Karol Bonenberg

Laboratorium Sztucznej Inteligencji

Projekt Delta

1. Historia

Delta to gra logiczna stworzona przez włoskiego szachistę, Fabiego Forzoni. Jest to stosunkowo mało popularna i znana gra planszowa, pierwszy raz pojawiła się w Internecie około roku 1999.

2. Opis gry

Plansza złożona jest z 5x5 pól oraz maksymalnie 9 pionków dla każdego z graczy. Początkowy stan gry to pusta plansza.

Grę zawsze rozpoczyna gracz czarny, umieszczając pionek w dowolnym wybranym przez siebie miejscu. Następnie gracz biały wykonuje ruch, wybierając w ten sam sposób dowolne puste pole na planszy.

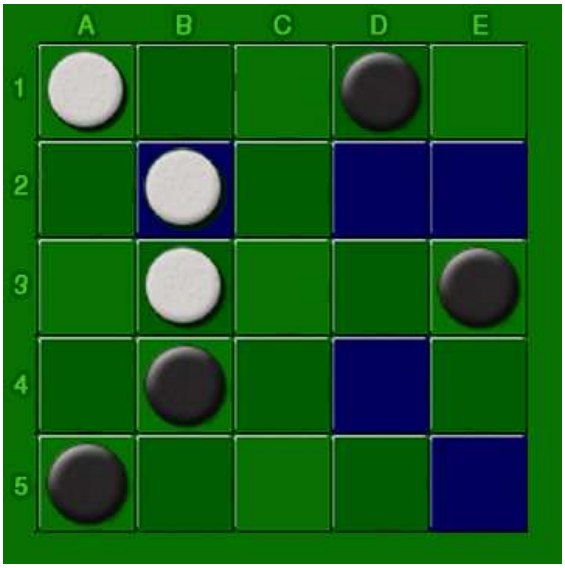
Kolejne ruchy przebiegają według następującego schematu. Każde posunięcie składa się z dwóch części. Pierwsza z nich to ruch nowego pionka o przynajmniej dwa pola w dowolnej linii poziomej, pionowej, lub ukośnej (nie przeskakując po drodze innych pionków). Druga część to postawienie kolejnego pionka na przedostatnim polu przebytej linii.

Po wystawieniu 9 pionków, gracz najpierw wykonuje ruch, następnie podnosi jeden z wcześniej umiejscowionych pionków i stawia go za poruszonym wcześniej pionkiem.

Wyjątek: w celu wyrównania szans, gracz biały ma prawo w drugim ruchu umieścić pionek w dowolnym miejscu na planszy, tak jak miało to miejsce podczas pierwszego ruchu.

Wygrywa gracz który pierwszy umieści cztery lub pięć pionków w linii (poziomej, pionowej, skośnej) lub też zablokuje ruchy przeciwnika.

3. Przykładowa plansza



Rys 1. Przykładowa plansza

Na przedstawionej na rysunku planszy gracz biały może wykonać pionkiem znajdującym się na polu B2 ruch na jedno z pól: D2 (dostawić na C2), E2 (dostawić na D2), D4 (dostawić na C3), E5(dostawić na D4). Może również wykonać ruch pozostałymi pionkami.

