Mikołaj Nadzieja, 184474

Dawid Łydka, 184440

# *Sprawozdanie*

Spis treści:



1. Wstęp

   Tematem projektu była implementacja gry w warcaby w języku python. Wizualna strona gry wykorzystuje bibliotekę pygame. Nacisk został położony na algorytm minmax i heurystyki, które używa.

2. Zasady gry Warcaby

   Klasyczna gra rozgrywana jest na planszy o rozkładzie pól 8 x 8. Każdy gracz rozpoczyna grę z dwunastoma pionkami.

   Celem gry jest zbicie wszystkich pionów przeciwnika albo zablokowanie go, pozbawiając przeciwnika możliwości wykonania ruchu.

   Zwykłe pionki mogą poruszać się o jedno pole do przodu po przekątnej na wolne pola. Bicie następuje przez przeskoczenie pionka przeciwnika na pole znajdujące się za nim po przekątnej. W jednym ruchu wolno wykonać więcej niż jedno bicie tym samym pionkiem.

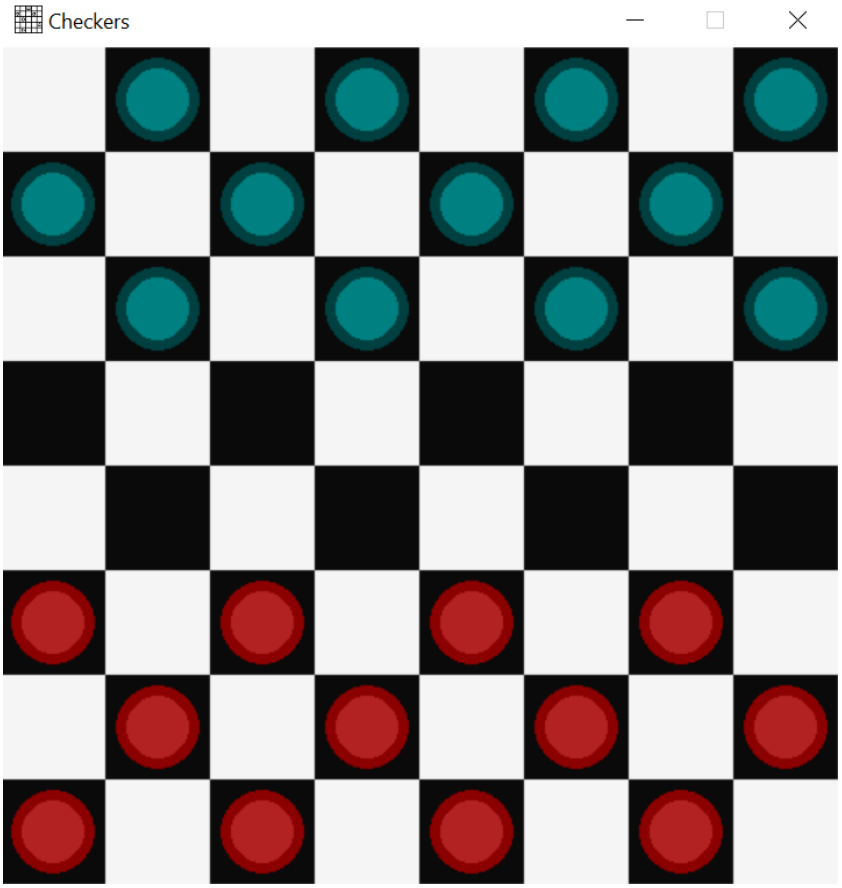
   Pionek, który dojdzie do ostatniego rzędu planszy, staje się damką. Damki mogą poruszać się do przodu lub do tyłu po przekątnej. Bicie damką wygląda tak samo, jak dla zwykłego pionka, ale możliwe jest do przodu i do tyłu.

3. Opis techniczny programu
   Program zbudowany jest z kilku klas: Board, BoardGraphics, Checkers, Game i minmax.

   Klasa Board odpowiedzialna jest za utworzenie planszy o podanym rozmiarze (zawsze kwadratowa). W miejsce pustych pól wstawiane są 0, natomiast w miejsce pionków ustawiane są odpowiednie wartości. Gracze na planszy nazywani są Gracz 1 i Gracz 2. Pionki gracza 1 oznaczane są 1 dla zwykłych pionków i 3 dla damek. Analogicznie pionki gracza 2 oznaczane są 2 dla zwykłych pionków i 4 dla damek. Pionki graczy rozkładane są na planszy we wszystkich rządach poza dwoma środkowymi (trzema środkowymi jeśli plansza ma nieparzyste wymiary). Zajmuje się ona również operacjami na planszy, takimi jak ruch pionka, zbicie pionków i ewaluacja wartości planszy za pomocą heurystyk.

```
[0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 2, 0, 2, 0, 2, 0]
[0, 2, 0, 2, 0, 2, 0, 2]
[2, 0, 2, 0, 2, 0, 2, 0]
```

