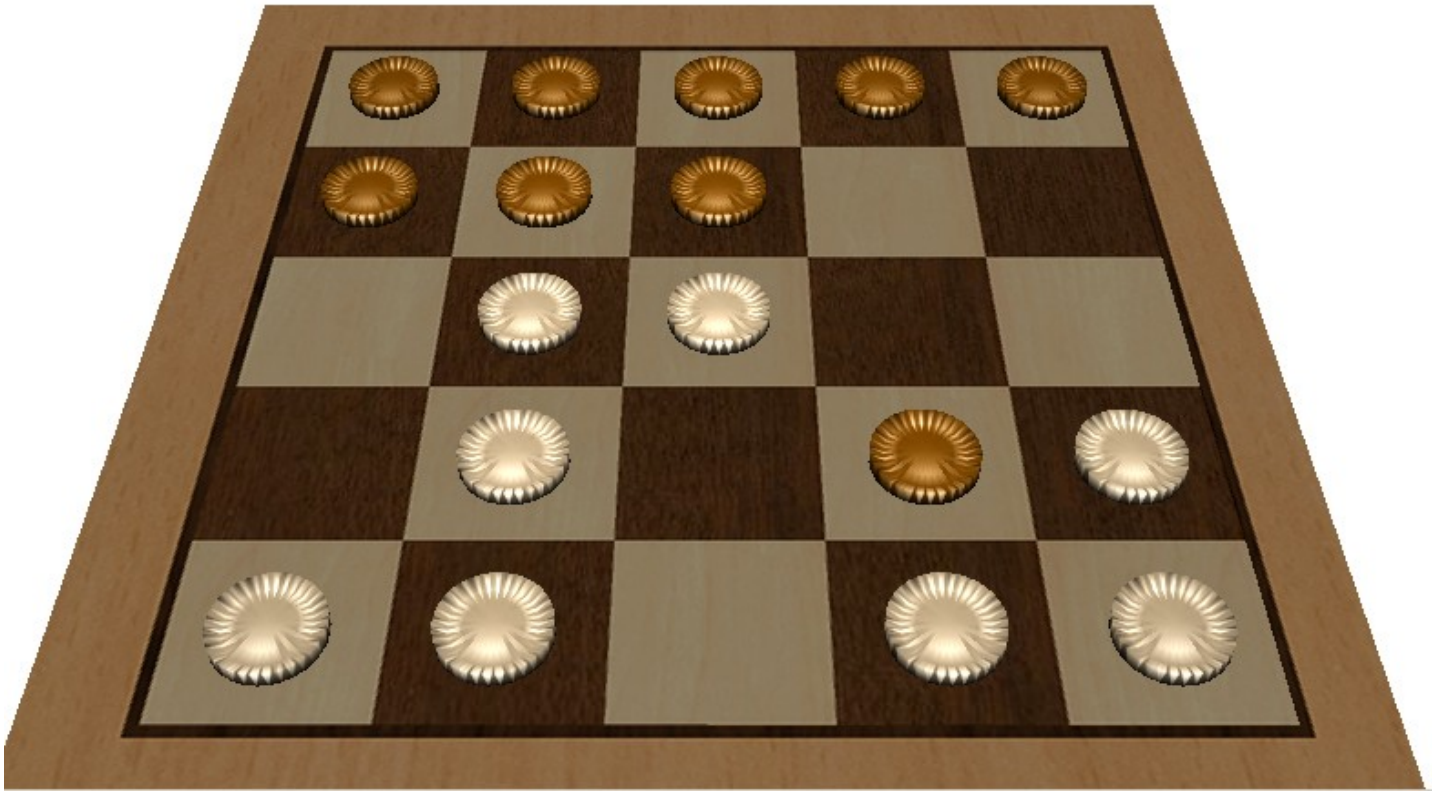
Sprawozdanie Sztuczna Inteligencja	Gra BreakThrough
Bartosz Kosarzycki 80080 Jacek Dalkowski 80046	

BreakThrough



1. Opis gry

i. Krótka historia gry

Zasady opisywanej gry zostały wymyślone w 2000 roku przez Dana Troyka'ę i zaimplementowane przy użyciu komercyjnego systemu umożliwiającego tworzenie gier planszowych (i aplikującego algorytmy sztucznej inteligencji do wprowadzonych reguł gry) – „Zillions of Games”.
Pierwotne zasady gry przewidywały przeprowadzanie rozgrywki na planszy o wymiarach 7 pól na 7, jednak pozostałe reguły gry nie zależą od wielkości planszy – jej wielkość może być parametryzowana. Dzięki temu, „Breaktrhrough” mogła wygrać w 2001 roku konkurs na projekt 8-polowej gry planszowej - „8x8 Game Design Competiton”.

ii. Opis zasad

A) Rozgrywka jest przeprowadzana na symetrycznej planszy („szachownicy”). Początkowe zasady przewidywały korzystanie z planszy o wymiarach 7x7, jednak wymiary mogą być z powodzeniem i bez naruszania pozostałych zasad parametryzowane. Plansza nie musi być kwadratowa. (Nasza implementacja gry przewiduje jednak kwadratowe plansze o rozmiarach: 5x5, 6x6, 7x7, 8x8, na takich też planszach rozgrywana jest ta gra w systemie Zillions of Games, gdzie została oryginalnie zaimplementowana (podobno, nie sprawdzaliśmy ponieważ dostęp do gry w tym akurat serwisie jest płatny)),

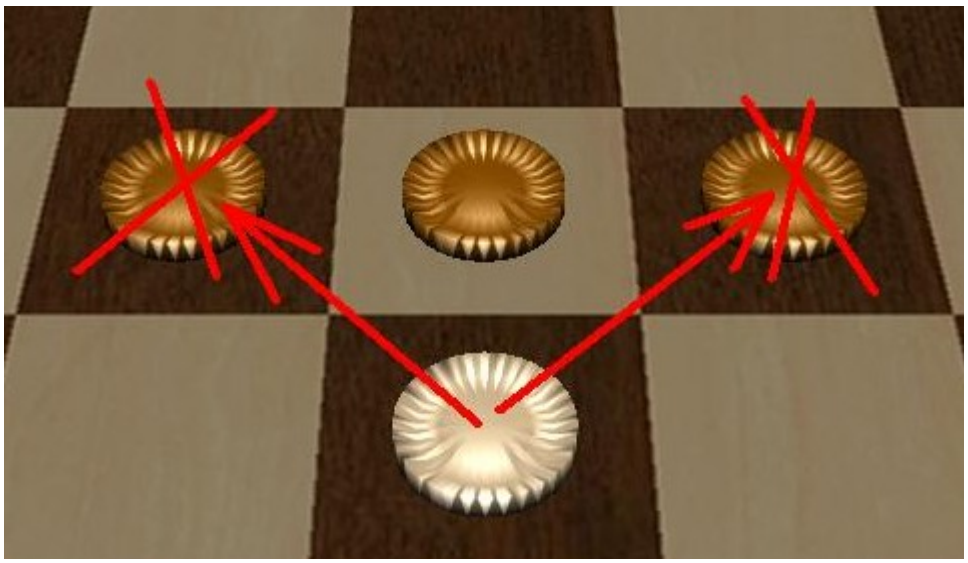
B) W rozgrywce uczestniczy dwóch graczy, którzy wykonują ruchy własnymi pionkami w kolejnych turach,