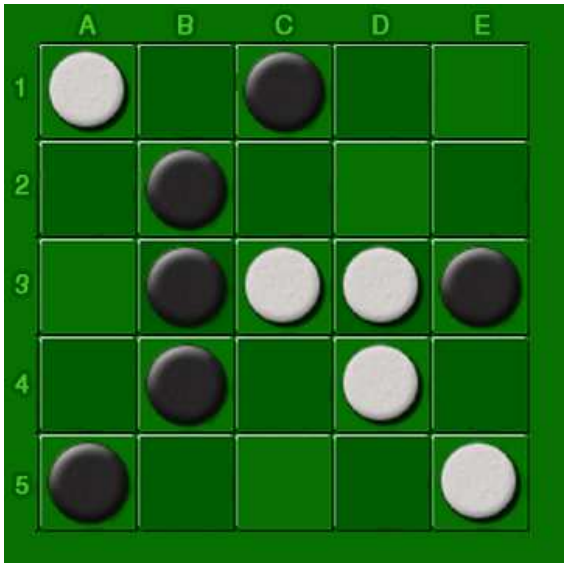
4. Reprezentacja stanu gry

Stan gry reprezentowany jest przez obiekt klasy **Board**. Rozmieszczenie pionków na planszy przechowywane jest w dwuwymiarowej tablicy liczb całkowitych (**int**) o wymiarach 5x5. Polu na którym znajduje się pionek czarny przypisana jest wartość **-1** polu wolnemu **0**, polu zajętemu przez pionek biały **1**.

Dodatkowo klasa **Board** zawiera pola opisujące liczbę nie wyłożonych pionków każdego z kolorów oraz kolor gracza do którego należy następny ruch. Są to informacje nadmiarowe, można je obliczyć na podstawie rozmieszczenia pionków na planszy, przechowywanie ich zwiększa efektywność oraz czytelność kodu.

Klasa graficznego interfejsu użytkownika wywołuje metodę **getBoard()** klasy **Board** otrzymując jako strukturę tablicę o rozmiarach planszy, następnie sprawdza iteracyjnie zawartość pól i rysuje pionki na zajętych polach.

Na potrzeby zapisu do pliku plansza jest konwertowana do formy w której każdemu wierszowi planszy odpowiada wiersz pliku składający się z pięciu znaków (zakończony znakiem '\n'). W zależności czy i jaki pion znajduje się na polu, w pliku zapisywany jest na odpowiedniej pozycji znak * - pionek czarny, **o** - pionek biały, **.** - brak pionka.




```
board = {
{ 1, 0,-1, 0, 0},
{ 0,-1, 0, 0, 0},
{ 0,-1, 1, 1,-1},
{ 0,-1, 0, 1, 0},
{-1, 0, 0, 0, 1}};
```

Rys 2. Przykładowa plansza oraz jej reprezentacja

5. Generacja ruchów dopuszczalnych

Procedura generująca ruchy rozróżnia trzy stany gry:

- Rozkładanie pionków (pierwszy ruch czarnego oraz pierwsze 2 białego)
- Ruchy standardowe (pierwsze 9 ruchów każdego gracza)
- Ruchy z podnoszeniem pionka (pozostałe)

Ruch reprezentowany jest w programie jako obiekt klasy **Move**. Zawiera on pola klasy **Point** opisujące pozycje na planszy z której pionek został przestawiony, na którą zostanie postawiony oraz pozycję na którą zostanie dostawiony dodatkowy pionek. Jeżeli jest to ruch z podnoszeniem zapisana jest również informacja skąd został podniesiony pionek.

Przykład:

Ruch z B2 na E2.

```
move.from = Point(1, 1)
move.to   = Point(4, 1)
move.drop = Point(3, 1)
move.pick = null
```

Ruchy generowane są dla danego stanu planszy za pomocą procedury **listMoves()**. Zwraca ona listę obiektów typu **Move** zawierającą wszystkie dozwolone w danym stanie gry ruchy.
Ruchy generowane są deterministycznie, kolejno dozwolone ruchy dla każdego pionka, w kolejności podyktowanej strukturą w której jest przechowywana plansza. Niedeterminizm gry algorytmu uzyskany został przez zastosowanie losowania pierwszego ruchu oraz, jeśli kilka ruchów uzyskało taką samą ocenę, losowy wybór jednego z nich.

6. Algorytmy przeszukiwania stanów gry

Ze względu na bogactwo materiałów dotyczących algorytmów przeszukiwania stanów gry w internecie oraz wysoką jakość dostarczonych materiałów dydaktyczne wybór oraz implementacja algorytmu grającego okazały się prostrze od wyboru heurystyki.

