Algorytmy Minimax i alfa-beta w grze Halma

Mikołaj Jastrzębski (266858)

Maj 2024

Spis treści

1 Opis problemu

Projekt realizowany w ramach kursu "Sztuczna Inteligencja i Inżynieria Wiedzy"miał na celu praktyczne zapoznanie się i zastosowanie algorytmów gier dwuosobowych o sumie zerowej, a konkretnie algorytmu Minimax i jego optymalizacji przy użyciu cięć alfa-beta. Wybrana gra to Halma, klasyczna gra planszowa, gdzie dwaj gracze starają się przesunąć wszystkie swoje pionki na przeciwną stronę planszy.

W ramach projektu, głównym zadaniem było stworzenie programu, który potrafi grać w Halmę, wykorzystując zaawansowane techniki algorytmiczne do analizy i podejmowania decyzji w trakcie gry. Zastosowane metody miały za zadanie nie tylko symulować pojedynczą partię, ale także ocenić skuteczność strategii przy różnych konfiguracjach gry i typach ruchów przeciwnika.

W dalszej części sprawozdania szczegółowo omówiono zastosowane algorytmy, metodę ich implementacji oraz wyniki eksperymentalne, które ukazują efektywność wybranych podejść w kontekście grania w Halmę. Raport zawiera również analizę teoretyczną zastosowanych algorytmów, formalne sformułowanie problemu, opis stanu gry oraz drzewa decyzyjnego, co pozwoli na głębsze zrozumienie mechanizmów rządzących zaimplementowanym rozwiązaniem.

2 Wykorzystane biblioteki oraz język programowania

W projekcie użyto języka programowania TypeScript, który umożliwił wygodne pisanie zaawansowanego interfejsu użytkownika. Wybór ten pozwolił na łatwiejsze testowanie algorytmów i zrozumienie problemu przez interaktywne eksperymentowanie z różnymi strategiami gry.

Do stworzenia aplikacji wykorzystano następujące główne biblioteki i narzędzia:

- **React** - biblioteka służąca do budowania interfejsu użytkownika, znana ze swojej efektywności oraz elastyczności w tworzeniu dynamicznych aplikacji webowych.
- **Zustand** - lekka i prostsza alternatywa dla Redux, używana do zarządzania stanem w aplikacjach React.
- **Vite** - nowoczesne narzędzie do budowania, które znacznie przyspiesza proces tworzenia aplikacji dzięki wykorzystaniu natywnych modułów ES.
- **TypeScript** - język programowania rozwijający JavaScript o statyczne typowanie, co przyczynia się do większej przejrzystości kodu oraz łatwiejszej detekcji błędów na etapie kompilacji.
- **ESLint** - narzędzie do statycznej analizy kodu, które pomaga utrzymać jednolity styl kodowania oraz wychwytywać błędy i potencjalne problemy.
- **TailwindCSS** - framework CSS służący do szybkiego stylizowania aplikacji bez potrzeby pisania dużej ilości własnego CSS.