C) Gracze posiadają jednakową liczbę pionów. Liczba ta jest zależna od rozmiaru używanej planszy – liczba pionów posiadana przez każdego gracza równa jest dwukrotnej szerokości użytej „szachownicy”, ponieważ:

D) Początkowy stan rozgrywki stanowi rozstawienie pionów graczy w skrajnych stronach planszy, po 2 rzędy pionów, wypełniających całą szerokość planszy i stojących jeden obok drugiego. Zasady nie określają, jakiego koloru piony powinny się znajdować na określonym skraju planszy:

E) W danej turze tylko jedna bierka gracza wykonującego ruch, może i musi zostać przesunięta,
F) Bierka może zostać przesunięta tylko „do przodu”, czyli o jeden rząd w stronę skraju planszy, na którym początkowo były ustawione pionki przeciwnika:
   1. przesuwając pionek do przodu, można go ustawić na polu który go bezpośrednio poprzedza, lub na polach znajdujących się po przekątnej od jego pierwotnej pozycji, jeśli nie są one zajęte prze któryś z pionów przeciwnika:

   1. Jeśli na którymś z wspomnianych wcześniej pól, znajduje się pionek przeciwnika, to:
      (1) Jeśli znajduje się on na polach „po przekątnej”, to może zostać „zbity”, a pionek zbijający musi się przesunąć na pole pionka zbitego
      (2) Jeśli znajduje się on bezpośrednio przez rozpatrywaną bierką, to nie może wykonać bicia ani też rozpatrywany pion nie może zająć pola znajdującego się przed nim:



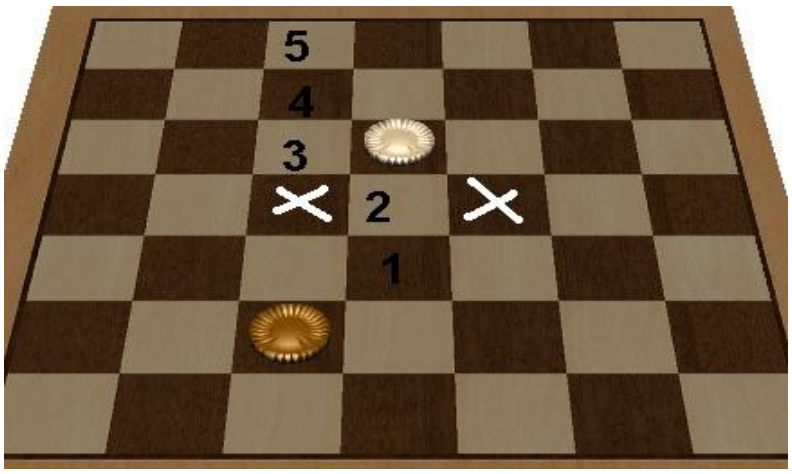
A) Wygrywa gracz, którego jeden z pionów, jako pierwszy osiągnie przeciwległy względem pozycji początkowej, rząd planszy (stąd nazwa gry – należy się przebić przez piony przeciwnika)

iii. Konsekwencje powyższych zasad.

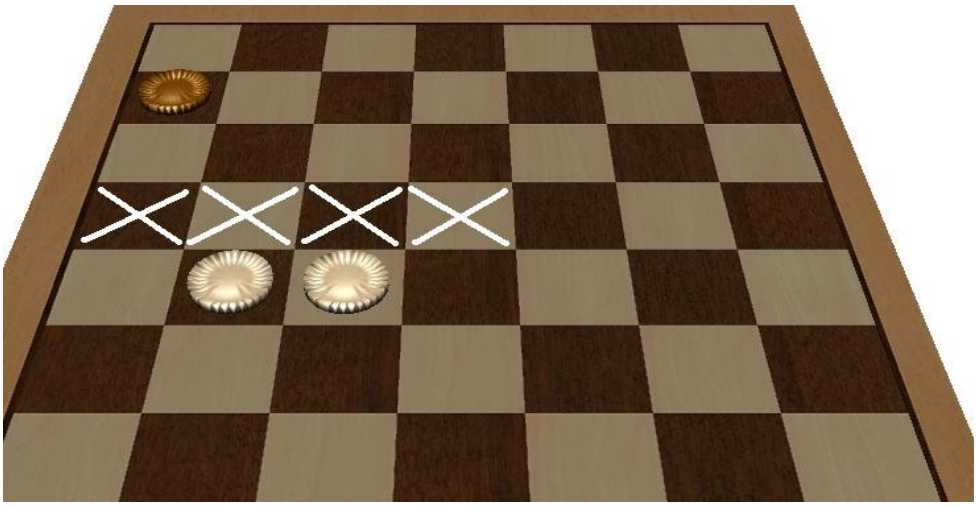
A) Remis jest niemożliwy – któryś z pionów dotrze pierwszy do przeciwległego rzędu, nie ma problemu dotyczącego cykli w grafie gry,

B) Każdy gracz stara się jako pierwszy osiągnąć ostatni rząd planszy zajęty pierwotnie przez przeciwnika, i równocześnie stara się nie dopuścić, by przeciwnik przebił się przez jego układ pionów. Aby grać efektywnie, należy prowadzić odpowiednią ofensywę i defensywę.

C) W „Breakthrough", istotne jest, jakie ułożenie na planszy tworzą grupy pionków danego gracza. Ponieważ bierki mogą bić i poruszać się tak, jak zostało to wcześniej opisane, pojedynczy pionek nie jest w stanie zablokować bierki przeciwnika, ponieważ ta po prostu jest w stanie ominąć, idąc cały czas na wprost niego (unikając tym samym bicia, ponieważ pionki nie mogą bić na wprost), a następnie omijając go wykonując ruch w bok. Po wykonaniu ruchu w bok, rozpatrywana bierka znajdzie się na w tym samym rzędzie co pionek „broniący", i już nic z jego strony jej nie zagraża – może kontynuować bieg do celu, jakim jest przeciwległy koniec planszy.



