly becomes a public good
2. The score of a game in [4] is binary: A game counts as either a win or a loss, and there is no reward for second place. In this case it makes sense for all other players to form an alliance against the player in 1st place, and conversely it makes sense for the player in 1st place to be paranoid. This, however, does not apply to the scoring system in Reversi++
3. A large part of paranoid search's superiority comes from the extra search depth alpha-beta pruning enables. With game boards as large as 2500 tiles in Reversi++ however, either algorithm can only scratch the surface of the whole game tree

1.3 Best Reply Search

Best Reply Search [4] offers a reasonable middle ground between Max^n and paranoid search. The game tree for Best Reply Search looks very much like the Minimax tree for 2-players, with alternating Max- and Min-layers where the Max-player represents us and the Min-player represents all other players (i.e. has moves of all other players available to him):

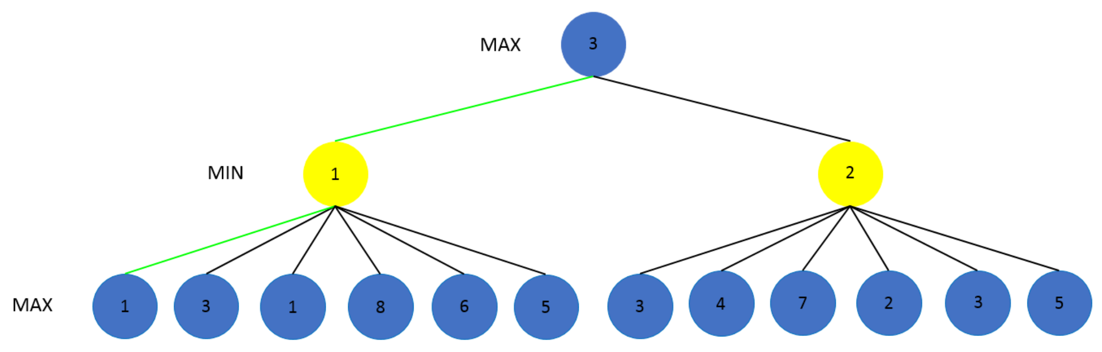

Figure 3: A Best Reply Search game tree. MIN has many more nodes than MAX since Min represents all other players

Like paranoid search, the value at each node is a simple scalar, and alpha-beta pruning is even more straightforward here. BRS also makes the paranoid assumption, but not as strongly as paranoid search since the Min-player will only have 50% of the turns instead of $\frac{N-1}{N}$. Additionally, it has the advantage of searching through more Max-layers – the layers most relevant to us – over both Max$^{\mathrm{n}}$ and paranoid search. This is especially justified in the early stages of the game where usually players' starting positions are divided into pairs, and the pairs are far apart from each other. This way, considering moves of any player other than ourselves and the closest opponent is a waste of computational resources which BRS avoids.

However, it should be noted that BRS mostly produces board states that cannot exist in reality: the BRS game tree is *not* a representation of possible future states. In experiments in [4], though, BRS produced good results against both Max$^{\mathrm{n}}$ and paranoid search nonetheless, which was the ultimate reason we settled on this algorithm. It would still be worth to conduct tests in Reversi++ with all three algorithms to determine empirically which is best suited for this particular game.

1.4 Beam Search

Even with alpha-beta pruning, the branching factor can still turn out to be too much to handle with the given search depth and time limit. Additionally, alpha-beta pruning requires reasonable ordering of child nodes to provide any significant speedup. To account for both, we introduced beam search into our algorithm: for each node in the tree (except for leave nodes) the AI first finds all child nodes and evaluates the board state of these child nodes directly (essentially a one-step search), then orders the best w of them in a list for the subsequent recursive call. w then is our beam width, or the effective branching factor, and the ordered list of child nodes is the “beam” we’re searching through. This way, we can control the branching factor of our game tree at any stage of the game:

