11.05.2023

Sztuczna Inteligencja i Inżynieria Wiedzy

Lab 2

Autor: Jakub Szwedowicz 243416

Prowadzący: dr inż. Piotr Syga

Zajęcia: czw 7:30

Spis treści

1. Zadania

1.1. Poprawne zdefiniowanie stanu gry i funkcji generującej możliwe ruchy dla danego stanu i gracza.

Bazując na drzewiastej strukturze przebiegu gry, utworzona jest klasa *Tree*, które przechowuje korzeń o typie *Node*. Każdy *Node* przechowuje swoje dzieci (pod warunkiem, że nie jest liściem), które reprezentują kolejne stany gry po wykonaniu jednego ruchu. W węzłach przechowywany jest między innymi stan gry *GameState*, który wewnętrznie zawiera już dwuwymiarową tablicę - listę list - bezpośrednio reprezentującą stan planszy. Rozważane było zastosowanie biblioteki NumPy oraz zastąpienie List[List[int]] bibliotecznym typem ndarray, natomiast charakter problemu sprzyjał bardziej listom. Powodami jakie za tym stoją, a które potwierdziło profilowanie wykazujące spadek wydajności po użyciu ndarray są:

- Częste tworzenie mało elementowych (64) i tylko dwu wymiarowych tablic jest szybsze przy użyciu list niż ndarray.
- Ndarray jest zoptymalizowany pod kątem działań macierzowych i jest w związku z tym wolniejszy w zastosowaniach, których oczekuje gra (zamiana pojedynczych elementów, losowe sprawdzanie pól tablicy)
- Najistotniejsze jednak okazało się często iterowanie po tablicach z równoczesnym sprawdzaniem wielu warunków. Problemem w tym przypadku są występujące w tablicach ndarray typy dostarczone przez bibliotekę numpy. W trakcie iterowania po takich tablicach dochodzi do unboxing-u na typy pythonowe. Efektem czego jest nawet dwukrotne zwiększenie czasu wykonywania niektórych funkcji np. tej odpowiedzialnej za weryfikację poprawności ruchu.

Samo generowanie ruchów graczy opera się na przejściu po tablicy i sprawdzeniu dla każdego pustego pola czy w 8 kierunkach odchodzących od tego pola zachodzi sytuacja, w której występuje pewna ilość pionków przeciwnego gracza a następnie występuje po nich pionek gracza wykonującego ruch. W takiej sytuacji możliwe jest zbicie pionków przeciwnika, a więc jest to możliwy do wykonania ruch.

1.2. Zbudowanie zbioru heurystyk oceny stanu gry dla każdego z graczy, każdy z graczy powinien mieć przynajmniej 3 różne strategie

W projekcie przewidziane zostały trzy heurystyki:

1. Heurystyka „wyniku gry” wyznaczające różnicę w ilości pionków jednego gracza od drugiego.
2. Heurystyka „premiująca rogi planszy” , która zwraca tym większą wartość im bliżej rogów planszy gracz maksymalizujący ma pionki. Oznacza to, że wraz z postawieniem nowego, najbliższego względem niezajętego rogu planszy pionka, rośnie wartość zwracana przez funkcję. Premiuje to zajmowanie rogów, po ich zajęciu zysk z danej operacji zostaje utrzymany, natomiast postawienie w rogu planszy pionka przez przeciwnika jest silnie karane poprzez utratę wszystkich punktów jakie gracz maksymalizujący zyskał zbliżając się do tego rogu.
3. Heurystyka „premiująca przewagę ruchów”, która zwraca tym większą liczbę im więcej dostępnych ruchów ma gracz maksymalizujący w stosunku do minimalizującego. Zamysłem stojącym za tą heurystyką jest idea że posiadanie większej liczby ruchów musi wynikać z

lepszej pozycji pionków w stosunku do przeciwnika, a także z samej liczy własnych pionków. W tym miejscu można zaznaczyć istotną wadę tej heurystyki, czyli że następstwem takie strategii jest karanie za wykonywanie ruchów zbijających większość pionków przeciwnika, gdyż może to doprowadzić do mniejszej liczby dostępnych ruchów w następnej turze gracza maksymalizującego.