Pewnym problemem okazała się znacznie zmieniająca się w trakcie gry liczba dozwolonych ruchów. Zmiany te wynikają z opisanego w poprzednim punkcie trójfazowego przebiegu gry. Przez większość gry, podczas wykonywania standardowych ruchów, liczba ruchów dozwolonych jest mniejsza od 10. Pozwala to na efektywne przeszukiwanie na głębokość 7-8.

Podczas rozkładania pionów liczba ta wynosi ok 20, a w fazie podnoszenia może przekroczyć nawet 50.

Rozważaliśmy dwa sposoby unikania przekraczania dozwolonego czasu (timeout'u):

- rozpoznawanie w jakim stanie znajduje się gra i dostosowanie głębokości do niego
- zastosowanie przeszukiwania z literacyjnym pogłębianiem

W zaimplementowanym środowisku gry po przekroczeniu dozwolonego czasu obliczenia są przerywane a jako ruch wynikowy brany jest najlepszy z dotychczas wyliczonych. Ostatecznie zastosowaliśmy drugie podejście (iterative deeping).

Innym problemem który początkowo zmniejszał szansę wygrania zaimplementowanych algorytmów był fakt, że zakładały bezbłędną grę przeciwnika, co szczególnie podczas gry człowiek kontra komputer, jest założeniem błędnym. Problem ten objawiał się tym, że w momencie gdy algorytm odnajdywał sekwencję ruchów prowadzącą przeciwnika do wygranej "poddawał się" i losował dowolny z dozwolonych ruchów.

Rozwiązaniem tego problemu było preferowanie późniejszej przegranej, uzyskane poprzez odejmowanie głębokości na której zauważono zwycięstwo od wartości oznaczającej zwycięstwo. Zastosowanie tej strategii znacznie zwiększyło szanse algorytmu w konfrontacji z człowiekiem (od tego momentu nie potrafiliśmy już wygrać z własnym programem).

6.1 Metody porządkowania następników z pamięcią

Jako przykład metody porządkowania następników z pamięcią zastosowaliśmy ***Iteracyjne Pogłębienie z Porządkowaniem w Korzeniu***.

Ruchy przechowywane są w generycznej liście **List<Move>** i sortowane przy użyciu metody **Collections.sort(List)**. Oprócz kryterium osiągnięcia maksymalnej głębokości przeszukiwania obliczenia przerywa również przekroczenie zadanego czasu.

6.2 Metody manipulowania zakresem alfa-beta

Jako przykład metody manipulowania zakresem alfa-bera zastosowaliśmy algorytm ***AspirationSearch***. Wartość parametry delta ustawiliśmy ręcznie na podstawie eksperymentów (dobrze sprawdzają się wartości z przedziału 10-30). Początkowa wartość guess jest równa 0, a następnie modyfikowana jest przez literacyjne pogłębianie.

6.3 Metody sterowania głębokością przeszukiwania

<TODO>

7. Funkcja oceny heurystycznej gry

Funkcja oceny heurystycznej stanu gry okazała się największym wyzwaniem podczas realizacji projektu. Niezbędne było znalezienie algorytmu który wiarygodnie odzwierciedli stan planszy oraz będzie działał stosunkowo szybko.

Po wielu testach wybraliśmy heurystykę biorącą pod uwagę dwa aspekty sytuacji na planszy:

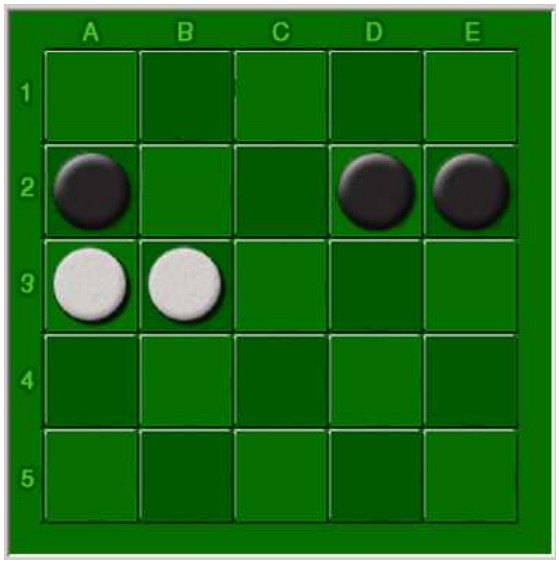
- zablokowanie przeciwnika
- ułożenie czterech pionków w linii