   Klasa BoardGraphics odpowiedzialna jest za utworzenie okna gry i narysowanie podanego ułożenia pionków na planszy. Narysowana plansza składa się z naprzemiennie białych i czarnych pól. Pionki graczy reprezentowane są kołami w danym kolorze. Na



damkach dodatkowo rysowana jest korona. Dla zaznaczonego pionka mniejszymi kołami reprezentowane są ruchy, które może wykonać. Aktualizuje ona okno gry po wykonaniu każdego ruchu.

Klasa Checkers odpowiedzialna jest za najważniejsze aspekty rozgrywki. Deleguje ona wydarzeniami na planszy i aktualizacją okna gry poprzez wyżej wymienione klasy. Pozwala na zaznaczenie pionka, którym chcemy zagrać i obliczyć dla niego możliwe ruchy. Utrzymuje ona aktualny stan gry, w który wchodzi plansza, tura, wybrany pionek, itp.

Klasa Game odpowiedzialna jest rozpoczęcie gry. Jej główną funkcją jest wytworzenie wszystkich potrzebnych obiektów i rozpoczęcie pętli gry.
Klasa minmax odpowiedzialna jest za implementację algorytmu minmax. Wybiera ona najlepszy możliwy ruch dla komputera.

4. Opis uruchomienia programu
   Przed uruchomieniem programu należy wybrać jaką wersję gry chcemy uruchomić i podać odpowiednią liczbę odpowiadającą danemu trybowi:
   - 1 – tryb gry gracz kontra gracz
   - 2 – tryb gry gracz kontra komputer
   - 3 – tryb gry komputer kontra komputer

   Po wybraniu trybu gry i uruchomieniu programu rozpocznie się rozgrywka. W przypadku wybrania trybu gry, gdzie występuje przynajmniej jeden gracz, podczas tury gracza będzie oczekiwanie na wybranie ruchu przez naciśnięcie na wybrany pionek i następnie na pole, w które chcemy nim poruszyć. Wybór pionka można zmieniać do momentu wykonania ruchu. Gra kończy się, gdy na planszy pozostaną pionki tylko jednego gracza.

5. Algorytm Minmax
   a) Pochodzenie i historia
      Algorytm Minimax to metoda wywodząca się z teorii gry o sumie zerowej, obejmującej dwa przypadki. Ten, w którym gracze wykonują ruchy naprzemiennie jak i drugi, gdzie wykonują ruchy jednocześnie. Minimax został rozszerzony na bardziej skomplikowane gry i ogólne podejmowanie decyzji przy problemach, którym towarzyszy niepewność. Algorytm min-max to metoda minimalizowania maksymalnych możliwych strat. Alternatywnie można je traktować jako maksymalizację minimalnego zysku (maximin). Twierdzenie to zostało ustanowione w XX wieku przez Johna von Neumanna, którego powiedzenie jest cytowane „Jak do tej pory widzę, nie mogłoby być żadnej teorii gier… bez tej teorii… Myślałem, że nic nie było warte publikowania, aż Teoria Minimax została udowodniona”

   b) Teoria min-max
      Odpowiednia strategia gracza 1. gwarantuje mu spłatę V niezależnie od strategii gracza 2. i podobnie gracz 2. może zagwarantować sobie spłatę -V. Nazwa Minimax pojawiła się, ponieważ każdy gracz minimalizuje maksymalną możliwą

spłatę dla drugiego – ponieważ gra jest grą o sumie zerowej, także maksymalizuje swoją minimalną spłatę. Do wizualizacji algorytmu min-max bardzo często korzysta się z drzewa binarnego, które dobrze obrazuje wpływ działań jednego gracza na drugiego i odwrotnie. Węzły drzewa reprezentują stan gry po wykonaniu ruchu przez jednego z graczy. Poziomy drzewa reprezentują wszystkie ruchy, jakie może wykonać jeden z graczy w danej turze. Każdy węzeł ma przypisaną liczbę wyrażającą wartość reprezentowanej pozycji z punktu widzenia gracza, dla którego zaczęliśmy konstruować drzewo. Zadaniem algorytmu jest teraz wyznaczenie ścieżki prowadzącej od stanu początkowego do tego ze stanów końcowych, który daje nam pewność, że nie zdobędziemy mniej jak x punktów, przy czym x jest największą z najmniejszych wartości jakie możemy zdobyć. Ostatnie stwierdzenie w ogólności sprowadza się do tego, że zdecydowanie lepiej dla gracza A wykonać taki ruch, który zagwarantuje mu, że po odpowiedzi gracza B straci jak najmniej, co jest esencją podejścia min-max.

