

## **STRESZCZENIE**

Celem niniejszej pracy jest implementacja i analiza wydajnościowa generatora silników gier logicznych. Warunkiem prawidłowego wykonania wskazanych zadań było m.in. dokonanie dokładnego przeglądu i analizy dostępnych implementacji silników popularnych gier.

Właśnie tego typu opisowej analizie, wraz z przytoczeniem kluczowych fragmentów kodów źródłowych programów grających, poświęcony jest cały pierwszy rozdział. Specyfika tematu implikuje omówienie tak istotnych zagadnień, jak dokładna analiza kodu kluczowego dla wydajności czy efektywne reprezentacje planszy i stanów gry. Dużo uwagi poświęcono efektywnym reprezentacjom planszy i stanów gry. Opisano również struktury danych i algorytmy AI dla generowania ruchów. W każdym z analizowanych silników, szczegółowo zostały opisane wszelkie optymalizacje sprzętowe. Dokładnie zanalizowano wszystkie wstawki asemblerowe, funkcje „intrinsic”, czy wołanie bezpośrednio instrukcji procesora. Używanie instrukcji dostępnych dla konkretnych, nowoczesnych architektur procesorów zostało znalezione i opisane w przypadku silników programów mistrzowskich. Łącznie analizie poddano dwa silniki gier logicznych. Analizowany silnik do gry w szachy – Stockfish, jest programem mistrzowskim i opisane, znalezione w nim rozwiązania dobrze zaprezentowały najnowsze trendy w programach grających. Analizie poddano także jedyny otwartoźródłowy silnik Connect-k dla stosunkowo nowej gry Connect-6.

W rozdziale drugim skupiono się na implementacji generatora silników dla gier z rodziny (m,n,k,p,q) (np. Connect-k, GoMoku, Tic-Tac-Toe). Celem było zaimplementowanie generatora, który na podstawie zdefiniowanych przez użytkownika parametrów tworzy silnik do konkretnej gry z tej rodziny. Implementacja okazała się w pełni sukcesem, a generowane silniki poprawne i szybkie. Co ważniejsze, generowane silniki wykorzystują efektywną reprezentację (kompresję) stanów gry z uwzględnieniem zdefiniowanych parametrów. Dla mniejszej planszy jest np. używana zmienna o mniejszym zakresie.

Celem trzeciego i zarazem ostatniego rozdziału jest dokładna analiza poprawnościowo-wydajnościowa zaimplementowanego generatora silników gier. W tym celu opracowano oddzielną aplikację, która była w stanie się komunikować z silnikiem. Testy poprawności wykazały 100% poprawności generowanych silników. Testy wydajności pozytywnie zaskoczyły autora i jednocześnie utrwalili w przekonaniu, że implementacja silnika gry w C/C++ to dobry wybór. Zbadano także zależności wydajności silników od parametrów m,n,k,p,q.

**Słowa kluczowe:** generator silników gier logicznych, gry logiczne, analiza silników gier logicznych, programy grające, connect-6, gomoku, szachy, hex, Stockfish, Connect-k, rozwiązywanie gier, reprezentacje stanów gry, reprezentacje planszy w grach, gry parametryzowane, silniki gier, gry z rodziną (m,n,k,p,q), monte carlo

**Dziedzina nauki i techniki, zgodnie z wymogami OECD:** 1.2 Nauki o komputerach i informatyka

## ABSTRACT

The aim of this work is the implementation and performance analysis of logical games engines generator. However, before implementation it was first necessary to do a thorough review and analysis of available implementations of popular games engines.

The first chapter is devoted to descriptive analysis with in-depth focus on key pieces of engines source code. This is typically descriptive section, but because of the focus on the engines and performance thorough analysis of the source code of each engine was done. Much attention was paid to effective representation of the board and the states of the game. This work also describes the data structures and algorithms for AI moves generation.

In each of the analyzed engines all the hardware optimizations are described in full detail. All the assembler functions, "intrinsic" or direct CPU calls are found and analyzed. Usage of instructions available for specific CPU architectures of modern processors have been found and described.

In total, two logical games engines were analyzed. First one is an engine to play chess - Stockfish is an awards-wining program and its solutions well-presented the latest trends in the logical games engines development. Second one was the only open source engine Connect-k for the relatively new game Connect-6.

The second chapter focuses on the implementation of the engines generator for games of  $(m, n, k, p, q)$  family (eg. Connect-k, Gomoku, Tic-Tac-Toe). The aim was to implement a generator, which based on user-defined parameters, creates the engine for a specific game of  $(m,n,k,p,q)$  family. Implementation proved to be fully successful, the engines were generated correctly and fast. More importantly, the generated engines are using effective representation (compressed) of states of games including the defined parameters. For example, a smaller board is using smaller type for variable used to store bitboard.

The aim of the third and final chapter is a thorough correctness and performance analysis of implemented engines generator. For this purpose, a separate application, which was able to communicate with the engine using shared interface was created. Validation tests showed 100% correctness of generated engines. Performance tests positively surprised the author and at the same time perpetuated the belief that the implementation of the game engine in C/C++ is a good choice. Engines performance depending on the parameters  $m,n,k,p,q$  was also tested.

**Keywords:** logical games engines generator, logic games, logical games engines analysis, programs playing, connect-6, gomoku, chess, hex, Stockfish, Connect-k, solving games, representations of states of the game, board representations, parameterized game, engines,  $(m,n,k,p,q)$  games family, monte carlo

## **SPIS TREŚCI**

Wykaz ważniejszych oznaczeń i skrótów.....	7
1. Wstęp i cel pracy .....	8
2. Przegląd i analiza dostępnych implementacji silników popularnych gier logicznych .....	10
2.1 Stockfish.....	12
2.1.1 Reprezentacja planszy – bitboard.....	15
2.1.2 Wykorzystanie zaawansowanych instrukcji assemblerowych procesora .....	16
2.1.3 Reprezentacja ruchu i wartości dla funkcji ewaluacji.....	19
2.1.4 Fishtest.....	21
2.2 Connect-k .....	22
2.2.1 Zasady gry .....	23
2.2.2 Plik shared.h – definicje stałych i typów używanych w całym programie .....	25
2.2.3 Plansza i stany gry .....	27
2.2.4 Plik state.c i funkcje modyfikujące stan.....	28
2.2.5 Sztuczna inteligencja – ogólnie.....	32
2.2.6 Monte Carlo.....	36
3. Implementacja generatora silników wybranej gry .....	39
3.1 Problem wyboru gry .....	39
3.1.1 Początkowy wybór – szachy .....	40
3.1.2 Wybór ostateczny – Connect6, Gomoku, (m,n,k,p,q) .....	41
3.2 Architektura projektu .....	41
3.3 Generator silników – (m,n,k,p,q)EnginesGenerator .....	42
3.3.1 Podstawowe struktury danych – EngineParameters i EngineScheme .....	44
3.3.2 Schematy silników.....	45
3.3.3 Generacja silników gier logicznych – klasa EnginesGenerator .....	47
3.3.4 Proste interakcje z wygenerowanymi silnikami.....	55
3.3.5 Tryb „batch” .....	61
3.4 Silnik gry w C++ z makrami – (m,n,k,p,q)GameEngine .....	67
3.4.1 Plik Types.h – definicje, makra i struktury danych .....	68
3.4.2 Klasa Board – reprezentacja planszy .....	69
3.4.3 Klasa Game – obsługa komunikacji, gry i jej stanu .....	73
3.4.4 Klasa AIPlayer – proste AI zwracające poprawne ruchy .....	81
3.5 Biblioteka ułatwiająca współpracę z silnikiem – (m,n,k,p,q)EngineWrapper .....	84
3.5.1 Klasa ProcessInBackground .....	84
3.5.2 Klasa EngineParameters i wyliczenie WinCondition .....	86
3.5.3 Klasa PerformanceInformation i struktura ValueWithUnit.....	90
3.5.4 Wyliczenie Player .....	92
3.5.5 Klasa Move .....	93
3.5.6 Wyliczenie GameState .....	94

3.5.7 Klasa EngineWrapper i wyliczenie WrapperMode, GameType .....	96
3.6 Aplikacja do analizy wydajnościowo-poprawnościowej generowanych silników – (m,n,k,p,q)EnginesAnalyzer .....	104
3.6.1 Klasa EnginesTester – testowanie silników.....	106
3.6.2 Klasa CorrectnessTests – testy poprawności.....	113
3.6.3 Klasa PerformanceTests – testy wydajności .....	115
4. Analiza wydajnościowo-poprawnościowa zaimplementowanego generatora silników gier .	116
4.1 Platforma testowa .....	116
4.2 Testy wydajności generatora .....	117
4.2.1 Wyniki wydajności generatora .....	117
4.3 Testy poprawności generowanych silników .....	118
4.3.1 GetMovesShouldReturnAvalaibleMoves .....	119
4.3.2 AllPossibleMovesGameShouldEndTest .....	119
4.3.3 GameOverTest.....	119
4.3.4 TwoTimesSameMoveShouldNotBePossible .....	120
4.3.5 AfterQMovesItsWhitePlayerTurn .....	120
4.3.6 FirstTurnHumanVsHumanMovesTest .....	121
4.3.7 FirstTurnHumanVsAiMovesTest .....	122
4.3.8 FirstTurnAiVsHumanMovesTest i FirstTurnAiVsAiMovesTest .....	122
4.3.9 HumanVsHumanBlackWinTest.....	123
4.3.10 HumanVsHumanBlackWinIfKOrMoreToWinTest .....	124
4.4 Wyniki poprawności generowanych silników .....	125
4.5 Testy wydajności generowanych silników.....	126
4.6 Wyniki testów wydajności.....	127
4.6.1 Wydajność silników a rozmiar planszy .....	127
4.6.2 Wydajność silników a parametr k.....	139
4.6.3 Wydajność silników a warunek wygranej.....	147
4.6.4 Wydajność silników a parametry p i q.....	152
5. Podsumowanie.....	163
Wykaz literatury.....	165
Wykaz rysunków .....	168
Wykaz tabel.....	171
Dodatek A: Zawartość załączonej płyty CD .....	172

## **WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW**

ICGA – International Computer Games Association

TCEC – Top Chess Engine Championship

WPF – Windows Presentation Foundation

GUI – graficzny interfejs użytkownika

UI – interfejs użytkownika

.NET – Microsoft .NET Framework

n – liczba wierszy planszy

m – liczba kolumn planszy

k – k-pod-rząd postawionych pionów jest potrzebnych, żeby wygrać

p – liczba ruchów (np. stawianych pionów) w turze gracza

q – liczba ruchów (np. stawianych pionów) przez pierwszego gracza w pierwszej turze

UCI – Universal Chess Interface

AI – Artificial Intelligence, sztuczna inteligencja

LERF – Little-Endian Rank-File

Wrapper – funkcje, klasy lub moduły opakowujące inny kod

Content – treść

VS – Visual Studio

QPC – QueryPerformanceCounter

MSBuild – narzędzie firmy Microsoft do procesu budowania aplikacji i projektów

## **1. WSTĘP I CEL PRACY**

W początkowym okresie rozwoju prac badawczych nad algorytmiką i komputerami, znacznie bardziej istotną cechą algorytmów była złożoność pamięciowa [1]. Wynikało to z faktu, że dostępne komputery miały mało pamięci operacyjnej, która w dodatku była niezwykle droga. Duża jej ilość jest natomiast niezbędna dla niektórych silników gier logicznych, ponieważ muszą one np. generować możliwe posunięcia lub trzymać „drzewo gry” w pamięci. Z powodu niewielkiej wydajności pierwszych komputerów, programy dobrze grające w gry logiczne, takie jak warcaby czy szachy, nie pojawiły się aż do późnych lat siedemdziesiątych XX wieku [2][3][4].

Dzisiaj dzięki dużej, taniej i łatwo dostępnej pamięci operacyjnej, wysokiej wydajności procesorów oraz rozwojowi algorytmiki mamy do czynienia z prawdziwym rozwitkiem dziedziny zajmującej się programami grającymi. Intensywność rozwoju wzmagana się dodatkowo pod wpływem nieustannej rywalizacji pomiędzy programami grającymi podczas takich wydarzeń, jak olimpiada ICGA [5][6] czy TCEC [7]. Na samej olimpiadzie ICGA rozgrywane są corocznie mistrzostwa programów grających w około 36 różnych gier logicznych [8], a autorzy zapewniają możliwość dodania kolejnych gier, jeżeli tylko znajdą się chętni [9].

Biorąc pod uwagę wskazane prawidłowości, oczywiste jest, że kolejne generacje silników gier będą coraz bardziej wydajne i efektywniejsze w wykorzystywaniu zasobów sprzętowych. Jednym z czynników umożliwiających ciągłe zwiększenie wydajności jest stosowanie optymalizacji sprzętowych – wykorzystywanie konkretnych funkcji architektury procesora, o ile są dostępne. Często wymaga to umieszczania wstawek asemblerowych w kodzie źródłowym programu grającego.

Trzeba przyznać, że dzięki ludzkiej inteligencji, która umożliwia szybki rozwój wydajności procesorów, pojemności pamięci i efektywności algorytmów, został osiągnięty w 2007 roku spektakularny sukces – udało się rozwiązać wariant angielski Warcabów [10]. Rozwiązanie jest określane jako *słabe*, tzn. algorytm zaimplementowany w programie Chinook nie ma gwarancji wygranej z każdym przeciwnikiem, ale jest zagwarantowane, że nigdy nie przegra. Tak czy inaczej robi to ogromne wrażenie i jest wspólnym osiągnięciem algorytmiki i inżynierii komputerowej – wyznacznikiem postępu ludzkości. Właśnie algorytmika, która korzysta z dobrodziejstw inżynierii komputerowej, można by rzec „algorytmy zoptymalizowane pod sprzęt”, to bardzo istotny element nowoczesnych programów grających, a tym samym ważna część tej pracy dyplomowej.

W niniejszej pracy, z racji złożoności tematyki i trudności implementacji wydajnych silników gier logicznych, cały pierwszy rozdział poświęcono przeglądowi i analizie dostępnych implementacji silników popularnych gier logicznych. Ma to przede wszystkim umożliwić scharakteryzowanie ogólnego trendu w rozwoju silników gier oraz, co ważniejsze, pozwolić autorowi na przeprowadzenie implementacji generatora silników dla wybranej gry logicznej (w rozdziale drugim).

Główym celem pracy jest implementacja i analiza generatora silników dla wybranej gry logicznej. Program powinien umożliwić obsługującemu go człowiekowi wygodne wygenerowanie konkretnego silnika wybranej gry logicznej, parametryzowanego takimi wartościami, jak: rozmiary

planszy, liczba pól aktywnych planszy, zasady gry, liczba graczy. W implementacji głównym wyzwaniem było napisanie takiego generatora, aby generowane silniki były przynajmniej równie dobre (równie szybkie, z równie mocnym AI) co znane silniki dla wybranej gry. Jest to zadanie zdecydowanie trudne, dlatego też cały trzeci rozdział jest przeznaczony na sprawdzenie poprawności i wydajności zaimplementowanego generatora.

Podsumowując, delimitacja pracy (wyodrębnienie trzech rozdziałów) wyznacza w pewnym sensie trzy cele, do których zrealizowania dążył autor. Część pierwsza jest czysto opisowa, zawiera przegląd i analizę dostępnych silników gier logicznych, co prowadzi do poznania trendów implementacyjnych współczesnych programów grających. Część druga jest w pełni implementacyjna, jej celem jest powstanie generatora wydajnych i poprawnych silników wybranej gry logicznej. W części trzeciej, o charakterze analitycznym i praktycznym, dogłębnie przetestowano wydajność zaimplementowanego generatora (głównie poprzez testowanie silników przez niego generowanych).

## 2. PRZEGŁĄD I ANALIZA DOSTĘPNYCH IMPLEMENTACJI SILNIKÓW POPULARNYCH GIER LOGICZNYCH

W niniejszym rozdziale postanowiono bliżej przyjrzeć się wybranym, dostępnym implementacjom silników popularnych gier logicznych. Ze względu na ograniczoną objętość pracy należało dokonać optymalnej selekcji, tj. wybrać nowoczesne i aktywnie rozwijane silniki (aby uwzględnić trend ich rozwoju), ale także przeznaczone do popularnych gier (większa rywalizacja, więc wyższa jakość implementacji). Autor jako miłośnik czystego kodu, w procesie wyboru silników do analizy, dużą wagę przywiązywał również do tego, czy udostępniony kod źródłowy programu grającego jest dobrze ustrukturalizowany i przejrzyście napisany.

Wstępnie zadecydowano również, że najodpowiedniejsze do analizy będą programy najlepsze w swojej dziedzinie, ponieważ interesujące są rozwiązania bliskie optymalnym – można zadać pytanie: „Co takiego znajduje się w kodzie programu grającego, że dało mu przewagę nad innymi?”. W tym celu przeanalizowano wyniki z olimpiad ICGA i TCEC.

**Tabela 2.1.** Wyniki turnieju "Top Chess Engine Championship" [5]

Sezon	Data	Zwycięzca	Drugie miejsce
1	Dec 2010 – Feb 2011	Houdini 1.5a	Rybka 4.0
2	Feb 2011 – Apr 2011	Houdini 1.5a	Rybka 4.1
3	Apr 2011 – May 2011	<i>brak (sezon niedokończony)</i>	
4	Jan 2013 – May 2013	Houdini 3	Stockfish 250413
5	Aug 2013 – Dec 2013	Komodo 1142	Stockfish 191113
6	Feb 2014 – May 2014	Stockfish 170514	Komodo 7x
6 (szachy losowe)	June 2014 – July 2014	Stockfish 260614	Houdini 4
7	Sep 2014 – Dec 2014	Komodo 1333	Stockfish 141214
8	Aug 2015 – Nov 2015	Komodo 9.3x	Stockfish 021115
9	May 2016 – Dec 2016		

W tabeli 2.1. zaprezentowano wyniki z wszystkich sezonów turnieju szachowego TCEC. Na ich podstawie można stwierdzić, że prawdopodobnie dwoma najlepszymi programami grającymi w szachy są aktualnie Stockfish oraz Komodo. Warto byłoby przeanalizować w niniejszym rozdziale oba, niestety Komodo, który jest prawdopodobnie najlepszym silnikiem szachowym na świecie [25] [26], jako komercyjny ma zamknięty kod źródłowy. Można jedynie kupić obecną wersję wykonywalną [27] lub poprzednią w wyprzedaży [28], dwie wersje wstecz są natomiast dostępne na licencji freeware [29]. W każdym razie, ponieważ kod źródłowy nie jest dostępny, analiza nie może zostać przeprowadzona. Szczęśliwie jednak miejsce drugie z TCEC z 2015 roku, czyli Stockfish, jest otwartoźródłowy i dostępny na publicznym repozytorium w serwisie github [30] [31]. W dalszej części tego rozdziału będzie on jednym z kluczowych silników oddanych analizie.

**Tabela 2.2.** Wyniki w grze Go 9x9 [11]

Miejsce	Nazwa programu	Punkty
1	<b>Zen</b>	9.5
2	<b>Abakus</b>	7.5
3	<b>CGI</b>	7.0
4	Nomitan	4.0
5	MC_ark	2.0
6	Wingo	0.0

Program Zen w 2015 roku wygrał we wszystkich rozgrywanych na Olimpiadzie Gier Komputerowych ICGA wersjach gry Go [11], nie tylko w wersji 9x9, której wyniki przedstawiono w tabeli 2.2., ale także w wersjach 13x13 i 19x19. Byłby więc doskonałym materiałem do analizy, ale ponieważ program został wydany komercyjnie, nie może być w zastosowany w niniejszej pracy [12]. Autor – japoński programista Yoji Ojima – po raz pierwszy opublikował go jako komercyjny program w roku 2009 i kolejne jego wersje w latach następnych. Najnowsza wersja 6. pochodzi z 2016 roku i można ją zakupić na stronie [13], nie ma więc mowy o jakiejkolwiek analizie kodu źródłowego.

**Tabela 2.3.** Wyniki w grze Hex 11x11 [11]

Miejsce	Nazwa	Punkty
1	Mo-Hex	7
2	Deep-Hex	5
3	Ezo	0

**Tabela 2.4.** Wyniki w grze Hex 13x13 [11]

Miejsce	Nazwa	Punkty
1	Mo-Hex	6
2	Deep-Hex	6
3	Ezo	0

Innym dominującym silnikiem na wspomnianej wcześniej olimpiadzie okazał się program MoHex, który wygrał w obu rozgrywanych odmianach gry Hex (11x11 i 13x13) [11], jak pokazano w tabelach 2.3 i 2.4. W tym przypadku okazało się, że przynajmniej jakaś (zapewne starsza) wersja kodu źródłowego jest dostępna [14][15]. Dodatkowym atutem jest fakt, że w mistrzowskim programie Mo-Hex został wykorzystany algorytm Polaka, Jakuba Pawlewicza – „new virtual connection implementation”. Po raz pierwszy ten algorytm został zaprezentowany w programie Polaka MIMHex, którego kod źródłowy jest dostępny w kilku wersjach, rozwijanych przez różnych autorów [16] [17] [18] [19] [20] [21] [22] [23] [24], co jeszcze bardziej ułatwia analizę.

Ostatnią już grą, wybraną do przeglądu i analizy zaimplementowanych i dostępnych silników, jest Connect6. Wpływła na to fascynacja autora zagadnieniem gier (n,m,k,p,q), a

Connect6 jest grą ( $\infty, \infty, 6, 2, 1$ ) [34] [35]. Istnieją też inne przyczyny, niezwiązane bezpośrednio z jakością dostępnych silników tej gry, które zostaną ujawnione w dalszej części pracy.

**Tabela 2.5.** Wyniki w grze Connect6 [11]

Miejsce	Nazwa	Punkty
2	Floating Cloud	6
1	<b>Explorer</b>	6
3	<b>USTB1</b>	0

Niestety, pomimo dogłębnego przeszukania Internetu, żaden z programów grających w Connect6 i biorących udział w Olimpiadzie Gier Komputerowych ICGA w 2015 roku nie został odnaleziony, ani w formie plików wykonywalnych, ani tym bardziej w formie kodu źródłowego. Po długich poszukiwaniach programów grających w Connect6 udało się znaleźć jeden dobrze napisany, nazwany „Connect-k”. Jego autor twierdzi, że jedynym innym dostępnym w Internecie programem grającym w Connect6 jest NCTU6 autorstwa dr-a I-Chen Wu (pomysłodawcy Connect6) [37]. NCTU6 zazwyczaj wygrywa z Connect-k, ale ponieważ ma zamknięty kod i jego interfejs jest w języku chińskim [37] [38], nie został poddany analizie. Autor niniejszej pracy znalazł też inny silnik – Connect6, zaimplementowany przez bliżej nieznaną osobę [39]. Może posłużyć on do ewentualnego porównania z Connect-k, który zostanie w dalszej części rozdziału szczegółowo zanalizowany.

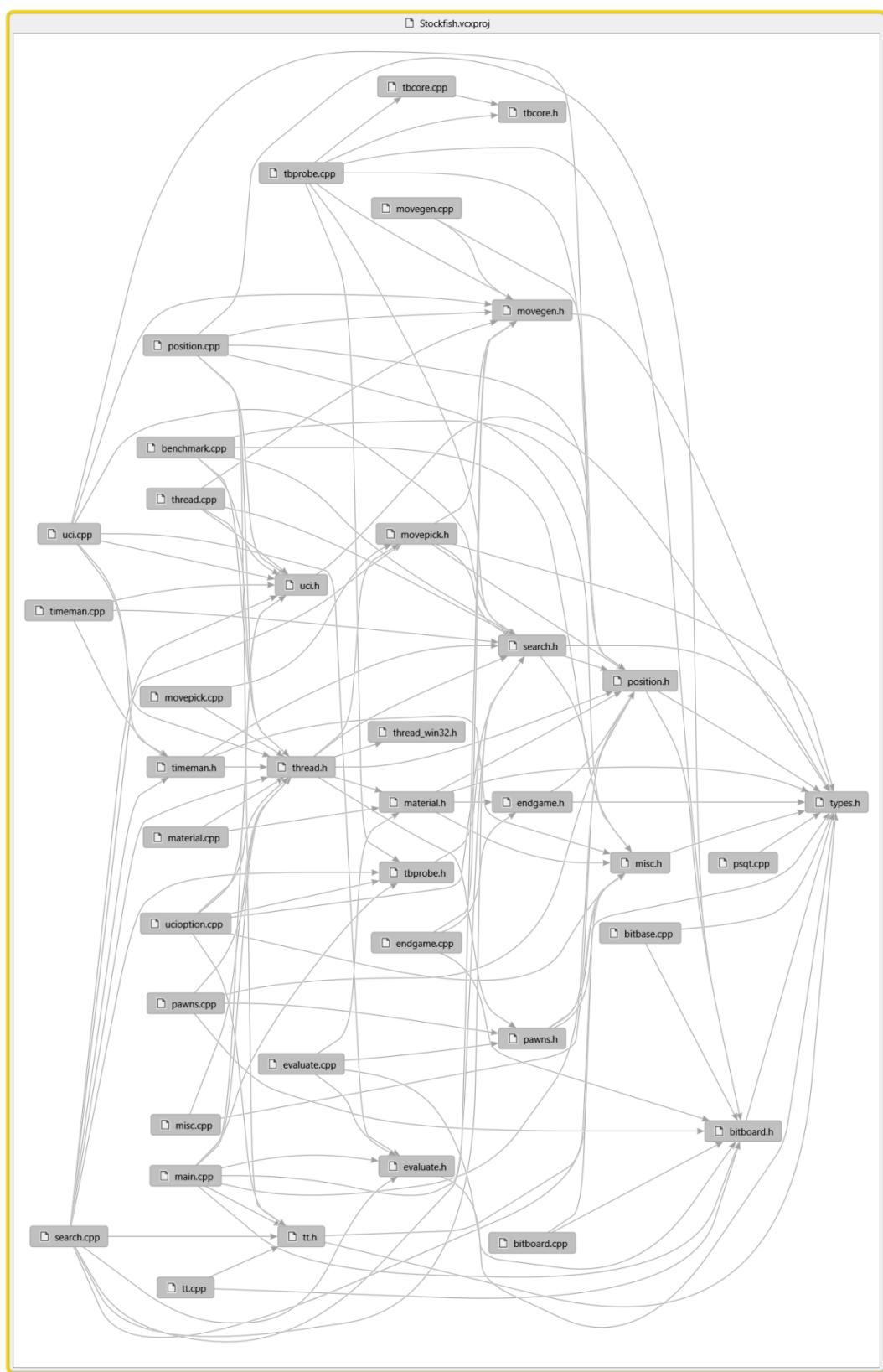
## 2.1 Stockfish

Dostępny na wszystkie popularne platformy sprzętowe Windows, Linux, Mac OS X, Android, iOS, darmowy i open source, udostępniony na licencji GNU GPL v3., rozwijany przez Marco Costalba, Joona Kiiski, Gary'ego Linscotta, Torda Romstada i internetową społeczność od 2004 roku i od ponad siedmiu lat pod własną nazwą (wcześniej Glaurung) [46], składający się z ponad 12 000 linii kodu, zajmujący regularnie pierwsze lub drugie miejsce na świecie w rankingu siły silników szachowych CCLR [47] [48] [49] jest Stockfish prawdopodobnie najlepszym silnikiem do analizy ze względu na cele pracy. Aby pokazać jego złożoność oraz zaawansowanie technologiczne i algorytmiczne, zebrano poniżej niektóre jego cechy i wykorzystywane przez niego techniki:

- wykorzystanie kilku reprezentacji szachownicy i przełączanie między nimi
  - bitboard z mapowaniem LERF
  - „magiczny” bitboard z podejściem „fancy”
  - bitboard BMI2 – PEXT
  - listy figur
- sam silnik (bez kodu UI) – korzysta z protokołu UCI
- support dla szachów losowych (Chess960)
- wykorzystanie tablic końca gry (Sygzy endgame tablabase)
- wykorzystanie dobrodziesztw nowoczesnych architektur procesorów
- wielowątkowość z wykorzystaniem do 128 rdzeni procesora

- zaawansowane funkcje ewaluacji figur i pozycji z uwzględnieniem:
  - fazy gry (tapered eval)
  - mobilności
  - struktury pionów
  - bezpieczeństwa króla
  - parowania gońców
  - tablic figura-pozycja
  - własnych, zdefiniowanych na starcie parametrów
- głębokie przeszukiwanie drzewa gry [56] [55] z uwzględnieniem:
  - iteratywnego pogłębiania (iterative deepening)
  - tablic transpozycji (transposition table)
  - haszowania Zobrista
  - przeszukiwania równoległego z użyciem wątków
  - heurystyki ruchów przeciwdziałających
  - heurystyki MVV-LVA
  - agresywnego odcinania [56]
    - odcinanie Null Move
  - wyszukiwania cichego (Quiescence search)
  - redukcji ruchów w późnej grze

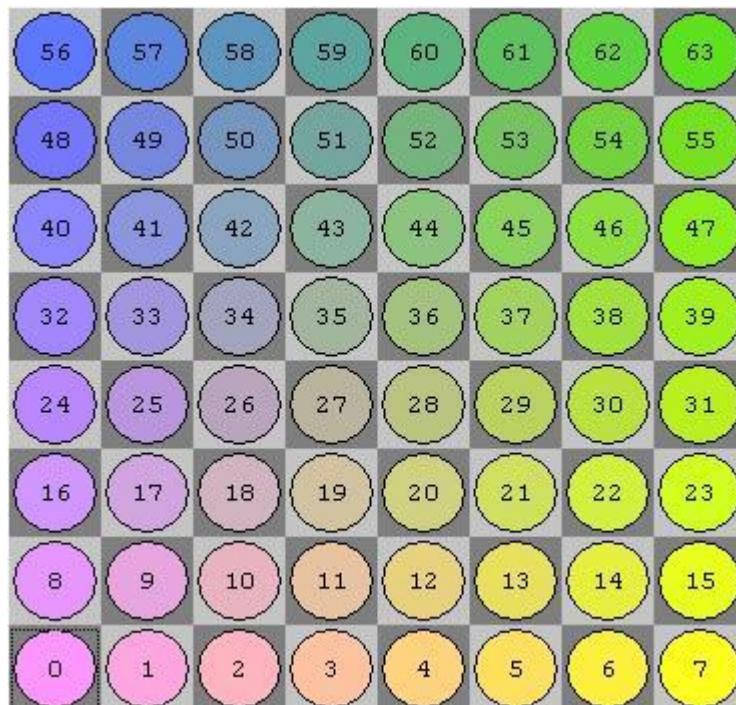
To zaledwie część cech i możliwości programu grającego oraz technik, które często są bardzo złożone i trudne pojęciowo. Jednakże z punktu widzenia celu tej pracy, większość z wymienionych technik nie ma aż tak wielkiego znaczenia, gdyż dotyczą sztucznej inteligencji programu grającego. O wiele bardziej interesujące są rozwiązania niskopoziomowe, które umożliwiły temu silnikowi wysoką wydajność generowania ruchów za pomocą algorytmów i technik AI. Dlatego też w dalszej części podrozdziału więcej uwagi poświęcone będzie reprezentacji planszy, stanów gry i innych danych dotyczących rozgrywki, a mniej wyszukanym technikom AI.



Rysunek 2.1. Graf zależności w programie Stockfish

### 2.1.1 Reprezentacja planszy – bitboard

Stockfish używa wewnętrznie prawdopodobnie najefektywniejszej możliwej reprezentacji planszy, o bardzo dużym stopniu gęstości informacji. Tą reprezentacją jest oczywiście bitboard. Tak się składa, że plansza w szachach ma 64 pola, tak samo jak rejestr w procesorach 64-bitowych. Przechowywanie więc planszy w 64-bitowej zmiennej na 64-bitowym procesorze oznacza, poza korzyściami pamięciowymi, możliwość wykonania niektórych operacji w jednym cyklu zegara – czyli najszybciej jak się da [50]. Sposób mapowania planszy na 64-bitową zmenną używany przez Stockfish, to mapowanie Little Endian Rank-File, czyli mniej znaczący bit odpowiada wcześniejszemu polu i poruszanie się po szachownicy poziomo zmienia wartość o jeden, a poruszanie się pionowo o 8 [52]. Całość zaprezentowano na poniższym rysunku.



Rysunek 2.2. Mapowanie szachownicy na bitboard Little-Endian Rank-File [53]

Oczywiście, jeżeli bit odpowiadający danej pozycji przyjmuje wartość 1, to oznacza, że pole jest zajęte, jeżeli 0, to jest wolne. Należy jednak zauważyć, że pojedynczy bitboard nie wystarczy do reprezentacji całej planszy – byłoby tak jedynie wtedy, gdybyśmy mieli tylko jeden rodzaj figur szachowych i grali bez rozróżnienia na figury białe i czarne. W rzeczywistości Stockfish trzyma po prostu dużą liczbę bitboardów w różnych celach – np. po dwa bitboardy (gracz czarny, biały) dla każdego rodzaju figury szachowej (pion, goniec, wieża, król, hetman, skoczek).

Typ Bitboard zdefiniowany w pliku types.h zgodnie z intuicją jest po prostu typedef'em na 64-bitową zmienną całkowitą.

```
typedef uint64_t Bitboard;
```

Przykładem użycia bitboardu są choćby maski, położenia i przesunięcia dla figur zdefiniowane w pliku bitboard.cpp.

```

Bitboard RookMasks [SQUARE_NB];
Bitboard RookMagics [SQUARE_NB];
Bitboard* RookAttacks[SQUARE_NB];
unsigned RookShifts [SQUARE_NB];

Bitboard BishopMasks [SQUARE_NB];
Bitboard BishopMagics [SQUARE_NB];
Bitboard* BishopAttacks[SQUARE_NB];
unsigned BishopShifts [SQUARE_NB];

Bitboard SquareBB[SQUARE_NB];
Bitboard FileBB[FILE_NB];
Bitboard RankBB[RANK_NB];
Bitboard AdjacentFilesBB[FILE_NB];
Bitboard InFrontBB[COLOR_NB][RANK_NB];
Bitboard StepAttacksBB[PIECE_NB][SQUARE_NB];
Bitboard BetweenBB[SQUARE_NB][SQUARE_NB];
Bitboard LineBB[SQUARE_NB][SQUARE_NB];
Bitboard DistanceRingBB[SQUARE_NB][8];
Bitboard ForwardBB[COLOR_NB][SQUARE_NB];
Bitboard PassedPawnMask[COLOR_NB][SQUARE_NB];
Bitboard PawnAttackSpan[COLOR_NB][SQUARE_NB];
Bitboard PseudoAttacks[PIECE_TYPE_NB][SQUARE_NB];

```

Jak widać, jest ich naprawdę sporo, w dodatku to zaledwie część bitboardów wykorzystywanych w projekcie. Przechowywane są tutaj np. wszystkie możliwe ataki wieży z każdej z 64 możliwych pozycji.

### 2.1.2 Wykorzystanie zaawansowanych instrukcji assemblerowych procesora

W celach wydajnościowych niektóre silniki gier logicznych wykorzystują wstawki kodu assemblerowego. Jednocześnie nowoczesne architektury procesorów rozwijają się i udostępniają coraz to nowe funkcje, zwłaszcza te dotyczące manipulacji na bitach. Szczęśliwie nowoczesne kompilatory C/C++, takie jak Microsoft Visual C++, Intel C/C++ Compiler i GCC, udostępniają funkcje „intrinsic”, które bezpośrednio mapują do funkcji assemblerowych, np. rozszerzeń SIMD instrukcji procesora (MMX, SSE, SSE2, SSE3, SSE4) [59]. Funkcje „intrinsic” są oczywiście tak implementowane przez współczesne kompilatory z przyczyn wydajnościowych. Opisywane w tym podrozdziale funkcje są właśnie funkcjami tego typu.

Wspomniane wcześniej użycie bitboardu ma jeszcze jedną zaletę, często np. w funkcjach ewaluacji (mających ocenić wartość sytuacji szachowej – choćby posiadanie dwóch gońców jest warte więcej niż posiadanie jednego) potrzeba szybko zliczyć ilość danego typu figur. Autorzy programu pokusili się tutaj o wykorzystanie dobrodziejstw nowoczesnej architektury procesorów – nowej instrukcji POPCNT wprowadzonej przez Intel w architekturze Nehalem i AMD w architekturze Barcelona [51]. Jeżeli tylko procesor, na którym jest wykonywany program, wspiera tę instrukcję, to przy każdym zliczaniu w funkcjach ewaluujących wystąpienie konkretnego typu figury zostanie na bitboardzie wywołana instrukcja procesora POPCNT, która zliczy liczbę bitów ustawionych na 1 w danej zmiennej, czyli liczby wystąpień figury w bitboardzie. Oczywiście w innym wypadku program korzysta z wolniejszej implementacji programowej.