1.3. Implementacja metody Minimax z punktu widzenia gracza 1

W zrozumieniu algorytmu minimax istotne jest zauważenie, że opiera się on na dwóch strategiach:

- Gracz maksymalizujący wykonuje ruchy w taki sposób aby wartość heurystyki była jak największa.
- Gracz minimalizujący wykonuje ruchy w taki sposób, aby możliwe najbardziej zmniejszyć wartość heurystyki. Bardzo ważne jest zauważenie, że zadaniem gracza minimalizującego generalnie nie jest żeby wygrać, ale żeby możliwie jak najbardziej utrudnić wygranie graczowi maksymalizującemu. Oczywiście, to czy gracz minimalizujący będzie dążył do wygranej czy raczej do uniemożliwienia graczowi maksymalizującemu wygranej, będzie zależeć od heurystyki. Dla przykładu heurystyka „wyniku gry" będzie prowadzić gracza minimalizującego do zwycięstwa, ale już „premiująca rogi planszy" niekoniecznie. Na pewno efektem utrudniania zwycięstwa drugiemu graczowi jest wygranie samemu natomiast nie to jest celem niektórych z tych heurystyk. Wspomniana właśnie heurystyka będzie np. powodować że celem gracza minimalizującego będzie zabranie graczowi maksymalizującemu ruchów, które zbliżają go do rogu, a w przypadku zbliżenia się – być może zabranie go w ostatnim momencie.

Zważywszy na powyższe, algorytm będzie opierał się na wyznaczeniu drzewa przebiegu gry z założeniem, że jeden z graczy maksymalizuje wartość heurystyki a drugi ją minimalizuje. Ważne zatem jest aby użyte heurystyki zwracały większe (a nie mniejsze) wartości dla ruchów gracza maksymalizującego, które prowadzą do zwycięstwa (według heurystyki).

Ważne z punktu wydajnościowego jest aby zauważyć, że algorytm minimax zawsze przeszukuje całe drzewo decyzyjne, zaś same drzewo rośnie wykładniczo względem swojej głębokości. Prowadzi to do bardzo szybkiego wzrostu liczy przeszukiwanych węzłów.

1.4. Implementacja alfa-beta cięcia z punktu widzenia gracza 1

Algorytm alfa-beta to usprawniona wersja algorytmu minimax, który wprowadza usprawnienie polegające na zapamiętywaniu najlepszego wyniku z innego poddrzewa, co pozwala na wczesne odrzucenie całego następnego poddrzewa jeśli pierwsza sprawdzona w nim ścieżka jest gorsza od najlepszego zapamiętanego wyniku. Problemem w takim podejściu staje się natomiast możliwość utraty lepszej ścieżki niż zapamiętana jeśli znajduje się one w odrzuconym poddrzewie.

Pewną narzucającą się w tym miejscu optymalizacją jest aby sortować drzewo decyzyjne tak aby węzeł końcowy z największą (lub najmniejszą) wartością znajdował się w pierwszej sprawdzonej ścieżce poddrzewa. Problemem z tą optymalizacją jest jednak dość spory narzut pamięciowy (żeby je stworzyć i posortować) i czasowy (żeby to w ogóle zrobić). Być może zatem dobrym konsensusem byłoby determinowanie, który z dostępnych w każdym stanie gry ruchów jest najlepszy i

wykonywanie go jako pierwszego. Taka funkcjonalność nie została jednak wprowadzona w rozwiązaniu.

1.5. Optymalizacja

Zmienione zostały wartości symbolizujące pionki graczy. Zamiast standardowego:

- Gracz1: 1
- Gracz2: 2
- Puste pole: 0

Wprowadzone zostało:

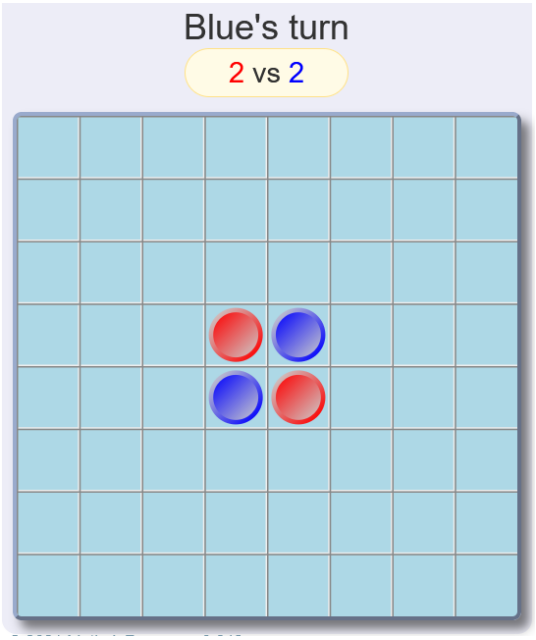
- Gracz1: 1
- Gracz2: -1
- Puste pole: 0