c) Algorytm

Załóżmy istnienie dwóch graczy, którzy grają w warcaby. Będzie to protagonista i antagonista, obaj grający algorytmem min-max. Podczas wyboru akcji dla protagonisty zakłada się, że antagonista wybierze najlepszą dla siebie akcję i vice-versa. Aby algorytm dobrze interpretował kolejne ruchy graczy, stany końcowe grafu gry muszą mieć liczbowy opis wyniku rozgrywki (np. funkcje heurystyczne). Nie ma jednego słusznego sposobu numerowania wyników, należy jedynie nadać takie liczby rzeczywiste, aby im wynik końcowy rozgrywki był lepszy, tym wyższa liczba go opisywała. Ponieważ algorytm ten podpowie jakie akcje powinien wybrać antagonista i protagonista, dlatego za pomocą algorytmu min-max można przewidzieć końcowy wynik rozgrywki, jeśli obaj gracze będą grali najlepiej jak to możliwe.

Algorytm min-max jest procedurą wywoływaną rekurencyjnie począwszy od korzenia grafu stanu gry, czyli zastanego na początku procedury stanu na planszy. Dla węzła algorytm ten wykona:

- Jeśli węzeł oznacza koniec gry lub głębokość jest równa zeru, wtedy zwróci wynik równy opisowi tego węzła,
- Jeśli aktualny węzeł dotyczy ruchu protagonisty, wtedy wybierz tę akcję, która daje maksymalny (najlepszy dla protagonisty) wynik, a następnie sam zwróć ten maksymalny wynik. Inaczej mówiąc dla takiego węzła należy znaleźć taki węzeł potomny, który daje maksymalny wynik i samemu go zwrócić.
- Jeśli aktualny węzeł dotyczy ruchu antagonisty, wtedy wybierz tę akcję, która daje minimalny (najlepszy dla antagonisty) wynik, a następnie sam zwróć ten minimalny wynik. Inaczej mówiąc dla takiego węzła należy znaleźć taki węzeł potomny, który daje minimalny wynik i samemu go zwrócić.

Poniższy pseudokod ilustruje jak można by zaimplementować przedstawione wcześniej rozumowanie.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

6. Wykorzystane heurystyki
   Fukcja count_pieces() jest dosyć prostą funkcją heurystyczną, dodatkowo jest intuicyjna, ponieważ gracz naturalnie dąży do posiadania większej liczby pionków od przeciwnika: Do zadań funkcji należą:
   - liczenie liczby pionków każdego z graczy
   - zwykły pionek to 10 punktów
   - za każdą damkę przyznawano 20 punktów
   - funkcja w zależności od gracza zwraca różnicę – liczba „moich" pionków – liczba pionków przeciwnika

   Funkcja evaluate_move_forward() jej zadaniem jest premiowanie graczy za przesuwanie pionków do przodu. Im dalej są one od bazy tym lepiej.
   - Dla każdego z graczy premiowanie tylko dla zwykłych pionków celem szybszej promocji do statusu damki
   - Punkty kumulowane są jako numery kolejnych rzędów licząc od bazy danego gracza

   Funkcja count_save_checkers() – zadaniem tej funkcji oceny gry jest zliczanie dla każdego z graczy liczbę pionków, które są bezpieczne.
   - Bezpieczny pionek to taki, który nie sąsiaduje w kierunku ruchu z przeciwnikiem. Mówiąc inaczej za bezpieczny brany jest pionek, który nie może zostać zbity w następnej turze
   - Funkcja zwraca liczbę bezpiecznych pionków

7. Testy
   Zaimplementowana aplikacja została poddana testom, tak aby znaleźć najoptymalniejszą z przygotowanych heurystyk. Dwaj agenci rozgrywają między sobą 100 partii, dla przejrzystości rezultatów, korzystając z różnych kombinacji opisanych

wcześniej heurystyk. Celem tych testów jest znalezienie najlepszej heurystyki, czyli takiej która pozwoli zmaksymalizować liczbę zwycięstw na dystansie 100 gier. Poniżej w tabeli przedstawiono podsumowanie uzyskanych rezultatów (gracz Blue i Red odzwierciedlają kolory pionków użytych w implementacji i dla ułatwienia będę się posługiwał tymi nazwami).