W pliku types.h załączanym prawie przez wszystkie pliki w projekcie znajdują się definicje załączające pliki nagłówkowe do obsługi instrukcji procesora POPCNT.

```
#if defined(USE_POPCNT) && (defined(__INTEL_COMPILER) || defined(_MSC_VER))
# include <nmmmintrin.h> // Intel and Microsoft header for _mm_popcnt_u64()
#endif
```

Funkcja popcorn w pliku bitboard.h jest inline'owana w celach wydajnościowych. Jak widać poniżej, jeżeli nie jest zdefiniowana stała USE\_POPCNT (które można zdefiniować jako flagę komplikacji), to wykorzystywana jest implementacja programowa. W innym wypadku zależnie od używanego kompilatora wywoływana jest funkcja wywołującą bezpośrednio instrukcję na procesorze.

```
inline int popcorn(Bitboard b) {

#ifndef USE_POPCNT

    extern uint8_t PopCnt16[1 << 16];
    union { Bitboard bb; uint16_t u[4]; } v = { b };
    return PopCnt16[v.u[0]] + PopCnt16[v.u[1]] + PopCnt16[v.u[2]] +
PopCnt16[v.u[3]];

#elif defined(_MSC_VER) || defined(__INTEL_COMPILER)

    return (int)_mm_popcnt_u64(b);

#else // Assumed gcc or compatible compiler

    return __builtin_popcountll(b);

#endif
}
```

Poza ewaluacją jest ona używana również w obliczaniu haszy na potrzeby przeszukiwania tablic końca gry, w tekstowym podsumowaniu stanu gry, w „magicznym bitboardzie”, w ewaluacji struktury pionów i w innych miejscach, związanych przede wszystkim z ewaluacją. Autor znalazł 19 wywołań funkcji popcorn w projekcie, każde o dużym znaczeniu dla programu grającego. Nie należy więc marginalizować znaczenia szybkiego wykonania tej funkcji dzięki instrukcjom asemblerowym procesora.

Inną funkcją zaimplementowaną podobnie jak popcorn jest w Stockfishu funkcja pext. Zdefiniowana w rozszerzeniu BMI2 zbioru instrukcji dla architektury procesorów x86 umożliwia zrównoległą ekstrakcję bitów [57]. Funkcja przydaje się w reprezentacji „magiczny bitboard”, gdzie po ustawnieniu mask można za jej pomocą łatwo i bardzo szybko sprawdzić wartość konkretnych bitów. Używana jest w pliku bitboard.cpp i bitboard.h. Jej definicja w types.h jest podobna do definicji POPCNT, tyle tylko że nie używa się implementacji programowej, jedyną zysk z tej funkcji pojawia się, jeżeli przy zdefiniowanej flagie do kompilatora USE\_PEXT korzysta się z funkcji wywołującej bezpośrednio instrukcję procesora.

```
#if defined(USE_PEXT)
# include <iimmmintrin.h> // Header for _pext_u64() intrinsic
# define pext(b, m) _pext_u64(b, m)
#else
# define pext(b, m) (0)
```

```
#endif
```

Ostatnią już funkcją, której definicję można manipulować flagami kompilatora, jest prefetch. W porównaniu jednak do poprzednich, których użycie wymagało zadeklarowania pewnej stałej, tutaj należy zadeklarować stałą NO\_PREFETCH, jeżeli funkcja prefetch nie ma wywoływać funkcji asemblerowej \_mm\_prefetch().

```
#if !defined(NO_PREFETCH) && (defined(__INTEL_COMPILER) || defined(_MSC_VER))
# include <xmmmintrin.h> // Intel and Microsoft header for _mm_prefetch()
#endif
```

Jak widać poniżej, jeżeli stała powodująca dla kompilatora Intela lub Microsoftu niezałączanie pliku nagłówkowego xmmmintrin.h jest zadeklarowana, to wywołanie funkcji prefetch nie wywołuje żadnych efektów.

```
#ifdef NO_PREFETCH

void prefetch(void*) {}

#else

void prefetch(void* addr) {

# if defined(__INTEL_COMPILER)
    // This hack prevents prefetches from being optimized away by
    // Intel compiler. Both MSVC and gcc seem not be affected by this.
    __asm__ ("");
# endif

# if defined(__INTEL_COMPILER) || defined(_MSC_VER)
    _mm_prefetch((char*)addr, _MM_HINT_T0);
# else
    __builtin_prefetch(addr);
# endif
}

#endif
```

Funkcja sama w sobie jest bardziej niskopoziomowa niż poprzednie z omawianych. Użycie jej to swego rodzaju optymalizacja, funkcja preloaduje dany adres w pamięci cache L1/L2 procesora. W dodatku wywołanie jest nieblokujące CPU czekaniem na dane do załadowania z pamięci, co mogłoby być dosyć wolne. Wykorzystywana jest m.in. w haszowaniu tablic transpozycji, w przeszukiwaniu, jak i haszowaniu tablic pozycji w pliku position.cpp.

```
// Update pawn hash key and prefetch access to pawnsTable
st->pawnKey ^= Zobrist::psq[us][PAWN][from] ^ Zobrist::psq[us][PAWN][to];
prefetch(thisThread->pawnsTable[st->pawnKey]);
```

Dwoma kolejnymi funkcjami wywołującymi, zależnie od kompilatora i platformy, bezpośrednio instrukcję procesora, są funkcje lsb i msb. Służą one odpowiednio do zwrócenia najmniej i najbardziej znaczącego bitu w niezerowym bitboardzie.

```

#if defined(__GNUC__)

inline Square lsb(Bitboard b) {
    assert(b);
    return Square(__builtin_ctzll(b));
}

inline Square msb(Bitboard b) {
    assert(b);
    return Square(63 - __builtin_clzll(b));
}

#elif defined(_WIN64) && defined(_MSC_VER)

inline Square lsb(Bitboard b) {
    assert(b);
    unsigned long idx;
    _BitScanForward64(&idx, b);
    return (Square) idx;
}

inline Square msb(Bitboard b) {
    assert(b);
    unsigned long idx;
    _BitScanReverse64(&idx, b);
    return (Square) idx;
}

#else

#define NO_BSF // Fallback on software implementation for other cases

Square lsb(Bitboard b);
Square msb(Bitboard b);

#endif

```

Widać więc, że podobnie jak `popcount` nie są to tylko funkcje optymalizujące, ale realnie potrzebne do obliczeń – więc w przypadku, gdy na komplikowanej platformie nie jest dostępna konkretna instrukcja procesora, korzysta się z implementacji programowej, która tutaj nie zostanie już omówiona z powodu jej trywialności. Instrukcja `clzll` procesora jest dostępna jednak na większości architektur sprzętowych [58].

### 2.1.3 Reprezentacja ruchu i wartości dla funkcji ewaluacji

Ruch jest reprezentowany w podobny sposób jak plansza. Wystarczy tutaj typ `short`, gdyż potrzebne jest 16-bitów, aby go przechować. Kolejne bity są przeznaczone odpowiednio na:

- bity 0-5: pozycja docelowa na szachownicy (wartości od 0 do 63),
- bity 6-11: pozycja źródłowa na szachownicy (wartości od 0 do 63),
- bity 12-13: typ figury podlegającej promocji,
- bity 14-15: flaga specjalnego oznaczenia ruchu (wartości: 0 – nic, 1 – promocja, 2 – bicie w przelocie, 3 – roszada).

W projekcie użyto w tym celu specjalnych enum'ów zaprezentowanych poniżej.

```
enum Move {
```

```

MOVE_NONE,
MOVE_NULL = 65
};

enum MoveType {
NORMAL,
PROMOTION = 1 << 14,
ENPASSANT = 2 << 14,
CASTLING = 3 << 14
};

```

Ciekawostką jest fakt wykorzystania dodatkowego miejsca w enumie Move na przemycenie dwóch innych rodzajów ruchu. Brak ruchu i ruch-null, o wartościach odpowiednio 0 i 65, zostały tam celowo umieszczone bez zwiększenia rozmiarów z 16-bitów dzięki faktowi, że każdy ruch będzie miał inną pozycję docelową niż źródłową (w szachach nie można dokonać ruchu w miejscu), więc wszystkie wartości Move z identycznymi bitami 0-5 i 5-11 nigdy w programie nie wystąpią. Jest to dosyć bezpieczna optymalizacja i na pewno całkiem wygodna, wygodniejsza niż dostawianie jakichś masek czy dodatkowych struktur.

Reprezentacja wyniku czy wartości dla funkcji ewaluującej jest też bardzo efektywna. Najpierw należy przyjrzeć się wagom przydzielonym konkretnym figurom.

```

enum Value : int {
VALUE_ZERO      = 0,
VALUE_DRAW      = 0,
VALUE_KNOWN_WIN = 10000,
VALUE_MATE       = 32000,
VALUE_INFINITE   = 32001,
VALUE_NONE       = 32002,

VALUE_MATE_IN_MAX_PLY = VALUE_MATE - 2 * MAX_PLY,
VALUE_MATED_IN_MAX_PLY = -VALUE_MATE + 2 * MAX_PLY,

PawnValueMg     = 198,    PawnValueEg     = 258,
KnightValueMg   = 817,    KnightValueEg   = 896,
BishopValueMg   = 836,    BishopValueEg  = 907,
RookValueMg     = 1270,   RookValueEg    = 1356,
QueenValueMg    = 2521,   QueenValueEg   = 2658,

MidgameLimit    = 15581,  EndgameLimit   = 3998
};

```

Zgodnie z tym, co wcześniej zostało wspomniane, Stockfish uwzględnia w ewaluacji fazę gry. Widać to w powyższym kodzie, w którym wartości dla figur z przyrostkiem Mg (mid-game) są dla środkowej fazy gry, a te z Eg (end-game) dla końcowej.

32-bitowy typ Score przechowuje wyniki w taki sposób, że najmniej znaczące 16-bitów to wyniki dla końcowej fazy gry, natomiast najbardziej znaczące 16-bitów to wartości dla środkowej fazy gry. Jest to kolejna optymalizacja, która co prawda wymaga wołania specjalnych funkcji (są one jednak inline'owane), ale jednak oszczędza pamięć w symulowanym drzewie gry.

```

enum Score : int { SCORE_ZERO };

inline Score make_score(int mg, int eg) {
return Score((mg << 16) + eg);

```

```

}

inline Value mg_value(Score s) {

    union { uint16_t u; int16_t s; } mg = { uint16_t(unsigned(s + 0x8000) >> 16) };
    return Value(mg.s);
}

inline Value eg_value(Score s) {

    union { uint16_t u; int16_t s; } eg = { uint16_t(unsigned(s)) };
    return Value(eg.s);
}

```

#### 2.1.4 Fishtest

Opisując sukces programu grającego Stockfish jako sukces nauki, otwartego oprogramowania i intelektualnej współpracy w ramach internetowej społeczności, nie można pominąć rozproszonego środowiska testów dla kolejnych wersji Stockfisha nazwanego Fishtest. Należy zauważać, że większość rozwiązań opisywanych przy okazji analizy tego programu grającego występuje też w innych tego rodzaju programach. Niekoniecznie więc efektywne reprezentacje planszy czy zaawansowane algorytmy AI zapewniły Stockfishowi sukces. Czymś, co wyróżnia go na tle innych silników (poza otwartością i dużą współpracą społeczności), jest właśnie Fishtest. Wystarczy tylko wspomnieć, że od powstania Fishtesta w zaledwie dwanaście miesięcy Stockfish zyskał ponad 120 punktów Elo na wszystkich głównych listach ratingowych silników szachowych [60] [61].

Idea działania jest stosunkowo prosta. Jako że repozytorium z kodem Stockfisa jest publicznie dostępne, zgodnie z modelem fork&pull, każdy użytkownik Internetu może zgłosić swoje zmiany (nie trzeba być zaakceptowanym członkiem zespołu). Commit z nowymi zmianami, jeżeli okaże się interesujący, może zostać włączony do testowego brancha. Ta nowa wersja zostaje wtedy wysłana do Fishtesta, gdzie gra dziesiątki tysięcy gier na komputerach ochotników należących do programu przeciwko starej wersji. Wyniki testów są weryfikowane testem chi-kwadrat i jeżeli tylko wyjściowa p-wartość nie jest statystycznie nieznacząca, to wynik testu jest uznawany za poprawny [46]. W uproszczeniu, jeżeli nowy commit jest zauważalnie silniejszy w testach od starszej wersji, to trafia do głównego brancha. W ten sposób można weryfikować każdą nową zmianę, oczywiście jest to dosyć kosztowne obliczeniowo. W sumie ochotnicy podarowali projektowi ponad 360 lat obliczeń, odgrywając na swoich komputerach ponad 240 milionów rozgrywek szachowych.

Sam Tord Romstad (jeden z autorów programu) przyznał, że dalsze rozwijanie siły Stockfisa to w dużej mierze modyfikowanie wartości różnych parametrów (np. wartości figur-położen-faz gry dla funkcji ewaluacji) i patrzenie, czy wyniki rozgrywek się poprawiły [54]. Przy takiej strategii dalszego rozwoju kodu grającego niezbędny jest dobry framework testowy, aby dokładnie sprawdzić każdą zmianę, coś, co Fishtest z pewnością umożliwia.

Poniżej zaprezentowano wycinek z głównej strony Fishtesta, gdzie widać 3 testowane nowe wersje. Trzecia wydaje się wnosić statystycznie znaczącą poprawę siły algorytmu

grającego o 35.12 punktów Elo. W tabeli przedstawiono m.in. wyniki testów, czas testów, datę ich uruchomienia i branch, z którego pochodzi testowany kod. W ostatniej kolumnie jest krótkie podsumowanie zmian w danej wersji testowej.

### Pending - 3 tests 22.3 hrs Hide

		11- 06-16	El lmr_cut4	diff	LLR: -0.90 (-2.94,2.94) [-3.00,1.00] Total: 34431 W: 4765 L: 4858 D: 24808	th 1	sprt @ 60+0.6	LTC: Simplification attempt for cut nodes
		10- 06-16	aj insert_pv	diff	LLR: 0.51 (-2.94,2.94) [0.00,5.00] Total: 6408 W: 932 L: 889 D: 4587	7	sprt @ 5+0.5 th	Insert only the best PV into TT. STC
		11- 06-16	aj master	diff	ELO: 35.12 +-8.9 (95%) LOS: 100.0% Total: 1211 W: 191 L: 69 D: 951	40000 @ 20+0.2 th 7	SMP Regression test 1. First SMP Regression test since SF-7 release 2. Framework is mostly empty	

Rysunek 2.3. Uruchomione obecnie testy na Fishteście [62]

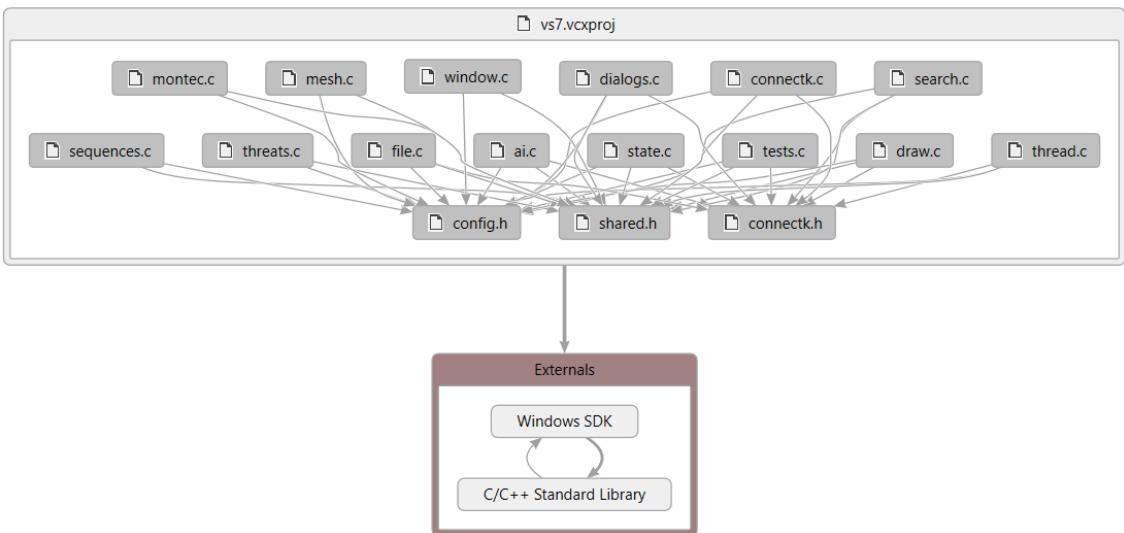
Dodatkowo można też porównać zmiany wprowadzane przez testowany kod w stosunku do głównej wersji, klikając „diff”. Przykładowo, dla pierwszego wiersza testowane zmiany są bardzo niewielkie.

```
2 src/search.cpp
@@ -1013,7 +1013,7 @@ namespace {
    + (fmh2 ? (*fmh2)[moved_piece][to_sq(move)] : VALUE_ZERO);
1014
1015     // Increase reduction for cut nodes
1016 - if (!PNode && cutNode)
1017 + if (cutNode)
1018     r += 2 * ONE_PLY;
1019
// Decrease reduction for moves that escape a capture. Filter out
```

Rysunek 2.4. Porównanie zmian wprowadzanych przez testowaną wersję [62]

## 2.2 Connect-k

Program Connect-k powstał w pierwotnej wersji na uniwersytecie w Minnesocie jako projekt semestralny z przedmiotu Sztuczna Inteligencja w 2007 roku [37]. Grupa projektowa składała się z czterech osób, ale tylko jedna postanowiła go dalej rozwijać. Jeff Deitch opracował AI oparte o metodę Monte Carlo i sekwencje, Gabe Emerson i Erik Shimshock pracowali nad interfejsem i częścią silnika gry, natomiast Michael Levin opracował silnik graficzny i strategię AI opartą o zagrożenie (Threat-Based Strategy). Ten ostatni pracuje nad programem do dziś.



Rysunek 2.5. Graf zależności w programie Connect-k

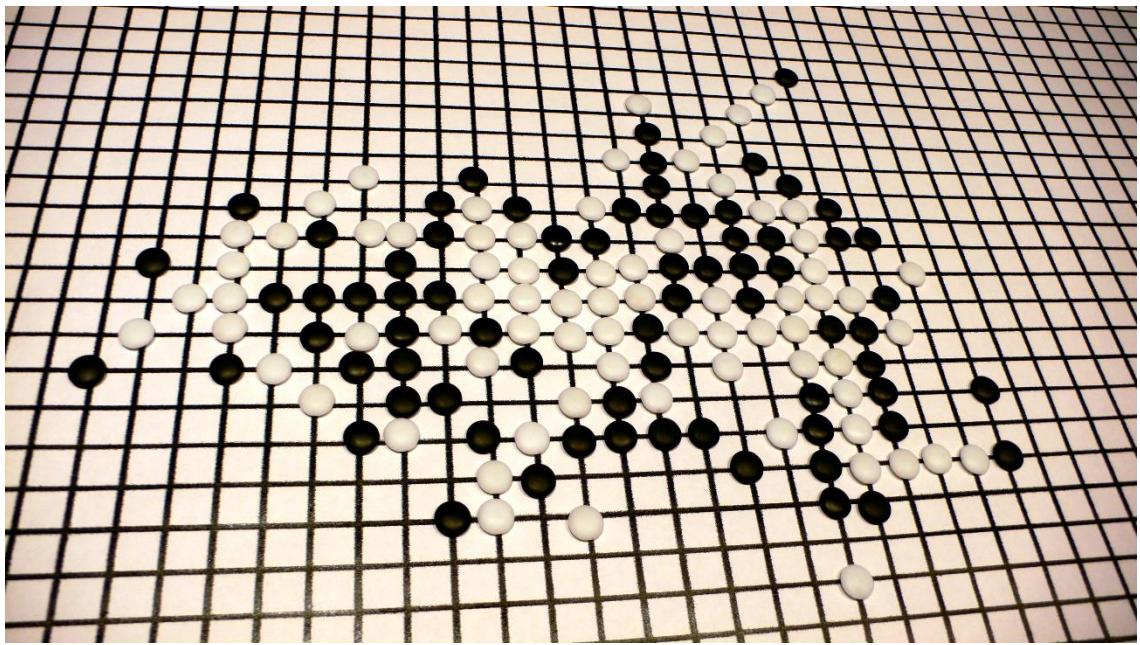
Niestety, część odpowiedzialna za implementację silnika gry nie jest w programie zbyt dobrze wyodrębniona, mimo że analizowana jest najnowsza (2.0) wersja programu [40]. Sytuacji nie poprawia fakt, że projekt był rozwijany w czystym języku C na Linux Debianie z bibliotekami GTK+. Kod odpowiedzialny za grafikę i interfejs jest mieszany z kodem odpowiedzialnym za logikę, natomiast AI jest dosyć dobrze wyodrębnione i podzielone na 3 wspomniane wcześniej zaimplementowane w programie metody.

### 2.2.1 Zasady gry

Aby dobrze opisać zasady gry w Connect-6, czy uogólnione Connect-k należy najpierw opisać zasady ogólne dla gier z rodziny m,n,k,p,q. Wśród reguł znajdują się takie, które są niezależne od wartości jakiegokolwiek parametru jak i opisywane konkretnym parametrem. Należy zacząć opis od tych pierwszych.

Podstawowe reguły gier m,n,k,p,q – niezależne od parametrów:

- liczba graczy: 2 (czarny i biały),
- kolejność graczy: grę zawsze zaczyna gracz czarny,
- każdy z graczy w swojej turze ustawia odpowiednią liczbę kamieni swojego koloru na planszy,
- tury graczy występują naprzemiennie,
- kamienie raz ustawione nie zmieniają położenia do końca rozgrywki,
- plansza jest skończonym prostokątem lub nieskończona,
- kamienie można kłaść tylko na pustych miejscach.



Rysunek 2.6. Przykładowa rozgrywka w Connect-6

Pozostałe reguły (lub ich uściślenie) zależą już od konkretnych parametrów:

- plansza ma rozmiar  $m \cdot n$ , gdzie  $m$  i  $n$  są liczbami naturalnymi, ale mogą też być nieskończone
- aby wygrać, gracz musi ustawić  $k$  kamieni swojego koloru „pod-rząd” (poziomo, pionowo lub po przekątnej)
- zależnie od konkretnej gry, więcej niż  $k$  kamieni pod rząd może również dawać wygraną (np. Gomoku Free-style, Connect-6) lub może być wymagane dokładnie  $k$  kamieni pod rząd (np. Gomoku tradycyjne)
- gracz czarny w pierwszej turze wykonuje  $q$  ruchów
- do końca gry każdy gracz wykonuje  $p$  ruchów w swojej turze

Konkretne definiują dokładnie wartości każdego z parametrów:

- Connect-6:  $m=\infty$ ,  $n=\infty$ ,  $k=6$ ,  $p=2$ ,  $q=1$  (przy tych parametrach przewaga gracza czarnego z tradycyjnego  $p=q=1$  wydaje się być wyeliminowana [35][36])
- Connect- $k$ :  $m=\infty$ ,  $n=\infty$ ,  $k=k$ ,  $p=2$ ,  $q=1$  (uogólnienie Connect-6)
- Gomoku:  $m=19$ ,  $n=19$ ,  $k=5$ ,  $p=q=1$  (istnieją różne warianty np.  $m=n=15$  lub takie w, których po przegranej rozgrywce gracz biały ma pierwszy ruch)
- Tic-Tac-Toe:  $m=n=k=3$ ,  $p=q=1$  (znane w Polsce jako kółko i krzyżyk)

Connect-6 jest szczególnie interesujące ze względu na wyeliminowanie przewagi gracza czarnego. Oczywiście ze względu ograniczeń fizycznych, gry te w praktyce muszą być rozgrywane na ograniczonych planszach – często wybiera się po prostu bardzo duże plansze typu 19x19 lub nawet 59x59)

### 2.2.2 Plik shared.h – definicje stałych i typów używanych w całym programie

Analizę należy rozpocząć od pliku nagłówkowego, który jest załączany dyrektywą include w prawie każdym innym pliku z kodem. To tutaj znajdziemy definicję podstawowych stałych, jak i ograniczeń dla nich. Poniżej przedstawiono istotny fragment z początku pliku. Zdefiniowane są ograniczenia wartości parametrów k,p,q oraz wielkości planszy. Nie ma generalnie jakichś odgórnych wymagań co do wartości tych ograniczeń. Autorzy programu zdefiniowali je tak, a nie inaczej między innymi po to, aby obraz gry powstający w plikach definiujących interfejs użytkownika był w miarę czytelny.

```
/* We limit the maximum values of these variables; note that these are not
   arbitrary limits and should be modified with care */
#define MAX_BOARD_SIZE 59
#define MAX_CONNECT_K 12
#define MAX_PLACE_P 12
#define MAX_START_Q 6
#define MAX_DEPTH 9
#define MAX_BRANCH 32
```

Z tych stałych możemy też wyciągnąć ciekawy wniosek, że chociaż autorzy nazwali swój program grający „Connect-k”, to tak naprawdę wartości p i q również są parametryzowane. Program jest więc dość uniwersalny i bliski prawdziwej grze (m,n,k,p,q) [32] [24] [35]. Część z zadeklarowanych stałych dotyczy zachowania AI, ograniczona jest maksymalna głębokość przeszukiwania jak i maksymalna ilość przeszukiwanych gałęzi gry.

Poniżej przedstawiono nienazwany enum, który przechowuje stałe oznaczające m.in. różne typy pól. Jeżeli pole na planszy gry jest puste, to przyjmuje wartość PIECE\_NONE, jeżeli zajęte przez pierwszego lub drugiego gracza, to odpowiednio PIECE\_BLACK i PIECE\_WHITE. Ciekawostką jest fakt wprowadzenia w reprezentacji pól definicji pola błędного PIECE\_ERROR, używane jest ono do otoczenia paskiem o szerokości 1 całej planszy, tak aby łatwo wykrywać wychodzenie poza planszę lub źle ustawiony indeks w tablicy. Jest to rozwiązanie o tyle dobre, że w języku C w porównaniu do języków zarządzanych jak C# nie ma standardowego sposobu wykrywania wyjścia poza granicę tablicy [41] [42] czy też innych tego typu błędów – tablica to po prostu wskaźnik, co może prowadzić do wielu błędów. Wprowadzając definicję pola błędnego PIECE\_ERROR, możemy uniknąć przynajmniej niektórych błędów wyjścia indeksu poza normalną planszą. Inne zadeklarowane stałe w tym enumie to m.in. znaczniki używane przez AI do określania np. ryzyka w strategii opartej o ryzyko, jak i znaczniki używane przy przeszukiwaniu drzewa / potencjalnego drzewa gry.

```
enum {
    PIECE_ERROR = -1,
    /* Error pieces form a one tile deep border around the board */

    PIECE_NONE = 0,
    PIECE_BLACK,
    PIECE_WHITE,
    /* Empty and played pieces */
```

```

PIECES,
/* Total number of normal pieces (2) */

PIECE_SEARCHED,
PIECE_SEARCHED_MAX = PIECE_SEARCHED + MAX_DEPTH,
/* Markers used by the search system */

PIECE_THREAT0,
PIECE_MARKER = PIECE_THREAT0,
/* These threat markers are usable by the AIs */
};

typedef int PIECE;

```

Do przechowywania informacji o typie pola na planszy (zajęte / wolne / błędne / znaczniki AI) używa się typu int poprzez odpowiednio zdefiniowany typedef. Wydaje się to na pozór nieoptymalne, gdyż jeżeli wartości byłyby ze zbioru (pierwszy gracz, drugi gracz, puste) lub nawet z polem błędu, to i tak wystarczyłby 8-bitowy char lub unsigned char. Jednak znaczniki używane przez AI, np. oznaczenia ryzyka, przyjmują w tej implementacji Connect-k wartości aż do INT\_MAX. Autorzy potrzebowali większej dziedziny, aby lepiej wartościować wagę pozycji czy posunięć. Niemniej jednak można sobie wyobrazić efektywniejszą implementację, gdzie ewentualna informacja o zagrożeniu posunięcia jest trzymana gdzieś indziej, a stany planszy są lżejsze. Innym rozwiązaniem byłoby chociaż ograniczenie dziedziny wag pozycji czy zagrożenia do przedziału, i tak dosyć obszernego, reprezentowanego przez np. typ short.

Poniżej przedstawiono kilka użytecznych makr zdefiniowanych również w pliku shared.h. Prawdopodobnie wybrano definicję makr zamiast funkcji z powodów wydajnościowych. Makra te służą do wyznaczania wartości znaczników pól używanych przez AI (np. zagrożenie). Makro piece\_empty(p) sprawdza za to, czy pole jest puste – nie jest to całkiem trywialne, ponieważ pole jest puste wtedy, kiedy jest oznaczone jako puste lub jest markerem używanym przez AI.

```

#define MAX_THREAT (INT_MAX - PIECE_THREAT0)
/* Highest value a threat marker can have */

#define PIECE_THREAT(n) (PIECE_THREAT0 + (n))
/* This marker represents a threat n-turns (of that player) away */

#define piece_empty(p) ((p) == PIECE_NONE || (p) >= PIECES)
/* Checks if a piece is an empty or a marker */

typedef unsigned int PLAYER;
/* Type for AIs, this is the index of the AI entry in ai.c */

typedef unsigned int BCOORD;
/* Type for board coordinates */

```

Powyżej przedstawiono również kolejne typedef'y używane tym razem jako odpowiednio indeks gracza AI w metodach sztucznej inteligencji i typ do trzymania współrzędnych na planszy. Po raz kolejny część kodu wydaje się nieoptymalna – współrzędne raczej nie potrzebują tak dużej dziedziny, jaką jest unsigned int. Tym razem jednak nieoptymalność nie jest tylko pozorna, ale rzeczywiście przy zdefiniowanej na początku pliku

stałej ograniczającej maksymalny rozmiar planszy na 59 (MAX\_BOARD\_SIZE) nie potrzebujemy aż tak dużego typu.

### 2.2.3 Plansza i stany gry

W dalszej części pliku shared.h znajduje się intuicyjnie podstawowa struktura reprezentująca stan gry. Jest nią struct Board reprezentujący planszę gry w danym momencie i wszystkich momentach w historii. Struktura ta jest niezwykle ważna, gdyż jest używana nie tylko do reprezentacji obecnego stanu gry, ale także przez wszystkie algorytmy AI do reprezentacji wszystkich stanów w drzewie gry itp. Struktura trzyma m.in. wskaźnik na poprzednią swoją wersję, czyli de facto poprzedni stan gry w zmiennej parent. Stosuje się także tutaj zmienną typu AllocChain reprezentującą pewien łańcuch alokacji obiektów (alokacji kolejnych stanów planszy, np. w algorytmach symulacyjnych AI), niezwykle przydatną do czyszczenia pamięci po wykonaniu obliczeń przez AI – po wykonaniu obliczeń i wyznaczeniu ruchu usuwa się np. potencjalne przyszłe plansze z algorytmów symulacyjnych AI. Zmienna ac musi być jako pierwsza zadeklarowana w strukturze, gdyż często wykonujemy rzutowanie tych struktur, aby łatwiej alokować / dealokować pamięć.

```
typedef struct AllocChain {
    gboolean free;
    /* Is this object unallocated? */

    unsigned int id;
    /* Each object has a unique id */

    struct AllocChain *next;
    /* Next object in the chain */
} AllocChain;

typedef struct Board {
    AllocChain ac;
    /* Allocation chain must be the first member */

    unsigned int moves_left;
    /* How many moves the current player has left */

    struct Board *parent;
    /* The board preceding this one in history */

    gboolean won;
    BCOORD win_x1, win_y1, win_x2, win_y2;
    /* On won boards, used to indicate where the winning line is */

    PIECE turn;
    /* Whose turn it is on this board */

    BCOORD move_x, move_y;
    /* The move to the next Board in history */

    PIECE data[];
} Board;
/* The board structure represents the state of the game board. Do NOT preserve
   board pointers across games. */
```

Biorąc pod uwagę optymalność silnika, niezwykle ważne jest, jak trzymane są konkretne pola planszy. Tutaj jest to zwykła tablica o nazwie data trzymająca wartości PIECE, które – jak wiemy z poprzedniego podrozdziału – są tak naprawdę typu int. Reprezentacja ta nie jest więc tak optymalna, jak byłby np. bitboard, trzeba jednak zauważyć, że ma pewne zalety [43] [44] i obliczenia / funkcje na niej działające są dzięki temu w miarę proste.

Do trzymania stanu użyto po prostu zmiennych globalnych typów opisanych wcześniej ze słowkiem extern – tak aby w każdym pliku z dyrektywą include do shared.h można było się do tych zmiennych odnosić.

```
extern AllocChain *board_root;
extern gsize board_mem;
/* Variables for the allocation chain */

extern Board *board;
/* This is the current board. Do NOT modify it, that's cheating. :) */

extern int board_size, board_stride, move_no, connect_k, place_p, start_q;
/* Board size (for all boards), moves in the game, connect_k to win, place_p
   moves at a time, black has start_q moves on the first move; do NOT modify
   these directly! */
```

Ostatnim ciekawym fragmentem kodu (poza definicjami struktur dla AI, o czym będzie w dalszym podrozdziale) są funkcje z dyrektywą inline dla wydajności i makra służące do wygodnego dostępu do pól planszy do gry.

```
static inline PIECE piece_at(const Board *b, BCOORD x, BCOORD y)
{
    return b->data[(y + 1) * board_stride + x + 1];
}
/* Returns the piece at (x, y) on board b. If the coordinates are out of range,
   this function will return PIECE_ERROR. */

static inline void place_piece_type(Board *b, BCOORD x, BCOORD y, PIECE type)
{
    b->data[(y + 1) * board_stride + x + 1] = type;
}
#define place_piece(b, x, y) place_piece_type(b, x, y, (b)->turn)
#define place_threat(b, x, y, n) place_piece_type(b, x, y, PIECE_THREAT(n))
/* Places a piece on board b, overwriting any piece that was previously in that
   place */
```

Z powodu użycia tablicy jednowymiarowej zamiast dwuwymiarowej oraz dodatkowego „paska błędu” wokół planszy niezbędne są pewne przeliczenia. Zmienna board\_stride reprezentuje rozmiar planszy z uwzględnieniem szerokości „paska błędu”, czyli jest to rozmiar planszy powiększony o dwa. Łatwo sprawdzić, że funkcje działają poprawnie, próbując podstawić za y lub x wartość 0.

#### 2.2.4 Plik state.c i funkcje modyfikujące stan

W pliku state.c znajduje się właściwa deklaracja, implementacja i ustawienie zmiennych używanych do trzymania globalnego stanu gry, dostępnych z niemal każdego pliku w projekcie za pomocą opisywanego w poprzednim podrozdziale pliku nagłówkowego. Są tu implementacje

funkcji „globalnych”, zadeklarowanych w pliku shared.h, jak i implementacje funkcji, które muszą być dostępne z punktu widzenia interfejsu użytkownika i są zadeklarowane w pliku connectk.h. Poniżej przedstawiono zmienne reprezentujące stan i parametry gry.

```
Board *board;
AllocChain *board_root = NULL;
int board_size, board_stride, move_no, move_last,
    connect_k = 6, place_p = 2, start_q = 1;
gsize board_mem = 0;
```

Inicjalizacja i alokacja pamięci na planszę to jeden z pierwszych kroków nowej rozgrywki. W poniżej przedstawionych dwóch funkcjach widać cały proces z m.in. ustawieniem wokół planszy pól typu PIECE\_ERROR, o czym pisano wcześniej.

```
static void board_init(Board *b)
{
    memset((char*)b + sizeof(AllocChain), 0, sizeof(Board) -
           sizeof(AllocChain));
}

AllocChain *board_alloc(AllocChain *ac)
{
    Board *b = (Board*)ac;
    int i;

    /* Clear the old board */
    if (b) {
        for (i = 1; i <= board_size; i++)
            memset(b->data + board_stride * i + 1, 0,
                   board_size * sizeof(PIECE));
        board_init(b);
        return (AllocChain*)b;
    }

    /* New boards are allocated with a 1-tile wide boundary of PIECE_ERROR
       around the edges */
    b = (Board*)g_slice_alloc0(board_mem);
    memset(b->data, PIECE_ERROR, sizeof(PIECE) * board_stride);
    for (i = 1; i <= board_size; i++) {
        b->data[i * board_stride] = PIECE_ERROR;
        memset(b->data + board_stride * i + 1, 0,
               board_size * sizeof(PIECE));
        b->data[(i + 1) * board_stride - 1] = PIECE_ERROR;
    }
    memset(b->data + board_stride * (board_stride - 1), PIECE_ERROR,
           sizeof(PIECE) * board_stride);
    board_init(b);
    return (AllocChain*)b;
}

void board_clean(Board *b)
{
    int y, x;

    for (y = 0; y < board_size; y++)
        for (x = 0; x < board_size; x++)
            if (piece_at(b, x, y) >= PIECES)
```

```

    place_piece_type(b, x, y, PIECE_NONE);
}

```

Kod jest dosyć trywialny, tak samo jak powyżej przedstawione “czyszczenie” planszy.