Co ma następujące zalety:

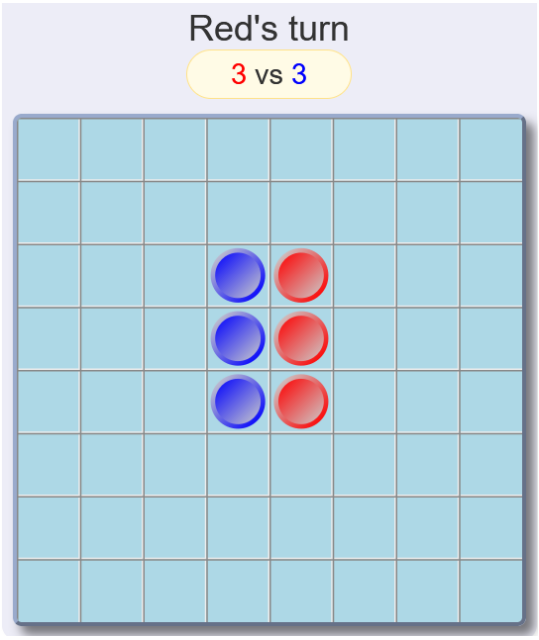
- Zamiana pionka jednego gracza na drugi polega wyłącznie na zamianie znaku (a jest to bardzo częsta operacja)
- Niektóre funkcjonalności polegają na wyliczeniu różnicy pomiędzy pionkami gracz1 i gracza2. W wariancie ze standardową numeracją pionków trzeba sprawdzać czy pole wynosi wartość 2 lub 1 i odpowiednio dodawać. W wariancie usprawnionym nie trzeba polegać na żadnych warunkach i można po prostu obliczyć sumę wszystkich pól na planszy. Znowu z racji na częstość tej operacji jest to odczuwalna poprawa prędkości działania. Zaznaczyć również można, że gdyby tablicą był ndarray to sumowanie pól planszy byłoby jeszcze szybsze natomiast straty powstałe przez iterowanie po takiej tablicy przewyższałyby zyski wynikające z nieco szybszego sumowania pól.

Ponad to wykryte zostają symetryczne ruchy na planszy i są one odrzucane. Pozwala to ponad 4 krotnie zmniejszyć liczbę występujących węzłów. Już w pierwszym ruchu ograniczona zostaje liczba dostępnych ruchów z czterech do jednego, gdyż występują pomiędzy tymi ruchami symetrie względem przekątnych planszy oraz złożenie tych dwóch w postaci symetrii względem punktu środka planszy. Istotne w zrozumieniu jak wykrywać takie przypadki jest podanie kilku przykładów.

Korzystając z webowej wersji reversi: https://www.mathsisfun.com/games/reversi.html można zauważyć kilka właściwości:



Rysunek 1.1. Początkowa sytuacja na planszy



Rysunek 1.2. Symetria lokalna

Na podstawie rysunku 1.1 . można zauważyć, że występuje kilka osi symetrii:

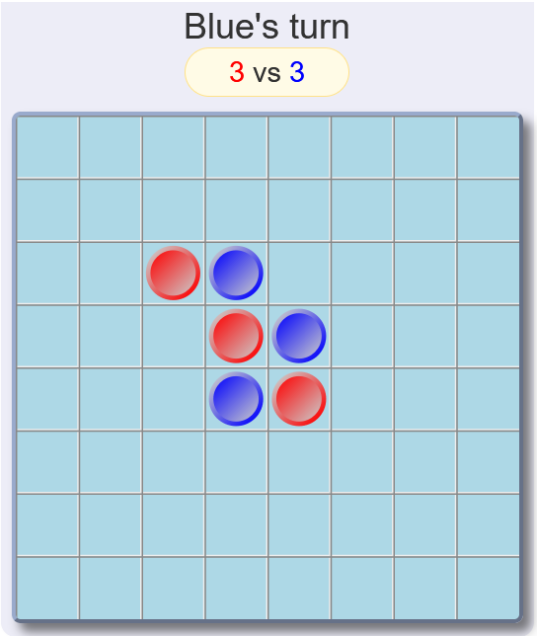
1. diagonalna przechodząca przez pionki gracza2 (niebieski)
2. druga diagonalna przechodząca przez pionki gracza1 (czerwony)

