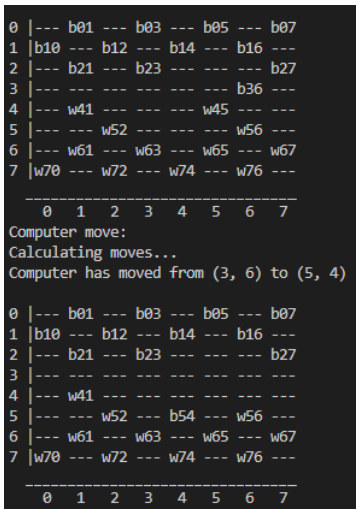
Projektowanie Algorytmów i Metody Sztucznej Inteligencji	Projekt 3	
Grzegorz Wojnarowicz [254056]	20.06.2022	Dr inż. Marta Emirsajłow

Projekt znajduje się pod poniższym linkiem:
`https://github.com/GrzegorzWojnarowicz/checkers.git`

1 Zasady i działanie

- W przypadku posiadania możliwości bicia jest ono wymagane.
- Królowa może ruszać się w drugą stronę, ale tylko o jedno pole
- Nie ma bić wielokrotnych



2 Algorytm min-max

Algorytm min-max polega na minimalizowaniu strat. Każdy ruch jest oceniany za pomocą wartości, na przykład w warcabach pionek ustawiony przy bandzie jest zabezpieczony przed zbiciem, dlatego ustawienie tak pionka zwiększa nam jego wartość. Ważne jest jednak to, że przeciwnik również dąży do jak najlepszej pozycji, dlatego przeszkadzanie mu również trzeba wziąć pod uwagę. Dlatego oceniana jest różnica pomiędzy oboma graczami i na jej podstawie podejmowana jest decyzja o kolejnym ruchu.

2.1 Algorytm alpha-beta

Zastosowanie tego algorytmu pozwala na zmniejszenie ilości obliczeń, ponieważ przeszukujemy wszystkie możliwe ruchy, po czym odcinamy te, które okazują się stawiać AI w gorszej pozycji od poprzednio badanych opcji. Dzięki temu uzyskujemy ten sam ruch co w przypadku algorytmu min-max, tyle że przy znacznie mniejszej ilości obliczeń. Oczywiście, są sytuacje gdzie odetniemy gałąź, która okazałaby się dobra, jednak są to niszowe przypadki, w których gracz poświęca coś, żeby zyskać coś po większej ilości ruchów niż głębia algorytmu min-max.

3 Wnioski

1. Przy wyższych poziomach głębi nie jestem w stanie wygrać z komputerem, co oznacza, że algorytm min-max i alpha-beta działają poprawnie.

4 Implementacja

Sprawdzanie dostępnych ruchów gracza

```
std::vector<Checkers::Move> Checkers::findPlayerMoves() const
{
    std::vector<Move> moves;
    std::vector<Move> jumps;

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            if (mBoard[i][j][0] == 'w')
            {
                if (checkMove({i, j, i - 1, j - 1}, true))
                    moves.push_back({i, j, i - 1, j - 1});
                if (checkMove({i, j, i - 1, j + 1}, true))
                    moves.push_back({i, j, i - 1, j + 1});
                if (checkJumps({i, j, i - 2, j - 2}, i - 1, j - 1, true))
                    jumps.push_back({i, j, i - 2, j - 2});
                if (checkJumps({i, j, i - 2, j + 2}, i - 1, j + 1, true))
                    jumps.push_back({i, j, i - 2, j + 2});
            }
            else if (mBoard[i][j][0] == 'W')
            {
                if (checkMove({i, j, i + 1, j + 1}, true))
                    moves.push_back({i, j, i + 1, j + 1});
                if (checkMove({i, j, i + 1, j - 1}, true))
                    moves.push_back({i, j, i + 1, j - 1});
                if (checkMove({i, j, i - 1, j + 1}, true))
                    moves.push_back({i, j, i - 1, j + 1});
                if (checkMove({i, j, i - 1, j - 1}, true))
                    moves.push_back({i, j, i - 1, j - 1});
                if (checkJumps({i, j, i + 2, j + 2}, i + 1, j + 1, true))
                    jumps.push_back({i, j, i + 2, j + 2});
                if (checkJumps({i, j, i + 2, j - 2}, i + 1, j - 1, true))
                    jumps.push_back({i, j, i + 2, j - 2});
                if (checkJumps({i, j, i - 2, j + 2}, i - 1, j + 1, true))
                    jumps.push_back({i, j, i - 2, j + 2});
                if (checkJumps({i, j, i - 2, j - 2}, i - 1, j - 1, true))
                    jumps.push_back({i, j, i - 2, j - 2});
            }

    if (jumps.size() == 0)
        return moves;
    return jumps;
}
```