W celu oszacowania możliwości zablokowania przeciwnika algorytm porównuje potencjalną ilość ruchów jaką mógłby wykonać każdy z graczy przy danym ułożeniu pionków. Szansa na ułożenie czterech pionków w linii szacowana jest poprzez dostawianie pionka kolejno na każdym wolnym polu i sprawdzanie czy na planszy powstała linia 4 pionków. Procedura powtarzana jest dla obu kolorów w celu zapewnienia symetrii funkcji oceny.

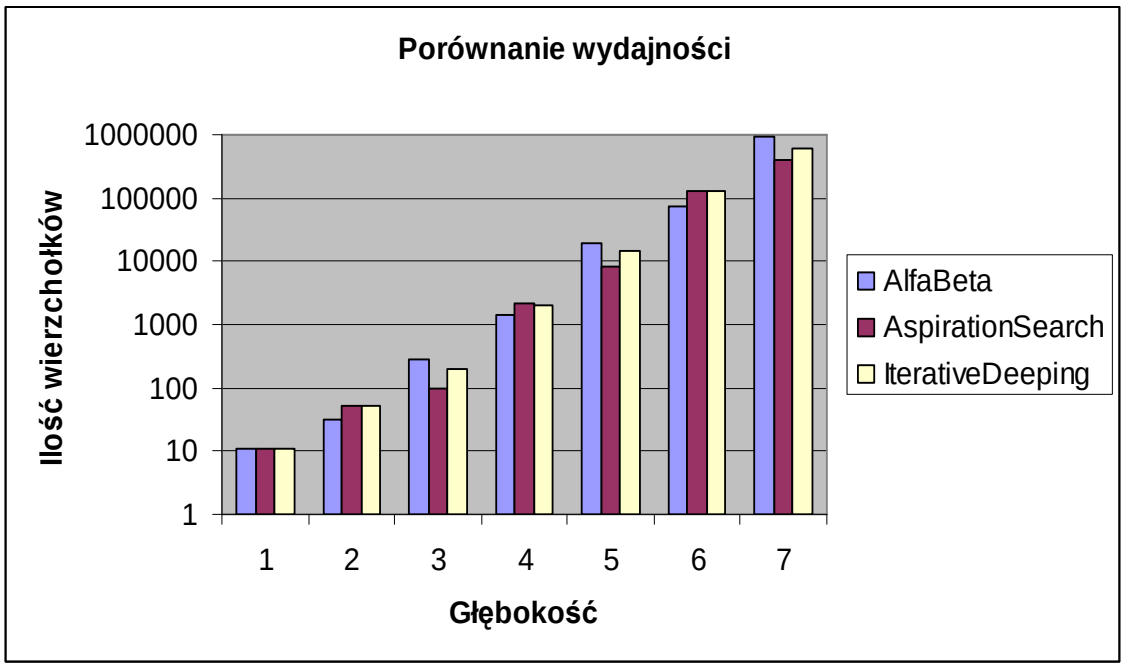
8. Testy wydajności i porównania

Wykonaliśmy testy dla osiemnastu przykładowych stanów gry, wszystkie wyniki znajdują się w załączniku, trzy najbardziej reprezentatywne wyniki prezentujemy poniżej.