Zmiana rozmiaru planszy jest z kolei znacznie bardziej złożonym procesem niż proste jej wyczyszczenie. W związku z tym, że w języku C nie ma garbage collectora, przydaje się tutaj zaimplementowany wcześniej AllocChain, który znacznie ułatwia usunięcie z pamięci kolejnych historycznych reprezentacji planszy (stanów gry). Oczywiście, jeżeli wołamy funkcję z parametrem równym rozmiarowi planszy, to nie robimy nic.

```

void set_board_size(unsigned int size)
{
    if (board_size == size)
        return;
    draw_marks(NULL, FALSE);
    achain_dealloc(&board_root, board_mem);
    achain_dealloc(&aimoves_root, aimoves_mem);
    board_size = size;
    board_stride = size + 2;
    board_mem = sizeof(Board) + board_stride * board_stride *
                sizeof(PIECE);
    aimoves_mem = sizeof(AIMoves) + size * size * sizeof(AIMove);
}

```

Funkcja achain\_dealloc wywoływana jest zarówno na wskaźniku na korzeń alokacji plansz gry, jak i korzeń alokacji ruchów AI. W ten sposób czyścimy całą zaalokowaną w trakcie gry pamięć.

```

static void achain_dealloc(AlocChain **root, gsize mem)
{
    AllocChain *ac = *root, *ac_next;

    while (ac) {
        ac_next = ac->next;
        g_slice_free1(mem, ac);
        ac = ac_next;
    }
    *root = NULL;
}

```

Wreszcie, nie pisząc już więcej o alokacji i dealokacji pamięci, przechodzimy do dwóch prawdopodobnie najważniejszych funkcji w pliku state.c. Są to funkcje sprawdzające warunek wygranej.

```

gboolean check_win_full(const Board *b, BCOORD x, BCOORD y,
                       BCOORD *x1, BCOORD *y1, BCOORD *x2, BCOORD *y2)
{
    int i, c1, c2, xs[] = {1, 1, 0, -1}, ys[] = {0, 1, 1, 1};
    PIECE type;

    type = piece_at(b, x, y);
    if (type != PIECE_BLACK && type != PIECE_WHITE)
        return FALSE;
    for (i = 0; i < 4; i++) {

```

```

    c1 = count_pieces(b, x, y, type, xs[i], ys[i], NULL);
    c2 = count_pieces(b, x, y, type, -xs[i], -ys[i], NULL);
    if (c1 + c2 > connect_k) {
        if (x1)
            *x1 = x + xs[i] * (c1 - 1);
        if (y1)
            *y1 = y + ys[i] * (c1 - 1);
        if (x2)
            *x2 = x - xs[i] * (c2 - 1);
        if (y2)
            *y2 = y - ys[i] * (c2 - 1);
        return TRUE;
    }
}
return FALSE;
}

```

Funkcja check\_win\_full sprawdza, czy świeżo po wykonaniu ruchu do współrzędnych x,y przez jednego z graczy ten ruch zapewnia mu wygraną. Zwraca również poprzez wskaźniki x1,x2,y1,y2 współrzędne linii dającej wygraną, jeżeli ktoś wygrał. Kluczową częścią tej funkcji są wywołania funkcji count\_pieces ze współrzędnymi x,y i kolejnymi czterema kierunkami od tych współrzędnych z rosnącymi i malejącymi współrzędnymi. Po podliczeniu, ile pod rząd pól tego samego gracza występuje, sprawdzamy, czy jest ich więcej niż k (c1+c2 daje o 1 więcej, bo pole x,y liczymy w obu wywołaniach) – należy tutaj zauważyc różnicę od gry Gomoku, gdzie musiałoby być dokładnie k+1 [34]. Jeżeli jest ich więcej niż k, to wyznaczamy z pomocą tablic przesunięć xs i wartości ilości c1, c2 prostym przekształceniem matematycznym wartości współrzędnych wygrywającej linii i zwracamy prawdę. W innym wypadku po sprawdzeniu wszystkich kierunków od x,y zwracamy po prostu fałsz.

Jednocześnie widać możliwość pewnej optymalizacji, której autorzy programu niestety uniknęli. Jako że przed wywołaniem tej funkcji wiemy, kto dokonał ruchu, nie ma potrzeby sprawdzania funkcją piece\_at typu pola, na które dokonano ruchu. Typ można by po prostu przekazać do funkcji, co zwiększyłoby w jakimś stopniu wydajność programu, biorąc pod uwagę, jak często dla algorytmów symulacyjnych AI ta funkcja jest wywoływana na symulowanych planszach.

Poniżej zaprezentowano jeszcze funkcję count\_pieces, jako że jest kluczowa dla całego procesu.

```

int count_pieces(const Board *b, BCOORD x, BCOORD y, PIECE type, int dx, int dy,
                 PIECE *out)
{
    int i;
    PIECE p = PIECE_NONE;

    if (!dx && !dy)
        return piece_at(b, x, y) == type ? 1 : 0;
    for (i = 0; x >= 0 && x < board_size && y >= 0 && y < board_size; i++) {
        p = piece_at(b, x, y);
        if (p != type)
            break;
        x += dx;
        y += dy;
    }
    if (out)
        *out = p;
}

```

```

    }
    if (out)
        *out = p;
    return i;
}

```

Funkcja jest dosyć trywialna. Zliczamy wszystkie wystąpienia pola typu type od miejsca na planszy o współrzędnych x,y w kierunku zdefiniowanym przez parametry dx,dy tak długo, aż się skończy plansza lub wystąpi inny typ pola. Opcjonalnie, jeżeli ostatni argument nie był Nullem, możemy zwrócić typ pola, które zatrzymało nasze zliczanie.

#### 2.2.5 Sztuczna inteligencja – ogólnie

Program Connect-k powstał w pewnym sensie jako pole testów różnych technik AI do problemu rozwiązywania gier z serii Connect [37]. W związku z tym istnieje w sumie aż 8 różnych algorytmów sztucznej inteligencji zaimplementowanych na potrzeby programu, z czego 6 jest udostępnionych przez interfejs graficzny, a przynajmniej 3 są na tyle silne, że mogą być wyzwaniem nawet dla ponadprzeciętnego gracza. Zaimplementowane algorytmy posortowane względem złożoności i efektywności to:

- ai\_random (stawia losowy ruch na wolne pole),
- ai\_adjacent (naiwne podejście do próby zbudowania sekwencji k-pod-rząd),
- ai\_threats (strategia oparta o zagrożenie, podobne do gry średnio doświadczonego człowieka),
- ai\_windows (ciekawe podejście polegające na zastosowaniu przesuwającego się okna poszukiwań wystąpień pod rząd pól tego samego gracza),
- ai\_priority (priorytetyzacja ruchów zwróconych przez ai\_threats),
- ai\_sequences (złożone podejście polegające na budowaniu konfiguracji, które potencjalnie mogą stać się później liniami wygrywającymi),
- DFS alpha-beta
- ai\_mesh (ciekawe podejście, siatka gęstości pól zajętych),
- ai\_monte\_carlo (metoda Monte Carlo).

Pierwsze dwa jako trywialne nie są warte omawiania. Algorytmy ai\_windows i ai\_priority z bliżej nieznanego powodu nie są uwzględnione w interfejsie użytkownika (być może są jeszcze niedokończone lub występują w nich błędy), więc również ich analiza nie ma większego sensu, tym bardziej gdy uwzględnili się zakres tematyczny niniejszej pracy (autor skupia uwagę na silnikach programów grających i ich implementacji, co wraz z AI ma wpływ na wydajność całego programu grającego, ale interesuje go przede wszystkim wpływ implementacji silnika programu grającego, a nie jego AI, na wydajność). Ostatecznie zadecydowano więc, że zostanie omówiony najciekawszy z nich: ai\_monte\_carlo.

Zanim dokonana zostanie analiza wspomnianych algorytmów, należy przybliżyć struktury i metody dedykowane AI i współdzielone przez wszystkie te algorytmy. Znajdziemy je przede wszystkim w plikach state.c, ai.c oraz shared.h.

Podstawą są struktury AIMove i AIMoves odpowiedzialne za reprezentacje ruchu AI oraz sekwencji ruchów AI. Ta druga struktura, trzymając AllocChain, przypomina nieco w swojej koncepcji strukturę planszy opisywaną kilka podrozdziałów wcześniej. Faktycznie możliwość alokacji i dealokacji łańcuchów ruchów AI jest przydatna w niektórych z algorytmów. Struktura AIMove posiada jedynie współrzędne (unsigned int) oraz wagę (int).

```

typedef struct {
    AIWEIGHT weight;
    BCOORD x, y;
} AIMove;
/* AIs return an array filled with these */

typedef struct AIMoves {
    AllocChain ac;
    /* Allocation chain must be the first member */

    unsigned int len;
    /* Number of members in data */

    AIWEIGHT utility;
    /* A composite utility value set by some AIs when producing a moves
       list */

    AIMove data[];
    /* Array of AIMove structures */
} AIMoves;
/* An array type for holding move lists */

```

Istotne są też niektóre z góry istniejące wagi przydzielane konkretnym ruchom w zależności od tego, co robią.

```

/* Some guideline values for move weights: */
#define AIW_MAX          INT_MAX           /* largest weight value */
#define AIW_MIN          INT_MIN           /* smallest weight value */
#define AIW_WIN           AIW_MAX          /* this move wins the game */
#define AIW_DEFEND        (AIW_WIN - 2)     /* defends from an opponent win */
#define AIW_NONE          0                 /* does nothing */
#define AIW_DRAW          AIW_NONE         /* draw game */
#define AIW_LOSE          (-AIW_WIN)       /* this move loses the game */
#define AIW_THREAT_MAX    262144          /* value of an immediate threat */

```

Liczne funkcje powiązane z alokacją i dealokacją łańcuchową ruchów sztucznej inteligencji zostaną tutaj pominięte ze względu na podobieństwo do analogicznych funkcji dla planszy. Dwie podobne do siebie funkcje, które służą do dodawania lub połączania nowych ruchów do struktury AIMoves, przedstawiono poniżej.

```

void aimoves_add(AIMoves *moves, const AIMove *move)
{
    int i;

    i = aimoves_find(moves, move->x, move->y);
    if (i < 0) {
        if (moves->len >= board_size * board_size)

```

```

        g_warning("Attempted to add a move to a full AIMoves");
    else
        moves->data[moves->len++] = *move;
    } else
        moves->data[i].weight += move->weight;
}

void aimoves_append(AIMoves *moves, const AIMove *move)
{
    int i;

    if (move->x >= board_size || move->y >= board_size)
        return;
    for (i = 0; i < moves->len; i++) {
        AIMove *aim = moves->data + i;

        if (aim->x == move->x && aim->y == move->y) {
            aim->weight = move->weight;
            return;
        }
    }
    if (moves->len >= board_size * board_size) {
        g_warning("Attempted to append a move to a full AIMoves");
        return;
    }
    moves->data[moves->len++] = *move;
}

```

Jak łatwo zauważyc, funkcje różnią się jedynie tym, że aimoves\_add może dodać wagę do już istniejącego ruchu, podczas gdy aimoves\_append z definicji umieszcza nowy ruch, więc jeżeli taki sam już istnieje, to waga zostaje podmieniona. Pierwsza funkcja znajduje zastosowanie w metodzie Monte Carlo i z siatkami gęstości, druga w metodzie z oknem przesuwnym, losowej, ai\_adjacent i przy łączeniu struktur AIMoves. Warto też podkreślić, że struktura AIMoves może przechowywać maksymalnie tyle ruchów, ile jest pól na planszy, co z jednej strony jest oczywiste, a z drugiej jeszcze bardziej upodabnia ją w koncepcji do struktury planszy.

W przeszukiwaniu DFS spore znaczenie ma też funkcja zwracająca najlepszy możliwy ruch ze struktury AIMoves. Wykonuje ona sortowanie po wagach i wybiera losowy indeks ze zbioru indeksów ruchów o najwyższej wadze (gdyby więcej niż jeden ruch był najlepszy).

```

int aimoves_choose(AIMoves *moves, AIMove *move)
{
    int i = 0, top = 0;

    if (!moves || !moves->len)
        return 0;
    aimoves_sort(moves);
    for (top = 0; top < moves->len &&
        moves->data[top].weight == moves->data[0].weight; top++);
    if (top)
        i = g_random_int_range(0, top);
    *move = moves->data[i];
    return 1;
}

```

Duże znaczenie ma więc wywołanie funkcji sortującej aimoves\_sort, jest to zwykły quicksort z wywołaniem poniższej funkcji porównującej.

```

int aimoves_compare(const void *a, const void *b)
{
    return ((AIMove*)b)->weight - ((AIMove*)a)->weight;
}

```

Sortowanie przebiega oczywiście po wagach ruchów.

Z punktu widzenia Monte Carlo, gdzie wstępnie interesuje nas n najlepszych ruchów wyznaczonych z innego algorytmu, ważna jest też zaprezentowana poniżej metoda aimoves\_crop, która zwraca n lub mniej najlepszych ruchów.

```

void aimoves_crop(AIMoves *moves, unsigned int n)
{
    if (moves->len < n)
        return;
    aimoves_shuffle(moves);
    aimoves_sort(moves);
    moves->len = n;
}

```

Jako że algorytm quicksort używany do sortowania ruchów w metodzie aimoves\_sort ma pesymistyczną złożoność  $O(n^2)$  w przypadku tablicy już posortowanej, autorzy na wszelki wypadek najpierw „przetasowują” tablicę ruchów, a potem wysyłają do sortowania. Również tutaj można by się pokusić o drobną optymalizację i np. w funkcji sort sprawdzać, czy tablica już jest posortowana.

W dalszej części kodu mamy także metody duplikujące, wyszukujące (ruch lub zakres wag), łączące z wykorzystaniem aimoves\_append, jak i scalające, a nawet odejmujące struktury AIMoves, jednak ich implementacja nie jest szczególnie ciekawa ani pod względem analizy wydajnościowej, ani widzenia trudności.

W usuwaniu konkretnego indeksu z pola data struktury AIMoves posłużono się drobną optymalizacją polegającą na szybkim usunięciu wartości z indeksu, wstawiając tam ostatni element i zmniejszając długość tablicy o 1.

```

void aimoves_remove_index_fast(AIMoves *moves, int i)
{
    if (moves->len > i)
        moves->data[i] = moves->data[moves->len - 1];
    moves->len--;
}

void aimoves_remove(AIMoves *moves, BCOORD x, BCOORD y)
{
    int i;

    for (i = 0; i < moves->len; i++) {
        AIMove *aim = moves->data + i;

        if (aim->x == x && aim->y == y) {
            aimoves_remove_index_fast(moves, i);
            return;
        }
    }
}

```

```

        }
    }

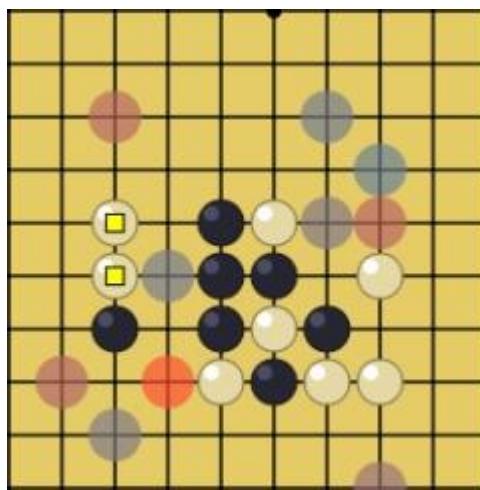
```

Jest to drobna optymalizacja, ale dla algorytmów AI znaczy wiele, ponieważ ponowna alokacja pamięci przy każdym usunięciu ruchu w przypadku Monte Carlo, który gra 1000 razy i w każdej grze po każdym ruchu usuwa ze struktur dostępnych ruchów ten świeżo wykonany, byłaby na tyle kosztowana, że algorytm stałby się bezużyteczny.

### 2.2.6 Monte Carlo

Inspiracją do zastosowania metody Monte Carlo w programie Connect-k były dla autorów wcześniejsze próby zastosowania symulowanego wyżarzania do gry w Go [37]. Po generacji niewielkiej liczby ruchów (np. 10) za pomocą innego algorytmu (np. sekwencji) pewna stosunkowo duża liczba gier losowych (np. 1000) jest odgrywana dla wzbogaconej o konkretny ruch planszy. Zostaje wybrany ruch, dla którego największa liczba gier losowych kończy się wygraną obecnego gracza.

Monte Carlo generuje w praktyce silne posunięcia, ale spędza sporo czasu obliczeniowego, wybierając ruch. To oczywiście wynika z faktu, że w pewnym sensie jest to jednak technika brute-force, tyle tylko że przeszukujemy ograniczoną ilość losowych gier zamiast wszystkie możliwe. Bardzo duża przestrzeń stanów ma niebagatelny wpływ na wydajność. Poniżej znajduje się obrazek planszy z programu, gdzie po uruchomieniu metody Monte Carlo dla czterech ruchów wygenerowanych z algorytmu „sekwencji” (algorytm ten optymalizuje długość sekwencji „pod-rząd” pionów tego samego gracza), zostanie wybrany tak naprawdę ten sam ruch (kolor pomarańczowy – wybrany, bordowy to analizowane), który wybrałby sekwencja.



Rysunek 2.7. Monte Carlo

Poniżej przedstawiono kod metody `ai_monte_carlo`, odpowiedzialnej za zwrócenie `MONTE_N` (zdefiniowane na 10) lub mniej najlepszych według metody ruchów. Jak widać, metoda alokuje pamięć na nową planszę, pozyskuje wstępnie dobre ruchy z metody `move_utilities`, wybiera z nich `MONTE_N` najlepszych. Następnie odpowiednio dla każdego ruchu:

- tworzy kopię planszy,

- wykonuje na niej ruch,
- jeżeli ruch jest wygrywający, to przydziela mu wagę MONTE\_NUM\_RUNS (czyli tak jakby wygrał wszystkie gry losowe),
- jeżeli nie, to rozgrywa MONTE\_NUM\_RUNS (domyślnie 1000) gier losowych i przydziela ruchowi wagę zgodnie z ilością wygranych gier – 1 za każdą wygraną,
- pole utility struktury moves przechowuje ogólną liczbę wygranych gier losowych przez wszystkie ruchy.

Na koniec jest zwalniana pamięć i zwracana jest struktura zawierająca wszystkie analizowane ruchu wraz z przydzielonymi wagami.

```

AIMoves *ai_monte_carlo(const Board *b)
/* chooses the best move based on which one wins the most random games */
{
    int i, k, wins, len;

    Board *new_board = board_new();

    AIMove move;
    AIMoves *moves = move_utilities(b);
    moves->utility = 0;
    aimoves_crop(moves, MONTE_N);

    len = moves->len;

    for (i = 0; i < len; i++) {

        move = moves->data[i];

        board_copy(b, new_board);
        place_piece(new_board, move.x, move.y);

        if (check_win(new_board, move.x, move.y)) {
            move.weight = MONTE_NUM_RUNS;
            moves->data[i] = move;
            moves->utility += MONTE_NUM_RUNS;
        } else {
            /* run the monte carlo trials */
            wins = 0;
            for (k = 0; k < MONTE_NUM_RUNS; k++) {
                wins += mc_run(new_board);
            }
            move.weight = wins;
            moves->data[i] = move;
            moves->utility += wins;
        }
    }

    board_free(new_board);
    return moves;
}

```

Należy jeszcze pokazać metodę mc\_run, która jest odpowiedzialna za rozegranie gry losowej na zmodyfikowanej o jeden, nowy – analizowany ruch na planszy.

```

int mc_run(Board *b)
/* plays a random game on board b, returns 1 if the current player wins
and 0 otherwise */
{
    Board *new_board = board_new();
    board_copy(b, new_board);

    AIMove move;
    AIMoves *empties;
    empties = empty_cells(new_board);
    int tries = 0;
    int i;

    while ( TRUE ) {

        /* if the board filled up, start over */
        if (empties->len == 0) {
            board_copy(b, new_board);
            empties = empty_cells(new_board);
            tries++;
            if (tries == 10) {
                g_debug("bailing");
                board_free(new_board);
                aimoves_free(empties);
                return 0;
            }
        }

        i = g_random_int_range(0, empties->len);
        move = empties->data[i];
        aimoves_remove_index_fast(empties, i);

        place_piece(new_board, move.x, move.y);

        if (check_win(new_board, move.x, move.y)) {
            if (new_board->turn == board->turn) {
                board_free(new_board);
                aimoves_free(empties);
                return 1;
            }
            else {
                board_free(new_board);
                aimoves_free(empties);
                return 0;
            }
        }

        new_board->moves_left--;
        if (new_board->moves_left == 0) {
            new_board->turn = other_player(new_board->turn);
            new_board->moves_left = place_p;
        }
    }
}

```

Aby nie tracić cennego czasu obliczeniowego, dla planszy, dla której po 10 losowych grach nie osiągnięto wygranej AI ani wygranej przeciwnika, zwracana jest po prostu wartość 0 – czyli AI nie wygrało. Ruchy stawiane przez graczy są losowe, ponieważ ze wszystkich pustych pól na

planszy zwracanych przez metodę empty\_cells wybieramy za każdym razem to o losowym indeksie.

### 3. IMPLEMENTACJA GENERATORA SILNIKÓW WYBRANEJ GRY

Na podstawie całej zdobytej z analizy silników popularnych gier wiedzy należało zaimplementować generator, który zależnie od zdefiniowanych parametrów będzie generować silnik do takiej sparametryzowanej gry. W uproszczeniu ideę można by zaprezentować w kilku punktach:

1. Implementacja silnika wybranej gry w C/C++, ale w taki sposób, aby jego kod był łatwy do modyfikacji – np. używanie flag, define'ów itp. możliwych do ustawienia przed komplikacją
2. Implementacja generatora w C#.NET, który jest aplikacją z interfejsem graficznym (WPF), z poziomu którego można zdefiniować wszystkie parametry dla gry (np. rozmiar planszy itp.)
3. Generator potrafi skompilować silnik w C/C++, ale wysyłając wartości odpowiednich stałych do kompilatora, wpływa mocno na wygląd ostatecznego kodu – np. dla mniejszej planszy używa mniejszych typów danych itp.
4. Implementacja programu GUI (C#.NET, WPF), odpowiedzialnego tylko za wyświetlanie stanu gry, planszy itp. (żadnej logiki)
5. GUI potrafi się komunikować ze skompilowanym silnikiem (wysyłanie ruchów człowieka, pobieranie generowanych ruchów przez AI z silnika)
6. Implementacja aplikacji testującej silniki (C#.NET, WPF)
7. Aplikacja-tester potrafi się komunikować z silnikiem i wysyłając predefiniowane ruchy, sprawdza poprawność i wydajność generowanych ruchów.

#### 3.1 Problem wyboru gry

Pierwsza decyzja, jaką należy podjąć przed rozpoczęciem implementacji, to wybór konkretnej gry logicznej. Możliwości jest oczywiście wiele, ale najpierw należy się zastanowić, co wynika z takiego lub innego wyboru. Wybranie gry, której silnik podlegał analizie w poprzednim rozdziale, mogłoby ułatwić implementację.

Inną sprawą jest wybór pomiędzy grą popularną a niszową. Z punktu widzenia parametryzacji ważny jest też wybór między grą złożoną i prostą. Wybór konkretnej gry jest więc przynajmniej problematyczny. W obu przypadkach zarówno jedna, jak i druga opcja ma swoje wady i zalety, które postaram się w skrócie przedstawić poniżej w tabeli 3.1.

**Tabela 3.1.** Zalety i wady – gry proste, złożone, znane i mniej znane

Typ gry	Zalety	Wady
Gra złożona	więcej wartości, które można parametryzować	więcej parametrów to więcej pracy, czasu i trudniejsza implementacja

Gra prosta	łatwiej i mniej pracy, trudniej o błędy	mała liczba parametrów może oznaczać, że tak naprawdę generator nie jest potrzebny – wystarczy sprytna implementacja silnika
Gra dobrze znana	dobre wsparcie ze strony społeczności, dużo materiałów i publikacji, dobre silniki, na których można się w pewnym stopniu wzorować	trudniej jest napisać coś szybszego lub sprytniejszego, dostępne rozwiązania są bardzo złożone
Gra słabiej znana	łatwiej napisać coś lepszego niż dostępne rozwiązania, można być pierwszym, rozwiązanie niekoniecznie musi być bardzo złożone	implementowany silnik prawdopodobnie będzie wolny (brak wyzwań), potrzeba napisania własnego GUI i opracowania własnych protokołów komunikacji

Jak widać, ciężko jest wskazać jednoznacznego zwycięzcę porównania. Dlatego też w kolejnych podrozdziałach zostaną przedstawione dwa wybory, każdy z innego etapu pracy.

### 3.1.1 Początkowy wybór – szachy

Po rozważaniu za i przeciw podjęto decyzję o wyborze gry w szachy. Za jego słusznością przemawiają argumenty zaprezentowane poniżej:

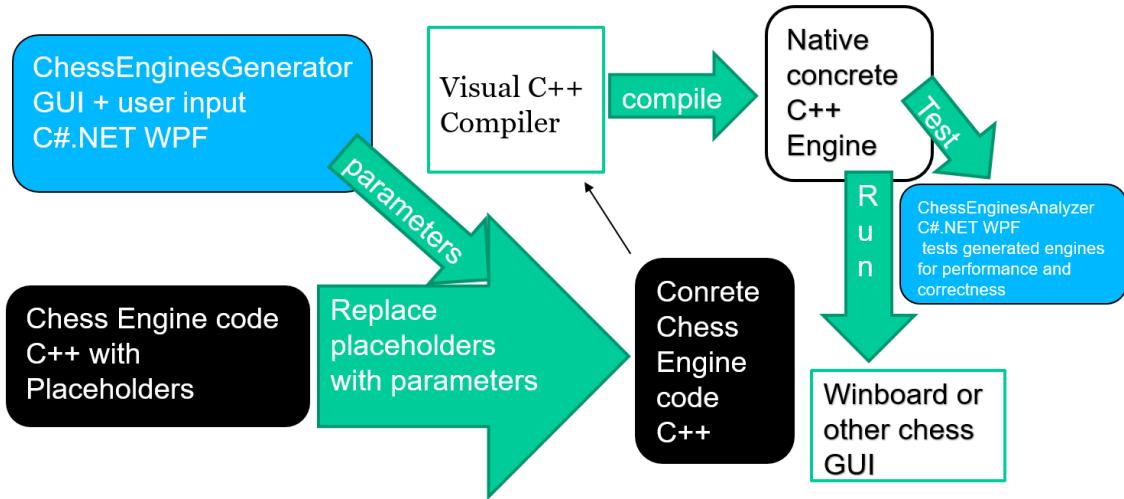
- bardzo dobre wsparcie ze strony społeczności,
- obfitość materiałów, opracowań, artykułów naukowych
- dostępne programy do GUI
- ustandaryzowane protokoły komunikacji silników z GUI
- bardzo dopracowane, bardzo wydajne i szeroko dostępne (otwarte) implementacje programów grających
- dużo wielkości nadających się do parametryzowania.

Szczególne wrażenie robi ilość wielkości, które możemy parametryzować. Autor wymyślił na początku 6 takich parametrów, szybko jednak okazało się, że jest ich nawet więcej. Przykładowe parametry to:

- liczba pól w formacie  $n \times m$  (czyli zarówno  $n$ , jak i  $m$ , szachownica nie musi być kwadratowa)
- kto ma pierwszy ruch
- ile ruchów jest wykonywanych do końca gry
- ilość figur każdego typu [szczególnie jeśli jakiegoś typu figury nie pojawią się (możliwość optymalizacji pamięci na stan gry)]
- rozstawienie początkowe bierek

- zasady promocji.

Po wstępny przemyśleniu parametrów nadszedł czas na rozpoczęcie implementacji. Na sam początek należało zastanowić się nad architekturą całego projektu. Wykorzystano w tym informacje z początku rozdziału nr 3. Wstępna architektura projektu widoczna jest na poniższym rysunku.



Rysunek 3.1. Architektura generatora silników szachowych

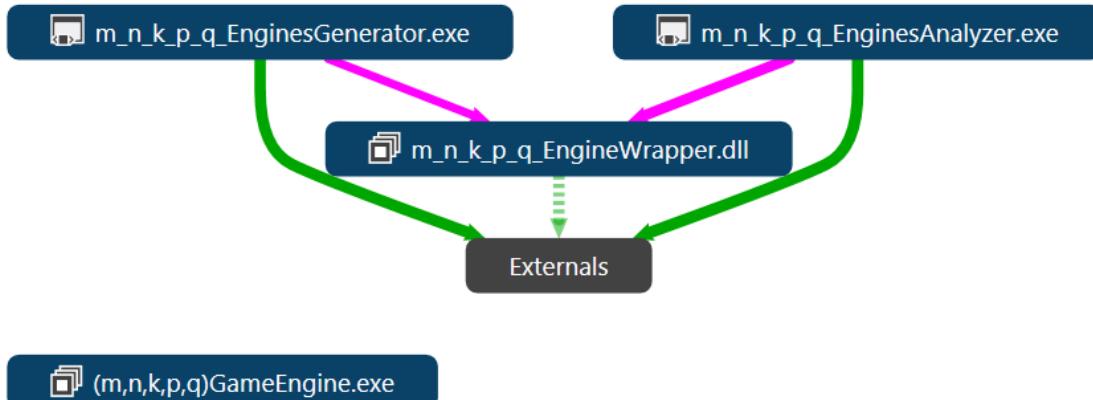
Zdecydowano się na użycie Winboard GUI (aplikacja GUI do szachów, korzystająca z ustandaryzowanego protokołu komunikacji z silnikami szachowymi UCI), co zdecydowanie jest praktyczniejszym rozwiązaniem, gdyż nie ma potrzeby pisania własnego programu do interfejsu graficznego gry.

### 3.1.2 Wybór ostateczny – Connect6, Gomoku, ( $m,n,k,p,q$ )

Z powodu analizy programu Connect-k i zauważenia jego niedociągnięć i wad, zamiast szachów, zdecydowano o realizacji generatora silników dla gier z rodziną ( $m,n,k,p,q$ ). Do tej rodziny należą takie właśnie gry, jak Connect6, Gomoku, Tic-Tac-Toe itp. Istnieje więc realna szansa napisania czegoś parametryzowanego i w dodatku lepszego oraz porównania tego z analizowanym wcześniej silnikiem. Przedstawiona wcześniej architektura generatora silników szachowych została dostosowana do nowego typu gry.

## 3.2 Architektura projektu

Architektura projektu dla gier ( $m,n,k,p,q$ ) jest tak naprawdę ewolucją architektury generatora silników szachowych. Główną zmianą jest definiowanie flag (wartości) w wywoływaniu kompilatora C++ zamiast zamianiania „placeholderów” w kodzie – wzorowano się tutaj na Stockfishu, gdzie odpowiednie flagi mówią, czy użyć danej funkcji. Wprowadzono także bibliotekę pośrednią ( $m,n,k,p,q$ )EngineWrapper, ułatwiającą współpracę z generowanymi silnikami z poziomu .NET-a. Podpatrując rozwiązania ze Stockfisha, generowane silniki zrealizowano jako aplikacje konsolowe, gdzie komunikacja opiera się na założonym protokole tekstowym.

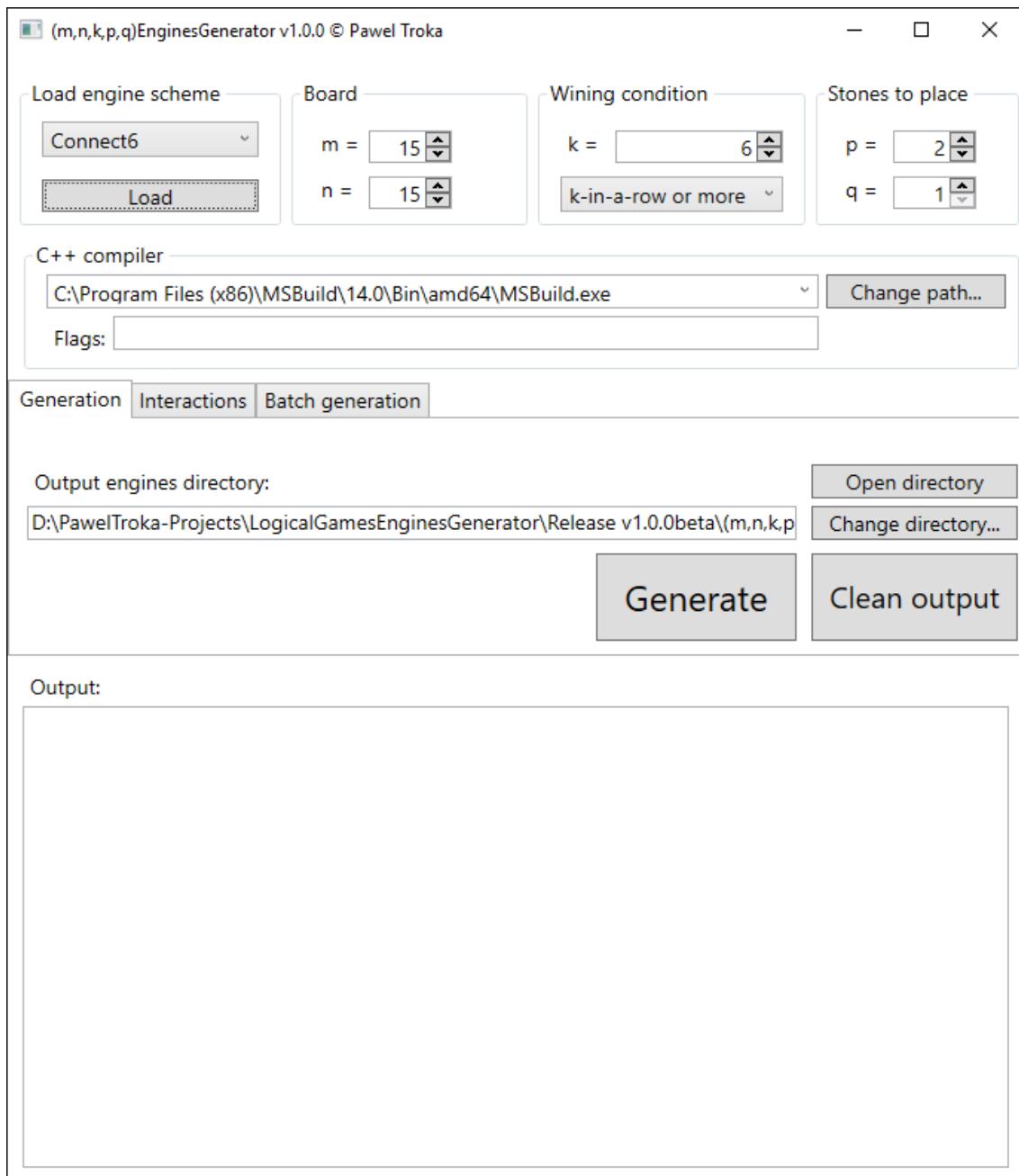


Rysunek 3.2. "Code map" dla solucji zawierającej wszystkie projekty

Projekt powstał w IDE Microsoft Visual Studio 2015 Ultimate. Oprócz przyzwyczajeń autora, dużym plusem z punktu widzenia projektu jest wsparcie wielu technologii – tutaj przydatne okazały się języki C++ (kod silników) i C# (kod pozostałych części projektu, w tym generatora). Visual Studio jest najlepszym IDE dla języka C# i środowiska .NET, a jego wsparcie dla najnowszych standardów C++11, C++14 i C++17 jest dość dobre [63][64].