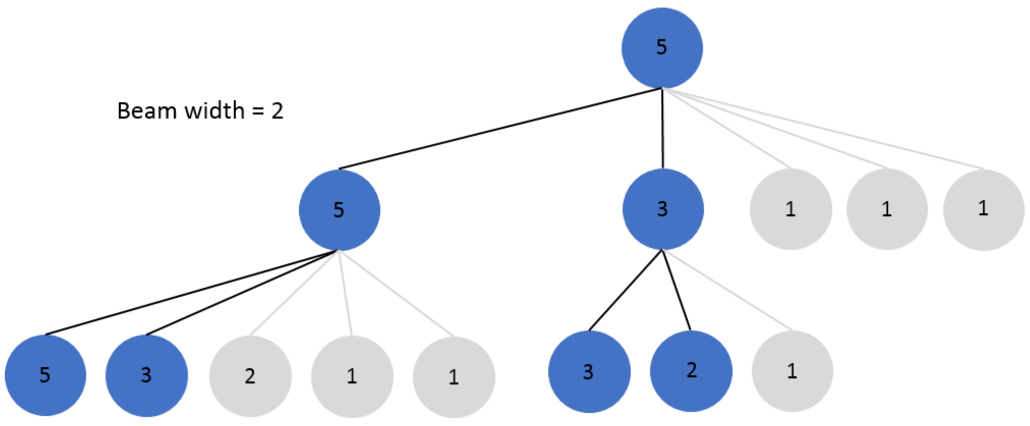

Figure 4: A beam search tree with beam width 2

This is however a greedy strategy since we might be discarding quite a few possible moves at each node on relatively little information. If this turns out to be a bad idea that affects the output quality too much, we can still effectively turn off beam search by setting beam width to infinity while retaining move ordering for alpha-beta pruning to be efficient.

1.5 Implementation

Our implementation of Best Reply Search consists only of one class: the BRSNode class. It contains two constructors and 6 methods:

- `public BRSNode` public constructor for the root node; used only by external calls
- `private BRSNode` private constructor for internal nodes; only used by (recursive) internal calls

- `getBestMove` simply returns the best move from the node that has already been evaluated; only used on the root node to return the final result
- `evaluateNode` recursive method and center point of the class; first computes search beam (using `computeBeam`); then constructs for beam entries child nodes and calls `evaluateNode` in them; finally updates its own value to the maximum or minimum (depending on whether it's a Max- or Min-Node) of its children values and returns its own value
- `computeBeam` computes the beam by fetching all legal moves (by using `getMaxMoves` or `getMinMoves`) and ordering the best w of them in a list; then returns the ordered list
- `getMaxMoves` fetches legal moves the Max-player (us) can make; prioritizes regular moves over override moves (it's a good heuristic to save override moves for last)
- `getMinMoves` fetches all other players' legal moves and merges them into one set (these are moves the Min-player can make by definition of BRS); prioritizes regular moves over override moves
- `evaluateCurrentState` evaluates the board state of the node according to heuristics; used to order child nodes and to evaluate leave nodes

Algorithm 1 Node Evaluation

```
function EvaluateNode()

    beam[move] ← ComputeBeam()

    if beam is empty then
        return value of current state

    else if current layer == search depth then
        return value of current state

    else if this node is Max node then
        for each move in beam do
            construct child node(current layer + 1, Min node)
            EvaluateNode(child node)
            if child node value > node value then
                node value = child node value

    else if this node is Min node then
        for each move in beam do
            construct child node(current layer + 1, Max node)
            EvaluateNode(child node)
            if child node value < node value then
                node value = child node value
```

To initialize BRS, one needs to construct the root node with parameters `depth` (search depth), `branching` (branching factor) and `prune` (alpha-beta pruning on/off), then evaluate the root node and finally get the best move of the root node. With current heuristics in place our BRS algorithm can search through $\approx$ 1000 nodes per second, pending future updates. It is only used for the first phase of the game as moves in the Bombing Phase are mostly independent of each other and the branching factor is very large, making a direct move evaluation more practical than any sort of tree search.

1.6 Alpha-beta pruning

Alpha-beta pruning can be seamlessly integrated into BRS. We initialize α and β as $-\infty$ and $+\infty$ respectively, and pass down these two parameters each time a child node is constructed. When a node is going through the beam and updating its own value, α or β (depending on whether it's a Max- or Min-Node) is updated as well and compared with each other. If $\beta \leq \alpha$, the loop gets broken and the rest of the beam discarded, as per the alpha-beta pruning algorithm.