3 Zdefiniowanie stanu gry i funkcji generującej możliwe ruchy

3.1 Stan gry

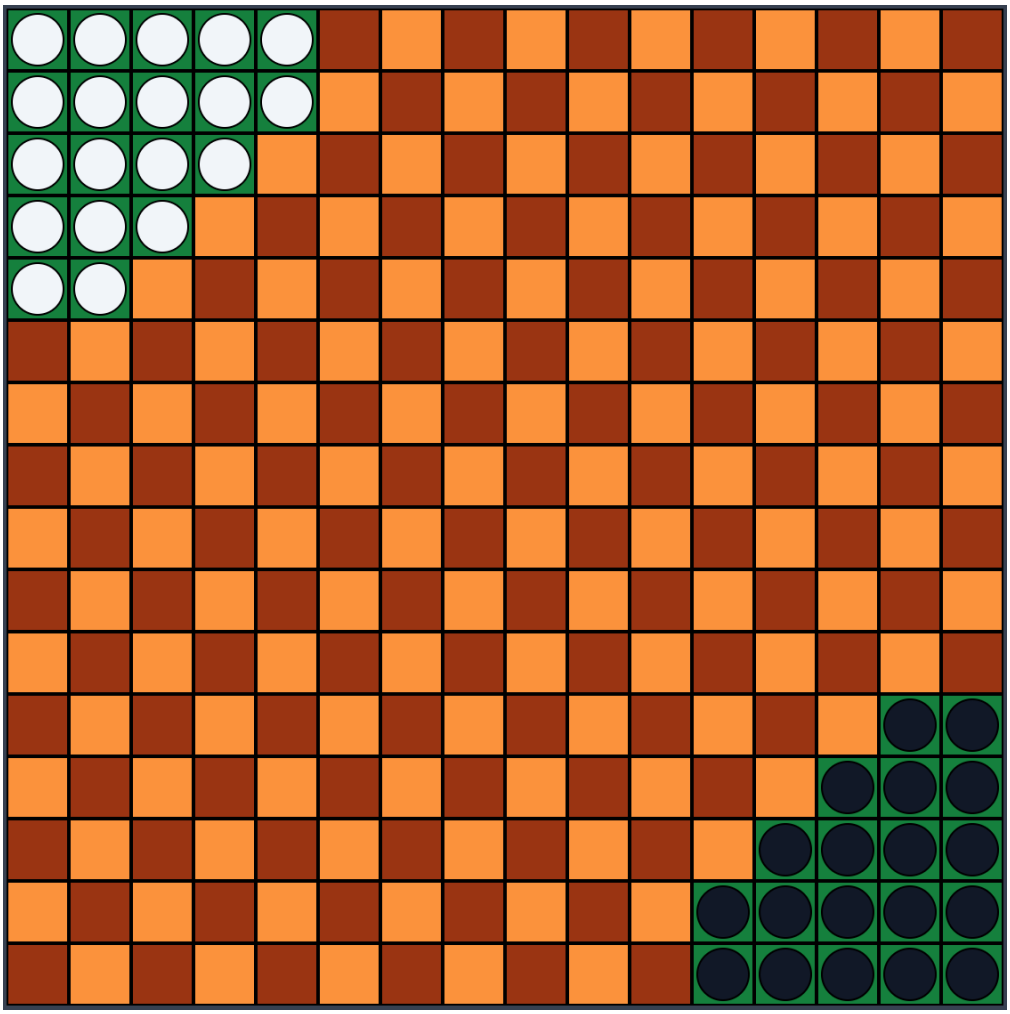
Halma jest grą planszową rozgrywaną na szachownicy o wymiarach 16x16, co daje łącznie 256 pól. Na początku gry każdy z dwóch graczy posiada swoje piony rozmieszczone w przypisanych narożnikach planszy. Gracz pierwszy w lewym górym rogu, Gracz drugi po przeciwnej stornie planszy. Każdy z graczy posiada 19 pionków. Celem gry jest przeprowadzenie wszystkich pionów do przeciwległego narożnika w możliwie najkrótszym czasie.

Stan gry w każdym momencie można zdefiniować jako macierz 16x16, gdzie każde pole może przyjąć jedną z trzech wartości:

- PLAYER.None - pole puste,
- PLAYER.Player1 - pole zajęte przez pion gracza pierwszego,

- PLAYER.Player2 - pole zajęte przez pion gracza drugiego.

Każde pole na planszy można identyfikować przez parę współrzędnych (x, y), gdzie x i y to liczby od 0 do 15. Stan gry jest dynamicznie aktualizowany w odpowiedzi na każdy wykonany ruch przez graczy.



Rysunek 1: Początkowy stan gry w Halma na planszy 16x16. Na planszy widoczne są dwa zestawy pionów: białe i czarne, każdy z graczy rozpoczyna grę z umieszczeniem swoich pionów w przeciwległych bazach.

3.2 Funkcja generująca możliwe ruchy

Implementacja funkcji generującej ruchy w programie polega na analizie każdego piona danego gracza, sprawdzaniu dostępnych sąsiadujących pól oraz potencjalnych skoków. Dla każdego piona generowana jest lista możliwych ruchów, które zawierają zarówno pojedyncze przesunięcia, jak i skomplikowane sekwencje skoków.