D) Istnieje bardzo wiele sposobów prowadzenia defensywy – ważne jest tylko, aby bierki pełniące tę funkcję był ustawione w taki sposób, by pola na które mogą bić tworzyły pewną całość, np.:



jak widać, już dwa pionki mają możliwość skutecznego blokowania przeciwnika. Takich układów pionków jest bardzo wiele i trudne jest ocenienie, jakie ułożenie jest lepsze, a jakiego gorsze.

3. Opis użytych algorytmów

Szczegóły implementacyjne.

- Na potrzeby interfejsu graficznego, plansza na której odbywa się rozgrywka, jest reprezentowana w pamięci komputera jako tablica o rozmiarze odpowiadającym wielkości planszy, której elementami są wskaźniki na obiekty pionków. Jeśli na danym polu nie ma pionka, wskaźnik odpowiadający temu polu przyjmuje wartość null, w przeciwnym razie, wskazuje na obiekt który odpowiada wyświetlanemu na ekranie pionkowi.
  Tablica, wykorzystywana przez algorytmy AI dla stanu gry:

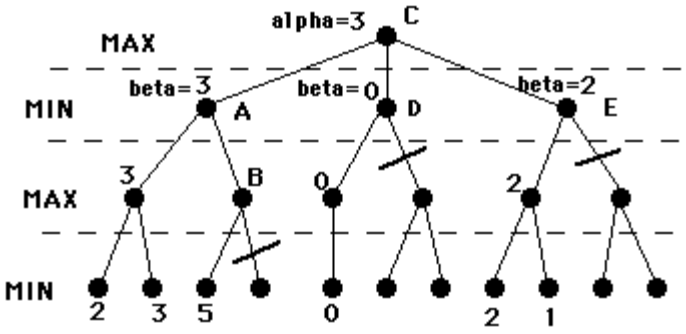
(Pionki ciemne – gracz MAX, pionki jasne – gracz MIN) ma postać:

	0	1	2	3	4
0	0	1	1	1	0
1	0	-1	-1	0	0
2	0	1	1	1	0
3	1	-1	-1	0	-1
4	-1	0	0	0	0

- Obiekt pionka posiada pola, określające czy dany pionek należy do gracza MAX czy też MIN. W naszej implementacji gracz MAX podąża „w dół” planszy (ku rzędom o wyższych numerach) a gracz MIN „w górę planszy”. Niezależnie od tego, do kogo należy dany pionek, może być on jasny lub ciemny (zależy to od początkowych ustawień gry, oczywiście wszystkie pionki danego gracza mają ten sam kolor). Obiekt pionka posiada też dane dotyczące jego położenia w przestrzeni trójwymiarowej oraz wskaźnik na obiekt sterujący jego przesuwaniem.

- Przesunięcie pionka na planszy odpowiada:
  - na tablicy używanej przez algorytmy AI – wyzerowaniu pola z którego pionek się przesuwa oraz umieszczeniu odpowiedniej wartości: 1 albo -1 w polu tablicy, odpowidającemu polu planszy, na które pionek jest przesuwany,
  - na tablicy stworzonej na potrzeby interfejsu – przekierowaniu wskaźnika odpowiadającego polu, na które jest przesuwany pionek, na obiekt tego pionka, oraz ustawieniu wskaźnika pola, z którego ten pionek jest przesuwany na null, oraz uaktywnieniu obiektu odpowiedzialnego za animację przesunięcia,
- Bicie pionka odpowiadaj
  - na tablicy używanej przez algorytmy sztucznej inteligencji – wyzerowaniu pola odpowiadającego polu planszy, na którym stoi zbijany pionek, a następnie przesunięciu zbijającego pionka wg powyższego opisu
  - na tablicy stworzonej na potrzeby interfejsu – zakończeniu animowania obiektu pionka, ustawieniu wskaźnika odpowiadającego polu, na którym stoi zbijany pionek, na null, a następnie przesunięciu zbijającego pionka wg powyższego opisu.
- Na algorytmów sztucznej inteligencji, może zostać pobrana dwuwymiarowa tablica liczba całkowitych odpowiadająca układowi pionków na planszy. Jeśli na danym polu nie znajduje się pionek, to wówczas odpowiednie pole tablicy dwuwymiarowej przyjmuje wartość 0, jeśli na danym polu znajduje się pionek gracza MAX, to pole to przyjmuje wartość 1, jeśli gracza MIN – to wartość (-1). Taką też reprezentację stanu gry wykorzystują opisane poniżej algorytmy. Należy też wspomnieć, że następniki poszczególnych ruchów również wykorzystują tak interpretowaną tablicę liczb całkowitych. Samo generowanie następników odbywa się w oczywisty sposób – następuje iteracja po wszystkich rzędach „szachownicy", a w każdym rzędzie, po wszystkich polach od strony lewej do prawej. Jeśli na badanym polu nie ma pionka, przechodzimy dalej. Jeśli znajduje się pionek gracza MAX, to sprawdzane są dal niego możliwości przesunięcia: „w dół – w lewo", „w dół - na wprost", „w dół – w prawo", natomiast dla gracza MIN analogicznie – najpierw „w górę – w lewo", „w górę, na wprost", „w górę – w prawo".

3.1 Algorytm Alpha-Beta



Jako pierwszy zaimplementowaliśmy algorytm Alpha-beta opisany w materiałach [1]. Jak wiadomo, powyżej opisany algorytm jest usprawnieniem Mini-max/Nega-max mogącym znacznie poprawić efektywność przeszukiwania stanów gry. „Wprowadzenie odcięć alpha-beta pozwala zazwyczaj dwukrotnie zwiększyć głębokość przeszukiwania przy tej samej objętości pamięci". Skuteczność odcięć mocno zależy jednak od uporządkowania wierzchołków „dzieci" danego stanu, czyli inaczej mówiąc możliwych ruchów z danego

1 http://www.cs.put.poznan.pl/amichalski/search/si2006.bw2pp.pdf

stanu.

3.2 Algorytm Breaker – algorytm dopasowany do specyfiki gry BreakThrough

Projektując algorytm dopasowany do specyfiki naszej gry wzięliśmy pod uwagę dwa główne kryteria:

1) przeszukiwanie przestrzeni stanów jest operacją kosztowną, nawet gdy wspomagamy przeszukiwanie wszystkimi możliwymi odcięciami, należy więc unikać przeszukiwania całej przestrzeni, jeżeli ruch jest oczywisty.
2) jeżeli ruch nie jest oczywisty, musimy przeszukać całą przestrzeń, należy jednak usprawnić to przeszukiwanie na tyle, na ile jest to możliwe.