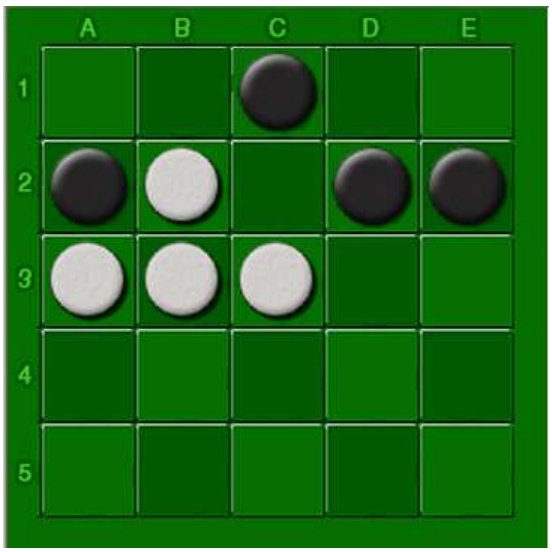
8.1. Test 1



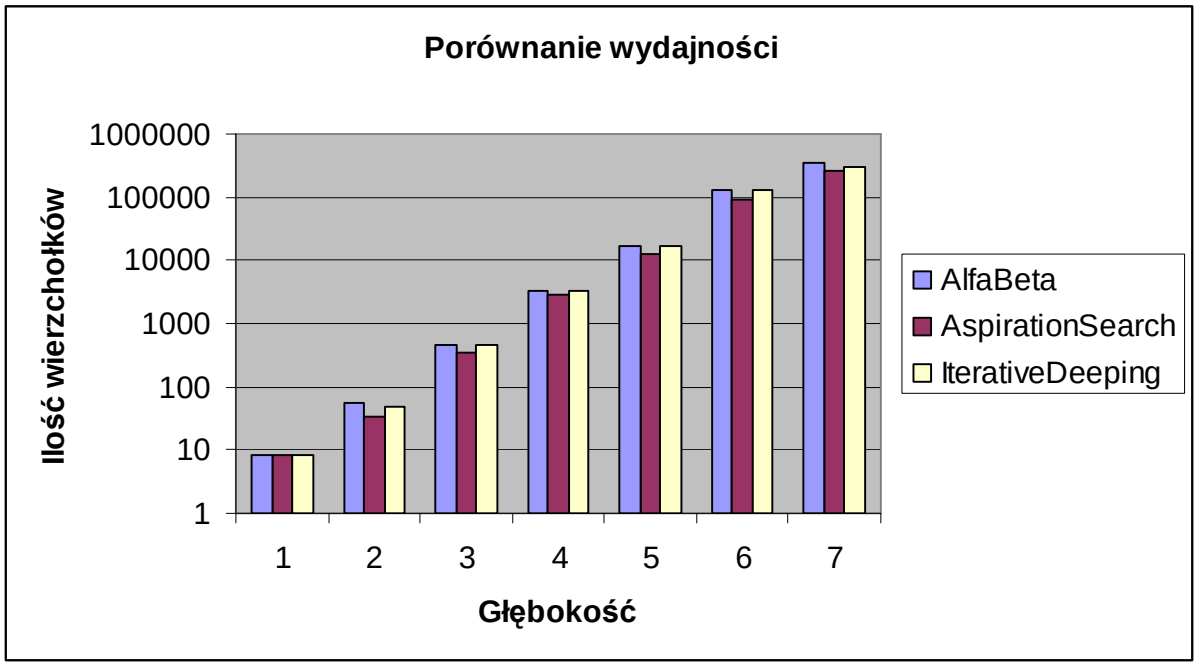
Alfa Beta	Aspiration Search	Iterative Deeping
11	11	11
31	53	53
277	97	198
1457	2116	2052
19963	8569	14598
75730	128428	126221
949308	406426	605253



