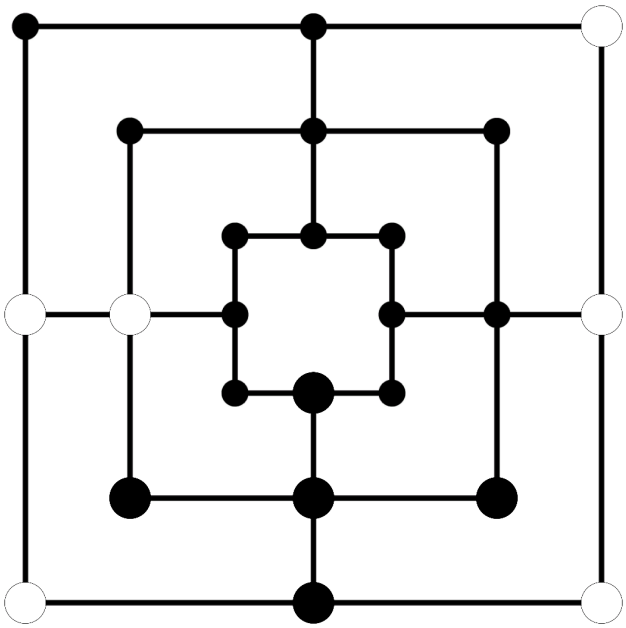
Algorytmy rozwiązywania gier o sumie zerowej

Szymon Woźniak, 235040

16.05.2019

1 Wstęp teoretyczny

1.1 Gra planszowa Młynek



Rysunek 1: Plansza do gry w młynek z kilkoma rozstawionymi pionkami

1.1.1 Skrót zasad

Młynek jest dwuosobową, turową, logiczną grą planszową. W rozpatrywanej wersji, na planszy znajdują się 24 rozmieszczone na 3 koncentrycznych kwadratach pola. Na każdym z tych pól gracze mogą umieszczać swoje pionki. Obaj gracze posiadają po 9 pionków do rozmieszczenia.
Gracze poprzez odpowiednie rozstawianie swoich pionków, mogą blokować lub zbijać pionki przeciwników. Bicie następuje gdy jeden z graczy ustawi 3 swoje pionki w linię. Może wtedy wybrać jeden z pionków przeciwnika, który zostanie usunięty z planszy.

1.1.2 Fazy rozgrywki

Pojedyncza partia młynka składa się z trzech faz.

- rozstawianie pionków,
- przesuwanie pionków,

- ”latanie”.

W pierwszej fazie rozgrywki gracze na zmianę umieszczają po jednym z dostępnych 9 pionków na wolnych polach planszy. Jeżeli któremuś z nich uda się ustawić młynek, może usunąć z planszy wybrany pionek przeciwnika. W rozpatrywanej wersji gry, jest to jedyny moment kiedy gracz może ustawić podwójny młynek.
W drugiej, podstawowej fazie rozgrywki gracze na zmianę swoje pionki. Mogą wybrać dowolne puste pole połączone linią z polem na którym znajduje się aktualnie pionek.
Trzecia faza następuje dla każdego gracza osobno, kiedy pozostaną mu tylko 3 pionki. Może on wtedy w swojej turze przemieszczać pionki na dowolne puste miejsca na planszy (stąd angielska nazwa *flying*).

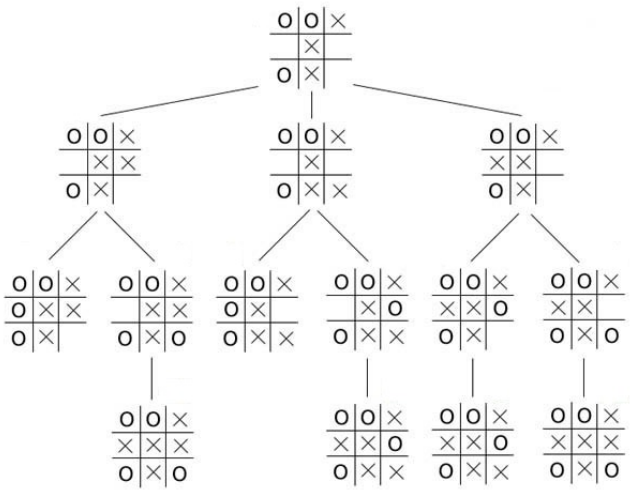
1.1.3 Cel rozgrywki

Celem rozgrywki jest doprowadzenie do sytuacji, w której przeciwnikowi pozostaną tylko 2 pionki, lub nie posiada on żadnego możliwego ruchu.