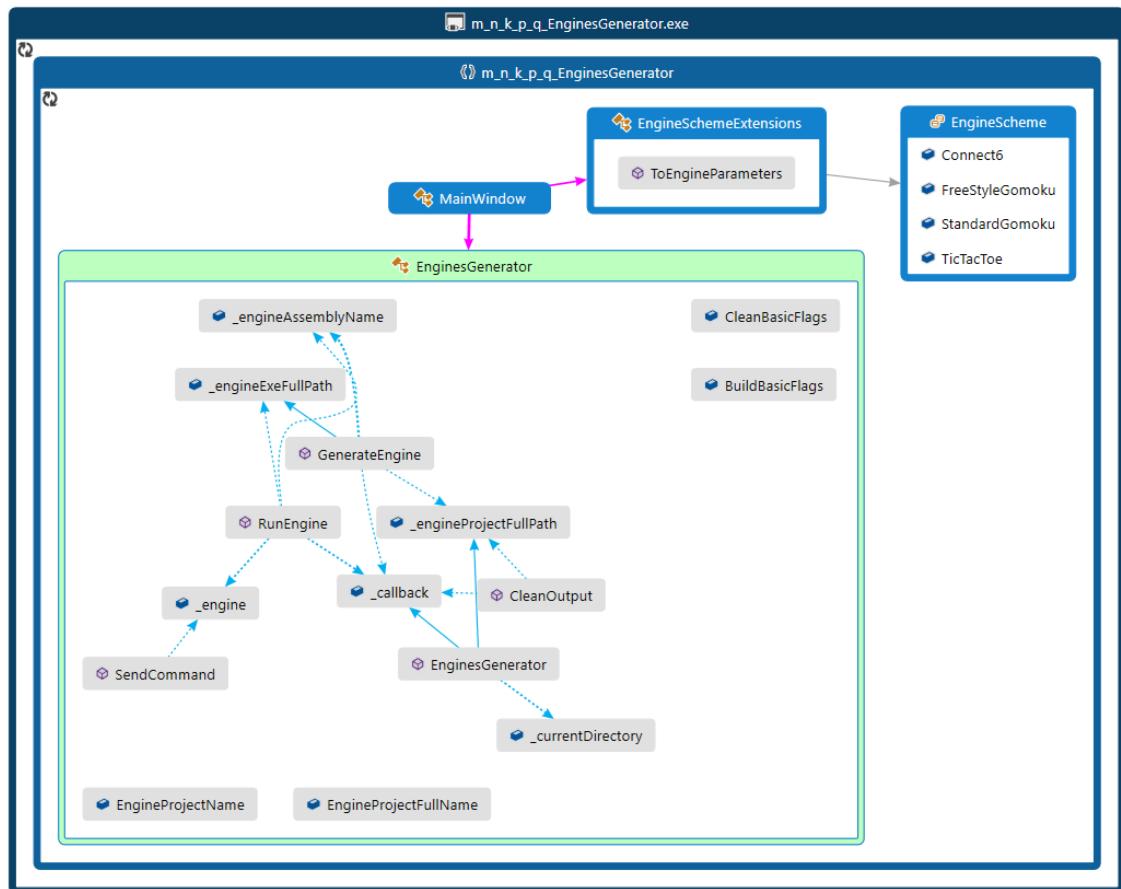
### 3.3 Generator silników – *(m,n,k,p,q)EnginesGenerator*

Najważniejszą aplikacją z całego projektu jest generator silników gier logicznych nazwany *(m,n,k,p,q)EnginesGenerator*. Napisany jest w C# 6.0 pod wersję 4.6.1 środowiska Microsoft .NET z graficznym interfejsem użytkownika zrealizowanym w XAML w WPF-ie. Odpowiada za generowanie silników gier logicznych w oparciu o kod silnika gry w C++ z makrami (opisany w następnym rozdziale) i wielkości zdefiniowane przez użytkownika w GUI. Poniżej zaprezentowano, jak wygląda program tuż po uruchomieniu.



Rysunek 3.3. Interfejs użytkownika aplikacji (m,n,k,p,q)EnginesGenerator

Architektura projektu rozdzielająca odpowiedzialności bardziej pomiędzy aplikacje aniżeli pomiędzy moduły jednej aplikacji spowodowała, że generator sam w sobie jest względnie prosty. Cała logika związana z interakcją z silnikami jest wyniesiona do biblioteki (m,n,k,p,q)EngineWrapper, a cała logika przełączania pomiędzy różnymi wersjami kodu w zależności od zdefiniowanych przez użytkownika w generatorze wartości jest zaimplementowana za pomocą makr w kodzie silników (m,n,k,p,q)GameEngine. Głównym zadaniem generatora jest wywoływanie kompilatora C++ na kodzie silników z odpowiednimi flagami. Dlatego też niżej zaprezentowany „code map” jest stosunkowo skromny, w dodatku usunięto z niego klasy mało znaczące.



Rysunek 3.4. "Code map" dla projektu `(m,n,k,p,q)EnginesGenerator`

Rozdzielenie logiki na kilka projektów pozwoliło dodatkowo uniknąć stosowania wzorców architektonicznych takich jak MVVM. W przyszłości wraz z rozwojem aplikacji nie powinno to być jednak specjalnie trudne – widok (View) jest napisany dosyć przemyślane w XAML-u, natomiast model (Model) jest spójny i w miarę prosty. Jedynym zadaniem na przyszłość pozostaje więc napisanie warstwy pośredniczącej (ViewModel), póki co jednak ze względu na rozmiary aplikacji autor uważa, że byłby to „overkill”.

### 3.3.1 Podstawowe struktury danych – `EngineParameters` i `EngineScheme`

Do modelowania parametrów generowanych silników gier (`m,n,k,p,q`), jak i definicji ich w interfejsie użytkownika niezbędne były przynajmniej dwie struktury. Zostały one zaprojektowane nie tylko jako pojemniki na dane (parametry), ale również tak, aby w przyszłości, gdy zostanie wprowadzony wzorzec MVVM, dobrze nadawały się do bindingu z warstwą pośrednią, która byłaby bezpośrednio zbindowana z widokiem. Pierwsza z nich z racji użyteczności nie tylko na potrzeby samego generatora – `EngineParameters` została opisana w rozdziale o bibliotece ułatwiającej współpracę z silnikiem – `EngineWrapper`. Druga natomiast występuje jedynie tutaj.

### 3.3.2 Schematy silników

Poza umożliwieniem użytkownikowi wygenerowania silnika z dowolnymi, zdefiniowanymi przez niego, parametrami warto było dać łatwe wczytanie gotowych parametrów dla znanych i lubianych gier. Ze wszystkich możliwych gier z rodziny (m,n,k,p,q) znane są przynajmniej cztery: Tic-Tac-Toe (polskie kółko i krzyżyk), standard Gomoku, freestyle Gomoku i Connect6. Postanowiono więc stworzyć enuma reprezentującego znane schematy silników.

```
public enum EngineScheme
{
    [Description("Tic Tac Toe")] TicTacToe, //3,3,3,1,1
    [Description("Connect6")] Connect6, //m,n,6,2,1
    [Description("Standard Gomoku")] StandardGomoku, //19,19,5,1,1, win
    exactly k
    [Description("Freestyle Gomoku")] FreeStyleGomoku //19,19,5,1,1, win k or
    more
}
```

Przy okazji wykorzystano atrybut Description, którego wartość jest potem wyświetlana w interfejsie użytkownika – jest to czytelniejsze niż sama nazwa zmiennej. Enumeracje domyślnie są pozbawione jakichkolwiek zachowań, więc reprezentują jedynie dane. Aby dodać do tego enuma logikę wykorzystano metody rozszerzające dostępne w C#. Rozszerzono go o jedną metodę pozwalającą na pozyskanie wartości parametrów ze schematu silnika.

```
public static EngineParameters ToEngineParameters(this EngineScheme
engineScheme)
{
    var engine = new EngineParameters();
    switch (engineScheme)
    {
        case EngineScheme.TicTacToe:
            engine.K = engine.M = engine.N = 3;
            engine.P = engine.Q = 1;
            break;
        case EngineScheme.Connect6:
            engine.K = 6;
            engine.M = engine.N = 15;
            engine.P = 2;
            engine.Q = 1;
            engine.WinCondition = WinCondition.K_OR_MORE_TO_WIN;

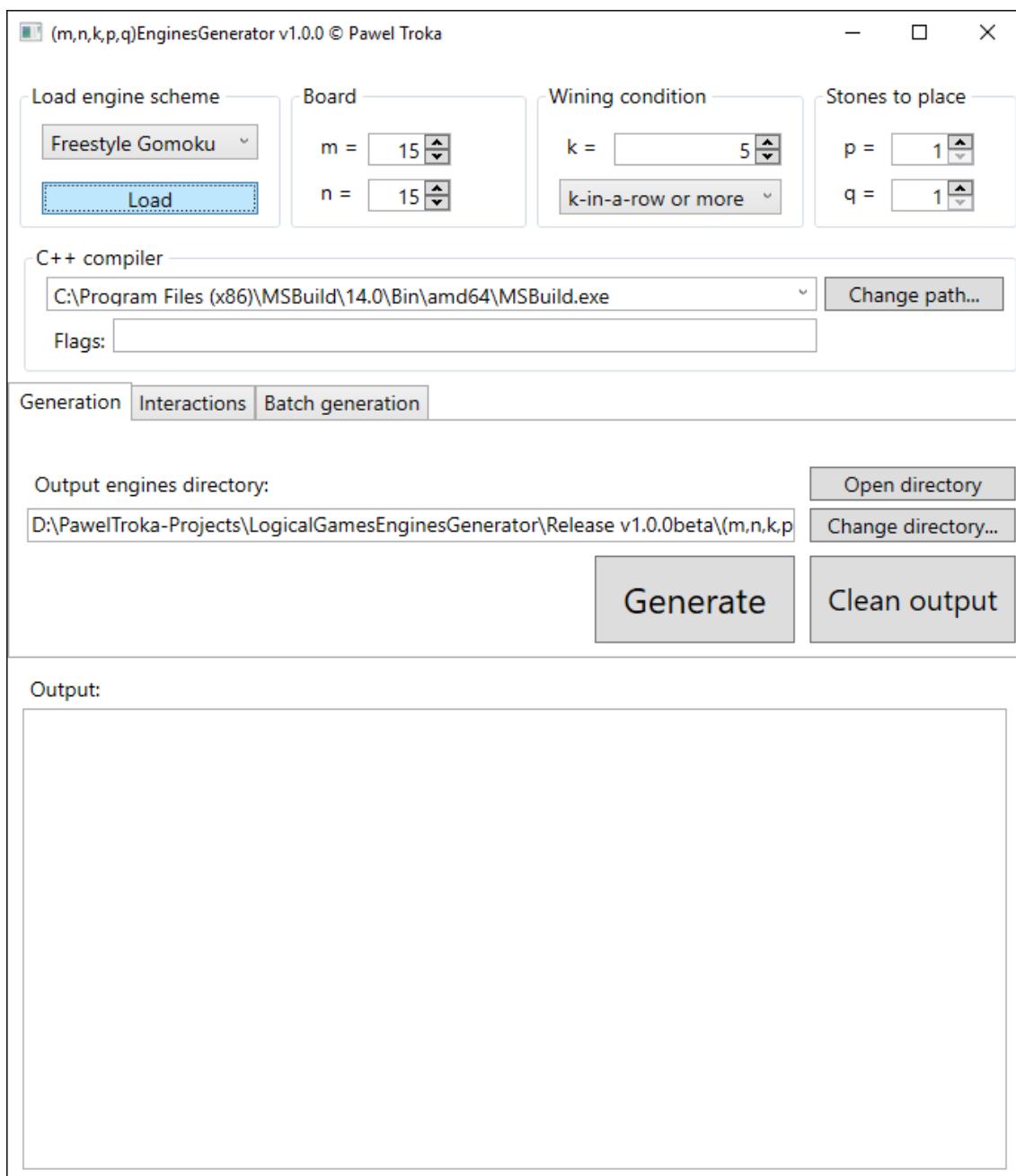
            break;
        case EngineScheme.StandardGomoku:
            engine.K = 5;
            engine.M = engine.N = 15;
            engine.P = engine.Q = 1;
            engine.WinCondition = WinCondition.EXACTLY_K_TO_WIN;
            break;
        case EngineScheme.FreeStyleGomoku:
            engine.K = 5;
            engine.M = engine.N = 15;
            engine.P = engine.Q = 1;
            engine.WinCondition = WinCondition.K_OR_MORE_TO_WIN;
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

```

    }
    return engine;
}

```

Metoda jest dosyć prosta – na podstawie konkretnej wartości schematu tworzy nowy obiekt klasy EngineParameters [klasa opisana w podrozdziale prezentującym bibliotekę (m,n,k,p,q)EngineWrapper] z odpowiednimi wartościami parametrów. Jak widać, w przypadku gry w kółko i krzyżyk nie musimy ustawać warunku wygranej, gdyżoba są równoznaczne – bo  $k=m=n$ . Standard i freestyle Gomoku z kolei różnią się tylko warunkiem wygranej. Schematy można łatwo ładować w interfejsie użytkownika, wybierając jeden z comboboxa i klikając „load”.



Rysunek 3.5. Załadowanie do interfejsu schematu silnika

Metoda obsługująca przycisk „load” po prostu ustawia uzyskane z wywołania metody rozszerzającej ToEngineParameters na wybranym schemacie silnika.

```
private void loadEngineSchemeButton_Click(object sender, RoutedEventArgs e)
{
    //load scheme
    var engineScheme = (EngineScheme) engineSchemeComboBox.SelectedIndex;
    var engineParameters = engineScheme.ToEngineParameters();

    kLongUpDown.Value = (long) engineParameters.K;
    mLongUpDown.Value = (long) engineParameters.M;
    nLongUpDown.Value = (long) engineParameters.N;
    pLongUpDown.Value = (long) engineParameters.P;
    qLongUpDown.Value = (long) engineParameters.Q;
    winingConditionComboBox.SelectedValue =
    engineParameters.WinCondition;
}
```

W przyszłości należy prawdopodobnie wraz z wprowadzeniem dodatkowej warstwy abstrakcji między modelem a widokiem przenieść tę prostą akcję do ViewModelu.

### 3.3.3 Generacja silników gier logicznych – klasa EnginesGenerator

Zakładając, że mamy zdefiniowane parametry do silnika (lub wczytaliśmy schemat) i poprawną ścieżkę do kompilatora C++ oraz ścieżkę, gdzie mają trafiać generowane silniki, możemy kliknąć przycisk „Generate”. Obsługę tego zdarzenia zaimplementowano z użyciem async i await, tak żeby nie mrozić interfejsu na czas generowania silnika.

```
private async void generateButton_Click(object sender, RoutedEventArgs e)
{
    generateButton.IsEnabled = false;

    var engineParameters = GetEngineParametersFromUI();
    var compilerPath = msbuildPathTextBox.Text;
    var outputDir = outputPathTextBox.Text;
    var flags = flagsTextBox.Text;

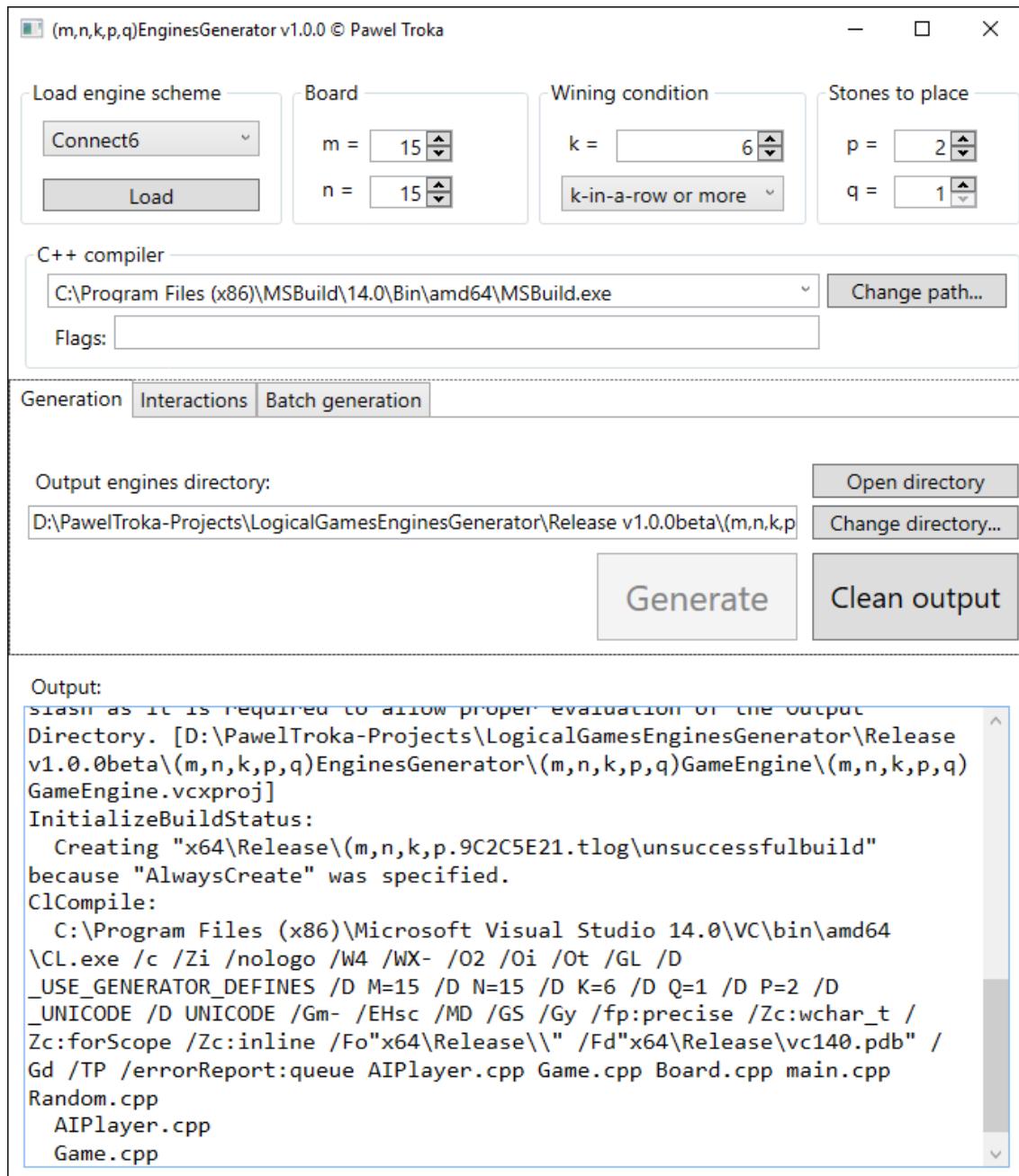
    await Task.Run(() => _generator.GenerateEngine(compilerPath,
outputDir, flags, engineParameters));

    generateButton.IsEnabled = true;
}
```

Jak widać, przycisk jest wyłączany, wartości są sczytywane z UI i uruchamiany jest awaitowany Task z wywołaniem metody GenerateEngine na obiekcie \_generator.

Poniżej zaprezentowano stan generatora w trakcie pracy po naciśnięciu przycisku „Generate”. Co istotne, cały „output” z wywoływanego kompilatora jest wyświetlany na bieżąco w interfejsie użytkownika – widać np. flagi przekazane do kompilatora. Szczególną uwagę należy zwrócić na takie wartości, jak „/D M=15” – jest to flaga kompilatora wstrzyknięcia przez generator w procesie wywołania kompilatora w metodzie GenerateEngine. Ma ona na celu powiedzieć kompilatorowi, że ma zdefiniować globalnie dla komplowanego kodu stałą M o wartości 15. Jest

to oczywiście wartość m z interfejsu użytkownika, oznaczająca jeden z wymiarów planszy (ilość wierszy planszy). Szczegóły implementacyjne i techniczne tego procesu są opisane dalej.



Rysunek 3.6. Generator w trakcie pracy

Aby dobrze zrozumieć działanie generatora, trzeba przede wszystkim zbadać metodę `GenerateEngine`.

```
public void GenerateEngine(string compilerPath, string outputDir, string flags,
                           EngineParameters engineParameters)
{
    if (outputDir.Last() == '\\')
        outputDir = outputDir.Substring(0, outputDir.Length - 1);

    _engineAssemblyName = engineParameters.ToString();
```

```

    _engineExeFullPath = Path.Combine(outputDir, _engineAssemblyName);

    var engineParametersAsCompilerFlags =
        $@"
/p:AdditionalPreprocessorDefinitions=""_USE_GENERATOR_DEFINES;{({engineParameters.
WinCondition ==

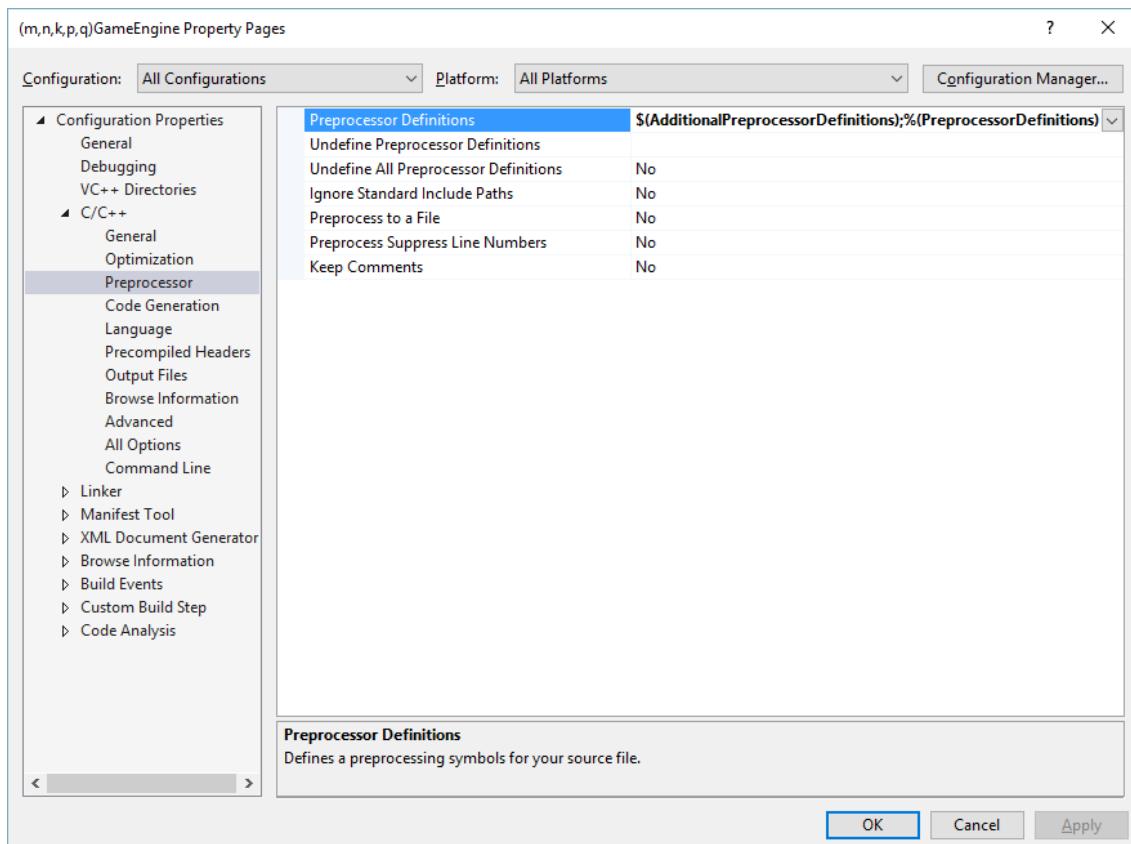
WinCondition.EXACTLY_K_TO_WIN
    ? "EXACTLY_K_TO_WIN;""
    :
""})}M={engineParameters.M};N={engineParameters.N};K={engineParameters.K};Q={engine
eParameters.Q};P={engineParameters
    .P};"" /p:OutDir=""{outputDir}"""
/p:AssemblyName=""{_engineAssemblyName}"";

    new ProcessInBackground(compilerPath,
        $"\"+_engineProjectFullPath+ "\" + BuildBasicFlags +
engineParametersAsCompilerFlags + flags, _callback, false)
    .Run();
}

```

Najpierw usuwamy ostatni slash z lokalizacji, gdzie mają trafić generowane silniki (pozostawienie go powodowało błędy). Potem ustalamy nazwę assembly generowanego silnika – jest to po prostu wywołanie metody ToString na parametrach silnika. Ścieżka do silnika to oczywiście połączenie tych dwóch.

Kolejny krok jest kluczowy w całym procesie generowania silników gier logicznych przyjętą w niniejszej pracy metodą. Zamieniamy wartości parametrów silnika na flagi kompilatora. Pod switchem /p:AdditionalPreprocessorDefinitions (gdzie \$AdditionalPreprocessorDefinitions zdefiniowaliśmy jako zmienną preprocesora w projekcie VC++ (m,n,k,p,q)GameEngine – o czym w dalszym podrozdziale) umieszczamy kolejno wartości parametrów silnika w formacie „NAZWA=WARTOŚĆ”. Na początek jednak definiujemy flagę \_USE\_GENERATOR\_DEFINES, która ma na celu użycie nowych – zdefiniowanych tutaj wartości parametrów. Definiujemy też stałą EXACTLY\_K\_TO\_WIN, jeżeli silnik ma WinCondition o takiej wartości.



Rysunek 3.7. Dodatkowa zmienna preprocesora \$(AdditionalPreprocessorDefinitions)

Pod switchem /p:OutDir przekazujemy również zdefiniowaną w interfejsie lokalizację dla generowanych silników, nazwa silnika trafia pod switch /p:AssemblyName – nie trzeba dopisywać rozszerzenia – MSBuild na podstawie projektu sam wie, jakie powinno być rozszerzenie.

Wyołujemy proces w tle (klasa opisana w podrozdziale dotyczącym wrappera silników). Oprócz flag kompilatora utworzonych z parametrów silnika przekazujemy również opcjonalnie zdefiniowane przez użytkownika flagi oraz lokalizację projektu.

Lokalizacja, gdzie znajduje się projekt, to z założenia folder `(m,n,k,p,q)GameEngine` w lokalizacji pliku wykonywalnego generatora. Po prostu kod silnika gry w C++ z makrami traktujemy w pewnym sensie jak dane / content /, treść, na której pracuje generator przy tworzeniu kolejnych silników. W poniższym fragmencie kodu widać bardzo dobrze, jak tworzona jest lokalizacja, w której z założenia powinien znajdować się projekt z silnikiem C++ gry.

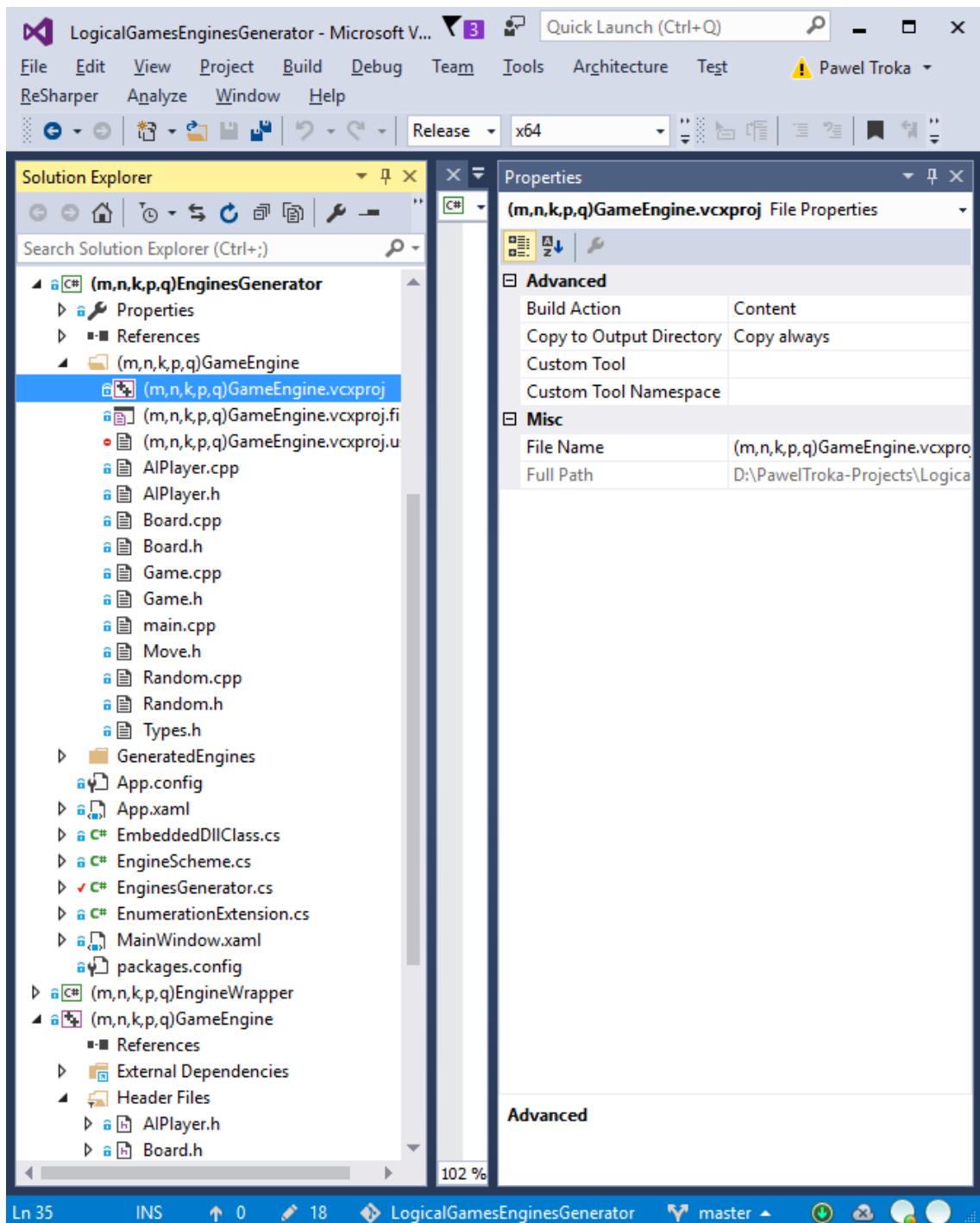
```

private const string EngineProjectName = "(m,n,k,p,q)GameEngine";
private const string EngineProjectFullName = EngineProjectName +
".vcxproj";
private readonly string _currentDirectory =
    Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
private readonly string _engineProjectFullPath;

public EnginesGenerator(Action<string> callback)
{
    _callback = callback;
    _engineProjectFullPath = Path.Combine(_currentDirectory,
EngineProjectName,
```

```
        EngineProjectFullName);  
    }
```

Dzięki mechanizmowi refleksji aplikacja – generator dowiaduje się, w jakiej lokalizacji się znajduje, i w konstruktorze ustala, że ścieżka do projektu silnika to lokalizacja generatora + (m,n,k,p,q)GameEngine + (m,n,k,p,q)GameEngine.vcxproj. W związku z takim podejściem kod silnika jest jednocześnie częścią procesu developmentu (ponieważ trzeba go programować, bo się zmienia) i treścią (gdyż na jego podstawie generator tworzy nowe silniki). Stąd, jak widać na poniższym obrazku, pliki projektowe (m,n,k,p,q)GameEngine są załączone do solucji zarówno jako oddzielny projekt C++, jak i content dla projektu generatora. Takie rozwiązanie generuje bardzo niewiele problemów, a sprawdza się bardzo dobrze i jest wygodne. Jedynym wymogiem jest dodawanie nowo powstałych plików z projektu C++ dodatkowo jako content do projektu generatora.



Rysunek 3.8. Pliki silników załączone zarówno jako oddzielny projekt, jak i content dla generatora

Do argumentów wysyłanych do procesu MSBuild kierowane są również flagi związane z budową lub wyczyszczeniem projektu (zależnie od przycisku wybranego przez użytkownika). Konfiguracją zawsze jest release – ze względu na potrzebę jak najwyższej wydajności generowanych silników. Również ze względów wydajnościowych, ale także z powodu trzymania dla pewnych rozmiarów planszy w zmiennej 64bitowej, tryb komplikacji jest też 64bitowy.