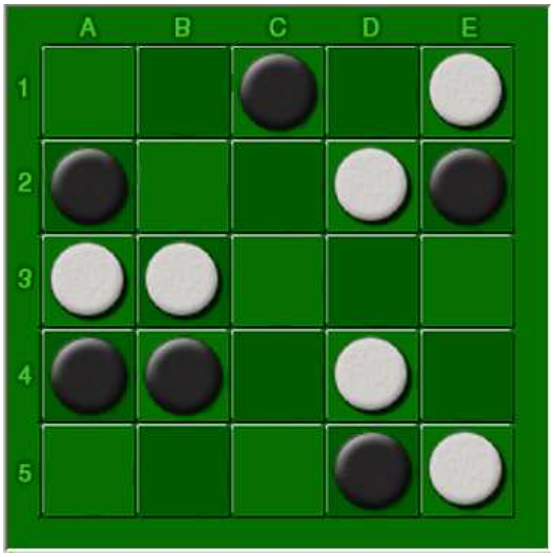
8.2. Test 2



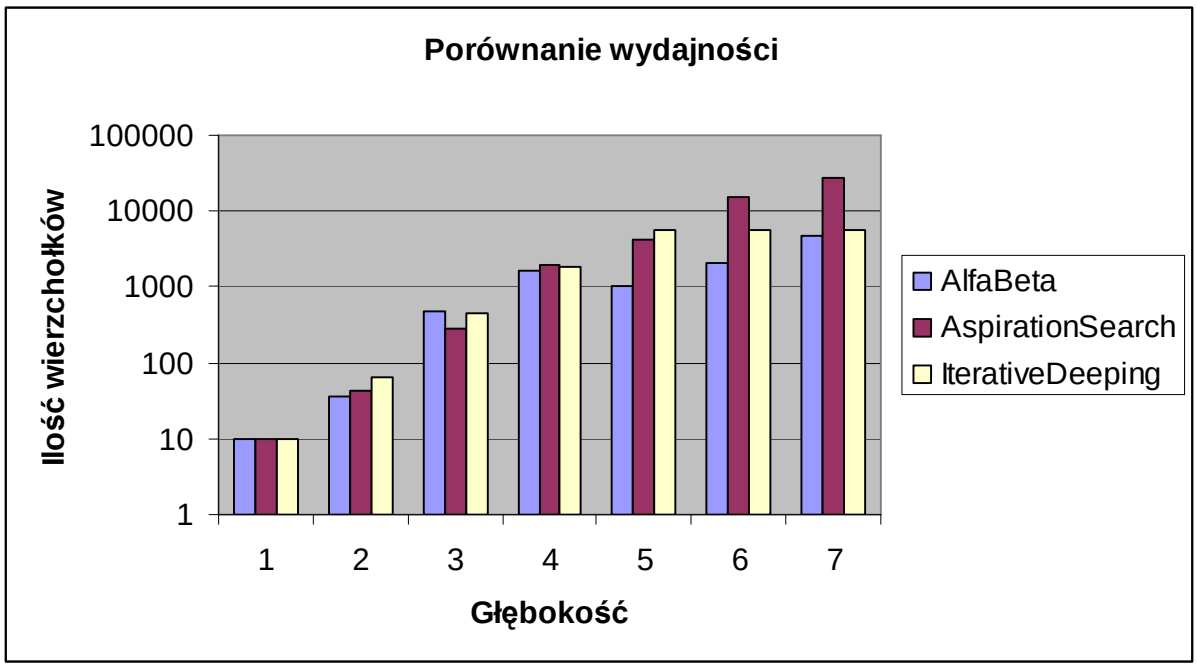
Alfa Beta	Aspiration Search	Iterative Deeping
8	8	8
57	33	49
475	350	458
3213	2842	3213
16580	12895	16239
125588	91589	127493
336799	266311	307458



8.3. Test 3



Alfa Beta	Aspiration Search	Iterative Deeping
10	10	10
35	43	63
489	286	460
1673	1986	1799
1044	4191	5725
2129	14891	5725
4595	27818	5725



9. Wnioski

9.1 Rozmier przestrzeni stanów

Rozmiar przestrzeni stanów jest równy:

S = 25! / (9!*9!*7!) = 2 3371 634 000

W porównaniu z grami takimi jak szachy, warcaby czy GO jest on bardzo niewielki. Przy dzisiejszych pojemnościach dysków twardych, biorąc pod uwagę, że stan całej gry może być zapamiętany w 61bitowym słowie, możliwe było by rozwiązanie i stablicowanie wyniku gry. Czyli utworzenie tablicy transpozycji zawierającej każdy możliwy stan gry.