1.1.4 Dodatkowe modyfikacje

W rozpatrywanej wersji gry stosuje się zasadę, że pionka nie można przesunąć na pole, z którego został przesunięty wcześniej. Tym samym gracze zmuszeni są budować młynki, zamiast korzystać z już istniejących.
W niektórych wersjach gry stosuje się też zasadę, że gracze mogą zbijać pionki przeciwników tylko pod warunkiem, że nie stoją w młynku. W tej pracy zasada ta nie została zastosowana.

1.2 Drzewo gry



Rysunek 2: Przykładowy fragment drzewa dla gry kółko i krzyżyk

Drzewo gry jest grafem skierowanym, w którym każdy z węzłów reprezentuje stan rozgrywki w danym momencie. Z każdego stanu, w którym gra jeszcze się nie skończyła, można wygenerować zbiór następnych stanów gry reprezentujących różne możliwe decyzje aktualnie ruszającego się gracza. Następnie z każdego z tych stanów można wygenerować ruchy przeciwnika itd.
Jak widać już na przykładzie kółka i krzyżyk, drzewo to rozrasta się bardzo szybko i w większości gier nie jest ono możliwe do zbudowanie i przejrzenia w całości.

1.3 Badane algorytmy

Rozpatrywane algorytmy przeglądają fragmenty operują na wspomnianym w sekcji 1.2 drzewie gry. Przeglądając jego fragment, estymują jakość możliwych do podjęcia decyzji, oceniając stan rozgrywki kilka ruchów dalej. Oba korzystają w tym celu z pewnej heurystycznej funkcji, oznaczanej dalej jako *heuristic*, do statycznej ewaluacji stanu rozgrywki.

1.3.1 Algorytm min-max

Algorytm min-max przegląda drzewo gry do pewnej zadanej głębokości *depth*, na zmianę wybierając odpowiednio stan oceniany jako najlepszy i jako najgorszy przez funkcję *heuristic*. Reprezentuje to podejmowanie możliwie najlepszych decyzji zarówno przez siebie jak i przez przeciwnika. Jego działanie przedstawia poniższy pseudokod.

Algorithm 1 Algorytm Min-Max

```
1: function MinMax(state, depth, maximizing)
2:     if game finished in state or depth = 0 then
3:         return heuristic(state)
4:     end if
5:     if maximizing then
6:         maxEval ← −∞
7:         childStates ← getAllNextStates(state)
8:         for child in childStates do
9:             eval ← MinMax(child, depth − 1, false)
10:            maxEval ← max(maxEval, eval)
11:        end for
12:        return maxEval
13:    else
14:        minEval ← ∞
15:        childStates ← getAllNextStates(state)
16:        for child in childStates do
17:            eval ← MinMax(child, depth − 1, true)
18:            minEval ← min(minEval, eval)
19:        end for
20:        return minEval
21:    end if
22: end function
```

1.3.2 Algorytm alfa-beta cięć

Algorytm alfa-beta cięć jest usprawnieniem algorytmu min-max. Przeglądając kolejne stany wgłąb drzewa podejmuje on decyzje czy rozpatrywana gałąź jest warta rozwijania. Jeżeli w dowolnym momencie nie istnieje możliwość znalezienia lepszego stanu na pewnym poziomie drzewa, algorytm nie przegląda kolejnych stanów. Pozwala to zaoszczędzić czas pracy procesora i potencjalnie przeglądać drzewo na większą głębokość w takim samym czasie jak algorytm min-max dla mniejszych głębokości. Jego działanie zostało przedstawione na poniższym pseudokodzie.

Algorithm 2 Algorytm Alfa-Beta

```
1: function AlfaBeta(state, depth, maximizing, α, β)
2:     if game finished in state or depth = 0 then
3:         return heuristic(state)
4:     end if
5:     if maximizing then
6:         maxEval ← −∞
7:         childStates ← getAllNextStates(state)
8:         for child in childStates do
9:             eval ← AlfaBeta(child, depth − 1, false, α, β)
10:            maxEval ← max(maxEval, eval)
11:            α ← max(α, eval)
12:            if α ⩾ β then
13:                break
14:            end if
15:        end for
16:        return maxEval
17:    else
18:        minEval ← ∞
19:        childStates ← getAllNextStates(state)
20:        for child in childStates do
21:            eval ← AlfaBeta(child, depth − 1, true, α, β)
22:            minEval ← min(minEval, eval)
23:            β ← min(β, eval)
24:            if α ⩾ β then
25:                break
26:            end if
27:        end for
28:        return minEval
29:    end if
30: end function
```