Na początek warto zauważyć, że obaj gracze mają na początku gry do wykonania po cztery ruchy z czego obie pary (dla czerwonego jest to np. para (2, 4) oraz (4,2)) są symetryczne względem którejś z osi. Można również zauważyć, że choć na początku gry gracz czerwony ma możliwość wykonać ruchy symetryczne względem osi przechodzącej przez pionki gracza niebieskiego to taka sytuacja występuje wyłącznie na początku partii. Zasadą, którą można zauważyć jest zatem że dany gracz ma możliwość wykonywania ruchów symetrycznie tylko względem osi przechodzącej na początku gry przez jego pionki. Wynika to z faktu, że w swoich kolejnych ruchach może wykonać takie same ruchy odbite względem tej właśnie osi. Ze względu jednak na początkowe ułożenie pionków, nie może on swoich ruchów odbijać względem drugiej osi.

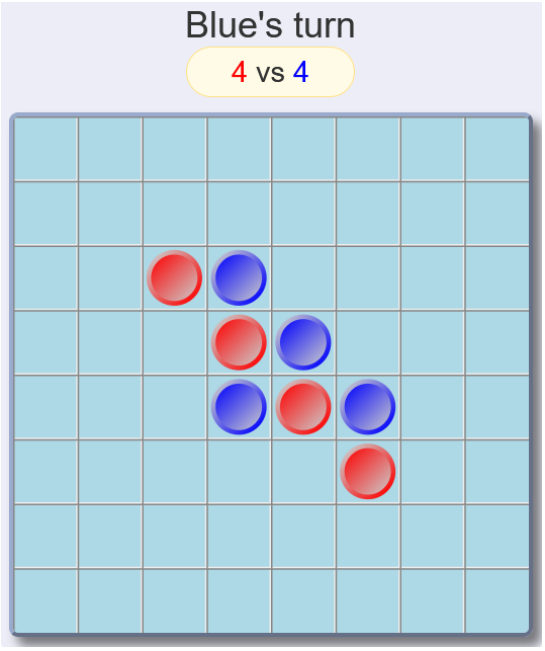
Rysunek 1.2 pokazuje ciekawą właściwość planszy polegającą na tym, że sam fakt występowania lokalnych osi symetrii (w tym przypadku oś przechodzi przez rząd 3) nie upoważnia do uproszczenia drzewa decyzyjnego. Dla przykładu z rysunku możemy zauważyć, że gracz czerwony ma dwa symetryczne względem wymienionej osi ruchy: (2, 2) oraz (4, 2) natomiast nie są one jednakowe w

skali całej gry. Pierwszy z nich doprowadzi do większego zajęcia lewej, górnej ćwiartki planszy natomiast drugi wzmocni pozycję na środku. W oczywisty sposób wpłynie to na dalszy przebieg gry. Powstaje zatem w tym miejscu wniosek, że same lokalne osie symetrii nie upoważniają do uproszczenia drzewa decyzyjnego.

Wydaje się zatem, że warunkiem dla uproszczenia drzewa decyzyjnego jest aby sytuacja na planszy była symetryczna względem globalnych osi symetrii. Oznacza to, że gdyby podzielić planszę z rys. 1.1. osią przechodzącą przez niebieskie pionki to aby można było uprościć ruchy to sytuacja w jednej połowie planszy (wyznaczonej przez oś symetrii) musi być identyczna po drugiej stronie. Innymi słowy każdy pionek ma taką samą wartość na polu po jednej stronie planszy oraz na polu obitym względem osi (czyli np. piony (3,3) oraz (4,4)).



Rysunek 1.3. Utrata symetrii



Rysunek 1.4. Przywrócenie symetrii względem osi

Przykład z rysunku 1.3 pokazuje sytuację, w której zaczynał gracz niebieski, który może grać symetrycznie względem osi przechodzącej przez pionki jego koloru. W wyniku sytuacji na planszy chwilowo układ pionów nie jest symetryczny względem żadnej z globalnych osi.