Wielkość grafu gry zmienia się podczas gry. Na początku oraz po wykonaniu 18 ruchów jest on znacznie większy. Dokładna przyczyna tych zmian została opisana w punkcie 6.

9.2 Zmniejszenie rozmiaru przeszukiwanej przestrzeni

Z punktu widzenia ograniczenia rozmiaru przeszukiwanej przestrzeni najlepszy okazał się algorytm AsprationSearch. Algorytm iteracyjnego pogłębiania również zwykle pozwalał zauważalnie zmniejszyć przestrzeń przeszukiwania ze względu na wielokrotne odwiedzanie tych samych wierzchołków czasami łączna liczba odwiedzin była większa niż dla algorytmu alfa-beta.

9.3 Gra człowiek vs komputer

Wygranie z algorytmem nie jest łatwe. Co ciekawe w konfrontacji z człowiekiem najskuteczniejszy okazał się algorytm skrajnie uproszczony algorytm nega-max z funkcją trójwartościową funkcją oceny heurystycznej zwracającą zawsze zero. Takie uproszczenie pozwoliło na zwiększenie głębokości przeszukiwania, co w połączeniu z losowaniem z pośród “remisowych” ruchów dało algorytm z którym człowiekowi naprawdę ciężko wygrać (trzeba mieć szczęście podczas losowań;)).

9.4 Problem cykli

Hipotetycznie wydaje się możliwe wystąpienie cykli w grafie gry. Podczas wielu partii testowych nie zdarzyło się jednak napotkanie cyklu. W zdecydowanej większości przypadków gra kończy się przed rozpoczęciem podnoszenia pionków, a więc w fazie

w której wystąpienie cykli jest niemożliwe. W związku z powyższym wykrywanie cyklów w grafie gry nie zostało zaimplementowane.

9.5 Implementacja

Implementacja środowiska gry nie była zadaniem trywialnym. W celu zapewnienia elastyczności oraz umożliwienia interakcji z interfejsem użytkownika podczas trwania obliczeń program został oparty na wątkach. Taka architektura wiązała się z typowymi dla aplikacji równoległych problemami synchronizacji oraz kontroli współbieżnego dostępu do danych. Innym poważnym utrudnieniem był fakt iż algorytm szybko zaczoł grać lepiej od autorów co uniemożliwiało ręczne testowanie.

9.6 Możliwości ulepszeń (bright future)

Zarówno funkcję oceny heurystycznej stanu gry jak i sposób reprezentacji stanu gry można by znacznie poprawić. O niedoskonałości zastosowanej funkcji oceny może świadczyć fakt iż algorytm iteracyjnego pogłębiania osiągał słabe wyniki dla stanów w których w liściach drzewa przeszukiwania gracz wygrywa.

Nasuwające się propozycje poprawy to:

- zapisywanie stanu planszy w 64 bitowym słowie
- generowanie ruchów oraz sprawdzanie wygranej za pomocą operacji XOR
- strumieniowym przetwarzaniu całych tablic podczas ww operacji
- udoskonalenie logiki funkcji heurystycznej (dokładniejsze poznanie gry)

10. Podsumowanie

Wykonanie powyższego projektu dało nam rozeznanie w problematyce działania gier minimaksowych jednocześnie, pozwoliło lepiej zrozumieć działanie metod omówionych na zajęciach.

Przekonaliśmy się, iż za pomocą mechanizmu odcięć możliwe jest znaczne skrócenie czasu przeszukiwania.

Mieliśmy okazję samodzielnie rozwiązywać problemy związane z programowaniem wielowątkowym wykorzystanym przy tworzeniu naszej aplikacji.