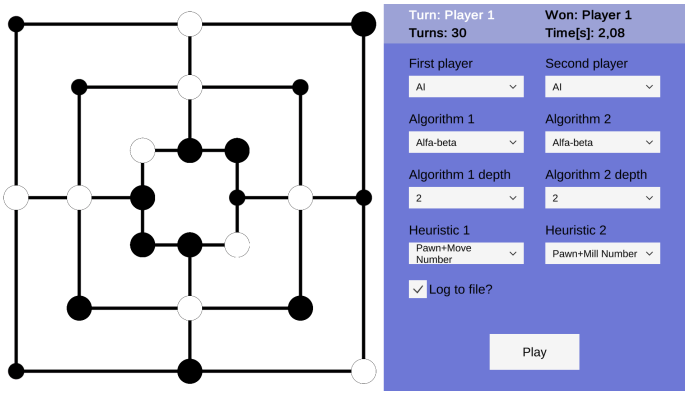
2 Plan pracy

W pierwszej kolejności gra młynek zostanie zaimplementowana w wybranym języku programowania i środowisku programistycznym. Implementacja będzie zawierać silnik gry pozwalający na grę zarówno graczy ludzkich jak i kierowanych przez algorytmy sztucznej inteligencji. Będzie również posiadać interfejs graficzny ułatwiający rozgrywkę i umożliwiający obserwowanie rozgrywek SI graczom ludzkim. W silniku gry zostaną również zaimplementowane algorytmy min-max i alfa-beta oraz różne heurystyki oceny stanu planszy.
Następnie przeprowadzone zostaną badania zaimplementownych rozwiązań. W pierwszej kolejności zostanie przeprowadzone badanie porównawcze czasów przetwarzania i liczby przeglądanych węzłów drzewa gry dla algorytmów min-max i

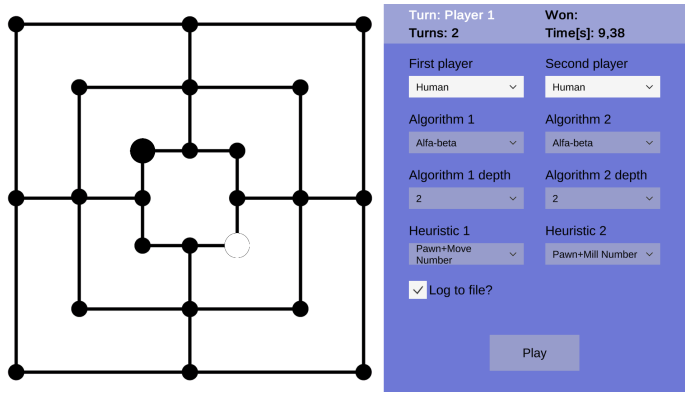
alfa-beta.
Jako drugie zostanie przeprowadzone badanie wpływu zastosowania heurystyki wyboru kolejności węzłów na czas przetwarzania, liczbę ruchów i przeglądanych węzłów drzewa gry dla algorytmu alfa-beta.
Wszystkie badania zostaną przeprowadzone na wersji z interfejsem graficznym.

3 Implementacja

Implementacja gry została przeprowadzona przy pomocy silnika Unity w technologiach .NET i C#. Całość implementacji została przystosowana do uruchamiania w trybie bez interfejsu graficznego. Zrzuty ekranu z wybranych fragmentów rozgrywki zostały przedstawione 3 i 4.



Rysunek 3: Interfejs graficzny po zakończeniu rozgrywki pomiędzy graczami sterowanymi przez SI



Rysunek 4: Interfejs graficzny w rozgrywce pomiędzy graczami ludzkimi

4 Heurystyki oceny stanu planszy

Do przybliżania oceny stanu planszy przez algorytmy należy zaproponować heurystyki, w których zakodowana jest specyficzna wiedza dla rozpatrywanego problemu. W tej pracy rozpatrzone zostaną 3 różne heurystyki. Dużą wartością funkcji oznaczony jest fakt, że plansza jest dobra dla gracza białego, natomiast małą że plansza jest dobra dla gracza czarnego.