Rysunek 1.4 pokazuje potencjalny rozwój sytuacji z rysunku 1.3. Na powyższym rysunku można zauważyć, że po wykonaniu pewnych ruchów po drugiej stronie planszy gracz niebieski (z pomocą czerwonego) przywrócił symetryczną sytuację pionków. Nasuwa się w tym miejscu zatem wniosek, że symetrię na planszy można chwilowo utracić i może ona ponownie wystąpić w późniejszych etapach drzewa decyzyjnego.

Kluczowe zatem dla eliminacji symetrycznych planszy i zmniejszenie rozmiaru drzewa decyzyjnego staje się rozpoznawanie symetrycznego układu pionów względem osi globalnych. Pewną optymalizacją jaką można wprowadzić na podstawie spostrzeżeń z rysunków 1.3 oraz 1.4 jest że symetria w niższych poziomach drzewa pojawia się ponownie wtedy gdy odbite zostaną wcześniejsze ruchy. Oznacza to że można zapamiętywać ruchy, które nie mają swojego symetrycznego

odpowiednika po drugiej stronie planszy oraz wykrywać sytuację, w której lista przechowująca te ruchy została wyczyszczona. Jeśli taka sytuacja nastąpi to na planszy powstałej z takiego ruchu wszystkie ruchy będą miały swoje odpowiedniki po drugiej stronie osi. Można zatem usunąć wszystkie ruchy występujące po jednej stronie osi. Funkcjonalność tę w aplikacji obsługuje klasa *MoveWatcher*.

1.6. Podsumowanie

Plik „benchmarkOutput_minimax_alphaBeta_all_heuristics” zawiera informacje z przebiegu algorytmów według poniższych kombinacji:

- Głębokość drzewa od 5 do 11
- Użyte algorytmy to minimax oraz alpha-beta cięcie
- Wykorzystane heurystyki to wspomniane wcześniej „wynik gry”, „premiująca rogi planszy” oraz „premiująca przewagę ruchów”.

Uzyskane wyniki prezentują się następująco:

Tabela 1.1. Czasowe przebiegi algorytmów z różnymi heurystykami i głębokościami drzewa

Głębokość	minimax			Alpha-beta		
	„wynik gry”	„premiująca rogi planszy”	„premiująca przewagę ruchów”	„wynik gry”	„premiująca rogi planszy”	„premiująca przewagę ruchów”
5	0.0037 [s] (80N)	0.01 [s] (159N)	0.037 [s] (159N)	0.0051 [s] (86N)	0.0072 [s] (82N)	0.0188 [s] (87N)
6	0.0277 [s] (849N)	0.0540 [s] (849N)	0.2064 [s] (849N)	0.0134 [s] (244N)	0.0374 [s] (332N)	0.0737 [s] (328N)
7	0.1634 [s] (4905N)	0.3119 [s] (4905N)	1.4564 [s] (4905N)	0.0512 [s] (851N)	0.0856 [s] (947N)	0.2114 [s] (843N)
8	0.9771 [s] (32 163N)	2.1153 [s] (32 163N]	8.2530 [s] (32 163N)	0.1042 [s] (1738N)	0.3351 [s] (4348N)	0.7727 [s] (3110N)
9	6.9463 [s] (225 363N)	14.2559 [s] (225 363N)	56.2581 [s] (225 363N)	0.3810 [s] (7452N)	0.6916 [s] (9171N)	1.9264 [s] (8543N)
10	46.8137 [s] (1 713 465N)	98.5586 [s] (1 713 465N)	423.9768 [s] (1 713 465N)	0.9594 [s] (21 676N)	2.1774 [s] (32 416N)	8.9158 [s] (40 056N)
11	376.6405 [s] (13 881 665N)	833.6964 [s] (13 881 665N)	3528.1982 [s] (13 881 665N)	3.9266 [s] (77 610N)	9.7860 [s] (128 223N)	30.3256 [s] (127 714N)

Gdzie:

- [s] – sekundy
- N – liczba odwiedzonych węzłów

Na ich podstawie można zgodnie z oczekiwania zauważyć, że czas przeszukiwania całego drzewa decyzyjnego rośnie wykładniczo i stąd nawet osiągając 1/6 głębokości całego drzewa decyzyjnego – algorytm minimax przetwarza drzewo w zależności od użytej heurystyki od 6 minut do prawie 1h. Dla porównania algorytm alpha-beta potrzebował od 4 do 30 sekund. Warto również zaznaczyć, że

wzrost czasu wykonania jest również znacznie mniejszy od wzrostu czasu dla minimax co każe sądzić, że dla większych drzew algorytm alpha-beta może być jedyną właściwą opcją z tych dwóch dostępnych.