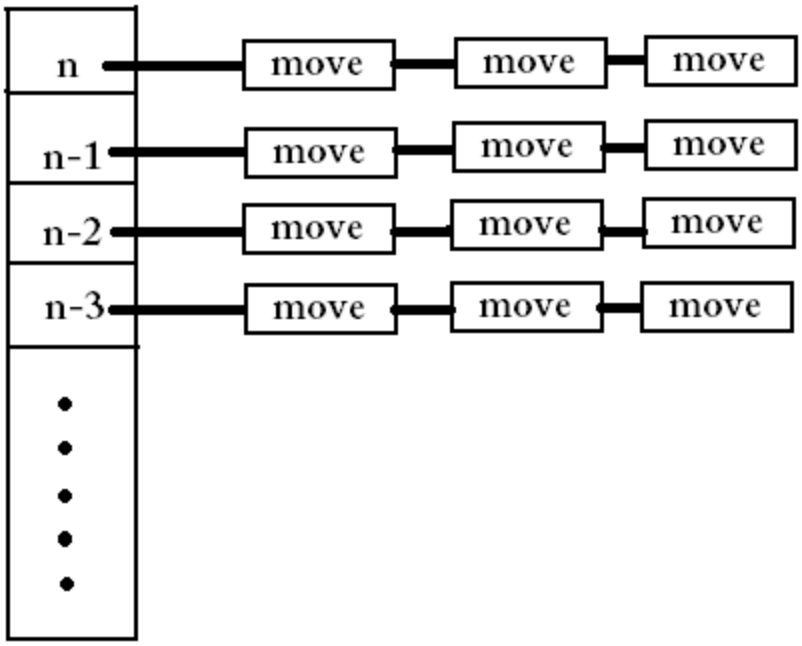
Odnośnie punktu pierwszego:

Po dogłębnej analizie sposobu gry postanowiliśmy wyłonić ruchy, które są oczywiste i należy je wykonać, aby nie przegrać. Analizowaliśmy zarówno ruchy defensywne, jak i ofensywne. W wyniku zaimplementowania różnych strategii powstało parę drobnych algorytmów, które zostaną opisane poniżej[2].

Odnośnie punktu drugiego:

Usprawniając przeszukiwanie stanów zdecydowaliśmy się na algorytm o nazwie „Alpha-Beta killer moves". Jak wspomnieliśmy wcześniej skuteczność alpha-bety zależy w dużej mierze od uporządkowania dzieci danego wierzchołka. Algorytm „LB-KillerMoves" próbuje uporządkować wierzchołki-następniki tak, aby najpierw przeglądane były ruchy, które w poprzedniej gałęzi drzewa, na tej samej głębokości, spowodowały odcięcia i polega on na zapamiętywaniu ruchów, które powodują odcięcia w tablicy, indeksowanej poziomami zagłębienia w drzewie.

2 Mowa tutaj o algorytmach: defenceMoves, killerMoves, sideAttack i safePassage



Strategia „LB-KillerMoves” opiera się w ogólności na założeniu, że jeżeli Twój przeciwnik ma jeden dobry ruch, a Twoim zadaniem jest powstrzymanie go przed wykonaniem go, to właśnie ten ruch stanie się ruchem zabójcą.

Heurystyka zabójcy zakłada, że ruch, który w poprzednich gałęziach na danym poziomie zagłębienia dawał dobre efekty (powodował odcięcia) być może również w tej gałęzi spowoduje odcięcie i zaoszczędzi czasu na przeszukiwanie innych ruchów. Dlatego też „ruchy zabójcy” muszą być przeszukiwani jako pierwsi.

Odnośnie punktu pierwszego:
Nasza strategia identyfikowania ruchów oczywistych składa się z 4 mniejszych algorytmów:
1) defenceMoves
2) killerMoves,
3) sideAttack,
4) safePassage

Podzieliliśmy planszę na „pasy”, które są następujące:

Zgodnie z zasadami gry, gdy pionek przeciwnika znajdzie się w pasie przegranej, gra kończy się naszą przegraną. Gdy pionek przeciwnika znajduje się w pasie obrony, jedynym poprawnym ruchem jest możliwa obrona. Jako ziemię niczyją przyjęliśmy dwa pasy, w których nie znajdują się pionki przeciwnika. Pas killer-moves to pas, z którego jedynym poprawnym ruchem jest atak.

1) **Algorytm defenceMoves**

opiera się na wykrywaniu stanu zagrożenia przegraną i polega na szukaniu pionków przeciwnika w naszym pasie obrony. Jeżeli taki pionek znajdziemy, próbujemy znaleźć pionek w naszym pasie przegranej, który mógłby danego pionka zbić. Gdy pionek taki zostanie znaleziony jako najlepszy ruch w danym momencie ustawiamy oczywiście owo bicie.

Kod źródłowy algorytmu znajduje się na końcu sprawozdania.

2) **Algorytm killerMoves**

jest naszą własną implementacją „killer moves" i ma znacznie mniejszą złożoność, gdyż nie korzysta oczywiście z alpha-bety. Polega on na wykrywaniu pionka danego gracza w pasie killer-moves. Jeżeli taki pionek zostanie znaleziony sprawdzamy jego dozwolone ruchy, jeżeli możliwa jest wygrana, wykonujemy ruch prowadzący do wygranej bez analizy stanów za pomocą alpha-bety.

Kod źródłowy algorytmu znajduje się na końcu sprawozdania.