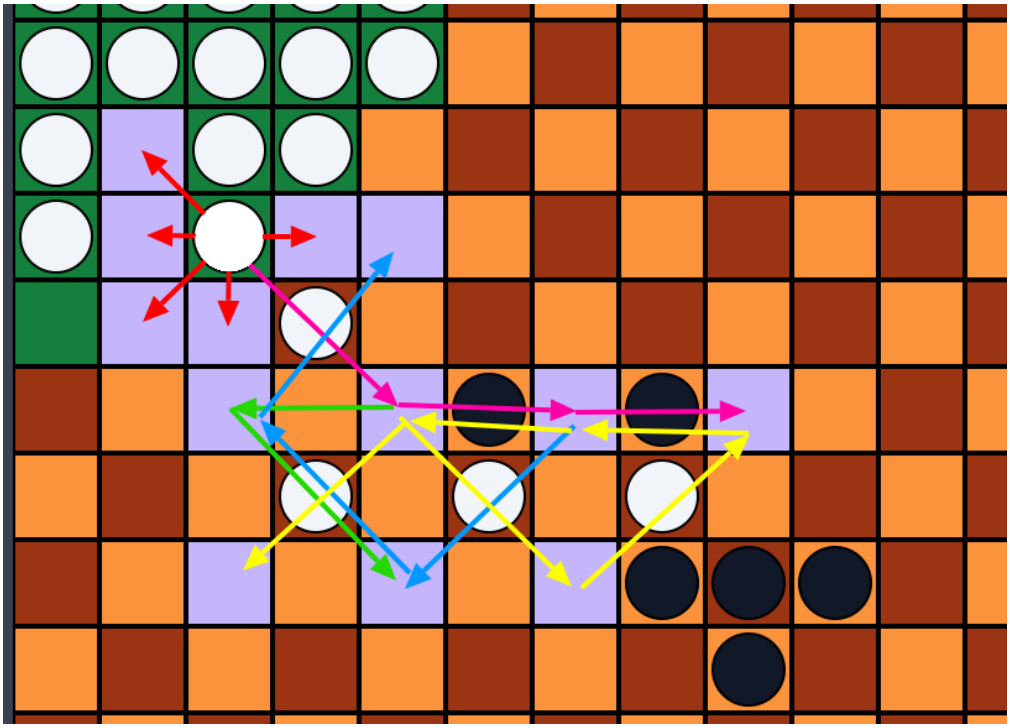
- **Parametry funkcji:**
  - `x, y` - współrzędne pionka na planszy.
  - `board` - dwuwymiarowa tablica reprezentująca stan planszy, gdzie każdy element jest typu `TypeTile`, który zawiera informacje o zawodniku (graczu) zajmującym dane pole.
  - `moves` - lista aktualnie zgromadzonych możliwych ruchów (domyślnie pusta).
  - `doubleJump` - flaga określająca, czy obecnie rozpatrywany ruch jest częścią serii skoków (domyślnie `false`).

– `visited` - zbiór odwiedzonych pól w formie łańcuchów znaków, zapobiegający ponownemu przetwarzaniu tych samych pól (domyślnie pusty zbiór).

- **Działanie funkcji:**

1. Na początku funkcja zbiera wszystkie wektory w których pionek może się poruszyć.
2. Dla każdego z kierunków ruchu, obliczane są nowe współrzędne (`nx`, `ny`).
3. Sprawdzana jest ważność nowej pozycji za pomocą funkcji `isValidMove`. Jeśli ruch jest nieważny (np. wyjście poza planszę), iteracja jest kontynuowana.
4. Jeżeli pole o współrzędnych (`nx`, `ny`) jest puste i nie jest to część skoku, dodaje się tę pozycję do listy możliwych ruchów, a pole jest oznaczane jako odwiedzone.
5. Jeśli pole (`nx`, `ny`) jest zajęte, funkcja próbuje wykonać skok na kolejne pole w tym samym kierunku (`nx2`, `ny2`).
6. Jeżeli skok jest możliwy (pole docelowe jest puste i nie było odwiedzane), ruch skokowy jest dodawany do listy ruchów, pole jest oznaczane jako odwiedzone, a funkcja jest rekurencyjnie wywoływana dla nowej pozycji z aktywowaną flagą `doubleJump`.

Dzięki rekurencyjnej naturze funkcji, możliwe jest generowanie złożonych sekwencji skoków, co jest kluczowe w grach, gdzie strategia i planowanie dalekosiężnych ruchów mogą decydować o wyniku partii.



Rysunek 2: Możliwe ruchy dla pionka gracza pierwszego w trakcie gry.

4 Zbudowanie zbioru heurystyk oceny stanu gry

Heurystyki służą do oceny jakości danego stanu gry, co umożliwia algorytmowi podejmowanie informowanych i optymalnych decyzji w sposób, który jest zdecydowanie szybszy niż pełne przeszukiwanie przestrzeni stanów. Heurystyka w kontekście halmy jest funkcją oceniającą, która przypisuje wartości liczbowe do różnych konfiguracji na planszy, reprezentując ich "dobre"lub "złe"aspekty względem celu gry.

Przykładowo, w Halma zastosowanie heurystyk umożliwia algorytmowi minmax + alfa-beta szybsze przeszukiwanie i bardziej efektywne podejmowanie decyzji poprzez skoncentrowanie uwagi na najbardziej obiecujących ścieżkach decyzyjnych bez konieczności rozważania każdego możliwego ruchu.