```
private const string BuildBasicFlags = @" /p:Configuration=Release  
/p:Platform=x64 /t:Build ";
```

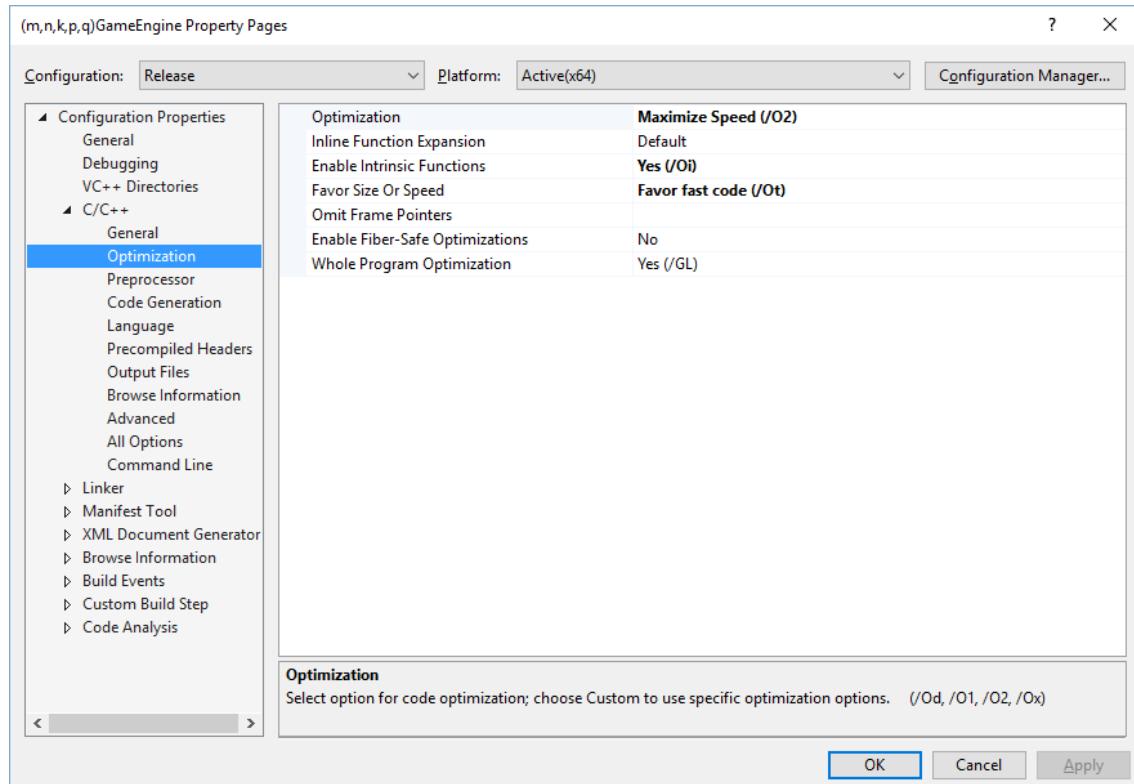
```

private const string CleanBasicFlags = @" /p:Configuration=Release
/p:Platform=x64 /t:Clean ";

```

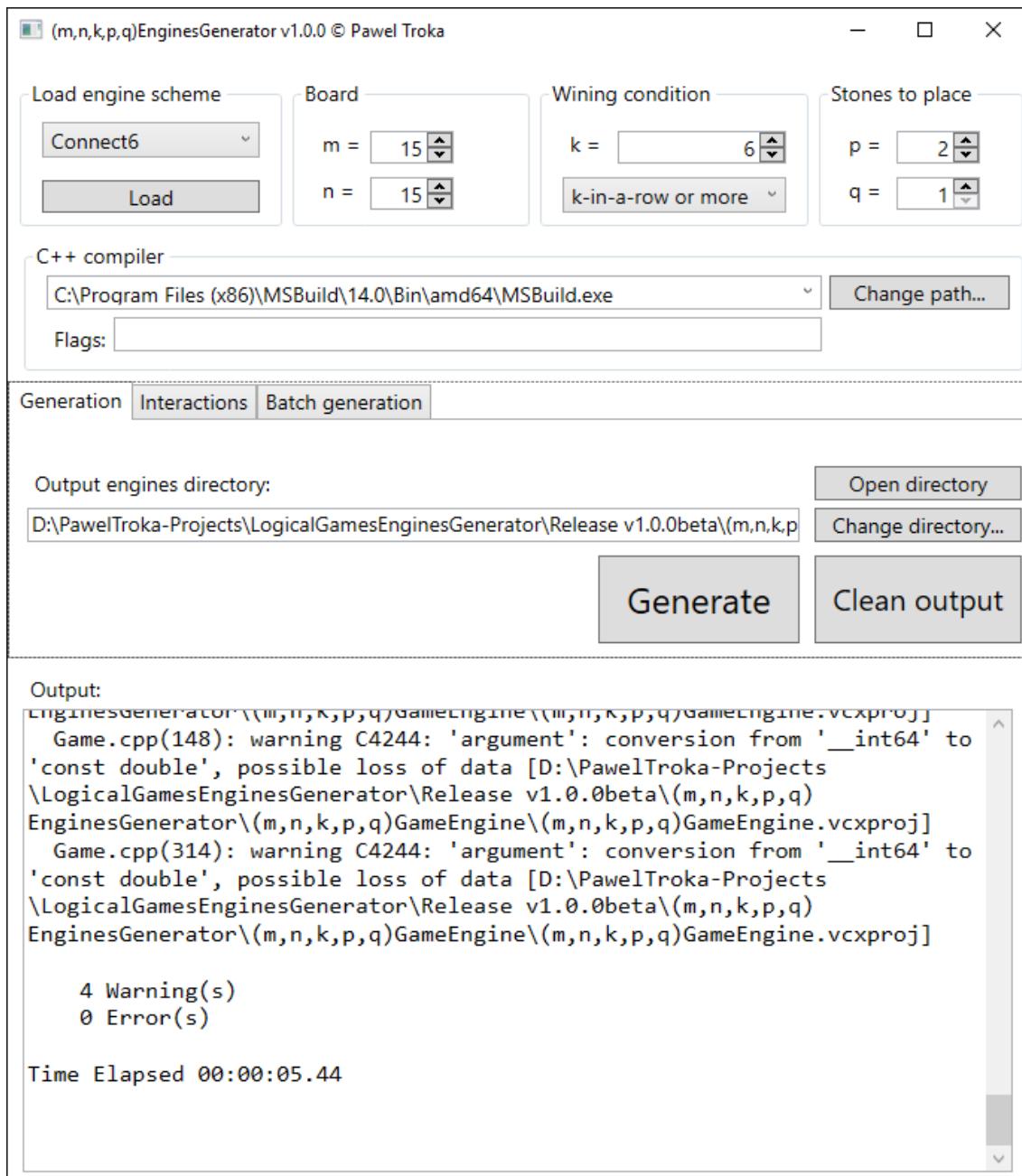
Ostatecznie w procesie generacji proces w tle reprezentujący MSBuild jest uruchamiany, a jego wyjście jest przekierowywane na akcję \_callback, która pisze do interfejsu.

Dla zwiększenia wydajności generowanego kodu ustawiono również liczne optymalizacje kompilatora w samych opcjach projektu.



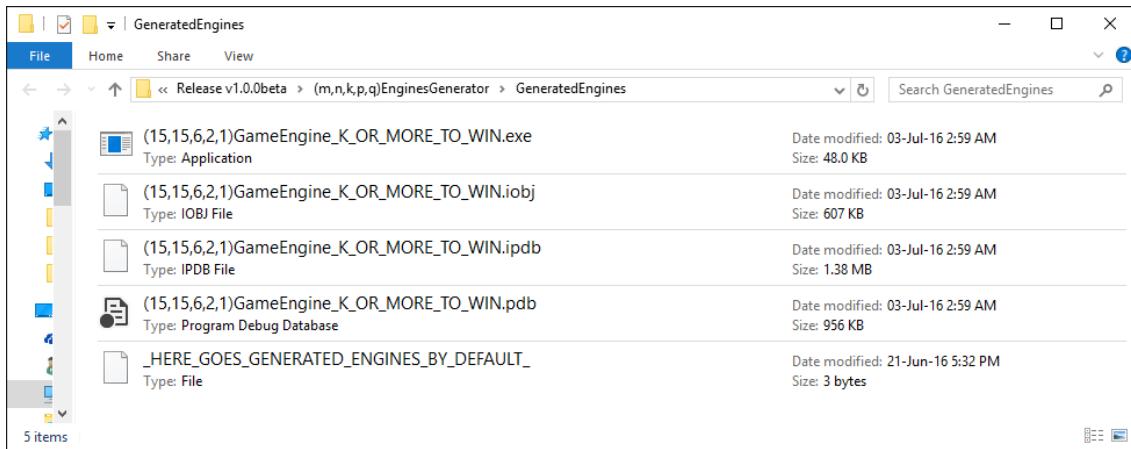
Rysunek 3.9. Optymalizacje kompilatora zastosowane w projekcie

Po wykonaniu procesu w bloku „Output” interfejsu jest całe wyjście z kompilatora, w tym czas komplikacji.



Rysunek 3.10. Generacja silnika zakończona

Po udanej generacji można również otworzyć lokalizację, gdzie trafił silnik, przy użyciu przycisku „Open directory”. Naszym oczom powinien się ukazać nowopowstały plik exe reprezentujący aplikację konsolową – skompilowany silnik gry dla konkretnych parametrów. Przykładową zawartość tej lokalizacji zaprezentowano na rysunku poniżej.

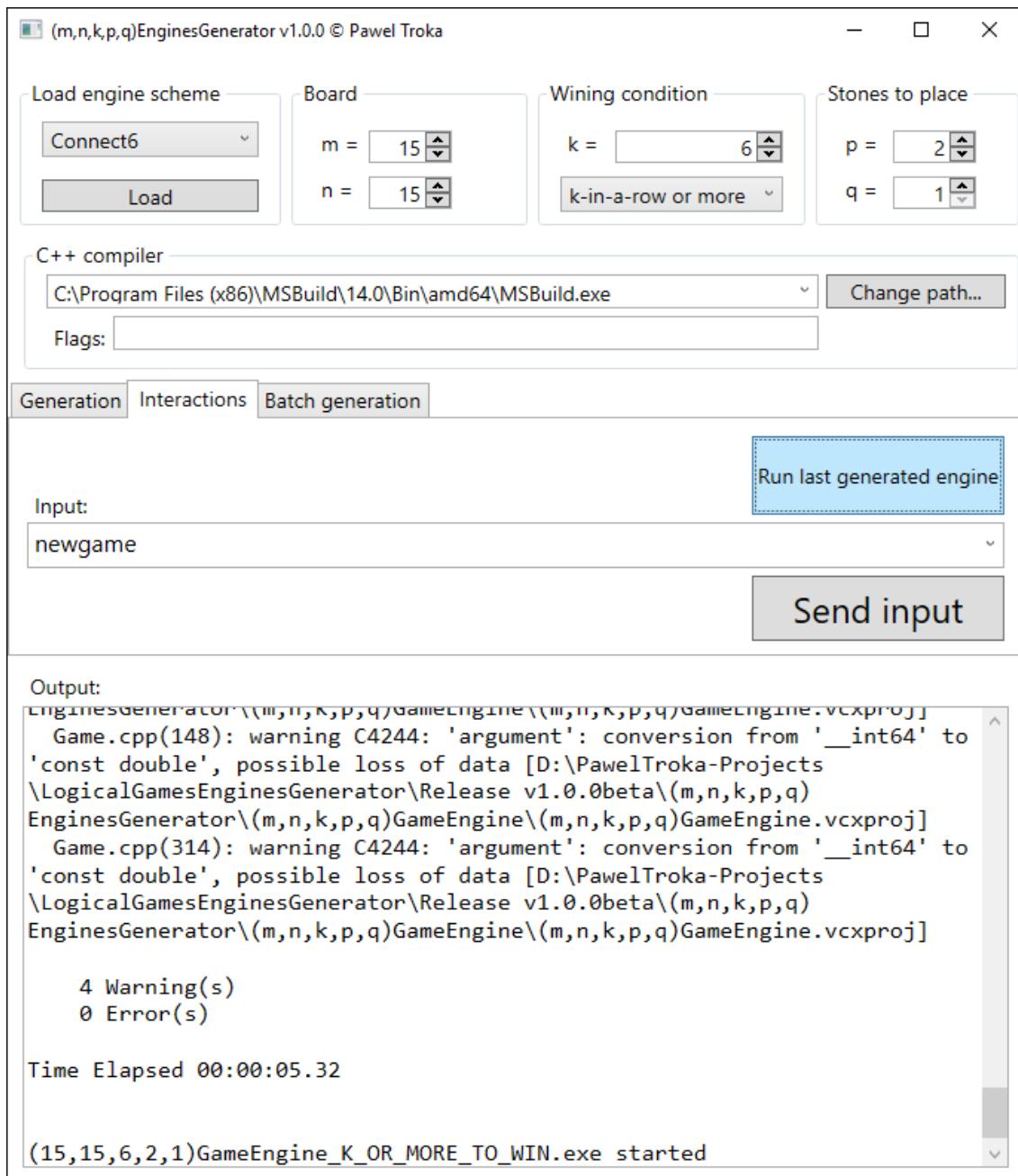


Rysunek 3.11. Lokalizacja z wygenerowanymi silnikami

Wygenerowane silniki można uruchomić – są to przecież aplikacje konsolowe. Oczywiście gra w taki sposób nie będzie zbyt wygodna, ale może być rozegrana chociażby w celu testów.

#### 3.3.4 Proste interakcje z wygenerowanymi silnikami

W dalszej części (po wygenerowaniu przynajmniej jednego silnika) aplikacja-generator umożliwia również prostą interakcję z nowym silnikiem. Interakcja ta różni się od współpracy programu testującego lub GUI z wygenerowanymi silnikami tym, że z założenia jest maksymalnie prosta. Oznacza to, że nie jest używana klasa EngineWrapper z biblioteki (m,n,k,p,q)EngineWrapper opisana później, ale zwykły proces w tle ProcessInBackground (również opisany później). Uzasadnienie prostoty tej interakcji jest takie, że interakcje z wygenerowanymi silnikami, to właściwie dodatkowy feature generatora, mający na celu jedynie sprawdzenie, czy silniki zostały wygenerowane prawidłowo. Na rysunku poniżej przedstawiono początek takiej interakcji – kliknięcie „Run last generated engines”, które powoduje uruchomienie procesu silnika w tle i wypisanie w sekcji „Output” informacji o tym.



Rysunek 3.12. Uruchomienie ostatnio wygenerowanego silnika

W kodzie źródłowym klasy EnginesGenerator uruchomienie silnika to względnie prosty proces.

```
public void RunEngine()
{
    _engine = new ProcessInBackground(_engineExeFullPath, "", _callback,
true);
    _engine.Run();
    _callback.Invoke(${@"{_engineAssemblyName}.exe started"});
}
```

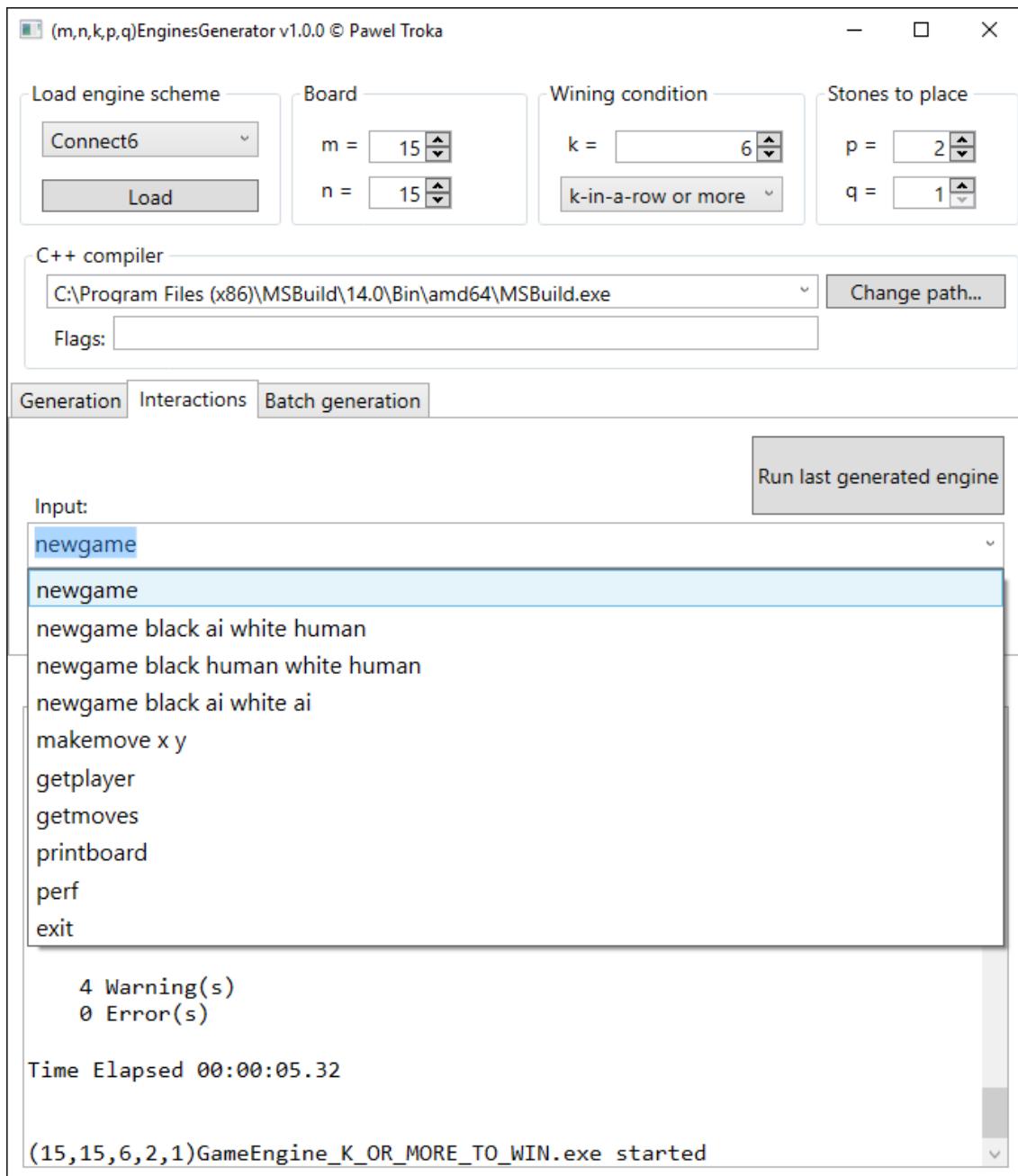
Ścieżka do pliku wykonywalnego silnika została uzyskana w procesie generacji, `_callback` to delegat `Action<string>` piszący do interfejsu. Ze względu na użycie `async await`, asynchroniczność wszelkich dłuższych zadań (takich jak generacja), zdefiniowany `_callback`, musi mieć możliwość pisania do interfejsu w wątku głównym (tylko main thread może

modyfikować UI). Dlatego też w konstruktorze MainWindow (głównego okna generatora) wstrzykiwany jest asynchroniczny delegat używający Dispatchera do konstruktora klasy EnginesGenerator.

```
_generator =  
    new EnginesGenerator(  
        async s => await Dispatcher.InvokeAsync(() =>  
            outputTextBox.AppendText(s + Environment.NewLine)));
```

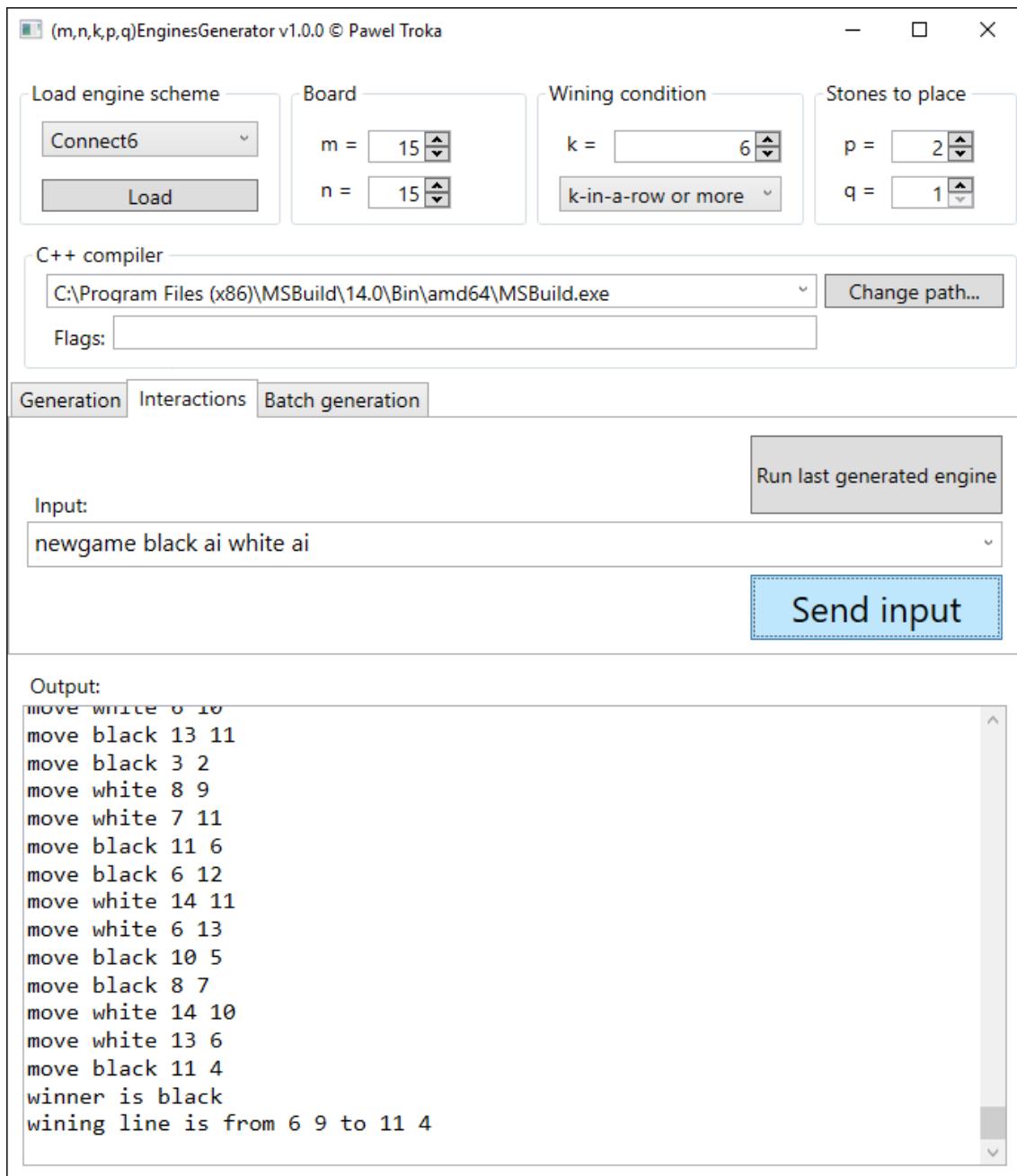
Konstruktor generator został pokazany w poprzednim podrozdziale.

Kolejnym krokiem w interakcji z wygenerowanym silnikiem jest wysłanie komendy. Jako że gra się jeszcze nie rozpoczęła, nie wszystkie polecenia mają sens – największy ma rozpoczęcie gry. Większość dostępnych komend jest dostępna w podpowiedzi, jak pokazano poniżej.



Rysunek 3.13. Podpowiedzi komend do wysłania do silnika

Po wybraniu lub napisaniu interesującej użytkownika komendy można wybrać przycisk „Send input”, który spowoduje jej wysłanie do uruchomionego w tle silnika, a wszelkie odpowiedzi silnika są asynchronicznie obsługiwane poprzez wywołanie delegata `_callback` opisanego wcześniej.



Rysunek 3.14. Wysłanie komendy newgame black ai white ai do silnika

Na rysunku powyżej pokazano zachowanie się silnika dla komendy „newgame black ai white ai”. Wybrano ją, gdyż powoduje automatyczne rozpoczęcie i w skończonym czasie zakończenie gry – komputer kontra komputer, więc nie wymaga dalszej interakcji ze strony użytkownika. Obsługa kliknięcia jest bardzo prosta – jest to wywołanie na silniku uruchomionym jako ProcessInBackground metody Send z argumentem będącym równym wpisanemu łańcuchowi znaków w sekcji „Input” interfejsu.

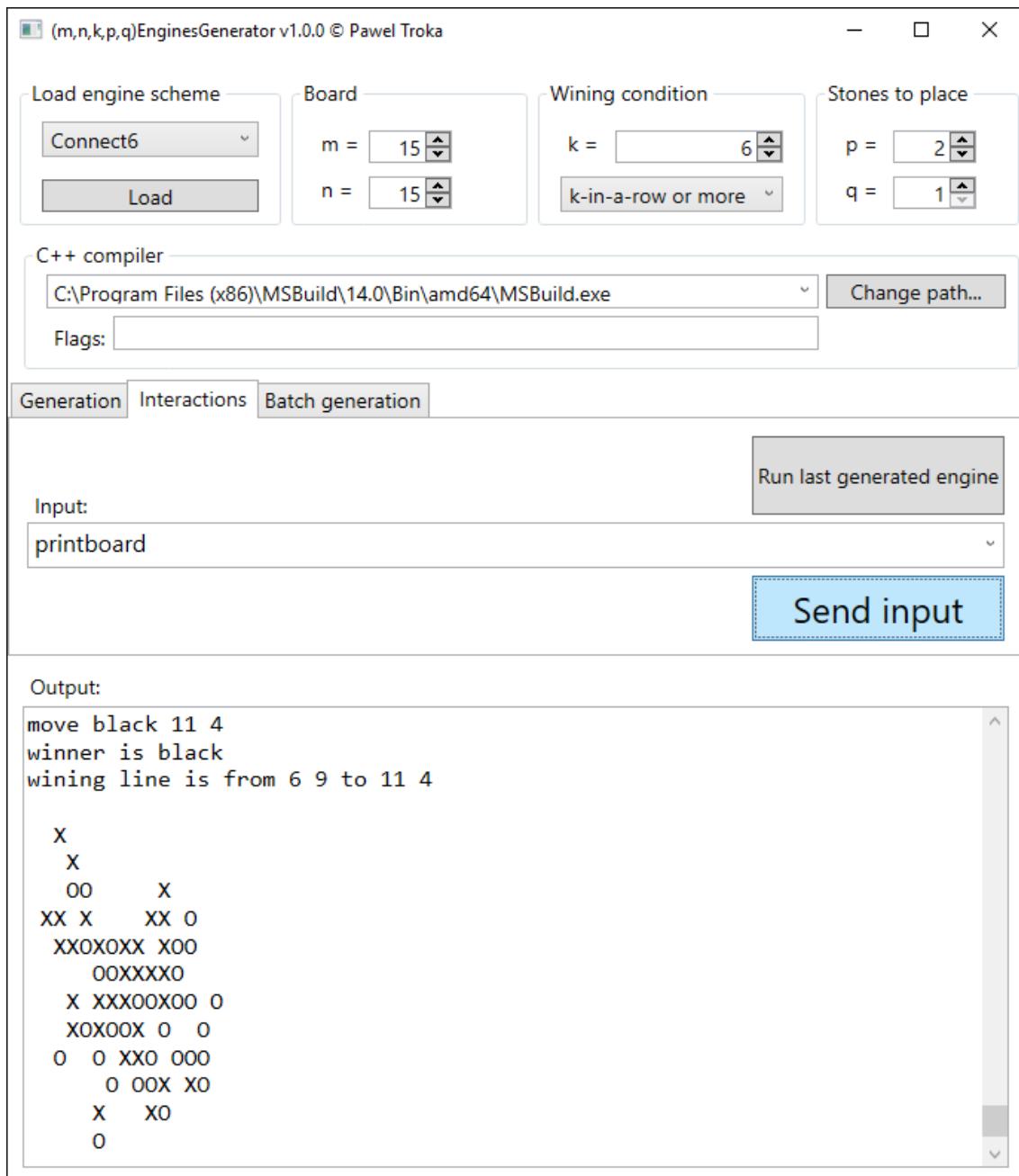
```
public void SendCommand(string cmd)
{
    _engine?.Send(cmd);
}
```

Dodatkowo zabezpieczamy się przed sytuacją, gdyby zmienna \_engine była nullem (np. jeżeli użytkownik nie wygenerowałby żadnego silnika), używając operatora propagacji nulli „?”. Oznacza to, że Send wykona się na \_engine, o ile \_engine nie jest nullem, jednak jeżeli jest, to nie zostanie rzucony wyjątek, tylko nic się nie stanie. W środku ProcessInBackground jest tak naprawdę zwykła klasa Process z .NET, wywołanie metody Send jest więc wywołaniem pisania do standardowego wejścia procesu.

```
public void Send(string cmd)
{
    _process.StandardInput.WriteLine(cmd);
}
```

Jak widać, pisana jest nowa linia na standardowe wejście, przekazująca łańcuch znaków będący zdefiniowaną komendą. Po raz kolejny pokazano prostotę interakcji z poziomu generatora z wygenerowanymi silnikami.

Inne komendy właściwie robią to, czego można by się spodziewać. Tak naprawdę znając implementację lub specyfikację silnika, możemy bez problemu się z nim w ten sposób komunikować. Co nie oznacza, że jest to optymalny sposób na granie czy testowanie – dlatego powstała biblioteka (m,n,k,p,q)EnginesWrapper (opisana w dalszej części). Z drugiej jednak strony, taki protokół tekstowy jest chyba najbardziej multiplatformowym i technologicznie agnostycznym rozwiązaniem.



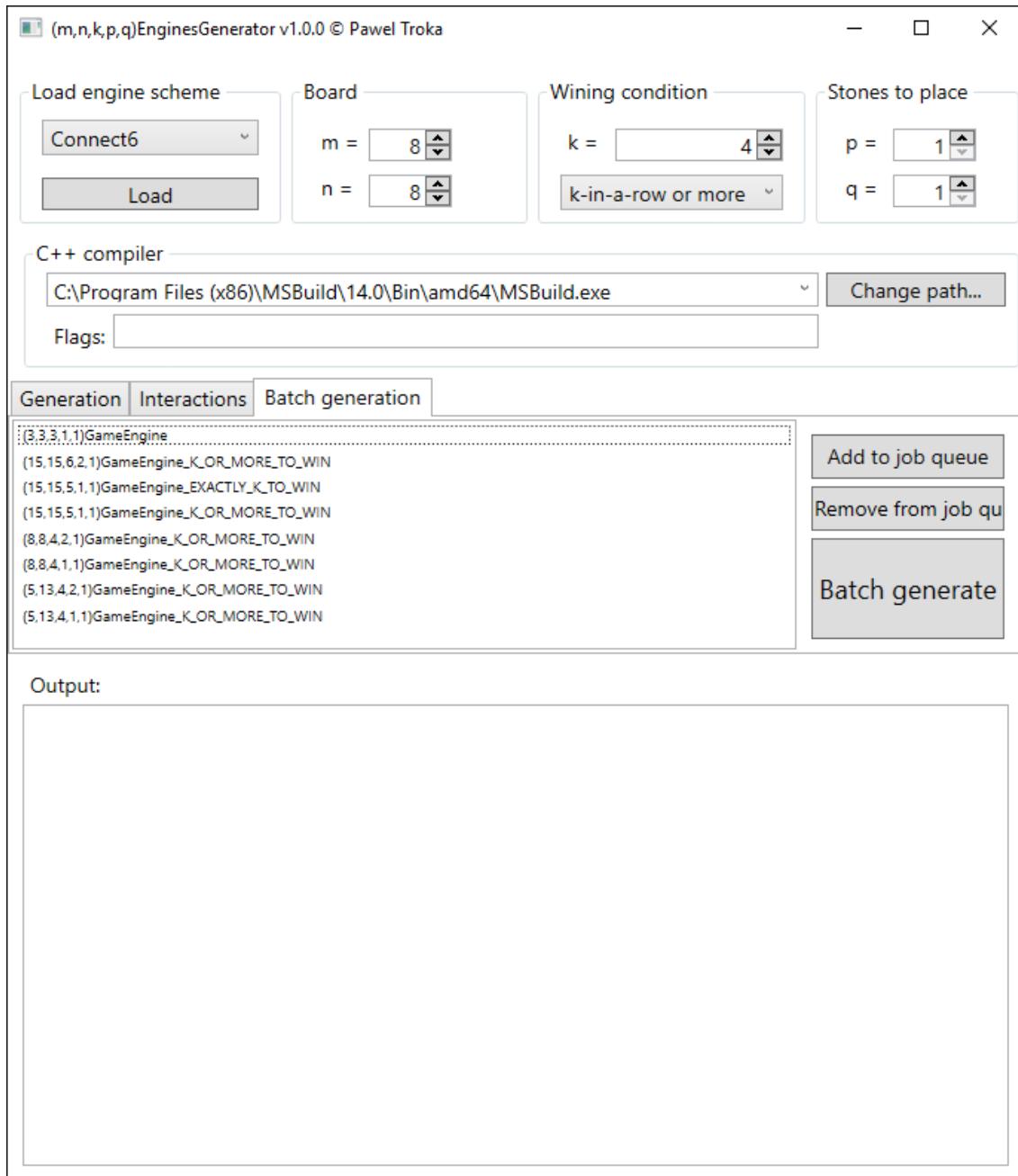
Rysunek 3.15. Wywołanie komendy "printboard" po zakończonej rozgrywce

W związku z tym, że kluczowa w całej interakcji jest klasa `ProcessInBackground`, została ona obszernie opisana w rozdziale poświęconym bibliotece `(m,n,k,p,q)EngineWrapper`.

### 3.3.5 Tryb „batch”

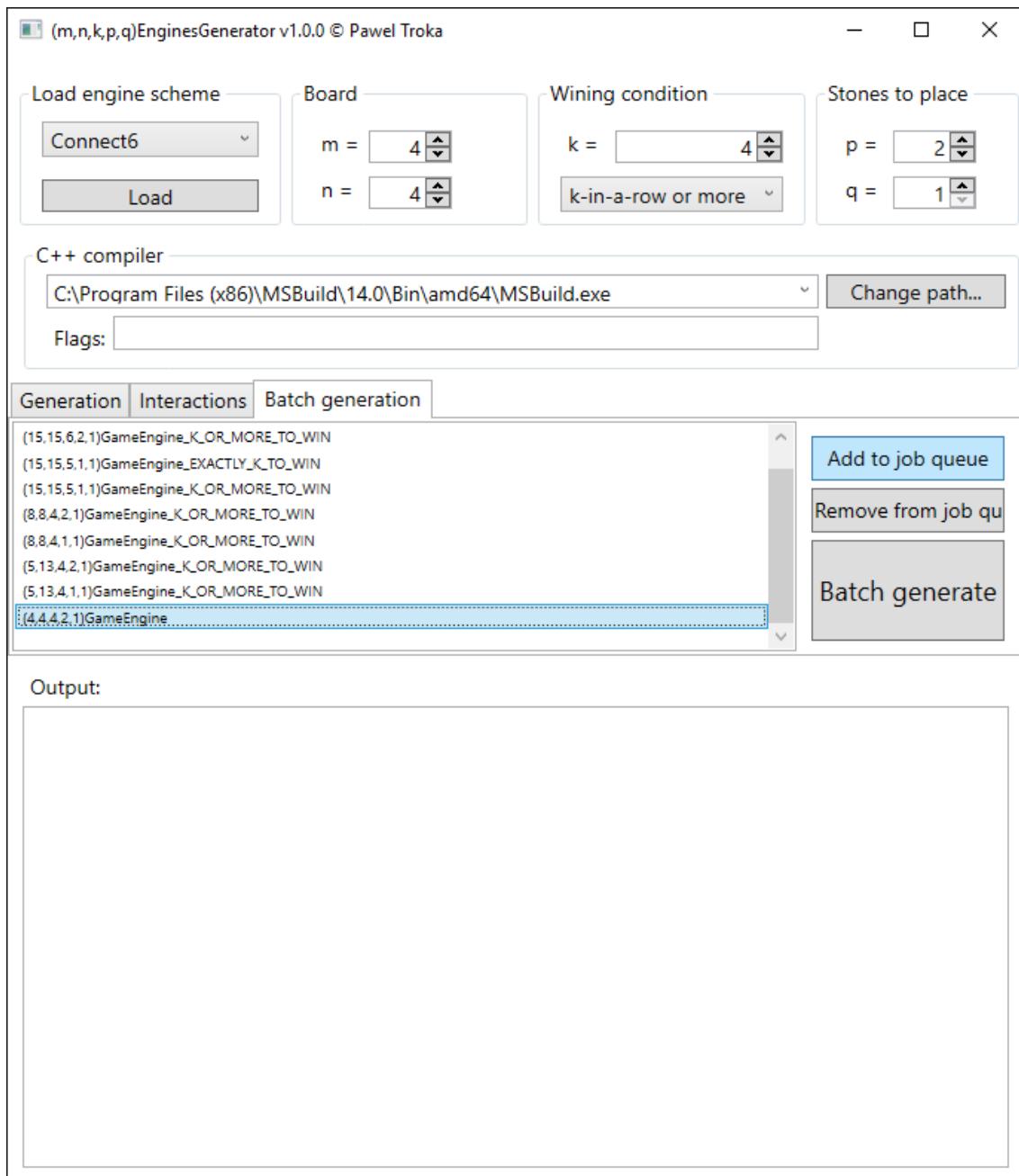
W trakcie rozwoju aplikacji – wraz ze zmieniającym się kodem silnika i dla potrzeb testów – powstała potrzeba, aby łatwo generować kilka silników jednocześnie. Wygenerowanie jednego silnika zajmuje około 5 do 8 sekund. W normalnym trybie użytkownik musiał każdorazowo oczekwać ten czas, aby zdefiniować nowe parametry i wygenerować kolejny silnik. Dla ośmiu silników cały proces trwał często blisko dwie minuty pracy, w trakcie których generalnie nie można było robić nic innego.

Rozwiązańiem wspomnianego problemu jest tryb „Batch generation”, w którym najpierw definiuje się, jakie silniki chce się wygenerować, a potem jednym przyciskiem uruchamia procesy generowania wszystkich zdefiniowanych silników. Na rysunku poniżej zaprezentowano widok w trybie „Batch generation” wraz z domyślnie zdefiniowaną listą silników do wygenerowania.



Rysunek 3.16. Tryb "Batch generation" generatora

Zadanie użytkownika sprowadza się tutaj do definiowania parametrów dla kolejnego silnika, który chce wygenerować, i dodania tego do listy zadań. Przykładowo poniżej pokazano sytuację po dodaniu silnika z parametrami (4,4,4,2,1).



Rysunek 3.17. Dodanie nowego silnika do kolejki zadań generatora

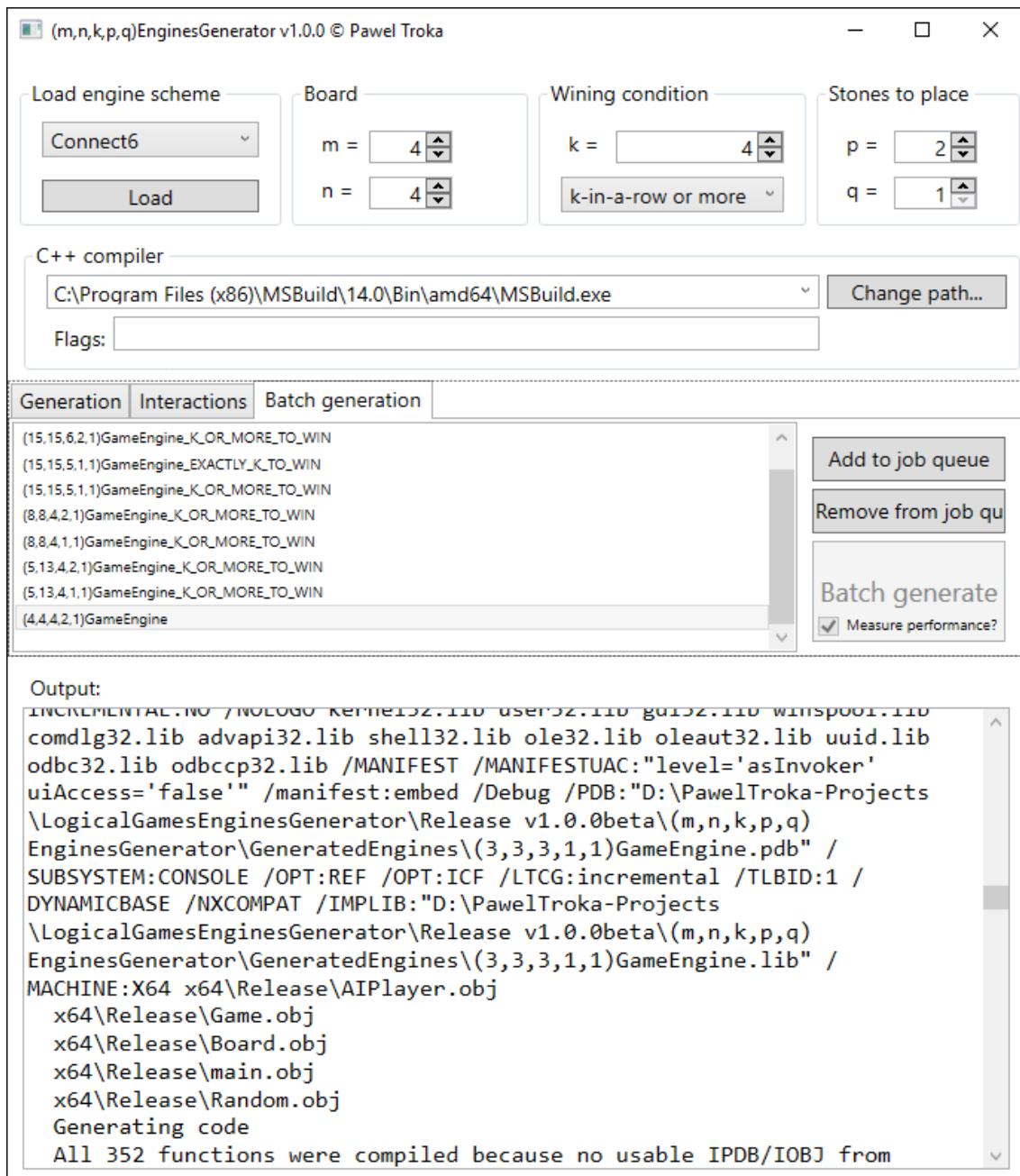
W kodzie źródłowym MainWindow operacja jest bardzo prosta – sprowadza się do sczytania wartości parametrów z UI, stworzenia na ich podstawie obiektu klasy EngineParameters i dodania go do kontrolki ListBox. Obiekty klasy EngineParameters są reprezentowane w ListBoxie jako swoje wywołania metody ToString() – domyślne zachowanie w WPF i właściwie w całym frameworku .NET.