3) **Algorytm sideAttack**

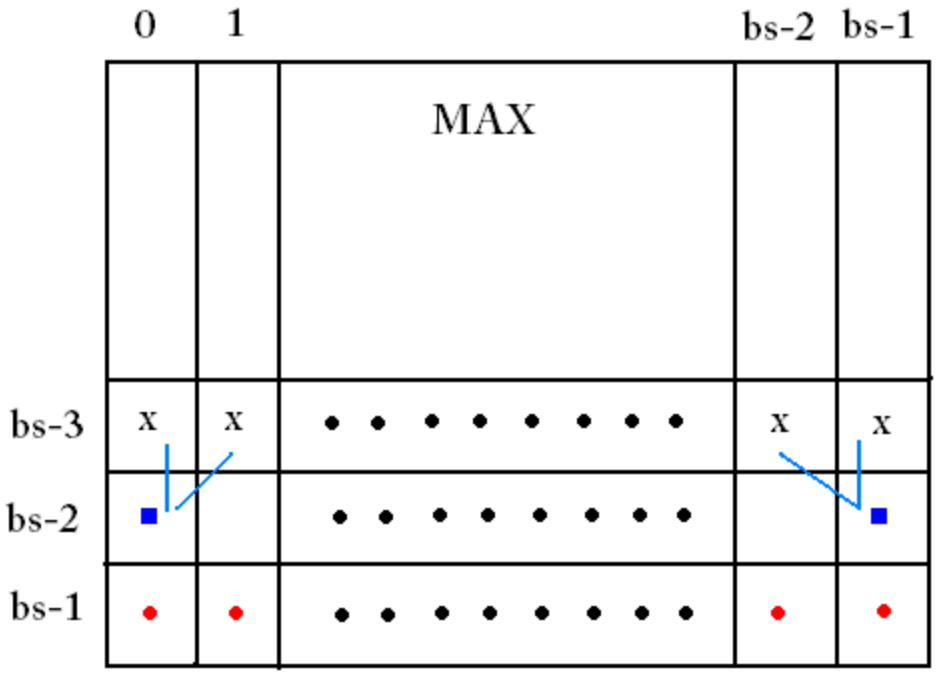
próbuje zaatakować wroga od boku – taki sposób jest niezwykle efektywny, gdyż bicie przeciwnika może nastąpić tylko z jednej strony. Jeżeli warunki zastosowania sideAttack – to znaczy odpowiednio puste:

a) w przypadku gracza MAX, pola w ostatnim dolnym rzędzie – dwa pierwsze od lewej lub dwa pierwsze od prawej oraz obecność pionka w trzecim rzędzie od dołu w kolumnie pierwszej lub drugiej od lewej lub w drugim przypadku obecność pionka w trzecim rzędzie od dołu w kolumnie pierwszej lub drugiej od prawej.

b) w przypadku gracza MIN, pola w pierwszym górnym rzędzie – dwa pierwsze od lewej lub dwa pierwsze od prawej oraz obecność pionka w trzecim rzędzie od góry w kolumnie pierwszej lub drugiej od

lewej lub w drugim przypadku obecność pionka w trzecim rzędzie od góry w kolumnie pierwszej lub drugiej od prawej.

Strategię side-attack dla gracza MAX przedstawia rysunek:



Gdy chcemy wykonać atak z pola (0, bs-3) sprawdzamy, czy na polu (0, bs-2) jest pionek przeciwnika. Jeżeli go nie ma, to możemy wykonać atak. Jeżeli jest to sprawdzamy, czy możemy wykonać jeszcze inny z ataków bocznych. Gdy chcemy wykonać atak z (1, bs-3) to nie jest ważne, czy na polu (0, bs-2) jest pionek, gdyż w razie czego możemy go zbić.

Po wykonaniu ruchu na pole (0, bs -2) sytuację końcową wykrywa algorytm *killerMoves.*

Podobnie algorytm zachowuje się dla „prawej strony planszy".

4) **Algorytm safePassage**

opiera się na założeniu, że gdy od wroga oddziela nas dwu-rzędowy pas ziemi niczyjej, próbujemy atakować wroga.

Algorytm sprawdza numer rzędu naszego najbardziej wysuniętego pionka. Jeżeli w obu rzędach przed nim nie ma żadnego pionka przeciwnika, oznacza to dla niego, że nie ma zagrożenia ze strony wroga i można przystąpić do ataku. Szukamy więc wszystkich naszych pionków w najbardziej wysuniętym rzędzie i poruszamy do przodu jednego z nich.

Należy zauważyć, że strategia ta jest bezpieczna, ponieważ wróg znajduje się na tyle daleko, że nam nie zagraża, a równocześnie zmusza gracza do ruchów ofensywnych. Obliczenia potrzebne do wykrycia takiej sytuacji są znikome w porównaniu z alpha-betą.

Podsumowując:

Algorytmy defenceMoves, killerMoves, sideAttack i safePassage są wykonywane przed alpha-betą i służą wykrywaniu ruchów, oczywistych – strategii, które można wykryć algorytmami o małej złożoności, a pokrywających dość sporo przypadków, a tym samym zmniejszyć częstość odpalania alpha-bety.

Jeżeli żaden ruch oczywisty nie występuje uruchamiana jest alpha-beta, którą wspomaga algorytm LBKillerMoves.

4. Złożoność obliczeniowa użytych algorytmów

1) Złożoność algorytmu alpha-beta:
Jak czytamy z materiałów[3] złożoność obliczeniowa wynosi:

a) w najlepszym przypadku: $O(b^{d/2})$
b) w najgorszym przypadku (czyli brak odcięć): $O(b^d)$
c) w średnim przypadku: $O((b/\log b)^d)$

gdzie:
b – branching factor
d – określona głębokość przeszukiwania

2)Złożoność algorytmu Breaker:

Breaker = defenceMoves, killerMoves, sideAttack i safePassage + LBKillerMoves

a) dla algorytmu: defenceMoves

O(BoardSize)

gdzie:
BoardSize – jest rozmiarem planszy

b) dla algorytmu: killerMoves

O(BoardSize)

gdzie:
BoardSize – jest rozmiarem planszy

c)dla algorytmu: sideAttack

O(1)

d)dla algorytmu: safePassage

$O(BoardSize^2)$

gdzie:
BoardSize – jest rozmiarem planszy

e)dla algorytmu: LBKillerMoves

Algorytm LBKillerMoves ma taka samą złożoność jak Alpha-Beta. Powinien jednak dużo szybciej odcinać.

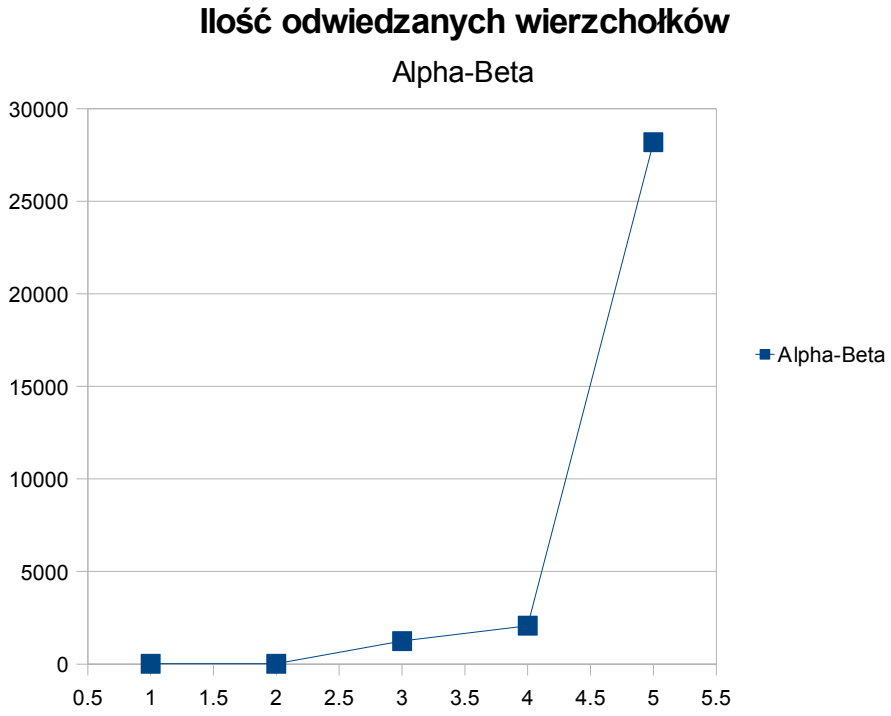
5. Badanie działania algorytmów:

3 http://www.cs.put.poznan.pl/amichalski/search/si2006.bw2pp.pdf

1) Badanie działania algorytmu alpha-beta

Uznaliśmy, że najlepiej będzie pokazać efektywność odcięć L-B poprzez wykreślenie ilości odwiedzanych wierzchołków do głębokości przeszukiwania.

								SREDNIO:
zagł.							zagł.	Alpha-Beta
1	19	20	20	20	19	20	1	17
2	19	20		20	19	20	2	16.67
3	1228	1762	1419	1536	1264	1454	3	1238
4	744	2853	3685	2999	4231	2	4	2074
5	34975	23726	36217	30182	37195	34981	5	28183



Badanie było przeprowadzane na planszy o rozmiarach 7x7, dla różnych sytuacji na planszy. Średnie wartości odzwierciedla wykres powyżej. Widzimy, że zgodnie ze złożonością średnią algorytmu wartości rosną wykładniczo. Badania nie przeprowadzaliśmy powyżej 5 wierzchołków ze względu na bardzo długi czas oczekiwania na wynik.

W grze głębokość została ustawiona na wartość 3, gdyż nie chcieliśmy, aby czas oczekiwania na ruch przeciwnika znacząco różnił się od reakcji człowieka na podobne zadanie.

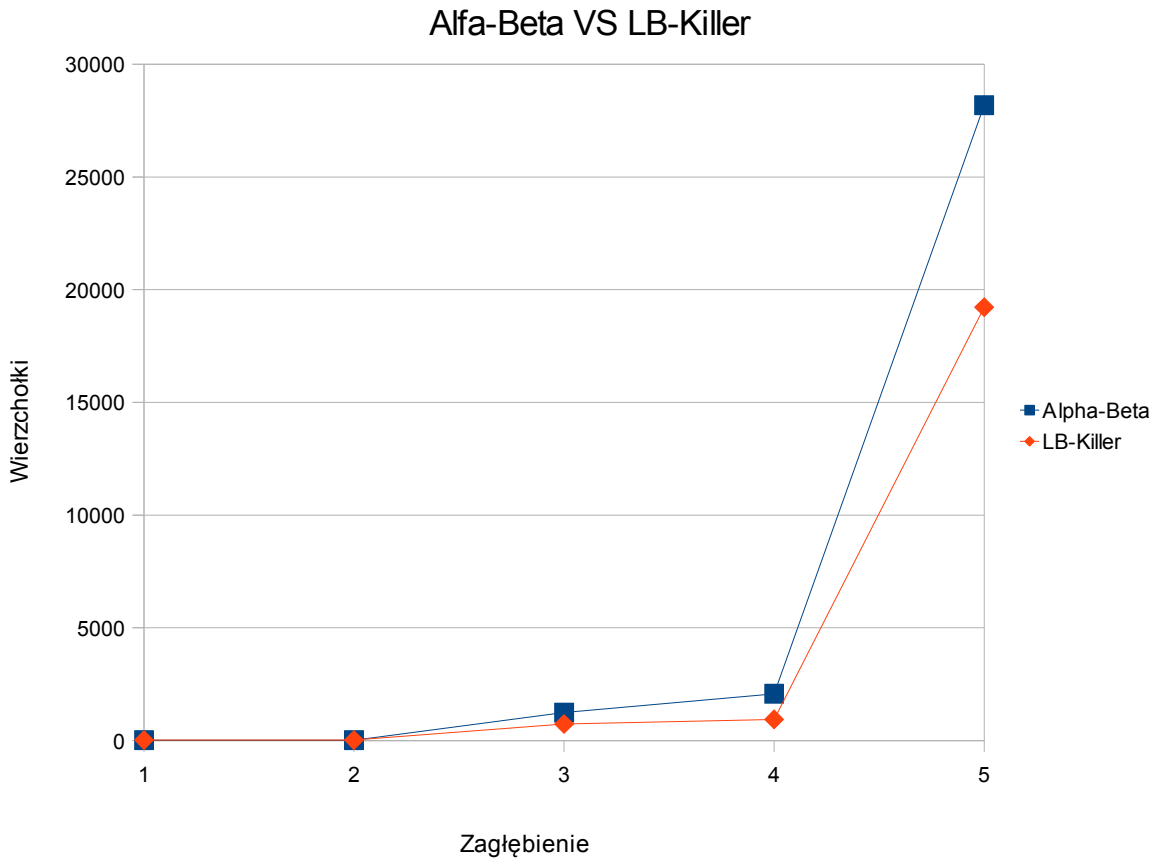
Wynik Alpha-bety nie jest zachwycający, ale i tak znacząco różni się od przeszukiwania brute-force.

2) Badanie działania algorytmu LB-KillerMoves

Te same pomiary zostały wykonane wykonane dla algorytmu LB-KillerMoves. Wyniki zgromadziliśmy w tabelce i przedstawiliśmy na wykresie poniżej:

zagł.						zagł.	LB-Killer
1	19	20	21	19	22	1	20.2
2	19	20	21	19	22	2	20.2
3	753	725	784	788	623	3	734.6
4	1257	790	881	830	945	4	940.6
5	20705	14090	21440	17867	22019	5	19224.2

Liczba odwiedzonych wierzchołków



Widzimy, że dla algorytmu LB-KillerMoves wykres rośnie dużo wolniej. Przykładowo dla 3 wierzchołków, przy zwykłej LB mamy średnio 1238 odwiedzonych wierzchołków, podczas gdy w LB-KillerMoves mamy ich tylko 734. Kształt wykresu jest oczywiście wykładniczy, gdyż KillerMoves nie zmieniają złożoności algorytmu, a jedynie poprawiają wykorzystanie odcięć.