Sprawdzanie dostępnych ruchów AI

```cpp
std::vector<Checkers::Move> Checkers::findComputerMoves() const
{
    std::vector<Move> moves;
    std::vector<Move> jumps;

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            if (mBoard[i][j][0] == 'b')
            {
                if (checkMove({i, j, i + 1, j + 1}, false))
                    moves.push_back({i, j, i + 1, j + 1});
                if (checkMove({i, j, i + 1, j - 1}, false))
                    moves.push_back({i, j, i + 1, j - 1});
                if (checkJumps({i, j, i + 2, j + 2}, i + 1, j + 1, false))
                    jumps.push_back({i, j, i + 2, j + 2});
                if (checkJumps({i, j, i + 2, j - 2}, i + 1, j - 1, false))
                    jumps.push_back({i, j, i + 2, j - 2});
            }
            else if (mBoard[i][j][0] == 'B')
            {
                if (checkMove({i, j, i + 1, j + 1}, false))
                    moves.push_back({i, j, i + 1, j + 1});
                if (checkMove({i, j, i + 1, j - 1}, false))
                    moves.push_back({i, j, i + 1, j - 1});
                if (checkMove({i, j, i - 1, j + 1}, false))
                    moves.push_back({i, j, i - 1, j + 1});
                if (checkMove({i, j, i - 1, j - 1}, false))
                    moves.push_back({i, j, i - 1, j - 1});
                if (checkJumps({i, j, i + 2, j + 2}, i + 1, j + 1, false))
                    jumps.push_back({i, j, i + 2, j + 2});
                if (checkJumps({i, j, i + 2, j - 2}, i + 1, j - 1, false))
                    jumps.push_back({i, j, i + 2, j - 2});
                if (checkJumps({i, j, i - 2, j + 2}, i - 1, j + 1, false))
                    jumps.push_back({i, j, i - 2, j + 2});
                if (checkJumps({i, j, i - 2, j - 2}, i - 1, j - 1, false))
                    jumps.push_back({i, j, i - 2, j - 2});
            }

    if (jumps.size() == 0)
        return moves;
    return jumps;
}
```

Funkcja odpowiedzialna za ruch figur

```cpp
void Checkers::makeMove(Move move, bool player)
{
    char letter = mBoard[move.oldI][move.oldJ][0];
    if (player)
    {
        if (move.newI == 0)
            letter = 'W';
    }
    else
    {
        if (move.newI == (size - 1))
            letter = 'B';
    }

    int i_diff = move.oldI - move.newI;
    int j_diff = move.oldJ - move.newJ;

    if (i_diff == -2 && j_diff == 2)
        mBoard[move.oldI + 1][move.oldJ - 1] = "---";
    else if (i_diff == 2 && j_diff == 2)
        mBoard[move.oldI - 1][move.oldJ - 1] = "---";
    else if (i_diff == 2 && j_diff == -2)
        mBoard[move.oldI - 1][move.oldJ + 1] = "---";
    else if (i_diff == -2 && j_diff == -2)
        mBoard[move.oldI + 1][move.oldJ + 1] = "---";

    mBoard[move.oldI][move.oldJ] = "---";
    mBoard[move.newI][move.newJ] = letter + std::to_string(move.newI) + std::to_string(move.newJ);
}
```

Funkcja zliczająca pozostałe figury

```
void Checkers::countPieces()
{
    mComputerPieces = 0;
    mPlayerPieces = 0;

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            if (mBoard[i][j][0] == 'b' || mBoard[i][j][0] == 'B')
                mComputerPieces++;
            else if (mBoard[i][j][0] == 'w' || mBoard[i][j][0] == 'W')
                mPlayerPieces++;
}
```

Funkcja odpowiedzialna za sprawdzanie dostępnych ruchów

```
bool Checkers::checkMove(Checkers::Move move, bool player) const
{
    if (move.newI >= size || move.newI < 0)
        return false;
    if (move.newJ >= size || move.newJ < 0)
        return false;
    if (mBoard[move.oldI][move.oldJ] == "---")
        return false;
    if (mBoard[move.newI][move.newJ] != "---")
        return false;
    if (player && (mBoard[move.oldI][move.oldJ][0] == 'b' || mBoard[move.oldI][move.oldJ][0] == 'B'))
        return false;
    if (!player && (mBoard[move.oldI][move.oldJ][0] == 'w' || mBoard[move.oldI][move.oldJ][0] == 'W'))
        return false;
    return true;
}
```