```
private void AddToBatchGenerationButton_OnClick(object sender,
RoutedEventArgs e)
{
    var engineParameters = GetEngineParametersFromUI();
    batchGenerationListBox.Items.Add(engineParameters);
}
```

W tym kodzie metoda GetEngineParametersFromUI robi dokładnie to, czego można by się spodziewać.

```
private EngineParameters GetEngineParametersFromUI()
{
    var engineParameters = new EngineParameters
    {
        K = (ulong) kLongUpDown.Value,
        M = (ulong) mLongUpDown.Value,
        N = (ulong) nLongUpDown.Value,
        P = (ulong) pLongUpDown.Value,
        Q = (ulong) qLongUpDown.Value,
        WinCondition = (WinCondition)
    };
    engineParameters.winingConditionComboBox.SelectedIndex
    return engineParameters;
}
```

Po zdefiniowaniu parametrów wszystkich silników, które użytkownik chce wygenerować, i dodaniu ich do listy zadań pozostaje kliknąć przycisk „Batch generate”. Ogromną zaletą tutaj jest to, że w czasie, gdy silniki się generują, użytkownik może robić coś innego, wiedząc, że nie musi ingerować w cały proces. Poniżej pokazano, jak wygląda interfejs w trakcie generacji silników.



Rysunek 3.18. Program w trakcie pracy w trybie "batch"

Należy jeszcze zauważyć, że przed kliknięciem przycisku „Batch generate” wybrano opcję „Measure performance?”. Jako że jednym z celów pracy jest zbadanie wydajności generatorów silników gier logicznych, postanowiono w trybie batch udostępnić opcję pomiaru czasów generacji wszystkich silników. Opcja ta mierzy czas generacji dla każdego silnika. Cały proces jest pokazany poniżej w kodzie obsługi kliknięcia przycisku „Batch generate”.

```
private async void BatchGenerateButton_OnClick(object sender,
RoutedEventArgs e)
{
    batchGenerateButton.IsEnabled = measurePerformanceCheckBox.IsEnabled
= false;
```

```

        var compilerPath = msbuildPathTextBox.Text;
        var outputDir = outputPathTextBox.Text;
        var flags = flagsTextBox.Text;
        var engineParameters =
batchGenerationListBox.Items.Cast<EngineParameters>();
        var measurePerformance = measurePerformanceCheckBox.IsChecked.Value;

        var performanceDict = new Dictionary<EngineParameters, double>();

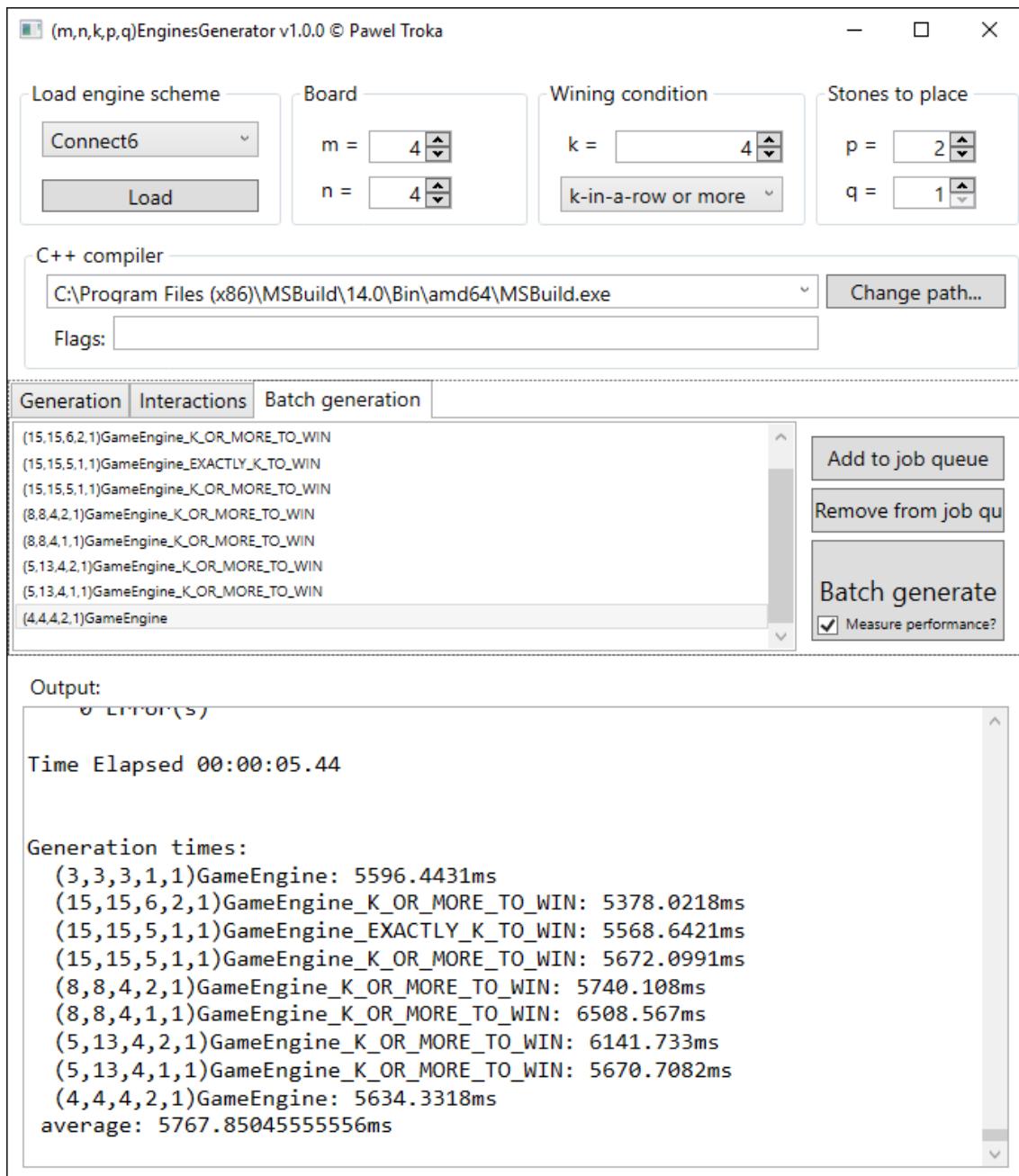
        await Task.Run(() =>
{
    foreach (var engine in engineParameters)
        if (measurePerformance)
    {
        _generator.CleanOutput(compilerPath);
        var stw = Stopwatch.StartNew();
        _generator.GenerateEngine(compilerPath, outputDir, flags,
engine);
        stw.Stop();
        performanceDict.Add(engine,
stw.Elapsed.TotalMilliseconds);
    }
    else
        _generator.GenerateEngine(compilerPath, outputDir, flags,
engine);

        if (!measurePerformance) return;
        _callback("Generation times:");
        foreach (var key in performanceDict.Keys)
            _callback($" {key}: {performanceDict[key]}ms");
        _callback($" average: {performanceDict.Values.Average()}ms");
    });
}

batchGenerateButton.IsEnabled = measurePerformanceCheckBox.IsEnabled
= true;
}

```

Jest to więc pobranie danych z interfejsu użytkownika i uruchomienie w awaitowanym Tasku pętli z wywołaniami metody GenerateEngine dla listy silników do wygenerowania. Jedyne różnice są takie, że jeżeli zaznaczono wcześniej opcję dla pomiaru wydajności – „measure performance?”, to przed wywołaniem metody wykonywany jest task „clean” procesu MSBuild na projekcie oraz uruchamiany jest stoper. Wyczyszczenie wyników komplilacji projektu przed kolejną komplikacją jest niezbędne, aby poprawnie zmierzyć prawdziwą wydajność generatora – w innym wypadku kompilator mógłby użyć części skompilowanych funkcji z poprzedniej komplikacji. Wartości pomiaru czasu wykonania metody GenerateEngine są wpisywane do słownika, w którym kluczami są parametry generowanych silników. Na koniec, jeżeli wybrano taką opcję, wyświetlane są informacje o wydajności. Efekt pokazano na poniższym rysunku.



Rysunek 3.19. Stan programu po wygenerowaniu silników w trybie "batch" wraz z informacją o wydajności generatora

Wydajność procesu generowania silników to oczywiście głównie wydajność kompilatora C++ i zależy od bardzo wielu czynników, nawet takich, jak losowe spowolnienia dysku czy zdarzenia systemu. Niemniej jednak widać, że jest to zazwyczaj 5-6 sekund [znacznie szybciej, gdy zezwala się na użycie części skompilowanego kodu z poprzedniej komplikacji (bez measure performance)].

### 3.4 Silnik gry w C++ z makrami – $(m,n,k,p,q)GameEngine$

Projekt silnika gry w C++ to kluczowy fragment pracy autora, jeżeli chodzi o wydajność. Dla przyjętych założeń projektowych, w których generator właściwie definiuje tylko pewne stałe i wykonuje komplikację silnika, kluczowy jest jak najlepiej opisany, jak najbardziej optymalny kod w

tym projekcie. Jest to wyzwanie również z tego powodu, że autor nie posiada dużego doświadczenia z językiem C++, gdyż zdecydowanie jest fanem C# i środowiska .NET.

Pisząc kod parametryzowanego silnika, postanowiono opierać się o dobre wzorce i wiedzę zaczerpniętą z części opisowej pracy, dotyczącej przeglądu dostępnych implementacji silników gier logicznych. Stąd też czytelnik słusznie zauważa duży stopień inspiracji tego projektu Stockfishem (bo jest to “state of the art” silnik szachowy) i Connect-k (bo jest to jedyny otwarty silnik dla gier z rodziny (m,n,k,p,q)).

### 3.4.1 Plik Types.h – definicje, makra i struktury danych

Zgodnie z treścią poprzedniego rozdziału, kod silnika ma być dostosowany do wprowadzanych przez kompilator definicji. Jednocześnie, aby łatwo się go programowało i testowało, należy posiadać „lokalne” definicje symboli używanych w wielu miejscach w kodzie. Zdecydowano o wprowadzeniu prostej flagi \_USE\_GENERATOR\_DEFINES, która jeżeli jest zdefiniowana, to ignorowane są lokalne definicje (zawsze definiuje ją generator), a używane są te zadeklarowane przez generator. W innym wypadku używane są lokalne definicje. Zademonstrowano to we fragmencie kodu poniżej.

```
#ifndef _USE_GENERATOR_DEFINES
#define _USE_LOCAL_DEFINES
#endif

#ifndef _USE_LOCAL_DEFINES
#define M 3
#define N 3
#define K 3
#define P 1
#define Q 1
//#define EXACTLY_K_TO_WIN
#endif

#ifndef EXACTLY_K_TO_WIN
#define K_OR_MORE_TO_WIN
#endif
```

Jak widać, w podobny sposób potraktowano stałą EXACTLY\_K\_TO\_WIN – jeżeli nie jest zdefiniowana, to definiujemy K\_OR\_MORE\_TO\_WIN.

Jako że wewnętrznie dla niektórych rozmiarów planszy używany jest bitboard (a wydajność jest kluczowa) potrzebne są makra do łatwej modyfikacji konkretnych bitów. Najpierw jednak należy policzyć rozmiar planszy i zdecydować, czy w ogóle istnieje typ prymitywny zmiennej, który pomieści bitboard. W innym wypadku definiowana jest stała REQUIRES\_ARRAYS – mająca taki sens, że należy do reprezentacji planszy użyć tablic, gdyż bitboard nie pomieści planszy w żadnym z typów prostych.

```
#define BOARD_SIZE M*N
#define REQUIRES_ARRAYS BOARD_SIZE > 64

#if BOARD_SIZE > 32
#define CHECK_BIT(var, pos) ((var) & (1i64<<(pos)))
```

```

#define SET_BIT(number,pos) number |= 1i64 << pos;
#else
#define CHECK_BIT(var,pos) ((var) & (1 << (pos)))
#define SET_BIT(number,pos) number |= 1 << pos;
#endif

```

Dodatkowo, jeżeli plansza jest większa od 32, to przesuwana jedynka jest 64-bitowa – unikamy w ten sposób ostrzeżenia kompilatora.

Na koniec zdefiniowano jeszcze kilka typedefów przydatnych z punktu widzenia aplikacji i enumy reprezentujące konkretne wartości koloru pól czy typ gracza.

```

typedef uint8_t coord;
typedef uint16_t arrayIndex_t;

enum Color
{
    Black,
    White,
    None
};

enum PlayerType
{
    Human,
    AI
};

```

Typedefy zostały użyte, aby – gdy to konieczne – łatwo było zmniejszyć lub zwiększyć zakres używanego typu dla konkretnych przypadków. Na przykład coord jest zawsze używany dla współrzędnych, więc hipotetycznie, jeżeli byłaby potrzeba umożliwienia planszy większych niż 255 w jednym z dwóch wymiarów, to trzeba by zmienić typ coord z uint8\_t na uint16\_t.

Z ważnych struktur danych dla projektu należy jeszcze wymienić strukturę reprezentującą ruch – Move. Struktura ta jest maksymalnie prosta – przechowuje jedynie współrzędne x,y i wagę. Waga może być używana przez AI do określenia jak dobry (lub jak zły) jest ruch.

```

struct Move
{
    coord x, y;
    int weight;
};

```

### 3.4.2 Klasa Board – reprezentacja planszy

Stan klasy Board odpowiedzialnej za reprezentację planszy to właściwie dwie zmienne – blackPieces, przechowująca informacje o obecności lub braku czarnego pionka w danym miejscu (1 lub 0 – wartość logiczna true lub false) i analogicznie – m whitePieces. Dla plansz większych niż 64 używamy tablic zmiennych typu bool, dla mniejszych jest to zależnie od rozmiaru planszy wystarczająco duża zmienna przechowująca bitboard.

```
#if BOARD_SIZE > 64
```

```

    bool blackPieces[M*N];
    bool whitePieces[M*N];
#elif BOARD_SIZE > 32
    uint64_t blackPieces;
    uint64_t whitePieces;
#elif BOARD_SIZE > 16
    uint32_t blackPieces;
    uint32_t whitePieces;
#elif BOARD_SIZE > 8
    uint16_t blackPieces;
    uint16_t whitePieces;
#else
    uint8_t blackPieces;
    uint8_t whitePieces;
#endif

```

Do łatwej i wydajnej modyfikacji konkretnych bitów, odpowiadających konkretnej pozycji, poza wcześniej opisanymi makrami zdefiniowano jeszcze makro POSITION zamieniające pozycje w reprezentacji x,y na komórkę tablicy lub bit bitboarda.

```

static inline coord Height()
{
    return M;
}

static inline coord Width()
{
    return N;
}

#define POSITION(x,y) (y*N + x)

```

Zdefiniowano także funkcje inline, opisujące dwa wymiary planszy. Są one w pewnym stopniu bardziej czytelne niż M i N, a dzięki słowa kluczowego inline ich wprowadzenie nie powinno kosztować nic z punktu widzenia wydajności, kompilator powinien po prostu wstawić odpowiednio wielkości M i N w miejsca ich wywołań [69].

Jedną z podstawowych funkcjonalności planszy, choćby do sprawdzenia poprawności proponowanego ruchu, jest ustalenie, czy dane pole jest puste. Przedstawiona poniżej funkcja planszy robi to dokładnie. Jej kod jest jednak inny, gdy można wykorzystać bitboard, i inny, gdy musiały zostać użyte tablice.

```

bool Board::IsEmpty(coord x, coord y) const
{
#if REQUIRES_ARRAYS
    return !blackPieces[POSITION(x, y)] && !whitePieces[POSITION(x, y)];
#else
    return !(CHECK_BIT(blackPieces, POSITION(x, y))) &&
    !(CHECK_BIT(whitePieces, POSITION(x, y)));
#endif
}

```

Stosunkowo mało znaną możliwością C++ jest oznaczenie metody jako const. Oznacza to, że taka metoda nie zmienia stanu wewnętrznego obiektu klasy, tzn. nie modyfikuje jego pól [70]. Może to być przydatne w poszukiwaniu błędów – jeżeli metoda nie zmienia stanu, to generalnie nie powinna być źródłem błędów, chyba że jej wartość jest używana w metodzie zmieniającej stan.

Podobnymi metodami są sprawdzające, czy pole na planszy jest danego koloru – IsColor i zwracające kolor danego pola – GetColor. Obie są const i wywołują także metodę IsEmpty, poza tym jedna z nich korzysta z drugiej.

```
Color Board::GetColor(coord x, coord y) const
{
    if (IsEmpty(x, y))
        return Color::None;
    if (IsColor(x, y, Color::Black))
        return Color::Black;
    return Color::White;
}

bool Board::IsColor(coord x, coord y, Color color) const
{
    if (color == Color::None)
        return IsEmpty(x, y);
    else if (color == Color::Black)
    {
#if REQUIRES_ARRAYS
        return blackPieces[POSITION(x, y)];
#else
        return CHECK_BIT(blackPieces, POSITION(x, y)) != 0;
#endif
    }
    else if (color == Color::White)
    {
#if REQUIRES_ARRAYS
        return whitePieces[POSITION(x, y)];
#else
        return CHECK_BIT(whitePieces, POSITION(x, y)) != 0;
#endif
    }
    return false;
}
```

Ich kod jest łatwy do zrozumienia. Warto jednak zauważyć, że wynik makra CHECK\_BIT jest porównywany z 0 zamiast z 1. Dzieje się tak, ponieważ CHECK\_BIT zeruje wszystkie bity, oprócz tego, który znajduje się na interesującej nas pozycji. To jednak nie znaczy, że wynikowa liczba będzie jedynką. Może to być np. dla planszy 2x2 wartość 0010, czyli 2. Porównanie do jedynki w takim wypadku zwróciłoby wartość false, dlatego sprawdzamy, czy liczba z makra CHECK\_BIT jest różna od 0, czyli czy sprawdzany bit ma wartość 1.

W kolejnej metodzie ujawnia się delikatna przewaga bitboardu nad tablicami. Dla każdej nowej rozgrywki potrzebna jest czysta plansza (bez żadnych pionów). Trzeba więc czyścić planszę dla każdego nowego wywołania gry. W takim celu powstała zaprezentowana poniżej metoda Clear(), ustawiająca wszystkie pola na 0 (wartość logiczną false).

```

void Board::Clear()
{
#if REQUIRES_ARRAYS
    for (arrayIndex_t i = 0; i<BOARD_SIZE; i++)
    {
        blackPieces[i] = false;
        whitePieces[i] = false;
    }
#else
    blackPieces = 0;
    whitePieces = 0;
#endif
}

```

W przypadku tablic wykona się tutaj aż  $2 \cdot M \cdot N$  instrukcji przypisania, podczas gdy dla bitboardu mamy zaledwie dwie takie operacje.

Kluczową metodą zmieniającą stan planszy jest metoda PlacePiece, służąca do umieszczenia w danej pozycji pionka o określonym kolorze. Korzysta ona ze wspomnianych wcześniej makr SET\_BIT i POSITION. Dla uproszczenia nie przyjmuje koloru, ale flaga czy pion są czarne (nie przewiduje się sytuacji, w której umieszcza się brak koloru – zawsze będzie to biały lub czarny pion).

```

void Board::PlacePiece(coord x, coord y, bool isBlack)
{
#if REQUIRES_ARRAYS
    if (isBlack)
    {
        blackPieces[POSITION(x, y)] = true;
    }
    else
    {
        whitePieces[POSITION(x, y)] = true;
    }
#else
    if (isBlack)
    {
        SET_BIT(blackPieces, POSITION(x, y));
    }
    else
    {
        SET_BIT(whitePieces, POSITION(x, y));
    }
#endif
}

```

Ostatnią metodą, która znalazła się w klasie Board ze względów użyteczności dla innych metod, jest CountPieces. Zlicza ona wystąpienia „pod rząd” danego koloru od pozycji x,y w kierunku wyznaczanym przez dx,dy tak długo, aż wystąpi inny kolor. Kolor łamiący sekwencję może być zapisywany we wskaźniku breakingColor, o ile nie jest on równy nullptr. Jak uważny czytelnik mógł zauważyc, implementacja metody jest inspirowana metodą o podobnej nazwie z wcześniejszej opisywanego silnika Connect-k.

```

uint16_t Board::CountPieces(char x, char y, Color color, char dx, char dy, Color* breakingColor) const
{
    uint16_t count;
    auto currentColor = Color::None;

    if (dx == 0 && dy == 0)
        return IsColor(x, y, color) ? 1 : 0;
    for (count = 0; x >= 0 && x < Width() && y >= 0 && y < Height(); count++)
    {
        currentColor = GetColor(x, y);
        if (color != currentColor)
            break;
        x += dx;
        y += dy;
    }
    if (breakingColor != nullptr)
        *breakingColor = currentColor;

    return count;
}

```

### 3.4.3 Klasa Game – obsługa komunikacji, gry i jej stanu

Cała logika gry, jej zasady, podział na tury i komunikacja znajdują się w pojedynczej klasie Game. Powoduje to jej całkiem znaczny rozmiar, prawdopodobnie niektóre funkcjonalności można by wydzielić do innych klas. Z braku czasu i dobrych pomysłów sporo kodu pozostało jednak w tejże klasie, dlatego też zaprezentowana poniżej definicja klasy może wydawać się obszerna.

```

class Game
{
public:
    Game();

    void StartGame();

    inline bool IsValidMove(coord x, coord y) const;

    bool MakeMove(coord x, coord y);
    bool GetMove();
    bool CheckWin(coord x, coord y, coord& x1, coord& y1, coord& x2, coord&
y2) const;
    void WriteMove(coord x, coord y) const;
    bool CheckGameEnd(coord x, coord y);
    void NextTurn();
    Move* GetMoves() const;
    static std::string engine_info(bool b);

    void GameLoop(int argc, char* argv[]);

private:
    std::vector<double> aiGetMoveTimes;
    std::vector<double> checkGameEndTimes;
    std::vector<double> getMovesTimes;
    uint16_t movesLeft = Q;
    uint16_t movesMade = 0;
}

```

```

    bool gameStarted = false;
    Board board;
    AIPlayer aiPlayer;
    Color currentColor = Color::Black;
    PlayerType players[2];
};


```

Metody zostaną omówione później. Jeżeli chodzi o pola, to pierwsze trzy kolekcje typu vector służą do zbierania informacji o czasach wykonania konkretnych procedur. Ułatwia to później analizę wydajnościową silnika, gdyż wszystkie niezbędne czasy są mierzone wewnętrznie, z maksymalną dostępną precyją i bez jakiegokolwiek narzutu, który ciężko byłoby potem wyeliminować. Kolejne dwa pola to „liczniki” – movesLeft, czyli ile ruchów pozostało do wykonania w bieżącej turze przez gracza, i movesMade, czyli ile ruchów zostało wykonanych od rozpoczęcia gry. Jest też flaga gameStarted oznaczająca, czy gra się już zaczęła, czy nie, oraz dwa obiekty: board reprezentujący planszę i aiPlayer reprezentujący gracza AI. Obecny kolor currentColor z powodu podziału na tury przyznaje raz jednemu, raz drugiemu graczowi prawo do ruchu. Ważna jest też dwuelementowa tablica players, przechowująca typ obu graczy (np. gracz nr 0 – czarny jest człowiekiem, a gracz nr 1 – biały jest AI). Tyle zmiennych wystarczy, aby precyzyjnie opisać stan gry z rodziny (m,n,k,p,q).

Konstruktor gry jest bardzo prosty, najważniejsze są w nim wywołania konstruktorów planszy i gracza AI.

```

Game::Game(): board(), aiPlayer(board)
{
    players[Color::Black] = PlayerType::Human;
    players[Color::White] = PlayerType::AI;
}

```

Bardziej złożona i dużo istotniejsza jest metoda StartGame(). Wywoływana jest przed rozpoczęciem nowej gry i ustawia wszystkie zmienne na startowe wartości. Wiadomo, że grę zawsze zaczyna gracz czarny, plansza jest czysta, ruchów wykonanych jest 0 i pierwszy gracz w pierwszej turze może wykonać ruchów Q. Dokładnie to jest ustawiane. Można by oczywiście za każdym wywołaniem nowej gry wywoływać konstruktor, ale byłoby to zdecydowanie drożej wydajnościowo – alokacja nowej pamięci, kreacja obiektów. Tymczasem tutaj jedynie zmieniamy stan zmiennych i możemy cały czas ponownie używać tych samych obiektów, w tym samym miejscu w pamięci komputera.

```

void Game::StartGame()
{
    board.Clear();
    movesLeft = Q;
    gameStarted = true;
    currentColor = Color::Black;
    movesMade = 0;
}

```

Oczywiście, aby program GUI czy aplikacja testująca mogły łatwo współpracować z silnikiem, potrzebny jest sposób na ujednoliconą komunikację. Wzorując się po części na

standardzie szachowym tekstowej komunikacji z aplikacją konsolową UCI i jego implementacji w omawianym wcześniej silniku Stockfish, zdecydowano się na podobne rozwiązanie. Zaimplementowano kilka komend tekstowych, które służą do komunikacji z silnikiem.

- Komendy wejścia (wpisywane przez komunikującego się z silnikiem, | oznacza lub, a kursywa oznacza opcjonalność)
  - newgame [white|black] [ai|human] [white|black] [ai|human]
    - rozpoczyna nową grę (wywołuje m.in. StartGame)
    - przykłady:
      - newgame
      - newgame white ai black ai
      - newgame white human black human
      - newgame black human white ai
    - odpowiedź:
      - game started
  - printboard
    - wypisuje planszę, używając notacji X dla pionów gracza czarnego i O dla gracza białego, użyteczne w celach debugowania
    - przykładowa odpowiedź:
      - XOX
      - OXOX
      - XXX
      - OOOO
  - getmoves
    - zwraca wszystkie dostępne (wolne i poprawne) ruchy
    - odpowiedź: moves: (x<sub>1</sub> y<sub>1</sub>) (x<sub>2</sub> y<sub>2</sub>) (x<sub>3</sub> y<sub>3</sub>) (x<sub>4</sub> y<sub>4</sub>)...
    - przykładowa odpowiedź:
      - moves: (1 1) (2 1) (3 1) (4 1) (1 2) (2 2) (3 2) (4 2) (1 3) (2 3) (3 3) (4 3) (1 4) (2 4) (3 4) (4 4)
  - getplayer
    - zwraca kolor gracza, do którego należy ten ruch
    - odpowiedź: black|white
  - movesleft
    - zwraca ilość ruchów, które zostały do wykonania przez obecnego gracza
    - przykładowa odpowiedź: 1
    - odpowiedź nigdy nie będzie większa od max(p,q)
  - movesmade
    - zwraca ilość ruchów wykonanych od początku rozgrywki
    - przykładowa odpowiedź: 0
    - wartość nigdy nie będzie większa niż m·n
  - perf

- zwraca informacje o metrykach wydajnościowych silnika
  - należy wywoływać po kilku, a najlepiej kilkunastu lub kilkudziesięciu (im więcej tym lepiej) rozgrywkach, aby dane miały mniejszy błąd pomiarowy
  - wypisuje średni czas wygenerowania ruchu przez AI, średni czas sprawdzenia warunku wygranej i średni czas wygenerowania listy dostępnych (wolnych) ruchów
  - przykładowa odpowiedź:  
 average AIPlayer::GetMove() execution is 10620.1 ns  
 average Game::CheckWin() execution is 1237.43 ns  
 average Game::GetMoves() execution is 3384.86 ns
- isready
  - używana do sprawdzenia, czy silnik odpowiada
  - odpowiedź: readyok
- info
  - wyświetla informacje o parametrach silnika
  - odpowiedź:  
 $(m,n,k,p,q)$  *K\_OR\_MORE\_TO\_WIN|EXACTLY\_K\_TO\_WIN*
  - przykładowa odpowiedź:  
 $(4,4,3,1,1)$ *K\_OR\_MORE\_TO\_WIN*
  - wywołuje metodę engine\_info z parametrem true
- makemove x y
  - obecny gracz wykonuje ruch w pole o współrzędnych x,y, czyli pole x,y zmienia kolor na kolor gracza
  - wykona się poprawnie tylko wtedy, gdy jest tura gracza i ruch jest poprawny, w innym wypadku odpowiedź z silnika będzie informowała o tym, że ruch był niepoprawny
  - przykładowe wywołanie:
  - makemove 2 3
  - przykładowe odpowiedzi:
  - poprawny ruch: move black 2 3
  - nieporawny ruch: invalid move
- quit|exit|stop
  - kończy rozgrywkę, wyłącza silnik
  - powoduje wyłączenie aplikacji konsolowej silnika wraz ze zwróceniem kodu 0 (aplikacja zakończyła się sukcesem)
  - odpowiedź:  $(m,n,k,p,q)$ GameEngine has exited
- Komendy wyjścia (wypisywane przez silnik)
  - move black|white x y
    - może to być odpowiedź z silnika na wykonany przez człowieka ruch lub ruch gracza AI, jeżeli była jego tura

- przykład:
  - move white 1 2
- winner is black|white
  - pojawia się na koniec rozgrywki w przypadku wygranej jednego z graczy
- wining line is from x1 y1 to x2 y2
  - pojawia się w przypadku wygranej
  - zawsze występuje po komendzie winner is black|white
  - informuje o współrzędnych końca i początku linii, dzięki której zwycięzca spełnił warunek wygranej – ułożył k pionów swojego koloru pod rząd
  - przykład: wining line is from 4 4 to 4 1
- draw
  - wypisywane jest przez silnik, w przypadku gdy żaden z graczy nie wygrał, a dalsza rozgrywka nie może być kontynuowana (bo np. cała plansza jest zapełniona)
  - oznacza remis

Implementacja głównej pętli gry poza podstawową logiką gry, jak pozyskiwanie ruchów od AI, zarządzanie turami itp., zawiera właśnie obsługę komunikacji, w tym implementacje tych komend. Metoda jest inspirowana podobną w silniku Stockfish, który – jak wcześniej zauważono – może być uznany de facto za standard dla silników gier logicznych z racji swojej jakości. Większość kodu obsługującego konkretne polecenia jest bardzo prosta, więc nie wymaga omówienia. Należy jednak zauważyć, że część logiki, która mogłaby być oddelegowana do innych klas lub przynajmniej do jakichś metod klasy Game, pozostała z powodu braku czasu na dopracowanie w metodzie obsługującej polecenia. Szczególnie istotne jest choćby obsługiwanie parametrów do komendy newgame, które ustawiają wielkości opisujące typ graczy w tablicy players. Jest to ważne, ponieważ następuje przed wywołaniem metody StartGame, a jest to jakby część jej odpowiedzialności. Należy również pamiętać, że każda zmiana kodu na późnych etapach jego wytwarzania może generować bardzo wysokie koszty w przypadku, gdyby wprowadziła nowe błędy.

W komunikacji istotną rolę odgrywa metoda engine\_info, która zwraca informacje o parametralach silnika. Może być niezbędna np. do wyrysowania prawidłowej planszy w aplikacji GUI lub poznania ograniczeń dla wielkości współrzędnych w aplikacji testującej. Dla parametru true metoda zwraca dokładne wielkości parametrów (m,n,k,p,q,) i warunku wygranej.

Ważne też może być pobranie dostępnych ruchów. Szczególnie w testowaniu wygodnie byłoby pobrać listę dostępnych ruchów i dopiero wybrać spośród nich. Metoda GetMoves wywoływana komendą getmoves iteruje po całej planszy i zwraca tylko ruchy na puste pola. Dzięki trzymaniu ilości wykonanych ruchów od początku rozgrywki w zmiennej movesMade można łatwo policzyć, że wolnych ruchów jest m·n – movesMade – co ułatwia alokację pamięci na tablicę z wolnymi ruchami.

```
Move* Game::GetMoves() const
{
```

```

auto moves = new Move[BOARD_SIZE - movesMade];
auto i = 0;
for (coord y = 0; y < board.Height(); y++)
    for (coord x = 0; x < board.Width(); x++)
        if (board.IsEmpty(x, y))
    {
        Move m;
        m.x = x;
        m.y = y;
        moves[i++] = m;
    }

return moves;
}

```

Jeżeli gracz decyduje się na wykonanie ruchu komendą makemove, to wywoływana jest metoda MakeMove. Wykona ona ruch jednak tylko wtedy, kiedy jest on poprawny, na co daje odpowiedź metoda IsValidMove.

```

inline bool Game::IsValidMove(coord x, coord y) const
{
    return board.IsEmpty(x, y) && players[currentColor] == PlayerType::Human
&& x < N && y < M && movesLeft > 0;
}

bool Game::MakeMove(coord x, coord y)
{
    if (IsValidMove(x, y))
    {
        board.PlacePiece(x, y, currentColor == Color::Black);
        return true;
    }
    return false;
}

```

Metoda IsValidMove jest inline'owana z przyczyn wydajnościowych. Sprawdza ona, czy pole jest wolne, czy gracz obecny to człowiek, czy graczowi pozostał przynajmniej jeden ruch i czy ruch jest w granicach planszy. Tylko po spełnieniu tych wszystkich warunków ruch jest poprawny i może zostać umieszczony na planszy. W odpowiedzi na komendę makemove, jeżeli MakeMove zwróciło prawdę, to jest jeszcze zwiększana ilość wykonanych ruchów, wypisywany jest ruch jako odpowiedź silnika (metoda WriteMove) i jeżeli gra się nie zakończyła, to wołana jest następna tura. Dla łatwiejszej notacji, dla osób nie będących programistami współrzędne na zewnątrz zaczynają się od 1,1, podczas gdy wewnętrz silnika od 0,0.

Omawiane w poprzednim akapicie metody dotyczyły gracza będącego człowiekiem, ale podobne występują dla gracza AI. Różnica jest m.in. taka, że analogiczna dla metody MakeMove metoda GetMove jest wywoływana za każdym razem w pętli GameLoop tak długo, jak długo zwraca true – czyli tak długo, jak wykonuje kolejne ruchy. Poniżej przedstawiono implementację metody GetMove().