4.1 Liczba pionków

Ta heurystyka ocenia liczbę pionków gracza i przeciwnika na planszy. Powinna faworyzować ruchy prowadzące do ustawiania młynków i tym samym bicia pionków przeciwnika, oraz ruchy blokujące ustawienie młynka przez oponenta, ponieważ prowadzą do utraty własnego pionka. Dodatkowo dodawana lub odejmowana jest wartość o dużej wadze, jeżeli gra została wygrana przez jednego z graczy. Przedstawia ją następujące równanie:

$$PAWN(state) = a \cdot (p_w(state) - p_b(state)) + b \cdot win(state) \tag{1}$$

, gdzie:

- a - waga dla liczby pionków,
- p_w - liczba pionków gracza białego,
- p_b - liczba pionków gracza czarnego,
- b - waga dla wygranej,
- win - wartość 1, -1 lub 0 w zależności od tego czy wygrał gracz biały, czarny lub gra się nie skończyła.

W tej pracy użyte zostały współczynniki $a = 1086$ i $b = 9$.

4.2 Liczba pionków i liczba młynków

Ta heurystyka różni się od poprzedniej tym, że kładzie dodatkowy nacisk na ustawianie młynków na planszy. Jest sformułowana następująco:

$$MILL(state) = PAWN(state) + c \cdot (mill_w(state) - mill_b(state)) \tag{2}$$

, gdzie:

- c - waga dla ustawionych młynków,
- $mill_w$ - liczba młynków gracza białego,
- $mill_b$ - liczba młynków gracza czarnego.

4.3 Liczba pionków i liczba możliwych ruchów

Ta heurystyka jest modyfikacją pierwszej. Kładzie dodatkowy nacisk na liczbę możliwych ruchów, które mogą wykonać gracze. Powinno to promować ruchy prowadzące do wygranej poprzez zablokowanie przeciwnika. Można ją można zdefiniować następująco:

$$MOVE(state) = PAWN(state) + c \cdot (move_w(state) - move_b(state)) \quad (3)$$

, gdzie:

- c - waga dla liczby ruchów,
- $move_w$ - liczba dostępnych ruchów gracza białego,
- $move_b$ - liczba dostępnych ruchów gracza czarnego.

5 Badania

5.1 Porównanie czasów przetwarzania i liczby instrukcji algorytmów min-max i alfa-beta

Badanie zostanie przeprowadzone dla jednego ustalonego przeciwnika - algorytmu alfa-beta z głębokością 2 i heurystyką oceniającą liczbę pionków na planszy. Przeciwko niemu zostanie na zmianę postawiony algorytm min-max i alfa-beta. Tym samym będą one przeglądały te same drzewa gry, a wyniki będą bezpośrednio przedstawiać wydajność obu algorytmów. Zarówno czas przetwarzania jak i liczba przeglądanych węzłów zostaną zsumowane z całego przebiegu rozgrywki.

Tabela 1: Wyniki porównania liczby przeglądanych węzłów i czasów przetwarzania dla różnych głębokości drzewa gry

Algorytm	**Min-max**		**Alfa-beta**	
Poziom drzewa	**Czas[s]**	**Liczba węzłów**	**Czas[s]**	**Liczba węzłów**
1	0,025	3408	0,018	874
2	4,797	949030	0,614	27492
3	n/a	n/a	23,004	911337

Wnioski W tabeli 5.1 widać dobrze przewagę algorytmu alfa-beta nad min-max. Ten pierwszy jest w stanie podejmować te same decyzje co min-max przeglądając dużo mniejszą część drzewa gry, tym samym oszczędzając moc obliczeniową i czas. Pozwala to potencjalnie przeglądać drzewa do większej głębokości co może skutkować podejmowaniem lepszych decyzji.

5.2 Porównanie heurystyk oceny stanu planszy

Badanie zostanie przeprowadzone przy użyciu algorytmu alfa-beta i ustalonej głębokości drzewa gry - 2. Zestawione zostaną wszystkie zaimplementowane heurystyki.

Tabela 2: Wyniki rozgrywek pomiędzy zaimplementowanymi heurystykami dla głębokości drzewa gry równej 2

P1/P2	**PAWN**	**MILL**	**MOVE**
PAWN	X	P1(63)	P2(52)
MILL	P1(77)	X	P1(49)
MOVE	P1(31)	P1(31)	X

W tabeli 5.2 widać, że najlepiej w rozgrywkach radzą sobie heurystyki MILL i MOVE. Ta druga radzi sobie wyjątkowo dobrze, gdy kontroluje pionki gracza rozpoczynającego rozpoczynającego rozgrywkę. Aby lepiej przeanalizować różnice między nimi zostało przeprowadzone badanie dla głębokości drzewa gry równej 3.