| Liczba wygranych gier | | Użyte funkcje i gracz zaczynający |
|---|---|---|
| Blue | Red | |
| 46 | 54 | (B): count_pieces()<br>(R): count_pieces()<br>Zaczyna: Blue |
| 15 | 85 | (B): count_pieces(), evaluate_move_forward()<br>(R): count_pieces()<br>Zaczyna: Blue |
| 17 | 83 | (B): count_pieces(), evaluate_move_forward()<br>(R): count_pieces()<br>Zaczyna: Red |
| 24 | 76 | (B): count_pieces(), evaluate_move_forward(),<br>count_save_checkers()<br>(R): count_pieces()<br>Zaczyna: Blue |
| 17 | 83 | (B): count_pieces(), evaluate_move_forward(),<br>count_save_checkers()<br>(R): count_pieces()<br>Zaczyna: Red |
| 79 | 21 | (B): evaluate_move_forward()<br>(R): count_save_checkers()<br>Zaczyna: Blue |
| 12 | 88 | (B): evaluate_move_forward(), count_save_checkers()<br>(R): count_pieces(), count_save_checkers()<br>Zaczyna: Blue |
| 24 | 76 | (B): count_pieces(), evaluate_move_forward()<br>(R): count_pieces(), count_save_checkers()<br>Zaczyna: Blue |

Na początku obaj agenci grali wykorzystując funkcje count_pieces(). Po wykonaniu kilku testów po 100 gier można zauważyć, że rezultaty uzyskane przez obu są dosyć bliskie siebie tzn. żaden z nich znacząco nie wygrywa częściej co nie powinno dziwić, ponieważ korzystają z tej samej funkcji heurystycznej. Co jednak jest warte uwagi widać delikatną przewagę na korzyść gracza który jako drugi wykonuje swój ruch.
Następnie agent Blue rozegrał partie z dodatkiem funkcji evaluate_move_forward(). Co ciekawe nie uzyskał lepszych rezultatów, co więcej niezależnie od tego kto rozpoczynał grę nie udało mu się przekroczyć 20% wygranych partii. Zmiana rozpoczynającego, kiedy to gracz Blue grał jako drugi, zwiększyła liczbę jego zwycięstw tylko o 2 punkty procentowe.

Dodanie kolejnej funkcji heurystycznej graczowi Blue zwiększyła jego skuteczność, ale dalej jest to znacząco gorszy rezultat, niż ten, który osiągnął gracz Red, korzystając tylko z jednej heurystyki. Analizując dwa testy gdy gracz Blue jest, wydawało by się, uzbrojony po zęby widać zaprzeczenie regule iż gracz zaczynający jako drugi ma przewagę, ponieważ zaczynając wygrał 24% partii a tylko 17% ruszając jako drugi.

Porzucając chwilowo funkcję count_pieces(), porównano funkcje evaluate_move_forward() z funkcją count_save_checkers(). Jak się okazało znacznie bardziej opłacało się poruszać do przodu niż za wszelką cenę unikać pionów przeciwnika. Ciężko ocenić czy w warcabach lepiej jest przeć do przodu, ponieważ liczba gier i testów była być może niewystarczająca albo zastosowany algorytm powinien przejść pewne modyfikacje.

Kolejno przetestowano funkcję evaluate_move_forward() z count_save_checkers() przeciwko count_pieces() i count_save_checkers(). Tutaj widać największą dominacje między zastosowanymi podejściami spośród wszystkich zastosowanych. Przyczyną może być charakterystyka użytych funkcji. Ślepe poruszanie się do przodu nie ma szans wobec zastosowaniu zliczania pionów i wybierania najkorzystniejszej opcji, prowadzącej do zwycięstwa, jaką jest zbicie wszystkich pionów rywala.

W ostatecznym teście wykorzystano dla obu graczy funkcję count_pieces(). Gracz Blue dodatkowo wykorzystał evaluate_move_forward(), a gracz czerwony count_save_checkers(). W tym przypadku gracz czerwony wygrał 76 partii ze 100 rozegranych, co pokazało, że bardziej opłacalne (w tej formie jak zostało to zaimplementowane) wykonywać bezpieczne ruchy z uwzględnieniem liczby pionków niż poruszać się do przodu, zajmując bardziej centralne pola planszy.

8. Wnioski końcowe

Biorąc pod uwagę wszystkie wykonane testy okazuje się, że najbardziej efektywna jest funkcja heurystyczna count_pieces(). Ten fakt jest dość interesujący, ponieważ jest ona jedną z bardziej podstawowych i intuicyjnych funkcji heurystycznych, które są stosowane do gier o sumie zerowej. Jak widać jej prostota wcale nie przeszkadza w osiąganiu dobrych rezultatów. W celu polepszenia efektywności agentów warto było by przeprowadzić większą liczbę testów z lepiej dostosowanymi funkcjami heurystycznymi. Pomysłem na rozwinięcie programu jest zastosowanie algorytmu min-max z odcięciem alfa-beta, które pozwala na przeglądanie drzewa gry na znacznie większej głębokości.