3) Procentowe odzwiercielenie poprawy po zastosowaniu LB-KillerMoves

Zagłębienie = 3
Rozmiar planszy 7x7

	Liczby odwiedzonych wierzch	
	LBKiller	Alpha-Beta
	760	1228
	784	1843
	963	1657
	810	7255
	583	1039
	505	1341
	722	1410
	704	1028
	824	1673
	781	1282
ŚREDNIO:	743.6	1975.6
POPRAWA:	62.36	%

Widzimy, że po zastosowaniu algorytmu LB-KillerMoves ilość przeglądanych wierzchołków spada o około 60%. Naszym zdaniem jest to rewelacyjny wynik. Poprawę szybkości w przeglądaniu możliwych ruchów da się odczuć podczas gry. Czas oczekiwania jest przecież średnio 2 razy krótszy!

W połączeniu z algorytmami defenceMoves, killerMoves, sideAttack i safePassage komfort gry wyraźnie się poprawia. Należy zauważyć także, że sytuacje pokrywane przez killerMoves i defenceMoves są częste, gdyż pokrywają ostateczny atak i obronę. Zmniejszenie czasu oczekiwania przy tych właśnie ruchach było dla nas szczególnie ważne.

6. Kody źródłowe algorytmów:

a) dla algorytmu: defenceMoves:

```
boolean defenceMoves(Integer[][] boardOfNumbers, int type){

    int boardSize = boardOfNumbers.length; //wielkosc planszy
    boolean defence = false;

      if (type == MAX){
         for (int x=0; x < boardSize; x++){
            if ((boardOfNumbers[x][1] == MIN) && (defence == false)){ //jezeli na przedostatnim polu znajduje sie pionek przeciwnika

               if ((x-1 >= 0) && (boardOfNumbers[x-1][0] == MAX)){ //po skosie, w lewo, do gory
                  defence = true;
                  bestMove.player = MAX;
                  bestMove.fromx = x-1; bestMove.fromy = 0;
                  bestMove.tox = x; bestMove.toy = 1;
               }
               else if ((x+1 <= boardSize -1) && (boardOfNumbers[x+1][0] == MAX)){ //po skosie, w prawo, do gory
                  defence = true;
                  bestMove.player = MAX;
                  bestMove.fromx = x+1; bestMove.fromy = 0;
                  bestMove.tox = x; bestMove.toy = 1;
               }

            }
         }
      }
      else if (type == MIN){
         for (int x=0; x < boardSize; x++){
```

```
            if ((boardOfNumbers[x][boardSize - 2] == MAX) && (defence == false)){ //jezeli na przedostatnim polu znajduje sie pionek przeciwnika

               if ((x-1 >= 0) && (boardOfNumbers[x-1][boardSize - 1] == MAX)){ //po skosie, w lewo, do gory
                  defence = true;
                  bestMove.player = MIN;
                  bestMove.fromx = x-1; bestMove.fromy = boardSize - 1;
                  bestMove.tox = x; bestMove.toy = boardSize - 2;
               }
               else if ((x+1 <= boardSize -1) && (boardOfNumbers[x+1][0] == MAX)){ //po skosie, w prawo, do gory
                  defence = true;
                  bestMove.player = MIN;
                  bestMove.fromx = x+1; bestMove.fromy = boardSize - 1;
                  bestMove.tox = x; bestMove.toy = boardSize - 2;
               }

            }
         }
      }

      if (defence == true){
         System.out.println("\ndefence!!!!");
         return true;
      }
      else
         return false;

   }
```

b) dla algorytmu: killerMoves

```
boolean killerMoves(Integer[][] boardOfNumbers, int type){

   int boardSize = boardOfNumbers.length; //wielkosc planszy
   boolean killer = false;

      if (type == MAX){
         for (int x=0; x < boardSize; x++){
            if ((boardOfNumbers[x][boardSize - 2] == MAX) && (killer == false)){ //jezeli na przedostatnim polu znajduje sie nasz pionek

               if (x-1 >= 0){ //po skosie, w lewo, do dolu
                  killer = true;
                  bestMove.player = MAX;
                  bestMove.fromx = x; bestMove.fromy = boardSize - 2;
                  bestMove.tox = x-1; bestMove.toy = boardSize - 1;
               }
               else if (boardOfNumbers[x][boardSize - 1] == 0){ //prost na dol
                  killer = true;
                  bestMove.player = MAX;
                  bestMove.fromx = x; bestMove.fromy = boardSize - 2;
                  bestMove.tox = x; bestMove.toy = boardSize - 1;
               }
               else if (x+1 <= boardSize -1){ //po skosie, w prawo, do dolu
                  killer = true;
                  bestMove.player = MAX;
                  bestMove.fromx = x; bestMove.fromy = boardSize - 2;
                  bestMove.tox = x+1; bestMove.toy = boardSize - 1;
               }

            }
         }
      }
      else if (type == MIN){
         for (int x=0; x < boardSize; x++){
            if ((boardOfNumbers[x][1] == MIN) && (killer == false)){ //jezeli na przedostatnim polu znajduje sie nasz pionek

               if (x-1 >= 0){ //po skosie, w lewo, do gory
                  killer = true;
                  bestMove.player = MIN;
                  bestMove.fromx = x; bestMove.fromy = 1;
                  bestMove.tox = x-1; bestMove.toy = 0;
               }
               else if (boardOfNumbers[x][0] == 0){ //prost do gory
                  killer = true;
```

```
                bestMove.player = MIN;
                bestMove.fromx = x; bestMove.fromy = 1;
                bestMove.tox = x; bestMove.toy = 0;
            }
            else if (x+1 <= boardSize -1){ //po skosie, w prawo, do gory
                killer = true;
                bestMove.player = MIN;
                bestMove.fromx = x; bestMove.fromy = 1;
                bestMove.tox = x+1; bestMove.toy = 0;
            }

        }
    }
}

if (killer == true){
    System.out.println("\nkiller move!!!!");
    return true;
}
else
    return false;

}
```