Tabela 3: Wyniki rozgrywek pomiędzy heurystykami MOVE i MILL dla głębokości drzewa gry równej 3

P1/P2	**MOVE**	**MILL**
MOVE	X	P1(21)
MILL	P2(18)	X

W tabeli 5.2 widać, że dla głębokości drzewa gry równej 3, heurystyka MOVE wygrała obie rozgrywki bardzo szybko.

5.3 Wpływ heurystyki wyboru kolejności węzłów

Algorytm alfa-beta można próbować dodatkowo usprawnić wprowadzając dodatkowe heurystyki służące do wyboru kolejności przeglądania węzłów na odpowiednich poziomach drzewa gry. Może to skutkować przycinaniem większej liczby gałęzi a tym samym szybszym działaniem. Należy tutaj jednak mieć na uwadze fakt, że takie sortowanie może skutkować innym prowadzeniem rozgrywki. Dzieje się tak, ponieważ przeprowadzane w ten sposób sortowanie nie zachowuje względnego porządku węzłów na różnych głębokościach.
Badanie zostanie przeprowadzone przy ustalonej głębokości drzewa gry - 2. Zestawione zostaną algorytm alfa-beta bez heurystyki wyboru kolejności węzłów, oraz z nią. Przeciwnikiem będzie algorytm alfa-beta z głębokością drzewa 2 i heurystyką PAWN. Przeprowadzone zostaną 3 różne rozgrywki - z różnymi heurystykami oceny planszy.

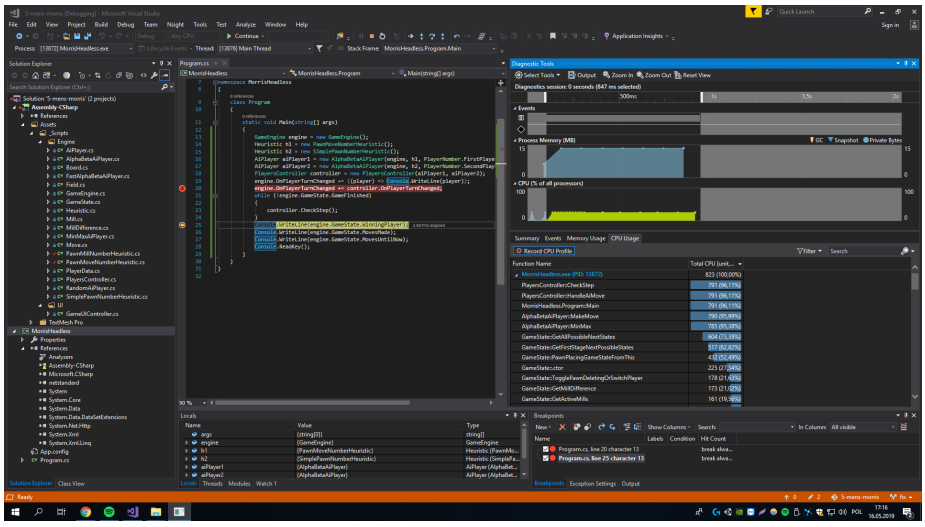
Tabela 4: Wyniki czasu działania i liczby przeglądanych węzłów dla algorytmu alfa-beta bez heurystyki i z włączoną heurystyką wyboru węzłów

	Bez heurystyki			Z heurystyką		
Lp.	**Czas[s]**	**Tury**	**Liczba węzłów**	**Czas[s]**	**Tury**	**Liczba węzłów**
1	0,681	61	27492	0,539	57	16457
2	0,689	77	25665	0,591	77	18186
3	0,768	31	35495	0,778	29	28365