Heurystyki zostały zaprojektowane w celu przypisywania wartości liczbowych stanom gry, które mogą być interpretowane jako korzystne dla jednego z graczy lub wskazujące na potencjalny remis. Wartość zwracana przez funkcję heurystyczną jest wskaźnikiem przewagi strategicznej — jeśli jest dodatnia, sugeruje to, że stan gry jest korzystniejszy dla gracza pierwszego. Natomiast wartość ujemna wskazuje, że lepszą pozycję na planszy ma gracz drugi. Dzięki temu, algorytmy podejmujące decyzje mogą efektywnie ocenić i porównać możliwe przebiegi gry, kierując się ku najbardziej obiecującym ruchom z punktu widzenia obecnego gracza.

4.1 Strategia losowa

Pierwszą heurystyką jest `Strategia losowa`, która generuje losową wartość dla każdego możliwego ruchu. Choć sama w sobie nie jest skuteczną strategią do gry, służy jako baza porównawcza dla bardziej zaawansowanych metod.

4.2 Strategia przejęcia bazy przeciwnika

Heurystyka koncentruje się na liczbie pionów znajdujących się w bazie przeciwnika. Jest to kluczowe dla osiągnięcia celu gry, jakim jest przemieszczenie wszystkich pionów do przeciwległego narożnika. Heurystyka ta jest szczególnie użyteczna w późnych etapach gry, gdy gracze zbliżają się do celu.

4.3 Strategia obrony bazy gracza

`Strategia obrony bazy gracza` odnosi się do liczby pionów przeciwnika, które znajdują się w bazie gracza. Strategia ta skupia się na obronie i próbie blokowania ruchów przeciwnika, co może zapobiegać szybkiemu zakończeniu gry przez przeciwnika.

4.4 Strategia odległości euklidesowej do bazy przeciwnika

Heurystyka wykorzystuje odległość Euklidesową od pionów do narożnika przeciwnika, aby ocenić postęp gracza w przemieszczaniu pionów na planszy. Mniejsza wartość odległości wskazuje na lepsze pozycje pionów gracza, co przyczynia się do skuteczniejszego nacisku na przeciwnika.

4.5 Strategia wielu skoków

Strategia wielu skoków ocenia możliwość wykonania wielokrotnych skoków w kierunku bazy przeciwnika. Skoki te są cenne, gdyż pozwalają na szybkie przemieszczenie pionów przez planszę, omijając blokady przeciwnika.

4.6 Strategia mobilności

`Strategia mobilności` mierzy liczbę pojedynczych skoków, które piony mogą wykonać w kierunku bazy przeciwnika. Wyższa mobilność może zwiększać ciśnienie na przeciwnika i poprawiać pozycje strategiczne gracza.

4.7 Strategie łączone

`Strategie łączone` łączy powyższe heurystyki, wykorzystując ważone wyniki z poszczególnych ocen. Strategia ta dostosowuje się do bieżącej sytuacji na planszy, maksymalizując potencjał ofensywny oraz defensywny gracza poprzez adaptacyjne podejście do każdego ruchu. W przypadku dynamicznego określania aktualnej heurystyki brane pod uwagę są wagi, które dynamicznie zmieniają wagę danej heurystyki w `Strategie łączone` w zależności od stanu gry.

Każda z tych heurystyk może być używana indywidualnie lub w kombinacjach, co pozwala na tworzenie zaawansowanych strategii dostosowanych do specyfiki danego przebiegu gry w Halma. Zastosowanie takiego

zbioru heurystyk umożliwia algorytmowi lepsze przewidywanie i planowanie ruchów, znacząco wpływając na ostateczny wynik partii.

4.8 Dynamiczne dostosowywanie strategii

Mechanizm dynamicznego dostosowywania opiera się na modyfikacji wag poszczególnych heurystyk w odpowiedzi na rozwijającą się grę. Wagi te są kluczowe dla oceny danego stanu planszy przez funkcję heurystyczną. W sytuacjach, gdy gracz jest blisko zwycięstwa, waga strategii oceniającej bliskość pionków do bazy przeciwnika jestzwiększona wraz z wagą przejęcia bazy przeciwnika, aby promować agresywne dążenie do celu. Z kolei, gdy przeciwnik ma przewagę, waga heurystyk skupiających się na obronie i zabezpieczaniu własnych pionków przed zajęciem może wzrosnąć, aby zminimalizować ryzyko przegranej.