Algorithm 2 Alpha beta Node Evaluation

```
function EvaluateNode()

    beam[move] ← ComputeBeam()

    if beam is empty then
        return value of current state

    else if current layer == search depth then
        return value of current state

    else if this node is Max node then
        for each move in beam do
            construct child node(current layer + 1, Min node, $\alpha$, $\beta$)
            EvaluateNode(child node)
            if child node value > node value then
                node value = child node value
                $\alpha$ = child node value
                if $\alpha \geq \beta$ then
                    break

    else if this node is Min node then
        for each move in beam do
            construct child node(current layer + 1, Max node, $\alpha$, $\beta$)
            EvaluateNode(child node)
            if child node value < node value then
                node value = child node value
                $\beta$ = child node value
                if $\alpha \geq \beta$ then
                    break
```

Task 3

First we compared our AI playing the same map against the same opponents in the same position with and without pruning enabled at depth limit 3. Because pruning does not change the decision outcome the resulting games where the same and easy to compare. Figure 5 shows that alpha-beta pruning can lower the amount of leaf

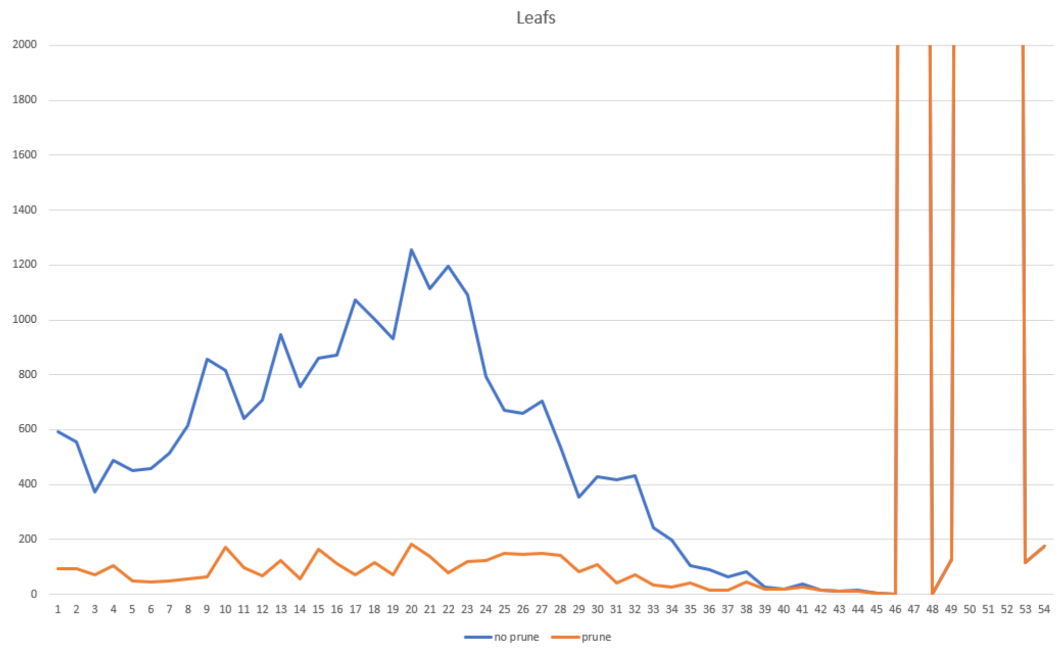

Figure 5: blue without pruning, orange with pruning

nodes and thereby computational work by an substantial amount. This is possible to such a degree in our case because we already use move sorting for beam search (the actual beam search is turned off in this experiment). The big spikes after Move 48 belong to the "every player places his override stones phase" (the blue curve is obstructed by the orange) and have the exact number of leafs in pruning and no pruning mode suggesting that our method of pruning somehow fails in this phase. This has to be further examined. The amount of states statistic looks basically like the amount of leafs statistic which is unsurprising (more states → more leaves) so we don't further discuss it here. What however is interesting is that the times per state shown in Figure 6 are higher with pruning and significantly higher in the "every player places his override stones phase". Figure 7 shows that even though times per state are higher with pruning the AI is able to chose moves faster because there are less states. It also shows how much faster beam search combined with