Poniżej znajduje się fragment pliku wynikowego z przebiegu profilera:

```
The execution took 37.8708379[s], visited nodes 856733
         76841525 function calls (75984677 primitive calls) in 37.872 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 50785458   14.511    0.000   14.511    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:85(_is_valid_direction)
  7211776    9.270    0.000   21.131    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:71(_is_valid_move)
  6696531    1.301    0.000    1.301    0.000 {built-in method builtins.sum}
  1139910    0.130    0.000    0.130    0.000 {method 'append' of 'list' objects}
   958461    0.507    0.000    0.507    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:136(_flip_direction)
856733/1    2.389    0.000   37.871   37.871 C:\Users\kubas\Dev\SIIW\lab2\algorithms.py:30(_minimax)
   856732    0.264    0.000    0.264    0.000 C:\Users\kubas\Dev\SIIW\lab2\tree.py:6(__init__)
   856732    0.759    0.000    0.759    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:46(<listcomp>)
   856732    0.940    0.000    2.067    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:35(__init__)
   856732    3.113    0.000    7.217    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:123(make_move)
   856732    0.368    0.000    0.368    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:171(__init__)
   856732    0.428    0.000    0.946    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:192(add_move)
   744059    0.313    0.000    2.856    0.000 C:\Users\kubas\Dev\SIIW\lab2\tree.py:16(calculate_heuristic)
   744059    0.800    0.000    1.947    0.000 C:\Users\kubas\Dev\SIIW\lab2\algorithms.py:128(<listcomp>)
   744059    0.441    0.000    2.543    0.000 C:\Users\kubas\Dev\SIIW\lab2\algorithms.py:118(heuristic_game_score)
   508313    0.053    0.000    0.053    0.000 {built-in method builtins.len}
   342580    0.147    0.000    0.197    0.000 C:\Users\kubas\AppData\Local\Programs\Python\Python39\lib\types.py:171(__get__)
   342579    0.050    0.000    0.050    0.000 C:\Users\kubas\AppData\Local\Programs\Python\Python39\lib\enum.py:792(value)
   342374    0.244    0.000    0.440    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:7(get_symmetrical_field)
   112684    1.780    0.000   23.003    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:55(get_valid_moves)
   112682    0.049    0.000    0.065    0.000 C:\Users\kubas\Dev\SIIW\lab2\reversi.py:204(is_board_symmetrical)
```

Rysunek 1.5. Fragment wyniku profilera wywołanego dla algorytmu minimax z heurystyką "wynik gry" i głębokością 10

Na jego podstawie można zauważyć, że program najwięcej czasu spędza na weryfikowaniu czy ze wskazanego pola występuje bicie we wszystkich ośmiu kierunkach. Byłoby to zatem potencjalne miejsce do usprawnienia czasu działania programu.

Dość intuicyjna jest również optymalizacja wynikająca ze sprawdzenia czy sam ruch jest poprawny. Program spędził dużo czasu wewnątrz tej funkcji prawdopodobnie z dwóch powodów: przez unpacking przy iterowaniu po możliwych kierunkach oraz właśnie z samego tytułu częstego iterowania po wszystkich polach planszy (funkcja zostaje wywołana dla każdego pola – w każdej turze na każdej planszy). Nasuwającym się zatem rozwiązaniem jest aby zapamiętywać wiersze i kolumny, w których doszło do zmiany pionów w poprzedniej turze i sprawdzać tylko te kolumny i rzędy, w których doszło do zmiany. Można również pójść o krok dalej i dołożyć ograniczenie sprawdzanych pól w danym szeregu lub kolumnie do wyłącznie pustych pionów graniczących z pionami przeciwnika. Powinno to w istotny sposób ograniczyć sprawdzenia dostępnych ruchów na pustych polach planszy (a w szczególności w początkowych fazach gry gdy większość planszy jest pusta).

Użyte biblioteki:

- cProfile
- timeit