5 Implementacja metody Minimax

Algorytm Minimax jest kluczowym narzędziem w teorii gier dwuosobowych o sumie zerowej, gdzie gracze naprzemiennie podejmują ruchy. Jest to metoda optymalizacyjna, która pozwala graczom maksymalizować minimalny zysk możliwy do osiągnięcia z danego stanu gry, biorąc pod uwagę możliwe ruchy przeciwnika. W kontekście gry Halma, algorytm ten jest wykorzystywany do analizowania potencjalnych ruchów i strategii, przewidując możliwe reakcje przeciwnika, co umożliwia podjęcie najkorzystniejszej decyzji.

5.1 Drzewo decyzyjne i głębokość algorytmu

Drzewo decyzyjne generowane przez algorytm Minimax rozwija się poprzez iteracyjne analizowanie możliwych ruchów dla obu graczy, począwszy od aktualnego stanu planszy. Każdy węzeł w drzewie reprezentuje potencjalny stan gry, a krawędzie odpowiadają przejściom między tymi stanami w wyniku ruchów graczy. Algorytm bada te możliwości, wypracowując ścieżkę decyzyjną od korzenia (obecnego stanu) do liści (potencjalne zakończenia gry lub osiągnięcia głębokości zerowej).

Głębokość drzewa decyzyjnego, czyli liczba poziomów rozważanych ruchów, jest kluczowym parametrem wpływającym na skuteczność i złożoność obliczeniową algorytmu. Dla gry Halma, nawet głębokość równa czterem może oznaczać konieczność analizy ponad 500 milionów potencjalnych stanów planszy, co stanowi ogromne wyzwanie obliczeniowe. Wynika to z eksponencjalnego wzrostu liczby możliwych stanów wraz z dodaniem każdego nowego poziomu głębokości.

5.2 Ograniczenia i optymalizacje

Z powodu ogromnej liczby potencjalnych ruchów w grze Halma, bezpośrednia implementacja algorytmu Minimax z pełną głębokością drzewa byłaby awykonalna obliczeniowo z wyjątkiem komputerów kwantowych. W praktyce głębokosc algorytmu szacuje się na maksymalnie równą 4 oraz stosuje się różne techniki optymalizacyjne, takie jak cięcia alfa-beta, które pomagają ograniczyć przestrzeń przeszukiwań przez eliminowanie oczywiście niekorzystnych ruchów i gałęzi drzewa, gdzie dalsza analiza nie zmieni już wyniku oceny.

Dodatkowo, stosowane mogą być różne heurystyki oceny stanu gry, które umożliwiają wstępne szacowanie wartości poszczególnych stanów bez konieczności pełnego przeszukiwania drzewa. Dzięki temu, algorytm może skupić się na najbardziej obiecujących scenariuszach, znacząco redukując złożoność obliczeniową i czas potrzebny do podjęcia decyzji.

5.3 Działanie zaimplementowanego algorytmu Minimax

Zaimplementowany algorytm Minimax zaczyna od oceny aktualnego stanu planszy, stosując funkcję `minimax`, która jest wywoływana rekurencyjnie. Początkowo algorytm wywoływany jest z określoną głębokością przeszukiwania, co w tym przypadku wynosi trzy. Ta wartość głębokości jest kompromisem między dokładnością a wydajnością obliczeniową i została dobrana tak, aby umożliwić efektywną analizę potencjalnych ruchów bez nadmiernego obciążania zasobów komputerowych.

- **Ocena terminalna:** Jeśli algorytm osiągnie maksymalną głębokość rekursji (w tym przypadku 3) lub jeżeli na planszy zostanie wykryty stan końcowy gry, tj. jeden z graczy wygrał, algorytm zwraca wartość heurystyczną dla danego stanu gry. Wartość ta jest obliczana przez funkcję `Strategii łączonej`, która integruje różne heurystyki, aby oszacować korzyść dla danego gracza w zależności od obecnego rozkładu pionów na planszy.
- **Generowanie ruchów:** Funkcja `getChildren` jest używana do generowania wszystkich możliwych stanów planszy wynikających z legalnych ruchów, które gracz może wykonać w danym stanie gry. Każdy z tych stanów jest potencjalnym dzieckiem węzła w drzewie decyzyjnym i jest oceniany przez rekurencyjne wywołania funkcji `minimax`.
- **Maksymalizacja i minimalizacja:** W zależności od tego, czy tura jest tury gracza pierwszego, czy drugiego, algorytm stara się maksymalizować lub minimalizować zwracaną wartość heurystyczną. Dla gracza pierwszego (Player.PLAYER1), algorytm szuka ruchu, który maksymalizuje wynik, podczas gdy dla gracza drugiego (Player.PLAYER2) — minimalizuje. To odzwierciedla klasyczne podejście Minimax, gdzie jeden gracz próbuje maksymalizować swoje korzyści, a przeciwnik minimalizować te korzyści.
- **Wybór najlepszego ruchu:** Po wygenerowaniu i ocenie wszystkich możliwych ruchów, algorytm wybiera ruch z najwyższą (dla Player.PLAYER1) lub najniższą (dla Player.PLAYER2) wartością heurystyczną i zwraca ten ruch jako optymalny dla aktualnego stanu gry.