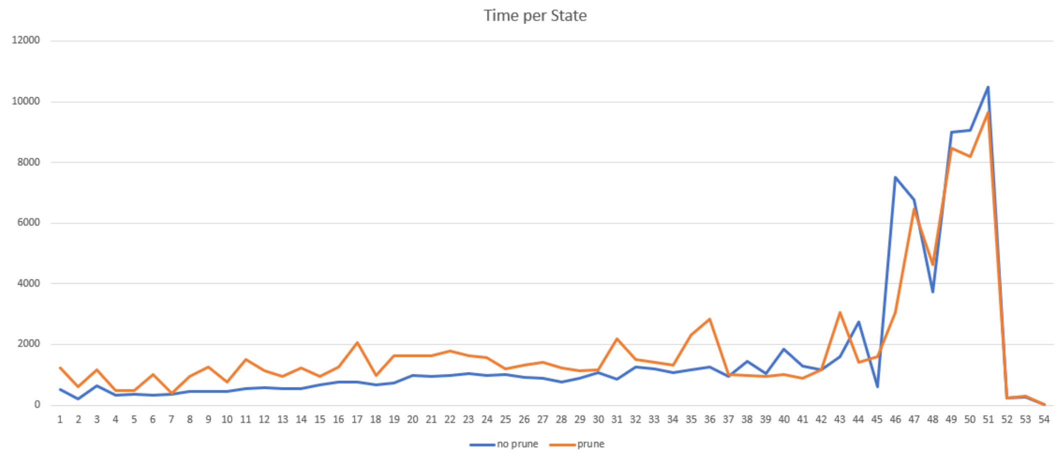

Figure 6: blue without pruning, orange with pruning

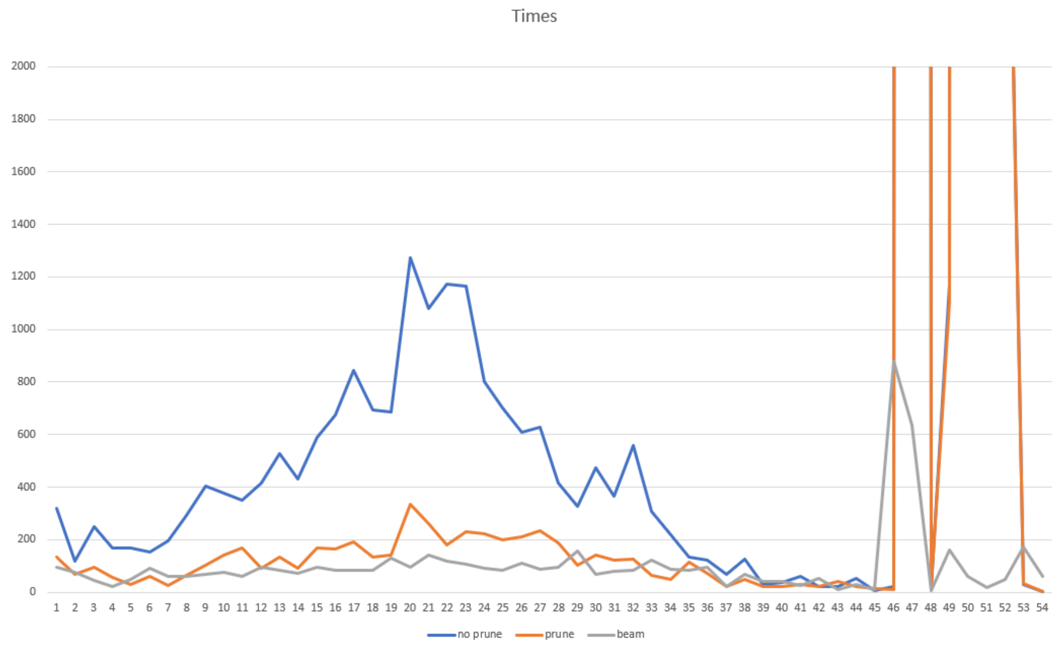

Figure 7: blue without pruning, orange with pruning, gray beam

pruning can be. Because beam search introduces a maximum branching factor it can also handle the extreme branching of the “every player places his override stones phase”.

This of course is not a big sample size but previous research from various people shows that alpha-beta pruning should be used when doing minMax, especially if you are able to do good (and fast) move sorting.

References

[1] R. E. Korf, “Multiplayer Alpha-Beta Pruning”, Artificial Intelligence, vol. 48, no. 1, pp. 99-111, 1991.

[2] Mikael Fridenfalk, “N-Person Minimax and Alpha-Beta Pruning”, NICOGRAPH International 2014, pp. 43-52

[3] Nathan Sturtevant, “A Comparison of Algorithms for Multi-player Games”, Computers and Games Third International Conference, CG 2002, pp. 108-122

[4] Maarten P.D. Schadd and Mark H.M. Winands, “Best-Reply Search for Multi-Player Games”