c)dla algorytmu: sideAttack

```
boolean sideAttack(Integer[][] boardOfNumbers, int type){

    int boardSize = boardOfNumbers.length; //wielkosc planszy
    boolean sattack = false;

    //System.out.println("\nboardSize: " + boardSize);

    sattack = false;

    if (type == MAX){
        if ((boardOfNumbers[0][boardSize - 1] == 0) && (boardOfNumbers[1][boardSize - 1] == 0) && (sattack == false)){ //lewy-dolny rog

            if ((boardOfNumbers[0][boardSize - 3] == MAX) && (boardOfNumbers[0][boardSize - 2] == 0) && (sattack == false)){
                sattack = true;
                bestMove.player = MAX;
                bestMove.fromx = 0; bestMove.fromy = boardSize - 3;
                bestMove.tox = 0; bestMove.toy = boardSize - 2;
            }
            else if ((boardOfNumbers[1][boardSize -3] == MAX) && (sattack == false)){
                sattack = true;
                bestMove.player = MAX;
                bestMove.fromx = 1; bestMove.fromy = boardSize - 3;
                bestMove.tox = 0; bestMove.toy = boardSize - 2;
            }

        }
         else if ((boardOfNumbers[boardSize -2][boardSize -1] == 0) && (boardOfNumbers[boardSize -1][boardSize -1] == 0) && (sattack == false)){//prawy-dolny
rog

            if ((boardOfNumbers[boardSize -1][boardSize -3] == MAX) && (boardOfNumbers[boardSize -1][boardSize - 2] == 0) && (sattack == false)){
                sattack = true;
                bestMove.player = MAX;
                bestMove.fromx = boardSize -1; bestMove.fromy = boardSize - 3;
                bestMove.tox = boardSize -1; bestMove.toy = boardSize - 2;
            }
            else if ((boardOfNumbers[boardSize -2][boardSize -3] == MAX) && (sattack == false)){
                sattack = true;
                bestMove.player = MAX;
                bestMove.fromx = boardSize - 2; bestMove.fromy = boardSize - 3;
                bestMove.tox = boardSize - 1; bestMove.toy = boardSize - 2;
            }

        }

    }
    else if (type == MIN){
```

```
        if ((boardOfNumbers[0][0] == 0) && (boardOfNumbers[1][0] == 0) && (sattack == false)){ //lewy-gorny rog

            if ((boardOfNumbers[0][2] == MIN) && (boardOfNumbers[0][1] == 0) && (sattack == false)){
                sattack = true;
                bestMove.player = MIN;
                bestMove.fromx = 0; bestMove.fromy = 2;
                bestMove.tox = 0; bestMove.toy = 1;
            }
            else if ((boardOfNumbers[1][2] == MIN) && (sattack == false)){
                sattack = true;
                bestMove.player = MIN;
                bestMove.fromx = 1; bestMove.fromy = 2;
                bestMove.tox = 0; bestMove.toy = 1;
            }

        }
        else if ((boardOfNumbers[boardSize - 2][0] == 0) && (boardOfNumbers[boardSize - 1][0] == 0) && (sattack == false)){//prawy-gorny  rog

            if ((boardOfNumbers[boardSize - 1][2] == MIN) && (boardOfNumbers[boardSize -1][1] == 0) && (sattack == false)){
                sattack = true;
                bestMove.player = MIN;
                bestMove.fromx = boardSize -1; bestMove.fromy = 2;
                bestMove.tox = boardSize -1; bestMove.toy = 1;
            }
            else if ((boardOfNumbers[boardSize -2][2] == MIN) && (sattack == false)){
                sattack = true;
                bestMove.player = MIN;
                bestMove.fromx = boardSize - 2; bestMove.fromy = 2;
                bestMove.tox = boardSize - 1; bestMove.toy = 1;
            }

        }

    }

    if (sattack == true){
        System.out.println("\nside attack!!!!");
        return true;
    }
    else
        return false;



}
```

d)dla algorytmu: safePassage

```
boolean safePassage(Integer[][] boardOfNumbers, int type){

    int boardSize = boardOfNumbers.length; //wielkosc planszy
    boolean sPassage = false;
    boolean found = false;
    Pole    pole1 = new Pole();
    boolean pionekPrzec = false;
    int[]   mojePionki = new int[boardSize];
    int     mojePionkiIter = 0;
    java.util.Random rand = new java.util.Random();

      sPassage = false;

      if (type == MAX){
          for (int y = boardSize - 1; y >= 0; y--){
              for (int x =0; x < boardSize; x++){
                  if (boardOfNumbers[x][y] == MAX){
                      found = true;
                      pole1.x = x;
                      pole1.y = y;
                  }
                  if (found == true)
                      break;
```

```
            }
            if (found == true)
                break;
        }

        System.out.println("found: " + found + " Pierwszy pionek: " + pole1.x + " " + pole1.y);

        //sprawdz, czy dwa rzedy do przodu sa wolne
        if ( (found == true) && (pole1.y + 2 <= boardSize - 1))
        {
            pionekPrzec = false;
            for (int x =0; x < boardSize; x++){
                if ((boardOfNumbers[x][pole1.y+1] == MIN) || (boardOfNumbers[x][pole1.y+2] == MIN)){
                    pionekPrzec = true;
                }
            }

            System.out.println("Pionek przeciwnika: " + pionekPrzec);

            if (!pionekPrzec){
                for (int x =0; x < boardSize; x++){
                    if (boardOfNumbers[x][pole1.y] == MAX){
                        mojePionki[mojePionkiIter++] = x;
                    }
                }

                pole1.x = mojePionki[rand.nextInt(mojePionkiIter)];

                sPassage = true;
                bestMove.player = MAX;
                bestMove.fromx = pole1.x; bestMove.fromy = pole1.y;
                bestMove.tox = pole1.x; bestMove.toy = pole1.y + 1;


            }
        }
    }
    else if (type == MIN){

        for (int y = 0; y < boardSize; y++){
            for (int x =0; x < boardSize; x++){
                if (boardOfNumbers[x][y] == MIN){
                    found = true;
                    pole1.x = x;
                    pole1.y = y;
                }
                if (found == true)
                    break;
            }
            if (found == true)
                break;
        }

        //sprawdz, czy dwa rzedy do przodu sa wolne
        if ( (found == true) && (pole1.y - 2 >= 0))
        {
            for (int x =0; x < boardSize; x++){
                if ((boardOfNumbers[x][pole1.y-1]) == MAX || (boardOfNumbers[x][pole1.y-2] == MAX)){
                    pionekPrzec = true;
                }
            }

            if (!pionekPrzec){
                for (int x =0; x < boardSize; x++){
                    if (boardOfNumbers[x][pole1.y] == MIN){
                        mojePionki[mojePionkiIter++] = x;
                    }
                }

                pole1.x = mojePionki[rand.nextInt(mojePionkiIter)];

                sPassage = true;
                bestMove.player = MIN;
                bestMove.fromx = pole1.x; bestMove.fromy = pole1.y;
```

```
            bestMove.tox = pole1.x; bestMove.toy = pole1.y - 1;


          }
        }

    }


    if (sPassage == true){
        System.out.println("\nsafe Passage :)");
        return true;
    }
    else
        return false;

}
```

e)dla algorytmu: LBKillerMoves