Zaimplementowany algorytm Minimax, dzięki swojej zdolności do przewidywania i oceny przyszłych możliwości, pozwala na strategiczne podejmowanie decyzji, znacznie przewyższając prostą analizę bezpośrednich ruchów dostępnych dla gracza. Użycie heurystyk, takich jak `CustomStrategy`, dodatkowo wzbogaca proces decyzyjny, zapewniając bardziej zróżnicowaną i adaptacyjną reakcję na zmieniające się warunki gry.

6 Implementacja algorytmu alfa-beta cięcia

Algorytm alfa-beta cięcia jest zaawansowaną wersją algorytmu Minimax, która polega na dodatkowej optymalizacji przeszukiwanego drzewa decyzyjnego. Dzięki wprowadzeniu dwóch parametrów, α i β, algorytm jest w stanie pomijać badanie gałęzi drzewa, które nie mają wpływu na ostateczny wybór ruchu. Ta metoda znacznie redukuje liczbę stanów, które muszą być oceniane w trakcie działania algorytmu, co pozwala na szybsze i bardziej efektywne przeszukiwanie przestrzeni możliwych ruchów.

6.1 Mechanizm alfa-beta cięcia

Alfa-beta cięcia wykorzystuje dwie wartości:

- α - najlepsza już zbadana opcja dla maksymalizującego gracza na ścieżce do korzenia. Oznacza to najgorszy wynik, na który musi się zgodzić gracz maksymalizujący.
- β - najlepsza już zbadana opcja dla minimalizującego gracza na ścieżce do korzenia. Jest to najgorszy wynik, na który musi się zgodzić gracz minimalizujący.

Te dwa parametry są aktualizowane w miarę przeszukiwania drzewa i wykorzystywane do przycinania gałęzi, które nie mogą wpłynąć na ostateczny wybór ruchu, zgodnie z zasadą, że jeśli istnieje lepsza opcja na wyższym poziomie drzewa, dalsze gałęzie mogą być pominięte.

6.2 Proces decyzyjny algorytmu

Podczas gdy algorytm przechodzi przez drzewo gry:

- Dla gracza maksymalizującego ($Player.PLAYER1$), każdy ruch jest oceniany, a wartość α jest aktualizowana, jeżeli analiza ruchu prowadzi do lepszego wyniku. Jeśli w dowolnym momencie β stanie się mniejsze lub równe α, dalsze przeszukiwanie obecnego węzła zostaje przerwane (tzw. cięcie β).
- Analogicznie, dla gracza minimalizującego ($Player.PLAYER2$), wartość β jest aktualizowana, a gdy α przekroczy β, przeszukiwanie jest również przerywane (cięcie α).

Ten mechanizm pozwala znacząco zredukować liczbę ruchów do rozważenia, co jest szczególnie ważne w grze takiej jak Halma, gdzie możliwości ruchów i ich konsekwencji może być ogromna.

7 Napotkane problemy implementacyjne

Podczas procesu implementacji algorytmu Minimax oraz alfa-beta cięcia w grze Halma, napotkano szereg problemów, które wpłynęły na skuteczność i wydajność algorytmu. Poniżej opisano dwa główne problemy, które miały znaczący wpływ na zachowanie algorytmu oraz strategię gry.

7.1 Problem niedostępności wolnego miejsca w bazie

Jednym z napotkanych problemów była sytuacja, w której ostatni pionek nie mógł dostać się do jedynego wolnego miejsca w bazie przeciwnika. Wynikało to z faktu, że dostępne miejsce było zablokowane przez inne pionki, które nie przemieszczały się, aby umożliwić dostęp, gdyż ruch taki zwiększałby ich odległość euklidesową od bazy. Problem ten był szczególnie widoczny przy ustawieniu głębokości przeszukiwania na 2. Przy takiej głębokości algorytm nie był w stanie przewidzieć korzystnych przesunięć, które w dłuższej perspektywie umożliwiłyby ostatniemu pionkowi dostęp do bazy.