Funkcja odpowiedzialna za sprawdzanie dostępnych bić

```
bool Checkers::checkJumps(Checkers::Move move, int viaI, int viaJ, bool player) const
{
    if (!checkMove(move, player))
        return false;
    if (mBoard[viaI][viaJ] == "---")
        return false;
    if (!player && (mBoard[viaI][viaJ][0] == 'b' || mBoard[viaI][viaJ][0] == 'B'))
        return false;
    if (player && (mBoard[viaI][viaJ][0] == 'w' || mBoard[viaI][viaJ][0] == 'W'))
        return false;
    return true;
}
```

Funkcja odpowiedzialna za wyliczenie wartości pozycji

```
int calculateHeuristics(const Checkers::Board& board)
{
    int result = 0;
    int computer = 0;
    int player = 0;

    for (int i = 0; i < Checkers::size; ++i)
        for (int j = 0; j < Checkers::size; ++j)
            if (board[i][j][0] == 'b' || board[i][j][0] == 'B')
            {
                computer++;

                if (board[i][j][0] == 'b')
                    result += 5;
                if (board[i][j][0] == 'B')
                    result += 10;
                if (i == 0 || j == 0 || i == (Checkers::size - 1) || j == (Checkers::size - 1))
                    result += 7;

                if (i + 1 > (Checkers::size - 1) || j + 1 > (Checkers::size - 1) || i - 1 < 0 || j - 1 < 0)
                    continue;

                if ((board[i + 1][j - 1][0] == 'w' || board[i + 1][j - 1][0] == 'W') && board[i - 1][j + 1] == "---")
                    result -= 3;
                if ((board[i + 1][j + 1][0] == 'w' || board[i + 1][j + 1][0] == 'W') && board[i - 1][j - 1] == "---")
                    result -= 3;
                if (board[i - 1][j - 1][0] == 'W' && board[i + 1][j + 1] == "---")
                    result -= 3;
                if (board[i - 1][j + 1][0] == 'W' && board[i + 1][j - 1] == "---")
                    result -= 3;

                if (i + 2 > (Checkers::size - 1) || (j - 2) < 0)
                    continue;

                if ((board[i + 1][j - 1][0] == 'w' || board[i + 1][j - 1][0] == 'W') && board[i + 2][j - 2] == "---")
                    result += 6;

                if (j + 2 > (Checkers::size - 1))
                    continue;

                if ((board[i + 1][j + 1][0] == 'w' || board[i + 1][j + 1][0] == 'W') && board[i + 2][j + 2] == "---")
                    result += 6;
            }
            else if (board[i][j][0] == 'w' || board[i][j][0] == 'W')
            {
                player++;
            }
    return result + (computer - player) * 1000;
}
```

Implementacja algorytmu min-max z zastosowaniem algorytmu alpha-beta

```cpp
int minMax(Checkers checkers, int depth, int alpha, int beta, bool player)
{
    if (depth == 0)
        return calculateHeuristics(checkers.getBoard());

    if (!player)
    {
        int max = -std::numeric_limits<int>::max();
        auto nodes = makeNodes(checkers, player);
        for (auto& node : nodes)
        {
            int val = minMax(node.checkers, depth - 1, alpha, beta, true);
            max = std::max(max, val);
            alpha = std::max(alpha, val);
            if (beta <= alpha)
                break;
        }
        return max;
    }
    else
    {
        int min = std::numeric_limits<int>::max();
        auto nodes = makeNodes(checkers, player);
        for (auto& node : nodes)
        {
            int val = minMax(node.checkers, depth - 1, alpha, beta, false);
            min = std::min(min, val);
            beta = std::min(beta, val);
            if (beta <= alpha)
                break;
        }
        return min;
    }
}
```

5 Literatura

- https://eduinf.waw.pl/inf/utils/001_2008/0415.php
- https://pl.wikipedia.org/wiki/Algorytm_alfa-beta
- https://pl.wikipedia.org/wiki/Algorytm_min-max
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/?ref=lbp
- https://github.com/dimitrijekaranfilovic/checkers