```

bool Game::GetMove()
{

```

```

        if (players[currentColor] == PlayerType::AI && movesLeft > 0 &&
gameStarted)
    {
        auto t1 = std::chrono::high_resolution_clock::now();

        auto aiMove = aiPlayer.GetMove();

        board.PlacePiece(aiMove.x, aiMove.y, currentColor == Color::Black);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::nanoseconds>(t2 - t1).count();
        aiGetMoveTimes.push_back(duration);

        WriteMove(aiMove.x, aiMove.y);
        movesMade++;
        if (!CheckGameEnd(aiMove.x, aiMove.y))
            NextTurn();
        return true;
    }
    return false;
}

```

Poza sprawdzeniem warunku, czy AI może wykonać obecny ruch, kluczowe jest tutaj wywołanie metody GetMove na obiekcie aiPlayer, które zwraca poprawny, wygenerowany przez gracza AI ruch. Dodatkowo mierzony jest w nanosekundach czas tego wywołania z użyciem high\_resolution\_clock, który pod Windowsem korzysta z API QueryPerformanceCounter [76] [77], co oznacza, że dokładność pomiaru typowo powinna być lepsza niż 1 $\mu$ s. Podobnie jak w przypadku gracza-człowieka, również tutaj ruch jest wypisywany na konsolę, ilość dokonanych ruchów jest inkrementowana i jeżeli gra się jeszcze nie skończyła, to wołana jest następna tura.

Metoda NextTurn() ustalająca stan gry na następną turę jest więc kluczowa dla obu opisywanych wcześniej przypadków. Jednocześnie jest stosunkowo prosta oraz na tyle przejrzysta i wygodna w użyciu, że wydaje się upraszczać logikę gry.

```

void Game::NextTurn()
{
    movesLeft--;
    if (movesLeft <= 0)
    {
        if (currentColor == Color::Black)
            currentColor = Color::White;
        else
            currentColor = Color::Black;
        movesLeft = P;
    }
}

```

Jak łatwo przewidzieć, najpierw zmniejszana jest liczba ruchów, które pozostały. Jeżeli jednak po zmniejszeniu liczby pozostałych ruchów osiągnie ona wartość 0, to należy przełączyć kolor na przeciwnego gracza. Jeżeli był czarny, to teraz kolej na białego i vice versa. W obu przypadkach ilość pozostałych ruchów ustawiana jest teraz na p. Pisząc to w taki sposób, wartości q używamy jedynie w metodzie StartGame do ustawienia movesLeft na Q. Potem po każdej pełnej turze zgodnie z wymogami gier z rodziny (m,n,k,p,q) ustawiamy wartość movesLeft na p.

Nie zawsze jednak poprawny ruch AI czy też człowieka wywołuje następną turę. Przed jej wywołaniem sprawdzany jest warunek końca gry. Efektywnie każdy ruch może zakończyć grę, dlatego po każdym poprawnym ruchu jest wywoływana metoda CheckGameEnd i tylko w przypadku, gdy zwróci on false, gra przechodzi do następnej tury. Poniżej podana jest implementacja tej metody.

```
bool Game::CheckGameEnd(coord x, coord y)
{
    coord x1, x2, y1, y2;

    auto t1 = std::chrono::high_resolution_clock::now();

    auto isWin = CheckWin(x, y, x1, y1, x2, y2);

    auto t2 = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(t2 - t1).count();
    checkGameEndTimes.push_back(duration);

    if (isWin)
    {
        sync_out << "winner is " << (currentColor == Color::Black ? "black" : "white") << sync_endl;
        sync_out << "wining line is from " << std::to_string(x1 + 1) << " "
        " << std::to_string(y1 + 1) << " to " << std::to_string(x2 + 1) << " "
        " << std::to_string(y2 + 1) << sync_endl;
        gameStarted = false;
        return true;
    }
    else if (movesMade == BOARD_SIZE)
    {
        sync_out << "draw" << sync_endl;
        gameStarted = false;
        return true;
    }
    return false;
}
```

Analogicznie, jak w poprzednich przypadkach, mierzona jest wydajność, tym razem dla wywołania CheckWin, sprawdzającego, czy któryś z graczy wygrał. Oczywiście w przypadku wygranej wypisywane są odpowiednie komunikaty, ale to nie jedyny przypadek, w którym gra ulega zakończeniu. W innym wypadku, jeżeli nie została stwierdzona wygrana, ale zostało wykonanych m·n ruchów, gra kończy się remisem, ponieważ cała plansza jest zajęta. W pozostałych przypadkach gra się nie zakończyła.

Została do wyjaśnienia metoda ustalająca, czy któryś z graczy wygrał – metoda CheckWin. Oczywiście w praktyce sprawdzamy, czy wygrał gracz, który właśnie wykonał ruch – sprawdzamy dla koloru currentColor – sprawdzanie dla obu kolorów byłoby niepotrzebne, gdyż sytuacja dla koloru poprzedniego nie uległa zmianie od poprzedniego sprawdzenia. Metoda została zaprezentowana poniżej.

```
bool Game::CheckWin(coord x, coord y, coord& x1, coord& y1, coord& x2, coord& y2)
const
{
```

```

    uint16_t c1, c2;
    char xs[] = {1, 1, 0, -1}, ys[] = {0, 1, 1, 1};

    if (board.IsEmpty(x, y))
        return false;

    for (char i = 0; i < 4; i++)
    {
        c1 = board.CountPieces(x, y, currentColor, xs[i], ys[i], nullptr);
        c2 = board.CountPieces(x, y, currentColor, -xs[i], -ys[i],
        nullptr);

#if defined(K_OR_MORE_TO_WIN)
        if (c1 + c2 - 1 >= K)
#endif defined(EXACTLY_K_TO_WIN)
        if (c1 + c2 - 1 == K)
#endif
    {
        x1 = coord(x + xs[i] * (c1 - 1));
        y1 = coord(y + ys[i] * (c1 - 1));
        x2 = coord(x - xs[i] * (c2 - 1));
        y2 = coord(y - ys[i] * (c2 - 1));
        return true;
    }
}
return false;
}

```

Ustawiamy przesunięcia xs i ys – czyli kierunki, w których będziemy szukać k pionów obecnego koloru pod rząd. Oczywiście, jeżeli miejsce, dla którego sprawdzamy, jest puste, to nie można stwierdzić wygranej. Należy tutaj nadmienić, że sprawdzamy wygraną zawsze dla świeżo wykonanego ruchu, czyli dla współrzędnych x,y. W głównej pętli metody CheckWin obliczamy ilość wystąpień obecnego koloru pod rząd zarówno w danym kierunku, jak i przeciwnym (zmiennie c1 i c2). Ich suma pomniejszona o 1 (bo dwa razy liczymy kolor z właśnie wykonanego ruchu), to ilość danego koloru pod rząd. Zależnie, czy jest to silnik typu dokładne k pod rząd, czy dopuszcza więcej niż k, sprawdzamy odpowiednią instrukcję warunkową. Jeżeli zwraca ona prawdę, to wyznaczamy współrzędne linii wygrywającej z prostych matematycznych zależności i zapisujemy je do przekazanych przez referencję zmiennych x1, x2, y1, y2. Zwracamy wtedy true. Gdy po sprawdzeniu wszystkich możliwych przesunięć i niestwierdzeniu w żadnym kierunku wymaganej liczby wystąpień pod rząd danego koloru, zwracamy po prostu false – nikt jeszcze nie wygrał. Do obliczania wystąpienia pod rząd koloru używamy metody planszy CountPieces (omawianej wcześniej) i przekazujemy nullptr jako wskaźnik na Color (parameter breakingColor), bo nie interesuje nas, jaki kolor złamał sekwencję. Metoda jest reimplementacją podobnej z silnika Connect-k po to, aby można było łatwo w przyszłości porównywać wydajność między silnikami.

#### 3.4.4 Klasa AIPlayer – proste AI zwracające poprawne ruchy

W związku z założeniem, że niniejsza praca nie powinna się skupiać na sztucznej inteligencji, klasa AIPlayer jest stosunkowo prosta. Algorytm AI jest analogiczny do algorytmu

ai\_adjacent z silnika Connect-k, co umożliwiłoby w przyszłości porównanie wydajności. Poniżej przedstawiono definicję klasy.

```
class AIPlayer
{
public:
    AIPlayer(const Board& board);
    ~AIPlayer();
    Move GetMove() const;
private:
    bool IsAdjacent(coord x, coord y) const;
    Random random;
    const Board* board;
};
```

Jak widać, klasa zawiera konstruktor przyjmujący referencję na planszę, a także pole będące wskaźnikiem na obiekt planszy. Aby klasa AIPlayer była w stanie w każdej turze wyznaczyć poprawny ruch, musi mieć podgląd na aktualną wersję planszy. Jednak aby w swoim działaniu AI nie zmieniało planszy (jest to odpowiedzialność klasy Game, jedyna odpowiedzialność klasy AIPlayer to generowanie poprawnych ruchów), plansza jest const. Oznacza to, że AIPlayer może wołać na board tylko metody, które są też const i nie może modyfikować pól obiektu board (chyba, że są mutable) [71] [72]. Możliwość takiego zadeklarowania pól i metod bardzo zabezpiecza kod przed nieoczekiwanymi błędami programisty. Poniżej implementacja konstruktora.

```
AIPlayer::AIPlayer(const Board& board) : random()
{
    this->board = &board;
}
```

Wyoływany jest tutaj również konstruktor obiektu random, który ma służyć klasie jako generator liczb pseudolosowych. Oczywiście można by przekazywać planszę jako const Board& za każdym razem do metody GetMove, ale wybrane rozwiązanie jest tańsze obliczeniowo, bo przekazujemy ją tylko raz, a potem korzystamy ze wskaźnika.

Klasa Random z kolei została tak zaprojektowana, aby była szybsza od zwykłego wywołania rand() jak i wszelkich implementacji generatorów liczb losowych z pakietu standardowego języka C++. Wykorzystano tutaj implementację fastrand(), która jest przynajmniej dwa razy szybsza [73].

```
class Random
{
public:
    Random();
    size_t GetValue(size_t min, size_t max) const;

    ~Random();
private:
    mutable size_t g_seed;

    inline size_t fastrand() const
    {
        g_seed = 214013 * g_seed + 2531011;
```

```

        return (g_seed >> 16) & 0x7FFF;
    }
};


```

Implementacje GetValue i konstruktora są trywialne.

```

Random::Random()
{
    g_seed = std::time(nullptr);

size_t Random::GetValue(size_t min, size_t max) const
{
    return (min + fastrand()) % max;
}

```

Najważniejszą metodą klasy AIPlayer jest GetMove, zwracająca poprawne ruchy po wywoaniu. Implementacja jest inspirowana analogiczną metodą z silnika Connect-k, dlatego też może ona się wydawać nieoptymalna. Zdecydowano się jednak na nią, aby mieć w przyszłości możliwość uczciwego porównania wydajnościowego.

```

Move AIPlayer::GetMove() const
{
    Move move;
    std::vector<Move> empties;

    for (move.y = 0; move.y < board->Height(); move.y++)
        for (move.x = 0; move.x < board->Width(); move.x++)
            if (IsAdjacent(move.x, move.y))
                empties.push_back(move);

    if (empties.empty())
    {
        move.y = board->Height() / 2;
        move.x = board->Width() / 2;
        return move;
    }

    auto index = random.GetValue(0, empties.size()); //rand() % empties.size();

    return empties[index];
}

```

Najpierw znajdują się wszystkie puste pola na planszy, które spełniają warunek sąsiadowania z jakimkolwiek zajętym polem (wywołanie metody IsAdjacent). Jeżeli nie zostały znalezione żadne takie pola, to wykonywany jest ruch w centrum planszy (oznacza to, że ruch jest tak naprawdę pierwszym ruchem – plansza jest pusta). Jeżeli znalezione zostały jakieś pola spełniające ten warunek, to zostaje wybrane jedno, losowe pole, które zwracane jest jako wybrany ruch. Jak można się domyśleć, duże znaczenie ma tutaj metoda IsAdjacent, która w przyszłości mogłaby być zastąpiona bardziej złożoną metodą obliczającą wagę dla każdego ruchu.

### **3.5 Biblioteka ułatwiająca współpracę z silnikiem – ( $m,n,k,p,q$ )EngineWrapper**

Z kilku powodów najlepszym rozwiążaniem problemu współpracy z silnikami okazało się napisanie biblioteki-wrappa dla środowiska .NET. Zdecydowano się na takie rozwiązanie m.in. po to, aby zapewnić ujednoliczoną współpracę z wygenerowanym silnikiem dla wszystkich aplikacji napisanych w .NET. Z uwagi na fakt, że autor preferuje pisanie programów w języku C#, każda kolejna aplikacja (czy to GUI, czy tester, czy może jeszcze coś innego współpracującego z silnikiem napisanym w C++) będzie korzystać ze środowiska .NET. Duplikowaniem kodu byłoby więc pisanie w każdym z tych programów fragmentów do kooperowania z natywnym silnikiem – rozpoznawania jego komend itp. Można by jednak zarzucić, że rozwiązanie to nie jest optymalne – przecież jeżeli potrzebujemy biblioteki współpracującej, to można było od razu napisać silnik jako bibliotekę C++ i używać „platform invoke” [85]. Owszem, wrapper byłby wtedy znacznie cieńszy i prostszy, ale jednocześnie silniki nie byłyby na tyle multiplatformowe, jak są w przypadku bycia aplikacjami działającymi w trybie tekstowym na zasadzie komunikatów.

Do głównych zadań tej biblioteki należy przetwarzanie i nadawanie komunikatów tekstowych z i do silnika. Poza tym potrzebne są też takie funkcjonalności, jak uruchamianie i zamknięcie silnika, w tym ukrywanie tego, że on działa w tle. Grając w aplikację gui czy robiąc testy, niekoniecznie przecież chcemy widzieć cały input i output aplikacji konsolowej z dodatkową możliwością przypadkowego jej wyłączenia.

#### *3.5.1 Klasa ProcessInBackground*

Biorąc pod uwagę podaną wcześniej motywację, autor stwierdził, że po pierwsze przydałaby się klasa, która uruchamia jakiś process (np. silnik gry), ale ukrywa jego działanie przed użytkownikiem. W środowisku .NET jest dedykowana klasa Process służąca do zarządzania procesami – ich uruchamianie, zatrzymywanie, ustawianie parametrów rozruchowych itp. jest wyeksponowane w wygodnym API.

Domyślnie jednak aplikacja w ten sposób uruchomiona będzie widoczna dla użytkownika z możliwością jej wyłączenia, co może być niebezpieczne. Napisana dla celów tej pracy klasa ProcessInBackground ukrywa przed użytkownikiem wszelkie znaki działania aplikacji w tle. Kluczowe znaczenie w tym ma pole \_baseStartInfo przedstawione wraz z innymi polami klasy poniżej.

```
private readonly ProcessStartInfo _baseStartInfo = new ProcessStartInfo
{
    UseShellExecute = false,
    RedirectStandardOutput = true,
    RedirectStandardError = true,
    CreateNoWindow = true,
    WindowStyle = ProcessWindowStyle.Hidden
};

private readonly bool _input;

private readonly Process _process;
```

Jak pokazano powyżej, ustawiamy takie wartości proporcji klasy ProcessStartInfo, aby ukryć działanie aplikacji przed użytkownikiem. Istotne są też flagi do przekierowania standardowego wyjścia i standardowego wyjścia błędu, ponieważ chcemy w bibliotece mieć dostęp do zwracanych poprzez uruchomiony process komunikatów. Poza tym przechowujemy również sam proces, jak i flagę `_input` oznaczającą, czy interesuje nas dalsza interakcja z nim (po jego uruchomieniu).

Bardzo ważny jest konstruktor – tworzy klasę procesu oraz ustawia przekierowanie standardowego wyjścia i wyjścia błędu dla niego.

```
public ProcessInBackground(string filename, string arguments,
Action<string> callback, bool input)
{
    _input = input;
    baseStartInfo.FileName = filename;
    baseStartInfo.Arguments = arguments;

    _baseStartInfo.RedirectStandardInput = input;

    _process = new Process { StartInfo = _baseStartInfo };

    _process.OutputDataReceived += (o, e) => callback?.Invoke(e.Data);
    _process.ErrorDataReceived += (o, e) => callback?.Invoke(e.Data);
}
```

Kontrola nad wyjściem oddawana jest parametrowi będącemu delegatem `callback`, ale jednocześnie dopuszcza się jego wartość równą null, oznaczającą tak naprawdę ignorowanie wyjścia. Wyjście i wyjście błędu są przekierowywane na wywołanie delegata `callback`.

Proces oraz przekierowania jego strumieni są tworzone w konstruktorze. Ale taki proces jeszcze nie działa. Uruchamia go dopiero metoda `Run`. Wewnątrz tej metody znajduje się przede wszystkim wywołanie metody `start` na obiekcie klasy `Process` i rozpoczęcie czytania/przekierowywania wyjścia z procesu. Co istotne, że jeżeli nie interesuje nas dwukierunkowa interakcja z procesem (flaga `_input` ustawiona na `false` – rezygnujemy z wysyłania komunikatów do procesu), to proces uruchamiany jest synchronicznie. Oznacza to, że w takim przypadku czekamy na niego, aż skończy swoje działanie.

```
public void Run()
{
    _process.Start();

    _process.BeginOutputReadLine();
    _process.BeginErrorReadLine();
    if (!_input)
        _process.WaitForExit();
}
```

Samo uruchomienie procesu to jednak w przypadku aplikacji silnika dopiero początek – kluczowe jest wysyłanie komunikatów. Odbieranie komunikatów obsługiwane jest przez przekazywany w konstruktorze delegate. Do wysyłania zdefiniowano bardzo prostą metodę

korzystającą z wyeksponowanego przez Process, pod właściwością StandardInput, obiektu StreamWriter.

```
public void Send(string cmd)
{
    _process.StandardInput.WriteLine(cmd);
}
```

Jak można zauważyć, klasa ProcessInBackground jest niezwykle prosta, ale przy tym – przynajmniej dla celów niniejszej pracy – również niezwykle użyteczna.

### 3.5.2 Klasa EngineParameters i wyliczenie WinCondition

EngineParameters implementuje interfejs INotifyPropertyChanged, który jest niezbędny, aby można było wykonać binding (nie tylko w WPF-ie, ale w całym środowisku .NET). Bez zastosowania kontenerów IoC tworzy to trochę nadmiarowy kod [65], ale w związku ze wspomnianym wcześniej „rozbiciem na projekty” można sobie na to, przynajmniej na razie, pozwolić. Poniżej zaprezentowano fragment klasy EngineParameters, aby nieco zobrazować idee przechowywanych parametrów.

```
public class EngineParameters : INotifyPropertyChanged,
IEquatable<EngineParameters>
{
    private ulong _k;
    private ulong _m;
    private ulong _n;
    private ulong _p;
    private ulong _q;
    private WinCondition _winCondition;

    public EngineParameters()
    {
    }

    public EngineParameters(ulong m, ulong n, ulong k, ulong p, ulong q,
    WinCondition w)
    {
        M = m;
        N = n;
        K = k;
        P = p;
        Q = q;
        WinCondition = w;
    }

    public ulong M
    {
        get { return _m; }
        set
        {
            _m = value;
            OnPropertyChanged();
        }
    }
}
```

Parametry do generacji silnika są przechowywane jako zmienne prywatne, ale eksponowane są na zewnątrz jako publiczne właściwości – tak aby każda zmiana przez property wywołała metodę OnPropertyChanged(), która wywołuje event PropertyChanged z INotifyPropertyChanged (o ile przynajmniej jeden obiekt subskrybował do tego eventu).

```
protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
{
    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
}
```

Jak widać, wykorzystano też atrybut [CallerMemberName], który umożliwia wywołanie metody bez parametru – jako parametr będzie dostępna wtedy nazwa metody/właściwości, w której dokonano wywołania.

W związku z tym, że jeżeli plansza jest przynajmniej w jednym rozmiarze (n lub m) większa od ilości wymaganych kamieni pod rząd (k), istnieje możliwość dwóch warunków wygranej, wprowadzono to jako dodatkowy parametr \_winCondition. Typ WinCondition to prosty enum.

```
public enum WinCondition
{
    [Description("exactly k-in-a-row")]
    // ReSharper disable once InconsistentNaming
    EXACTLY_K_TO_WIN,

    [Description("k-in-a-row or more")]
    // ReSharper disable once InconsistentNaming
    K_OR_MORE_TO_WIN
}
```

Pierwsza wartość oznacza, że jest potrzebne dokładnie k kamieni jednego koloru pod rząd, żeby wygrać. W takiej sytuacji k+1 lub więcej nie oznacza wygranej. Druga wartość daje wygraną we wszystkich przypadkach, gdy pod rząd jest k lub więcej kamieni tego samego koloru. Wykorzystano to m.in. do stworzenia dwóch wersji Gomoku (freestyle i standard) [34].

W związku z powyższym, parametr ten może mieć znaczenie lub nie – zależy to od rozmiaru planszy. Dlatego w metodzie przeciążonej ToString nie zawsze go wypisujemy.

```
public override string ToString()
{
    var ret = ${M},{N},{K},{P},{Q})GameEngine";
    if (Math.Max(M, N) > K)
    {
        ret += ${_WinCondition}";
    }
    return ret;
}
```

Jak widać, dla prostoty kodu zastosowano też stosunkowo nowy feature języka C# 6.0, jakim jest interpolacja stringów [66]. Dopisywany jest również fragment GameEngine, głównie z powodów identyfikacyjnych – wywołanie ToString na obiekcie klasy EngineParameters jest używane do wielu celów w różnych obszarach projektu, na co będzie zwrócona uwaga w kolejnych rozdziałach.

Kluczową częścią wrappera silników jest interpretowanie komunikatów tekstowych z silnika gry, jako że generowane silniki są aplikacjami konsolowymi i pracują w trybie tekstowym. W związku z tym należy gdzieś umieścić parsowanie części tekstowych na obiekty biznesowe. Wzorując się nieco na rozwiązaniach z samego framework'a .NET, gdzie np. typy liczbowe, jak int mają metody TryParse zwracające wartość logiczną, reprezentującą sukces procesu parsowania, zastosowano podobne rozwiązania. Klasa EngineParameters ma również metodę TryParse, której ciało pokazano poniżej.

```
public static bool TryParse(string str, out EngineParameters engineParameters)
{
    engineParameters = new EngineParameters();
    var match = EngineParametersRegex.Match(str);
    if (!match.Success)
        return false;

    engineParameters.M = ulong.Parse(match.Groups[1].Value);
    engineParameters.N = ulong.Parse(match.Groups[2].Value);
    engineParameters.K = ulong.Parse(match.Groups[3].Value);
    engineParameters.P = ulong.Parse(match.Groups[4].Value);
    engineParameters.Q = ulong.Parse(match.Groups[5].Value);

    engineParameters.WinCondition =
        match.Groups[6].Value.ToLowerInvariant().Contains("EXACTLY_K_TO_WIN".ToLowerInvariant())
            ? WinCondition.EXACTLY_K_TO_WIN
            : WinCondition.K_OR_MORE_TO_WIN;
    return true;
}
```

Metoda korzysta z wyrażenia regularnego – jeśli do niego nie pasuje, to od razu może zwrócić false, parametr przekazywany przez out w tym przypadku nie ma żadnego znaczenia, nie powinien być używany przez wołającego metodę. Jeżeli jednak proces dopasowywania do wyrażenia regularnego zakończył się sukcesem, to zgodnie z wyrażeniem kolejne jego grupy przechwytyujące są parsowane jako liczby typu ulong, odpowiadające kolejnym parametrom silnika. Na koniec sprawdzone jest zawieranie warunku wygranej, o którym wspomniano już wcześniej. Dopuszcza się napisanie go w dowolnym rozmiarze literek, ale musi być z przedziału dwóch wartości. Niżej zaprezentowane wyrażenie regularne używane w metodzie TryParse pokazuje, jak mniej więcej łańcuch znaków informujący o parametrach silnika powinien wyglądać.

```
private static readonly Regex EngineParametersRegex =
    new Regex(
```

```

@"\s*(\s*(\d+)\s*,\s*(\d+)\s*,\s*(\d+)\s*,\s*(\d+)\s*,\s*(\d+)\s*)\s*?\s*(EXAC
TLY_K_TO_WIN|K_OR_MORE_TO_WIN)?",
    RegexOptions.IgnoreCase | RegexOptions.Compiled);

```

Aby klasa mogła być sensownie używana jako struktura danych, należy jeszcze przeciążyć metody GetHashCode i Equals. Domyślnie obiekty klasy są porównywane w C# po referencjach. Oznacza to, że dwa obiekty o takiej samej wartości pól nie są sobie równe, chyba że są tym samym obiektem (tą samą instancją, tym samym miejscem w pamięci). Ze względu na typowy charakter „pojemnika na dane” klasy EngineParameters i użycie jej w kolekcjach z wyszukiwaniem, usuwaniem itp. trzeba zaimplementować interfejs IEquatable<EngineParameters>.

```

public bool Equals(EngineParameters other)
{
    if (ReferenceEquals(null, other)) return false;
    if (ReferenceEquals(this, other)) return true;
    return _m == other._m && _n == other._n && _k == other._k && _p ==
other._p && _q == other._q &&
        _winCondition == other._winCondition;
}

public override bool Equals(object obj)
{
    if (ReferenceEquals(null, obj)) return false;
    if (ReferenceEquals(this, obj)) return true;
    if (obj.GetType() != GetType()) return false;
    return Equals((EngineParameters) obj);
}

```

Metody zostały wygenerowane przy użyciu narzędzia ReSharper, ale tak naprawdę są bardzo proste i „z ręki” zostałyby napisane tak samo. Pierwsza z nich będzie wołana w np. kolekcjach generycznych, porównuje ona bowiem ten obiekt klasy EngineParameters do innego obiektu tej klasy. Są one równe wtedy, gdy są tym samym obiektem [ReferenceEquals (this, other) domyślne zachowanie w C#] i gdy mają wszystkie interesujące nas pola takie same – (m,n,k,p,q) i warunek wygranej. Różne będą we wszystkich pozostałych przypadkach, jak i wtedy, gdy do instancji tego obiektu będzie porównywany null, co jest jednocześnie zaleceniem Microsoftu dotyczącym implementacji interfejsu IEquatable. Druga wersja metody – porównanie z object jest specjalnie dla kolekcji z czasów .NET 1.1 i starszych, czyli przed ery generyków. Tamte kolekcje przechowywały po prostu object, więc porównanie będzie z object. Zachowuje się ona dokładnie tak jak poprzednia, ale dodatkowo w przypadku, gdy porównuje się obiekt typu EngineParameters z obiektem innego typu, z góry zwraca false. Jeżeli oba obiekty są instancjami klasy EngineParameters, to porównanie odbywa się zgodnie z wcześniej omawianą metodą.

Dla kolekcji haszowanych, takich jak słownik czy tablica haszująca, potrzebna jest jeszcze metoda GetHashCode. Wygenerowana przez ReSharper metoda przeciążona może wyglądać całkiem interesująco na pierwszy rzut oka.

```
public override int GetHashCode()
```

```

{
    unchecked
    {
        var hashCode = _m.GetHashCode();
        hashCode = (hashCode*397) ^ _n.GetHashCode();
        hashCode = (hashCode*397) ^ _k.GetHashCode();
        hashCode = (hashCode*397) ^ _p.GetHashCode();
        hashCode = (hashCode*397) ^ _q.GetHashCode();
        hashCode = (hashCode*397) ^ (int) _winCondition;
        return hashCode;
    }
}

```

W rzeczywistości jest dosyć podobna do Equals – przechodzimy po wszystkich istotnych dla nas polach i xor-ujemy ich hasze. Z małym zastrzeżeniem, że dla enuma bierzemy jego wartość i dla kolejnych pól po pierwszym bierzemy mnożnik \*397. Zespół ReSharpera zapewne testował różne parametry i ich wpływ na prawdopodobieństwo wystąpienia kolizji i ta wartość musiała być dosyć dobra. Należy też zauważać, że wyliczenie hasz kodu znajduje się w bloku unckecked. Blok ten zapobiega sprawdzaniu, czy wystąpiło przepełnienie (overflow). Mnożenie i xorowanie kolejnych wartości może doprowadzić do wyjścia poza zakres zmiennej typu int – w zależności od ustawień środowiska .NET. Taka sytuacja może doprowadzić do rzucenia wyjątku (domyślnie nie wyrzuca wyjątku) – słówko unchecked jest tutaj po to, aby wyjątek się nie pojawił [67][68]. Poza tym umieszczenie tego w bloku unchecked oznajmia dodatkowo innemu programiście, że autor wiedział, co robi – tutaj może wystąpić overflow, ale to dobrze, bo hasz kod może mieć dowolną wartość (jedynie ważne, żeby nie dochodziło do zbyt częstych kolizji).

### 3.5.3 Klasa PerformanceInformation i struktura ValueWithUnit

W związku z koniecznością pomiaru wydajności silnika, wynikającą z celów niniejszej pracy dyplomowej, należało po stronie biblioteki współpracującej zaimplementować logikę dla rozpoznawania i przechowywania informacji o wydajności silnika (odpowiedzi silnika na komendę „perf”). Jednym z problemów jest to, że wartości te występują wraz z jednostką czasową i może ich być kilka. Problem związany z kilkoma metrykami wydajnościowymi rozwiązano w tej wersji biblioteki w sposób trywialny – opierając się o wiedzę na temat generowanych silników, przyjęto trzy propercje reprezentujące trzy mierzone czasy. Są to: średni czas wygenerowania ruchu przez AI, średni czas sprawdzenia, czy któryś z graczy wygrał grę po danym ruchu i średni czas wygenerowania listy dostępnych (wolnych) ruchów. Właściwości te, znajdujące się w klasie PerformanceInformation, pokazano poniżej.

```

public ValueWithUnit AverageAiGetMoveExecution { get; set; }
public ValueWithUnit AverageCheckWinExecution { get; set; }
public ValueWithUnit AverageGetMovesExecution { get; set; }

```

Jak można się domyśleć, problem z przechowywaniem jednostek został rozwiązany przez implementację struktury ValueWithUnit, która jest typem każdej z trzech metryk. Kod struktury ValueWithUnit wklejono poniżej.

```

public struct ValueWithUnit
{
    public ValueWithUnit(double value, string unit)
    {
        Value = value;
        Unit = unit;
    }

    public double Value { get; }
    public string Unit { get; }

    public override string ToString()
    {
        return $"{Value} {Unit}";
    }
}

```

Jak na razie jednostka to jedynie prosty łańcuch znaków bez wyróżniania rozmaitych wielkości – może to być zadanie na przyszłość i wtedy też ta struktura wzbogaciłaby się o logikę przeliczającą z jednej jednostki czasowej na drugą. Na razie przyjmuje po prostu wartość numeryczną i jednostkę w formie łańcucha znaków i w podobnym formacie je wyświetla.

Poza stanem klasy PerformanceInformation, kluczowa jest logika odpowiedzialna za zamianę odpowiedzi z silnika na instancję klasy. Zdecydowano rozwiązać problem trzema wyrażeniami regularnymi – po jednym na każdą metrykę wydajnościową i metodą TryParse; jest to analogiczne rozwiązanie projektowe do tych z .NET Framework, np. dla struktury Double. Poniżej zaprezentowano wyrażenia regularne.

```

private static readonly Regex AiGetMovePerfCallbackRegex = new Regex(
    @"^.*ai.+?move.+?(" + NumberFormat + @")\s*(\w*)",
    RegexOptions.IgnoreCase | RegexOptions.Compiled);

private static readonly Regex GameCheckWinPerfCallbackRegex =
    new Regex(@"^.*checkwin.+?(" + NumberFormat + @")\s*(\w*)",
    RegexOptions.IgnoreCase | RegexOptions.Compiled);

private static readonly Regex GameGetMovesPerfCallbackRegex =
    new Regex(@"^.*getmoves.+?(" + NumberFormat + @")\s*(\w*)",
    RegexOptions.IgnoreCase | RegexOptions.Compiled);

```

Wyrażenia te można by rozdzielić na trzy części. Występuje jakiś identyfikator metryki (np. checkwin), potem wartość numeryczna w grupie przechwytyjącej i po dowolnej ilości białych znaków jednostka będąca słowem (ale dopuszcza się również brak jednostki). Identyfikator rozpoznawany jest bardzo liberalnie – dopuszcza się inne ciągi znaków przed nim i po nim, całe wyrażenie ignoruje wielkość liter itp. Jednostka to po prostu jakieś słowo po dowolnej ilości spacji po numerze, ale może nawet nie wystąpić. Najciekawsza zatem jest wartość numeryczna – jej format jest ukryty pod stałą NumberFormat, którą pokazano poniżej.

```
private const string NumberFormat = @"[+-]?\d+(?:[.,]\d+)?(?:[Ee][+-]?\d+)?";
```

Autor postarał się tutaj o dopuszczenie maksymalnej liczby sensownych formatów liczbowych – stąd długość tego wyrażenia. Przed liczbą może się znajdować plus lub minus (ale nie jest to warunek konieczny), następnie musi wystąpić przynajmniej jedna cyfra. Kolejna część jest opcjonalna – po przecinku lub kropce (separator dziesiętny) musi wystąpić przynajmniej jedna cyfra (oczywiście cały blok może w ogóle nie wystąpić, ale jeżeli jest separator dziesiętny, to musi być też cyfra). Ostatnia część również jest opcjonalna – to zapis notacji wykładniczej z wykorzystaniem E lub e. Musi więc wystąpić litera E lub e, następnie może wystąpić jeden plus lub minus i musi wystąpić przynajmniej jedna cyfra. Założono tutaj, że wykładnik jest liczbą całkowitą, ale pomimo tego to wyrażenie regularne dopuszcza naprawdę o wiele więcej formatów liczbowych niż potrzeba.

Metoda TryParse z kolei nie zawiera zbyt dużo logiki – przede wszystkim, jest to sprawdzenie, czy łańcuch znaków spełnia wszystkie trzy wyrażenia regularne, tzn. czy zawiera informacje o trzech metrykach wydajnościowych. Jeżeli nie zawiera chociaż jednej, to metoda TryParse zwraca false. W innym wypadku informacje o wszystkich trzech metrykach są tworzone na podstawie grup przechwytyujących – wartości numeryczne są przetwarzane na liczby typu double za pomocą metody double.Parse z użyciem kultury globalnej, a jednostka to po prostu przechwycone przez drugą grupę słowo.

### 3.5.4 Wyliczenie Player

Założenie, że w grze mamy dwóch graczy – czarny i biały, znacznie upraszcza logikę w niektórych miejscach. Wyliczenie Player jest właśnie jednym z taki przypadków. Oczywiście samo w sobie jest bardzo proste, jak widać poniżej.

```
public enum Player
{
    Black,
    White
}
```

Ale również metody rozszerzające (z racji statyczności nie są to tak naprawdę metody rozszerzające) zostały radykalnie uproszczone. Nie zdecydowano się tutaj nawet na użycie wyrażeń regularnych – poza zwykłym wycięciem spacji i innych białych znaków. Po tzw. operacji normalizacji, czyli wycięcia właśnie białych znaków i zamiany stringa na małe litery, zależnie od jego wartości ustalamy wartość zmiennej player. Należy tutaj zauważyć, że metoda TryParse została napisana na tę samą modę, co odpowiadające jej metody z .NET Framework, np. TryParse dla struktury Double.

```
public static class PlayerExtensions
{
    public static bool TryParse(string str, out Player player)
    {
        player = default(Player);
        str = Regex.Replace(str.ToLowerInvariant(), @"\s+", "");
        if (str == "black")
        {
```

```

        player = Player.Black;
        return true;
    }
    if (str == "white")
    {
        player = Player.White;
        return true;
    }
    return false;
}
}

```

Oczywiście, jeżeli nie udało się przetworzyć łańcucha znaków na wartość enuma player, to zwracamy false. W przypadku poprawnego parsowania zwracamy prawdę.

### 3.5.5 Klasa Move

Nieco bardziej złożoną klasą jest klasa Move. Przechowuje ona wartości współrzędnych x,y jako propercie X,Y, ale ponieważ jest zaprojektowana pod odpowiedzi z silnika, przechowuje również informację o tym, kto tego ruchu dokonał – propecję Player.

```

public Move(byte x, byte y)
{
    X = x;
    Y = y;
}

public byte X { get; set; }
public byte Y { get; set; }
public Player Player { get; set; }

```

W celu dokładnego odwzorowania zachowania silnika użyto 8-bitowych zmiennych typu byte do przechowywania współrzędnych (zakres 0 do 255). Prosty konstruktor nie przyjmuje jednak enuma player, ponieważ klasa jest używana również w miejscach, gdzie wartość Player nie ma dużego sensu – np. dostępne lub wygenerowane ruchy.

Jak wspomniano wcześniej, kluczowe jest przetwarzanie odpowiedzi z silnika na instancję obiektu Move. Podobną metodą, jak w innych klasach z biblioteki EngineWrapper, napisano więc odpowiednie wyrażenie regularne i metodę TryParse.