7.2 Różnice w wynikach w zależności od głębokości algorytmu

Kolejnym interesującym problemem była obserwacja, że zmiana głębokości przeszukiwania z 2 na 3 miała znaczący wpływ na wyniki gry, skutkując przewagą białych pionków przy głębokości 2 i czarnych przy głębokości 3. Przyczyny tego zjawiska mogą być wielorakie:

- **Złożoność obliczeniowa i heurystyki**: Głębsze przeszukiwanie umożliwia algorytmowi lepsze rozpoznanie potencjalnych strategii przeciwnika oraz bardziej zaawansowane planowanie własnych ruchów. Przy głębokości 3 algorytm może "dostrzec"ruchy, które początkowo wydają się niekorzystne, ale prowadzą do znaczących korzyści w kolejnych turach.
- **Struktura planszy i rozmieszczenie pionków**: Różnice w wydajności mogą wynikać także z asymetrii w rozmieszczeniu pionków na początku gry, co wpływa na dostępność poszczególnych strategii dla różnych głębokości przeszukiwania.

Analiza i zrozumienie tego problemu wymaga dalszych badań i eksperymentów, być może z zastosowaniem innych wariantów heurystyk lub modyfikacją samej strategii przeszukiwania.

8 Pomiary

Aby ocenić efektywność zastosowanych strategii algorytmicznych w grze Halma, przeprowadzono serię pomiarów, które miały na celu porównanie różnych podejść decyzyjnych. W szczególności, skoncentrowano się na analizie i porównaniu wydajności algorytmu Minimax, jego rozbudowanej wersji z cięciem alfa-beta, oraz zastosowania prostszych strategii heurystycznych bez głębokiego przeszukiwania przestrzeni stanów.

W ramach badań przeprowadzono porównania obejmujące:

- Wydajność algorytmu Minimax w porównaniu do jego wersji z optymalizacją alfa-beta przy różnych głębokościach przeszukiwania.
- Ocena wpływu zwiększenia głębokości przeszukiwania na skuteczność i czas wykonania algorytmów.
- Porównanie czysto heurystycznych strategii z algorytmami wykorzystującymi przeszukiwanie drzew decyzyjnych, aby ocenić, czy bardziej złożone podejścia przynoszą wymierne korzyści w kontekście gry.

8.1 Porównanie prędkości algorytmów

Poniższa tabela przedstawia porównanie czasów wykonania ruchów oraz liczby ruchów wykonanych przez algorytmy Minimax i Alpha-Beta przy różnych ustawieniach głębokości i zastosowaniu optymalizacji ruchów jedynie w stronę przeciwnika.

Algorytm	**Głębokość**	**Optymalizacja**	**Średni czas ruchu (ms)**	**Liczba ruchów**
Minimax	2	Nie	366.67	270
Minimax	2	Tak	82.71	290
Minimax	3	Tak	73314.45	262
Alpha-Beta	2	Nie	160.72	302
Alpha-Beta	2	Tak	39.63	250
Alpha-Beta	3	Nie	3219.94	272
Alpha-Beta	3	Tak	1449.12	272

Tabela 1: Porównanie wydajności algorytmów Minimax i Alpha-Beta

Jako optymalizację rozumie się ogarnieczenie możliwości znajdywania ruchów dla pionka jedynie w kierunku nieoddalającym go od bazy przeciwnika. Zastosowanie optymalizacji, polegającej na ograniczeniu ruchów do tych prowadzących w stronę przeciwnika, znacząco przyspiesza działanie obu algorytmów. Na przykład, dla algorytmu Minimax przy głębokości 2, optymalizacja zmniejsza średni czas ruchu z 366.67 ms do 82.71 ms. Podobnie, w przypadku algorytmu Alpha-Beta, redukcja czasu jest jeszcze bardziej wyraźna, gdzie przy tej samej głębokości, czas spada z 160.72 ms do 39.63 ms. Wynika to z mniejszej liczby stanów do rozważenia, co skutkuje szybszym dochodzeniem do decyzji.

Zwiększenie głębokości przeszukiwania z 2 do 3 prowadzi do znacznego wzrostu czasu wykonania ruchu, co jest szczególnie widoczne w przypadku algorytmu Minimax, gdzie średni czas ruchu wzrasta do 73314.45 ms. Podobnie, dla algorytmu Alpha-Beta, czas wzrasta z 160.72 ms do 3219.94 ms bez optymalizacji i do 1449.12 ms z optymalizacją. Wzrost ten jest spowodowany eksponencjalnym wzrostem liczby możliwych stanów gry do analizy przy głębszym przeszukiwaniu.

