Spellchecker

Martijn Gösgens 0914954
Dominique Sommers 0895679

March 27, 2016

1 Algorithms

1.1 Alpha-Beta Search

Algorithm 1 Alpha-Beta-Search(int depth)

```
Let s be the current state
bestMove ← null
bestValue ← −∞
for all Moves m do
    Do(m)
    value ← AlphaBetaMin(s, −∞, ∞, depth)
    if value > bestValue then
        bestValue ← value
        bestMove ← m
    Undo(m)
```

Algorithm 2 Alpha-Beta-Min(State s, double alpha, double beta, int d)

```
Check whether the time is up and throw an exception if so
if d = 0 then return State-Evaluation(s)
else
    for all Moves m do
        Do(m)
        max ← Alpha-Beta-Max(s, alpha, beta, d-1)
        beta ← Min(beta, max)
        Undo(m)
        if alpha ≥ beta then return alpha
    return beta
```

Algorithm 3 Alpha-Beta-Max(State s, double alpha, double beta, int d)

```
Check whether the time is up and throw an exception if so
if d = 0 then return State-Evaluation(s)
else
    for all Moves m do
        Do(m)
        max ← Alpha-Beta-Min(s, alpha, beta, d-1)
        alpha ← Max(beta, max)
        Undo(m)
        if alpha ≥ beta then return beta
    return alpha
```

We implemented the Alpha-Beta algorithm as described in "Introduction to Articial Intelligence" [2]. We decided not to use GameNode's since we did not think that would be necessary and this might be more efficient.

1.2 Iterative Deepening

Algorithm 4 Iterative-Deepening()

```
bestMove ← null
for d = 1, 2, ... do
    try bestMove ← Alpha-Beta-Search(d) and return bestMove if exception thrown
```

We call our Alpha-Beta-Search algorithm with increasing depths and when the AI is stopped (an exception is thrown), we return the move found by the last finished run of the algorithm.

1.3 Evaluation Function

Algorithm 5 State-Evaluation[State s]

```
if AI has no moves left then
    Return Integer.MINVALUE
else if Opponent has no moves left then
    Return Integer.MAXVALUE
else
    for all pieces p do
        pieceValue = 100
        if p is a king then
            p.pieceValue* = 2
            if p is on the side then
                p.pieceValue+ = 20
        else if p is black then
            if p is on row 9 then
                p.pieceValue+ = 20
            else
                p.pieceValue+ = (9 − rowNumber) * 3
        else
            if p is on row 0 then
                p.pieceValue+ = 20
            else
                p.pieceValue+ = rowNumber * 3
    blackValue ∑_{p∈pieces} p.pieceValue where p is black.
    whiteValue ∑_{p∈pieces} p.pieceValue where p is white.
    if AI is playing black then
        return blackValue/whiteValue
    else
        return whiteValue/blackValue
```

For the evaluation function, a score system is made for the draughts pieces. Initially, every piece is worth 100 points. A king is worth twice as much as a normal piece. Now, for kings it is better to stay at the side of the board rather than somewhere in the middle [1]. So there are 20 points added to the score of a king when it is positioned at the side in order to give this state of position a higher score. For normal pieces, the defense line is valuable to keep the opponent from getting a king. So again, 20 points are added to the score of pieces positioned on the first line. Pieces further on the board are more likely to get jumped, but is also more likely to jump it self. Also, the closer it is to the opponent's side, the closer it is to become a king. For every line it is moved forward, the piece receives an extra 3 points to add to it's score. So, a maximum of 24 points can be added, when the piece is one row from becoming a king.
After evaluating each piece value, the final value of the state is calculated as the ratio between the value of the AI's pieces and the value of the opponent's pieces. This will give a better approximation to the value of the state than taking the difference between the two, since for

example the state with 5 pieces to 1 is much strong than with 20 pieces to 16, although the difference is the same.

2 Results

Depending on the allowed time and the state of the board, the Alpha-Beta search usually reaches depths between 6 and 10. This can be seen by letting the player play a match and watching the application output (the depths are printed). The player itself also gives good results; it wins from all the default players and it also won from every human being that we let it play to (we tested it on seven people). This result can be reproduced by trying to play against it, you will very likely lose. Our evaluation function also gave good results. If we copy the AlphaBetaPlayer and replace its evaluationfunction with the more obvious evaluation function of counting every piece as one, counting kings as two and returning the difference of scores between the team, then this player will lose against our AlphaBetaPlayer.

3 References

1. http://www.checkerstip.com/handling-a-king-piece-in.html
2. Ertel, W. Introduction to Artificial Intelligence.

4 Contribution

Martijn Gösgens: I implemented and tested the Alpha-Beta search with iterative deepening. I also tested the evaluation function and contributed to it slightly (I came up with the idea of looking at the relative difference of the scores instead of the absolute difference).
Dominique Sommers: I designed the evaluation function for the AI. In order to do this, I studied good draughts patterns and searched for strong positions of particular pieces. These could be defended pieces (having a piece behind them, so they can not be jumped), defending pieces (having a piece in front of them), elbow pattern (pieces defending and being defended). However, I decided to keep the evaluation simple, since the method is called many times.

5 Manual

Add the AI player name to the Draughts game and it can play against other AI's or against a human. You can let it play against the default players and see it win every time, you can let a few average people play against it and see the majority lose. You can also let it play against the older versions of the AlphaBetaPlayer (AlphaBetaPlayerV0, with evaluation function as described in the results) and see it win.