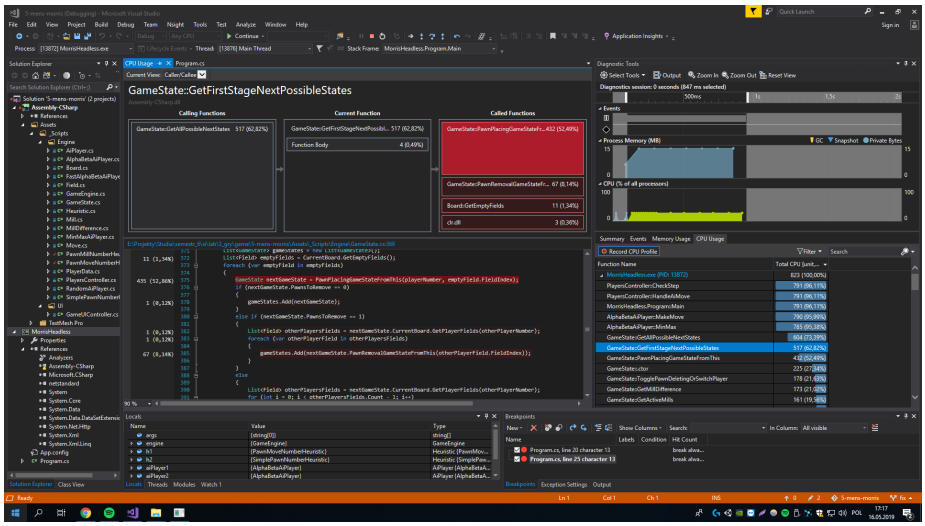
Wnioski Jak widać w tabeli 5.3 heurystyka wyboru kolejności węzłów może pozytywnie wpłynąć na efektywność algorytmu alfa-beta. Należy jednak, tak jak wspomniano już wcześniej, mieć na uwadze fakt, że algorytm z taką heurystyką może podejmować inne decyzje niż bez niej.

6 Profilowanie kodu

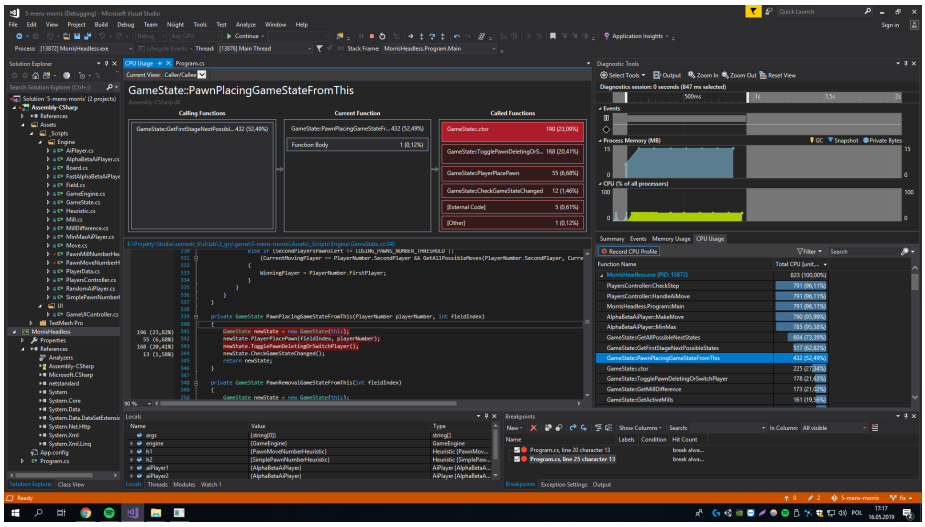
Profilowanie kodu zostało z przyczyn technicznych przeprowadzone dla wersji programu bez interfejsu graficznego, przy pomocy profilera dostępnego w programie Visual Studio 2017.



Rysunek 5: Zrzut ekranu z profilera - cały program



Rysunek 6: Zrzut ekranu z profilera - funkcja zwracająca stany gry dla pierwszej fazy rozgrywki



Rysunek 7: Zrzut ekranu z profilera - funkcja tworząca następny stan gry poprzez umieszczenie pionka na planszy

Jak widać na obrazkach 5 6 i 7 najwięcej czasu procesora wykorzystywane jest na działanie funkcji tworzących następne stany gry z aktualnego. Jako że

jest to w najczęściej wykonywana w badanych algorytmach operacja, to nacisk na jej optymalizację został położony już wcześniej.

7 Podsumowanie

Algorytmy przeszukujące drzewo gry stanowią dobre rozwiązanie dla rozwiązywania gier turowych. Przy zakodowaniu odpowiedniej wiedzy w heurystykach oceny stanu planszy, algorytmy te potrafią sprawiać wrażenie naprawdę inteligentnych i grać dużo lepiej niż przeciętny człowiek. Z dwóch przebadanych algorytmów lepszym wyborem jest algorytm alfa-beta, ponieważ potrafi podejmować te same decyzje szybciej niż min-max bez żadnych usprawnień. Tym samym może on przeglądać drzewo gry do większej głębokości w tym samym czasie i podejmować lepsze decyzje.