Algorytm Alpha-Beta wykazuje znacznie lepsze wyniki czasowe w porównaniu do Minimax przy każdej konfiguracji. Dzięki cięciom alfa-beta, które eliminują niepotrzebne gałęzie drzewa decyzyjnego, algorytm jest w stanie osiągnąć szybsze czasy odpowiedzi przy zachowaniu tej samej głębokości przeszukiwania. Optymalizacja ta jest szczególnie wartościowa przy większych głębokościach, gdzie różnica w czasach staje się bardziej wyraźna.

8.2 Porównanie heurystyk

Analiza heurystyk stosowanych w algorytmach decyzyjnych dla gry Halma wykazała, że pojedyncze heurystyki, z wyjątkiem heurystyki oceniającej odległość do bazy przeciwnika, często nie dostarczają wystarczających informacji, aby samodzielnie skutecznie kierować decyzjami algorytmu. W związku z tym, znacząco większą efektywność wykazuje strategia polegająca na kombinowaniu różnych heurystyk z odpowiednio dobranymi wagami.

Indywidualne heurystyki, takie jak liczba pionów w bazie przeciwnika (Occupy Enemy Base) czy możliwość wykonania podwójnego skoku (Double Jump), choć użyteczne w specyficznych scenariuszach, nie zapewniają kompleksowego obrazu sytuacji na planszy. Ich ograniczenie polega na skupieniu się na pojedynczym aspekcie gry, co może prowadzić do przeoczenia istotnych zagrożeń lub szans pochodzących z innych elementów strategii gry.

Heurystyka odległości do bazy przeciwnika okazała się być najbardziej wartościową ze względu na bezpośredni wpływ na główny cel gry — przesunięcie wszystkich pionów do przeciwnego narożnika. Jednakże, stosowanie jej samodzielnie może nie zawsze prowadzić do optymalnych decyzji w kontekście bardziej złożonych scenariuszy taktycznych.

Najlepsze wyniki osiągnięto stosując kombinację wszystkich dostępnych heurystyk:

- Odległość do bazy przeciwnika — 40%
- Liczba pionów w bazie przeciwnika — 40%

- Możliwość wykonania podwójnego skoku — 10%
- Zablokowanie ruchów przeciwnika — 5%
- Mobilność pionów — 5%

Taka zbalansowana kombinacja heurystyk pozwala na bardziej kompleksową ocenę stanu gry, gdzie każdy aspekt jest odpowiednio uwzględniony, co prowadzi do bardziej strategicznych i przemyślanych decyzji.

9 Podsumowanie

Projekt skoncentrowany był na praktycznym zastosowaniu i ocenie algorytmów Minimax oraz alfa-beta cięcia w kontekście gry planszowej Halma. Celem było nie tylko zaimplementowanie efektywnego systemu decyzyjnego, ale także dogłębne zrozumienie i ocena różnych strategii oraz heurystyk pod kątem ich skuteczności w dynamicznym środowisku gry.

Przez proces implementacji i testowania, projekt ujawnił znaczące wyzwania implementacyjne oraz różnice w wydajności między zastosowanymi metodami:

- Algorytmy Minimax i alfa-beta wykazały się znaczącymi różnicami w czasie wykonania, szczególnie przy różnych poziomach głębokości, gdzie algorytm alfa-beta cięcia zapewnił lepszą wydajność dzięki efektywniejszemu przeszukiwaniu przestrzeni stanów.
- Heurystyki, zwłaszcza te skoncentrowane na odległości do bazy przeciwnika, okazały się kluczowe dla oceny stanów gry. Jednak ich izolowane zastosowanie bywało niewystarczające, co podkreśliło wartość kombinacji wielu heurystyk z odpowiednio dostosowanymi wagami w celu osiągnięcia najlepszej wydajności.
- Dynamiczne dostosowywanie strategii, bazujące na zmieniających się warunkach gry, umożliwiło algorytmom adaptację i skuteczniejsze reagowanie na ruchy przeciwnika, co jest istotne w grach o wysokim stopniu nieprzewidywalności i zmienności.

Napotkane problemy, takie jak ograniczenia dostępu do kluczowych pól przez inne pionki, czy różnice w wynikach w zależności od głębokości przeszukiwania, zwróciły uwagę na potrzebę dalszego badania i optymalizacji algorytmów. Wyniki te wskazują na potencjalne obszary do poprawy, zarówno w strategiach przeszukiwania, jak i w podejściu do dynamicznego dostosowywania parametrów algorytmu.