```

private static readonly Regex MoveCallbackRegex = new
Regex(@"\s*move\s+(black|white)\s+(\d+)\s+(\d+)\s*",
      RegexOptions.IgnoreCase | RegexOptions.Compiled);

public static bool TryParse(string str, out Move move)
{
    var match = MoveCallbackRegex.Match(str);
    if (match.Success)
    {
        move = new Move(byte.Parse(match.Groups[2].Value),
                      byte.Parse(match.Groups[3].Value));
        Player player;
        if (PlayerExtensions.TryParse(match.Groups[1].Value, out player))
        {
            move.Player = player;
        }
    }
}

```

```

        return true;
    }
}
move = null;

if (str.ToLowerInvariant().Contains("invalid move"))
    return true;

return false;
}

```

Wyrażenie regularne dopuszcza dosyć liberalny format – nie jest wrażliwe na wielkość liter ani na białe znaki. Wymaga jedynie określenia, że jest to ruch, kto dokonał tego ruchu i jakie są współrzędne.

W metodzie TryParse najpierw sprawdzamy oczywiście, czy testowany łańcuch znaków spełnia wyrażenie regularne. Następnie dzięki zastosowanym w wyrażeniu grupom przechwytyującym parsujemy kolejne części – dwie odpowiedzialne za współrzędne X i Y jako normalne liczby naturalne oraz część oznaczającą kolor gracza za pomocą metody opisanej w poprzednim podrozdziale – metody TryParse dla enuma Player. Jeżeli w jakimkolwiek TryParse wartością zwracaną będzie false, to cały TryParse się nie powiedzie dla ruchu, chyba że zawiera napis “invalid move”, co jest przypadkiem szczególnym – odpowiedzią silnika na niepoprawny ruch – wtedy wartość parametry out move jest ustawiana na null, żeby oznaczyć niepoprawność takiego ruchu.

Z przydatnych jeszcze metod klasa Move implementuje interfejs IEquatable<Move>, co oznacza przeciążenie metod Equals i operatorów równości

Przydatną jeszcze metodą jest IsAdjacent, umożliwiającą stwierdzanie, czy dwa ruchy ze sobą sąsiadują. Dwa ruchy ze sobą sąsiadują wtedy i tylko wtedy, gdy różnią się na każdej ze współrzędnych maksymalnie o jeden.

### 3.5.6 Wyliczenie GameState

Kolejne proste wyliczenie, tym razem pozwalające rozpoznawać, w jakim stanie znajduje się gra. Przewidziano stany określające kolejno: gra jeszcze się nie zaczęła, gra właśnie wystartowała, wygrał gracz czarny, wygrał gracz biały i nastąpił remis. Jednocześnie można zauważyc, że trzy ostatnie stany to także stany określające koniec gry. Stąd przewidziano enum GameState, ale również metodę rozszerzającą do niego IsGameOver(). Kod zaprezentowano poniżej.

```

public enum GameState
{
    NotStarted,
    Started,

    WinnerIsBlack,
    WinnerIsWhite,
    Draw
}

```

```

public static class GameStateExtensions
{
    public static bool IsGameOver(this GameState gs)
    {
        return gs == GameState.WinnerIsBlack || gs == GameState.WinnerIsWhite
        || gs == GameState.Draw;
    }
}

```

Tym razem jest to metoda rozszerzająca z prawdziwego zdarzenia – rzeczywiście operuje na instancji GameState.

Oddzielną kwestią jest natomiast przetwarzanie odpowiedzi z silnika na wartości enuma GameState, które już jest metodą statyczną, ale nierozszerzającą. Duże znaczenia mają tutaj zdefiniowane wyrażenia regularne.

```

private static readonly Regex GameStartedCallbackRegex = new
Regex(@"\s*game\s+started\s*",
      RegexOptions.IgnoreCase | RegexOptions.Compiled);

private static readonly Regex GameEndedCallbackRegex = new
Regex(@".*has\s+exited\s*",
      RegexOptions.IgnoreCase | RegexOptions.Compiled);

private static readonly Regex DrawCallbackRegex = new
Regex(@"\s*draw\s*",
      RegexOptions.IgnoreCase | RegexOptions.Compiled);

private static readonly Regex WinnerIsCallbackRegex = new
Regex(@"\s*winner\s+is\s+(black|white)\s*",
      RegexOptions.IgnoreCase | RegexOptions.Compiled);

```

Wykryć remis jest bardzo łatwo, podobnie jest z końcem i początkiem gry – po prostu szukamy konkretnych łańcuchów znaków przedzielonych niezerową liczbą spacji i otoczonych dowolną liczbą spacji. W przypadku końca gry dopuszczałybyśmy dowolne znaki przed komunikatem „has exited”, ponieważ w silniku pada tutaj jego nazwa. Metoda TryParse – bardzo podobna jak w innych klasach – została zaprezentowana poniżej.

```

public static bool TryParse(string message, out GameState gs)
{
    gs = default(GameState);

    if (GameStartedCallbackRegex.IsMatch(message))
    {
        gs = GameState.Started;
        return true;
    }

    var match = WinnerIsCallbackRegex.Match(message);
    if (match.Success)
    {
        Player player;
        if (PlayerExtensions.TryParse(match.Groups[1].Value, out player))
        {
            if (player == Player.Black)
            {

```

```

        gs = GameState.WinnerIsBlack;
        return true;
    }
    if (player == Player.White)
    {
        gs = GameState.WinnerIsWhite;
        return true;
    }
}
}

match = DrawCallbackRegex.Match(message);
if (match.Success)
{
    gs = GameState.Draw;
    return true;
}

return false;
}

```

Po kolej sprawdza się, czy łańcuch znaków będący odpowiedzią z silnika pasuje do wyrażenia regularnego. Jeżeli pasuje do jakiegokolwiek, to jest zwracana wartość reprezentująca prawdę, a parametr wyjściowy typu GameState ustawiany jest na odpowiadającą danemu regexowi wartość. W przypadku dopasowania do regixa WinnerIsCallbackRegex przetwarzana jest jeszcze grupa przechwytyująca, która reprezentuje informacje o kolorze gracza, który zwyciężył.

### 3.5.7 Klasa EngineWrapper i wyliczenie WrapperMode, GameType

Główna klasą w bibliotece (m,n,k,p,q)EngineWrapper jest klasa EngineWrapper – odpowiada ona za obudowanie całego silnika w wygodne do użytkowania z punktu widzenia programisty .NET-a funkcje i własności. Jeżeli uruchomiony jako aplikacja konsolowa silnik to warstwa 0, a klasa ProcessInBackground go obudowująca to warstwa 1, wtedy klasa EngineWrapper z pewnością jest warstwą 2, gdyż obudowuje ProcessInBackground.

Opisywane wcześniej klasy biblioteki służyły głównie do przechowywania/reprezentacji danych i miały raczej niewiele logiki. Inaczej jest z EngineWrapper – ona korzysta z wcześniejszych opisywanych klas, ale poza tym zawiera znaczącą ilość logiki, ważnej z punktu widzenia osoby chcącej skorzystać z biblioteki. Zaczniemy jednak od stanu, czyli pól i własności.

```

private readonly ProcessInBackground _engine;

private readonly Action<GameState> _gameStateChangedCallback;
private readonly Action<Move> _moveMadeCallback;

private bool _gameOver;
private WrapperMode _mode = WrapperMode.Async;

private ConcurrentQueue<string> _messages = new
ConcurrentQueue<string>();

private readonly List<string> _engineOutputs = new List<string>();
private readonly List<Move> _movesOutput = new List<Move>();

public string EngineName { get; }

```

Jednym z najbardziej podstawowych pól jest tutaj wspomniany ProcessInBackground, czyli silnik działający w tle \_engine, nazwa silnika EngineName i flaga \_gameOver oznajmująca czy rozgrywka się zakończyła. Istotne są dwa delegaty Action, pod które można się podpiąć przez konstruktora, aby otrzymywać informacje o zmianie stanu gry (\_gameStateChangedCallback) i o tym, że został zrealizowany ruch (\_moveMadeCallback). Z kolei tryb działania wrapper-a – \_mode może przyjmować wartości reprezentujące współpracę asynchroniczną (przez wspomniane callbacki) i synchroniczną – przez kolejkę wiadomości. Do wspomnianej kolejki wiadomości służy kolejka równoległa (bo z różnych wątków można czytać i pisać do niej) ConcurrentQueue \_messages. Listy wyjść z silnika \_engineOutputs i \_movesOutputs służą właściwie tylko do debugowania. Poniżej pokazano konstruktor.

```
public EngineWrapper(string path, Action<GameState>
gameStateChangedCallback, Action<Move> moveMadeCallback)
{
    _gameStateChangedCallback = gameStateChangedCallback;
    _moveMadeCallback = moveMadeCallback;
    _engine = new ProcessInBackground(path, "", CallbackHandler, true);
    EngineName = Path.GetFileName(path);
}
```

Jest on stosunkowo prosty – przyjmuje i przypisuje dwa wspomniane wcześniej delegaty oraz ścieżkę do pliku wykonywalnego silnika, który opakowuje w ProcessInBackground. Również ze ścieżki jest pozyskiwana nazwa silnika jako nazwa pliku. Dodatkowo funkcja klasy EngineWrapper nazwana CallbackHandler jest przekazywana do ProcessInBackground w celu asynchronicznej obsługi odpowiedzi silnika. Funkcja ta jest pokazana poniżej.

```
public void CallbackHandler(string message)
{
    _engineOutputs.Add(message);

    if (_mode == WrapperMode.Sync)
    {
        _messages.Enqueue(message);
    }
    else
    {
        Move move;
        if (Move.TryParse(message, out move))
        {
            _movesOutput.Add(move);
            _moveMadeCallback?.Invoke(move);
            return;
        }

        GameState gs;
        if (GameStateExtensions.TryParse(message, out gs))
        {
            _gameStateChangedCallback?.Invoke(gs);
            _gameOver = gs.IsGameOver();
        }
    }
}
```

Zgodnie z tym, co powiedziano wcześniej, obsługa odpowiedzi z silnika działa inaczej dla trybu synchronicznego, a inaczej dla asynchronicznego. W pierwszym przypadku obsługa komunikatu z silnika jest bardzo prosta – dodajemy go do kolejki wiadomości. W drugim natomiast próbujemy sparsować go jako ruch – jeżeli się uda, to wołany jest podany wcześniej do konstruktora delegat informujący o tym, że jakiś gracz wykonał ruch. Jeżeli się nie uda, to drugim wariantem jest zmiana stanu gry – identycznie jak w poprzednim przypadku, tylko że inny delegat. W przypadku, gdy żadne parsowanie się nie powiedzie, to nic nie jest robione – wszak aplikację GUI interesują tylko te zdarzenia. Zawsze jednak (niezależnie od trybu) odpowiedź z silnika dodawana jest na listę odpowiedzi z silnika – głównie po to, aby w debug-u podejrzeć, jakie odpowiedzi kolejno przychodząły. Podobnie dla trybu asynchronicznego – poza wysyłaniem ruchu delegatem zapisywany on jest na listę ruchów otrzymanych z silnika.

Aby jednak pojawiały się jakiekolwiek odpowiedzi z silnika – nieważne, czy synchroniczne, czy asynchroniczne, należy najpierw silnik ten uruchomić – zgodnie z implementacją klasy `ProcessInBackground`. Służy do tego prosta metoda `Run()` – sama w sobie jednak poza uruchomieniem silnika jako proces działający w tle jedynie wywołuje delegat `_gameStateChangedCallback`. Należy jednak zauważyć, że zastosowano tutaj wyrażenie „?.”, czyli w przypadku, gdy callback jest Nullem, to po prostu nic się nie wywoła – nie zostanie rzucony wyjątek null pointer exception [79].

```
public void Run()
{
    _engine.Run();
    _gameStateChangedCallback?.Invoke(GameState.NotStarted);
}
```

Wywołanie tej metody to jedynie uruchomienie silnika jako procesu, a nie rozpoczęcie gry. Do rozpoczęcia gry przeznaczona jest metoda `StartGame`. Rozpoczyna ona grę, wysyłając do silnika komendę “newgame” z odpowiednimi argumentami. Argumenty te powstają na podstawie argumentu będącego enumem, przekazanego do owej metody. W chwili rozpoczęcia nowej gry czyszczona jest również kolejka komunikatów i w przypadku pracy w trybie asynchronicznym włączany jest na czas całej operacji tryb synchroniczny. Metoda czeka również przed swoim zakończeniem na odpowiedź z silnika, świadcząc o rozpoczęciu gry. Wysyłany jest również sygnał do odpowiedniego delegata informujący o zmianie stanu gry na rozpoczętą

```
public void StartGame(GameType gameType)
{
    _messages = new ConcurrentQueue<string>();

    var restoreAsync = false;
    if (_mode == WrapperMode.Async)
    {
        restoreAsync = true;
        StopAsync();
    }

    _gameOver = false;
```

```

        switch (gameType)
    {
        case GameType.TwoHumans:
            _engine.Send("newgame black human white human");
            break;
        case GameType.BlackHumanVsWhiteAi:
            _engine.Send("newgame black human white ai");
            break;
        case GameType.BlackAIVsWhiteHuman:
            _engine.Send("newgame black ai white human");
            break;
        case GameType.TwoAIs:
            _engine.Send("newgame black ai white ai");
            break;
        default:
            throw new ArgumentOutOfRangeException(nameof(gameType),
gameType, null);
    }
    while (!GetLine().Contains("game started")) { }

    _gameStateChangedCallback?.Invoke(GameState.Started);

    if (!restoreAsync)
        StartAsync();
}

```

GameType to tak naprawdę bardzo proste wyliczenie zawierające wszystkie możliwe tryby gry dwuosobowej.

```

public enum GameType
{
    TwoHumans,
    BlackHumanVsWhiteAi,
    BlackAIVsWhiteHuman,
    TwoAIs
}

```

Kolejności graczy nie musimy ustalać, bo reguły gier z rodziny (m,n,k,p,q) mówią, że grę zawsze zaczyna gracz czarny. Cztery wartości wyliczenia to wystarczająca ilość.

Z kolei na koniec gry należy zamknąć aplikację silnika – można to zrobić komendą „quit” wysyłaną do procesu. Dokładnie tę funkcjonalność wraz ze sprawdzeniem, czy rzeczywiście silnik zakończył swoją pracę, realizuje metoda Close().

```

public void Close()
{
    StopAsync();
    _engine.Send("quit");
    while (!GetLine().ToLowerInvariant().Contains("has exited"))
    {
    }
}

```

Należy zauważyć, że tutaj już można bezkarnie korzystać z trybu synchronicznego, zatrzymując tryb asynchroniczny metodą StopAsync() – zakończymy przecież pracę silnika, więc nie ma

znaczenia wznowienie jego pracy w trybie asynchronousznym, jeżeli w takim wcześniej się znajdował.

Co jednak wtedy, gdy zapomnimy o włączonej instancji silnika, uruchamiając w aplikacji ich bardzo wiele? Jeżeli mamy nieużywane obiekty w aplikacji, to wcześniej czy później zostaną one sprzątnięte przez Garbage Collector. Sprząta on jednak jedynie zarządzalne zasoby, a uruchomiony proces jest tak naprawdę zasobem natywnym. Na szczęście w .NET Framework dla każdego obiektu można przeciążyć metodę `Dispose()` wywoływaną przez GC przed sprzątaniem obiektu w celu zwolnienia innych zasobów (natywnych i zarządzalnych), nie sprzątanych przez GC. Dokładnie to właśnie uczyniono w klasie `EngineWrapper`.

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // free managed resources
    }
    // free native resources if there are any.
    Close();
}
```

W `Dispose()` wywoływana jest metoda `Close()`, która kończy działanie aplikacji silnika – jest to bardzo proste, a zarazem sprytne rozwiązanie.

Zarówno w `StartGame`, jak i w `Close`, po włączeniu trybu synchronousznego sprawdzana była odpowiedź z silnika wywołaniem metody `GetLine()`. Metoda ta działa następująco – pobiera synchronicznie odpowiedź z silnika, ale ponieważ silnik w swoim działaniu jest generalnie asynchronousny (a przynajmniej tak jest obudowywany w klasie `ProcessInBackground`), synchroniczność ta jest tak naprawdę udawana. Udwawianie to jest realizowane z pomocą wcześniej wspomnianej kolejki komunikatów. O ile metoda `CallbackHandler` „wkłada” do tej kolejki odpowiedzi z silnika, o tyle metoda `GetLine()` wyciąga te odpowiedzi z kolejki zgodnie z zasadą FIFO (first in, first out).

```
public string GetLine()
{
    string ret;
    while (!_messages.TryDequeue(out ret))
    {
        Thread.Sleep(20);
    }

    return ret;
}
```

Metoda `GetLine()` to podstawa w wielu zapytaniach do silnika o naturze typowo synchronousznej – część z tych zapytań jest często zadawana po zakończonej rozgrywce lub jeszcze przed jej rozpoczęciem. Przykładem jest zapytanie o parametry silnika.

```

public EngineParameters GetEngineInfo()
{
    var restoreAsync = false;
    if (_mode == WrapperMode.Async)
    {
        restoreAsync = true;
        StopAsync();
    }

    _engine.Send("info");
    var info = GetLine();

    EngineParameters engineParameters;

    if (EngineParameters.TryParse(info, out engineParameters))
    {
        if (restoreAsync)
            StartAsync();
        return engineParameters;
    }
    throw new Exception($"engine info wrong {info}");
}

```

Używana jest wcześniej opisywana klasa EngineParameters do parsowania odpowiedzi, jak i jej przechowywania. Również analogicznie, jak we wcześniejszych przypadkach asynchroniczność, jeżeli występowała, to jest przywracana na koniec zapytania.

Podobnie sytuacja wygląda dla informacji o wydajności. Tutaj jednak jest ona podawana przez silnik w trzech oddzielnych liniach, dlatego mamy trzy wywołania metody GetLine().

```

public PerformanceInformation GetPerformanceInformation()
{
    var restoreAsync = false;
    if (_mode == WrapperMode.Async)
    {
        restoreAsync = true;
        StopAsync();
    }

    _engine.Send("perf");
    var perf = GetLine() + GetLine() + GetLine();

    PerformanceInformation pi;

    if (PerformanceInformation.TryParse(perf, out pi))
    {
        if (restoreAsync)
            StartAsync();
        return pi;
    }
    throw new Exception($"GetPerformanceInformation failed for {perf}");
}

```

Nieco inną klasą metod są metody pobierające synchronicznie dane, które są w naturalny sposób udostępniane przez silnik w systemie callbacków. Na przykład silnik w formie wywołań delegatów o wykonaniu ruchu oddaje tę informację w trybie asynchronicznym, ale klasa metod z przyrostkiem „Sync” pozwala również na pozyskanie tej informacji synchronicznie. Przykładem

jest właśnie metoda `GetMoveSync()` – pobierająca ostatnio dokonany ruch w trybie synchronicznym.

```
public Move GetMoveSync()
{
    StopAsync();
    Move move;
    while (!Move.TryParse(GetLine(), out move))
    {
    }
    return move;
}
```

Bardzo analogicznie wygląda metoda `GetGameStateSync()`.

```
public GameState GetGameStateSync()
{
    StopAsync();
    var line = GetLine();
    GameState state;
    while (!GameStateExtensions.TryParse(line, out state))
    {
        line = GetLine();
    }
    return state;
}
```

Są też metody, które mają sens w trakcie rozgrywki – zwłaszcza w trybie człowiek vs człowiek. Przykładem jest pobranie obecnego gracza (koloru).

```
public Player GetCurrentPlayer()
{
    var restoreAsync = false;
    if (_mode == WrapperMode.Async)
    {
        restoreAsync = true;
        StopAsync();
    }
    _engine.Send("getplayer");
    Player player;

    while (!PlayerExtensions.TryParse(GetLine(), out player)) { }

    if (restoreAsync)
        StartAsync();

    return player;
}
```

Nieco inna, bo bez dedykowanej klasy z logiką parsującą, jest metoda pozyskująca listę dostępnych ruchów. Metoda `GetMovesSync` posiada po prostu dedykowane wyrażenie regularne `GetMovesRegex` umieszczone w klasie `EngineWrapper`.

```
public IEnumerable<Move> GetMovesSync()
{
    var restoreAsync = false;
```

```

        if (_mode == WrapperMode.Async)
        {
            restoreAsync = true;
            StopAsync();
        }
        _engine.Send("getmoves");

        var line = "";
        while (!GetMovesRegex.IsMatch(line))
        {
            line = GetLine();
        }

        var moves = new List<Move>();

        var match = MoveFromGetMovesRegex.Match(line);
        while (match.Success)
        {
            moves.Add(new Move(byte.Parse(match.Groups[1].Value),
byte.Parse(match.Groups[2].Value)));
            match = match.NextMatch();
        }

        if (restoreAsync)
            StartAsync();
        return moves;
    }
}

```

Wysłana jest do silnika komenda „getmoves” – odpowiedź z silnika dopasowywana jest do regesta reprezentującego wolne ruchy. Następnie po kolej wolne ruchy z odpowiedzi dodawane są do listy wolnych ruchów za pomocą innego regesta – MoveFromGetMovesRegex. Operacja powtarzana jest tak długo, jak długo są kolejne dopasowane fragmenty do tegoż wyrażenia regularnego.

```

private static readonly Regex GetMovesRegex = new Regex(@"\s*moves:\s*");
private static readonly Regex MoveFromGetMovesRegex = new
Regex(@"\s*\(\s*(\d+)\s+(\d+)\s*\)\s*");

```

Pierwsze wyrażenie regularne szuka po prostu łańcucha znaków “moves:” otoczonego dowolną liczbą spacji – czyli rozpoczęcia sekwencji odpowiedzi zawierającej wolne ruchy. Drugie wyrażenie regularne to z kolei notacja wolnego ruchu w tejże odpowiedzi. Każdy wolny ruch w odpowiedzi oddzielony jest od reszty nawiasem, wewnątrz którego znajdują się kolejno współrzędne x i y przedzielone przynajmniej jedną spacją. Przykład: (2 3). Współrzędne są parsowane metodą byte.Parse i dodawane do listy ruchów. Poza umieszczeniem logiki zamieniającej łańcuch znaków na instancję obiektów wewnątrz, metoda jest generalnie bardzo zbliżona do pozostałych wspomnianych wcześniej.

Do kontroli trybu współpracy z silnikiem służą bardzo proste metody StartAsync i StopAsync.

```
public void StopAsync()
```

```

{
    _mode = WrapperMode.Sync;
}

public void StartAsync()
{
    _mode = WrapperMode.Async;
}

```

WrapperMode to dwuelementowe wyliczenie.

```

public enum WrapperMode
{
    Async,
    Sync
}

```

Wreszcie kluczowa metoda ze względu na wprowadzanie danych do silnika – MakeMove. Jednocześnie jest ona generalnie niezależna od trybu pracy z silnikiem – tak czy inaczej próbę wykonania ruchu należy przesyłać do silnika komendą makemove.

```

public void MakeMove(Move move)
{
    _engine.Send($"makemove {move.X} {move.Y}{Environment.NewLine}");
}

```

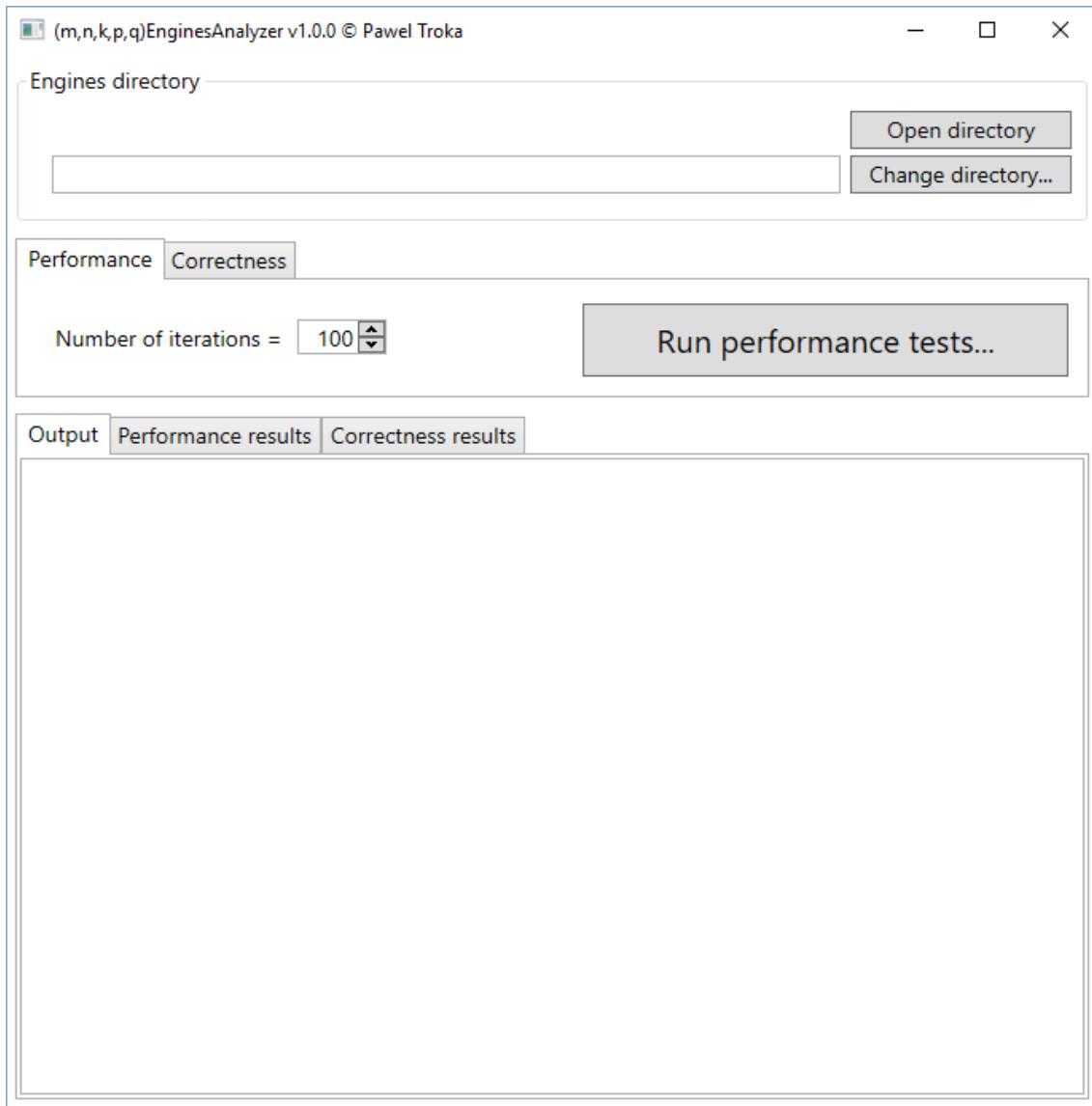
### **3.6 Aplikacja do analizy wydajnościowo-poprawnościowej generowanych silników – (m,n,k,p,q)EnginesAnalyzer**

W celu ułatwienia i, co ważniejsze, zautomatyzowania analizy generowanych przez generator silników, należało opracować aplikację testującą te silniki zarówno pod względem wydajności, jak i poprawności. Aplikacja (m,n,k,p,q)EnginesAnalyzer umożliwia taką automatyzację i łatwe testowanie silników. Nadal należy jednak pamiętać, że aplikacja ta dużo zawdzięcza bibliotece (m,n,k,p,q)EngineWrapper – używając jej do każdego testu, konstrukcja aplikacji testującej staje się o wiele prostsza.

Analiza poprawnościowo-wydajnościowa przeprowadzana jest w aplikacji (m,n,k,p,q)EnginesAnalyzer poprzez serie odpowiednio przygotowanych testów. Z pomocą biblioteki (m,n,k,p,q)EngineWrapper kolejne silniki ze wskazanej lokalizacji są uruchamiane. Następnie dla każdego silnika przeprowadzany jest każdy test z wybranej grupy. Aby zweryfikować poprawność wykonania testu, ma on wiedzę, jaka powinna być odpowiedź silnika na daną komendę, a w przypadku testów wydajności po prostu sprawdzany jest czas. Konkretne testy są opisane w rozdziale poświęconym analizie.

Jako że jest to rozdział implementacyjny, w pierwszej kolejności należy pokazać efekty implementacji. Poniżej przedstawiono, jak wygląda aplikacja tuż po uruchomieniu. W dolnej części aplikacji widoczne są zakładki powiązane z wyjściem aplikacji. Są to „Output”, czyli po prostu wyjście tekstowe informujące o tym, co aplikacja teraz robi lub zrobiła, „Performance results”, czyli wyniki testów wydajnościowych i „Correctness results”, czyli wyniki testów

poprawności. W środkowej części aplikacji są widoczne dwie zakładki „Performance” i „Correctness”, służą one ustawianiu parametrów i uruchamianiu odpowiednio testów wydajności i poprawności. W górnej części aplikacji można z kolei ustawić lokalizację, w której znajdują się już wygenerowane silniki, które chcemy przetestować.

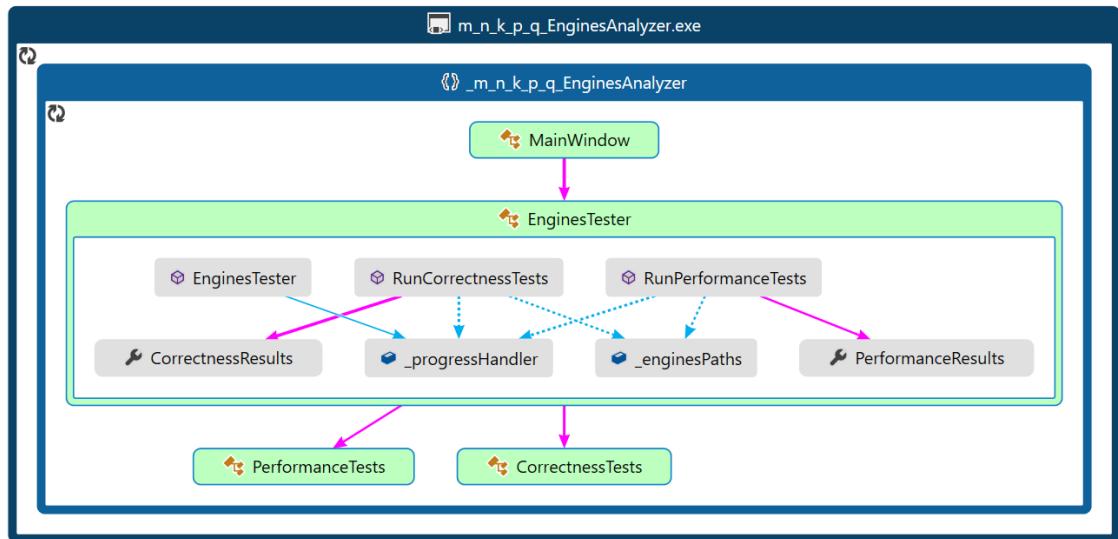


Rysunek 3.20. Aplikacja (m,n,k,p,q)EnginesAnalyzer tuż po uruchomieniu

Aplikacja została oczywiście napisana w C#.NET, korzystając ze stosu WPF, wygląd zaprojektowano w Xaml. Dbano o to, żeby była przejrzysta i łatwa w obsłudze. Ponieważ jednak jest to narzędzie potrzebne do realizacji celu pracy, a nie cel pracy sam w sobie, nie przywiązywano zbyt dużej uwagi do obiektywnej oceny wyglądu czy stylu aplikacji. Wszystkie użyte style są domyślnymi dla WPF-a.

Struktura kodu i niektórych klas aplikacji została z kolei przedstawiona na poniższym grafie zależności. Niektóre elementy pominięto, ale kluczowe jest to, że okno główne aplikacji korzysta z klasy EnginesTester, która zarządza ścieżkami do silników, uruchamianiem i oceną testów, ale sama w sobie nie ma wiedzy o testach. Wiedza o konkretnych testach ukryta jest w

kolejnych dwóch klasach PerformanceTests (testy wydajności) i CorrectnessTests (testy poprawności). Taki design ułatwia dalszy rozwój w przyszłości, polegający choćby na dodawaniu nowych testów czy grup testów.



Rysunek 3.21. Graf zależności w programie (m,n,k,p,q)EnginesAnalyzer

Oczywiście część klas i innych bytów została pominięta, żeby zmieścić to, co najistotniejsze, na grafie. W dalszej części pracy zostaną szerzej omówione najważniejsze elementy z tego grafu wraz z zaprezentowaniem efektów – części interfejsu korzystającej z danego bytu.

### 3.6.1 Klasa EnginesTester – testowanie silników

EnginesTester to główna klasa, jeżeli chodzi o sens całej aplikacji. Poniżej przedstawiono jej pola, proporcje i konstruktor. Przez konstruktor wstrzykiwany jest obiekt implementujący interfejs `IProgress<string>`, który służy oczywiście do zgłoszania postępów w formie komunikatów będących łańcuchami znaków. Te właśnie komunikaty są wyświetlane w obszarze „Output” interfejsu użytkownika, o czym wspomniano wcześniej.

```

public class EnginesTester
{
    private readonly IProgress<string> _progressHandler;
    private string[] _enginesPaths;

    public EnginesTester(IProgress<string> progressHandler)
    {
        _progressHandler = progressHandler;
        CorrectnessResults.Columns.AddRange(
            CorrectnessTests.GetTests().Select(t => new DataColumn(t.Name,
typeof(bool))).ToArray());
    }

    public DataTable PerformanceResults { get; } = new DataTable
    {
        Columns =
        {
            {"Engine", typeof(string)},

```

```

        {nameof(PerformanceInformation.AverageAiGetMoveExecution) + " ns",
        typeof(double)},
        {nameof(PerformanceInformation.AverageGetMovesExecution) + " ns",
        typeof(double)},
        {nameof(PerformanceInformation.AverageCheckWinExecution) + " ns",
        typeof(double)}
    }
};

public DataTable CorrectnessResults { get; } = new DataTable
{
    Columns =
    {
        {"Engine", typeof(string)}
    }
};

```

Jeżeli chodzi o pozostałe elementy stanu tej klasy, to istnieje zmienna prywatna `_enginesPaths`, służąca do przechowywania ścieżek do silników oraz dwie proporcje typu `DataTable`, służące do wygodnego – z punktu widzenia prezencji w interfejsie użytkownika – przechowywania wyników testów wydajności i poprawności. Właściwości typu `DataTable` posiadają inicjowane kolumny zgodnie z etykietami, które mają się w kolumnach pokazywać. W przypadku wyników wydajności sprawa jest dosyć prosta – są to nazwy odpowiednich metryk wydajnościowych z dodanym przyrostkiem, świadczącym o jednostce czasowej. Dla wyników poprawności jest to odrobinę bardziej skomplikowane. Pierwszą kolumną jest oczywiście i w tym przypadku silnik, ale kolejnymi są już konkretne nazwy testów zwracane z klasy `CorrectnessTests` przez metodę `GetTests()`.

Jedną z dwóch metod klasy `EnginesTester`, odpowiadającej odpowiedniej akcji w interfejsie jest `RunPerformanceTests()`. Uruchamia ona oczywiście na silnikach z danej lokalizacji i przy założeniu podanej w interfejsie liczby iteracji testy wydajności. Najpierw są czyszczone wyniki wydajności z poprzednich uruchomień metody, następnie są wczytywane wszystkie silniki (programy z rozszerzeniem `.exe`) z danej lokalizacji metodą `Directory.GetFiles()`. Po drodze są wysyłane komunikaty do UI.

```

public void RunPerformanceTests(string enginesDirectory, long iterations)
{
    PerformanceResults.Rows.Clear();

    _enginesPaths = Directory.GetFiles(enginesDirectory, @"*.exe");
    _progressHandler.Report(
        $"{Environment.NewLine}---- Testing performance of engines from
{enginesDirectory} ----{Environment.NewLine}");
    foreach (var path in _enginesPaths)
    {
        using (var engine = new EngineWrapper(path, null, null))
        {
            engine.Run();
            var performanceTests = new PerformanceTests(engine,
iterations);

            foreach (var test in performanceTests.GetTests())
            {

```

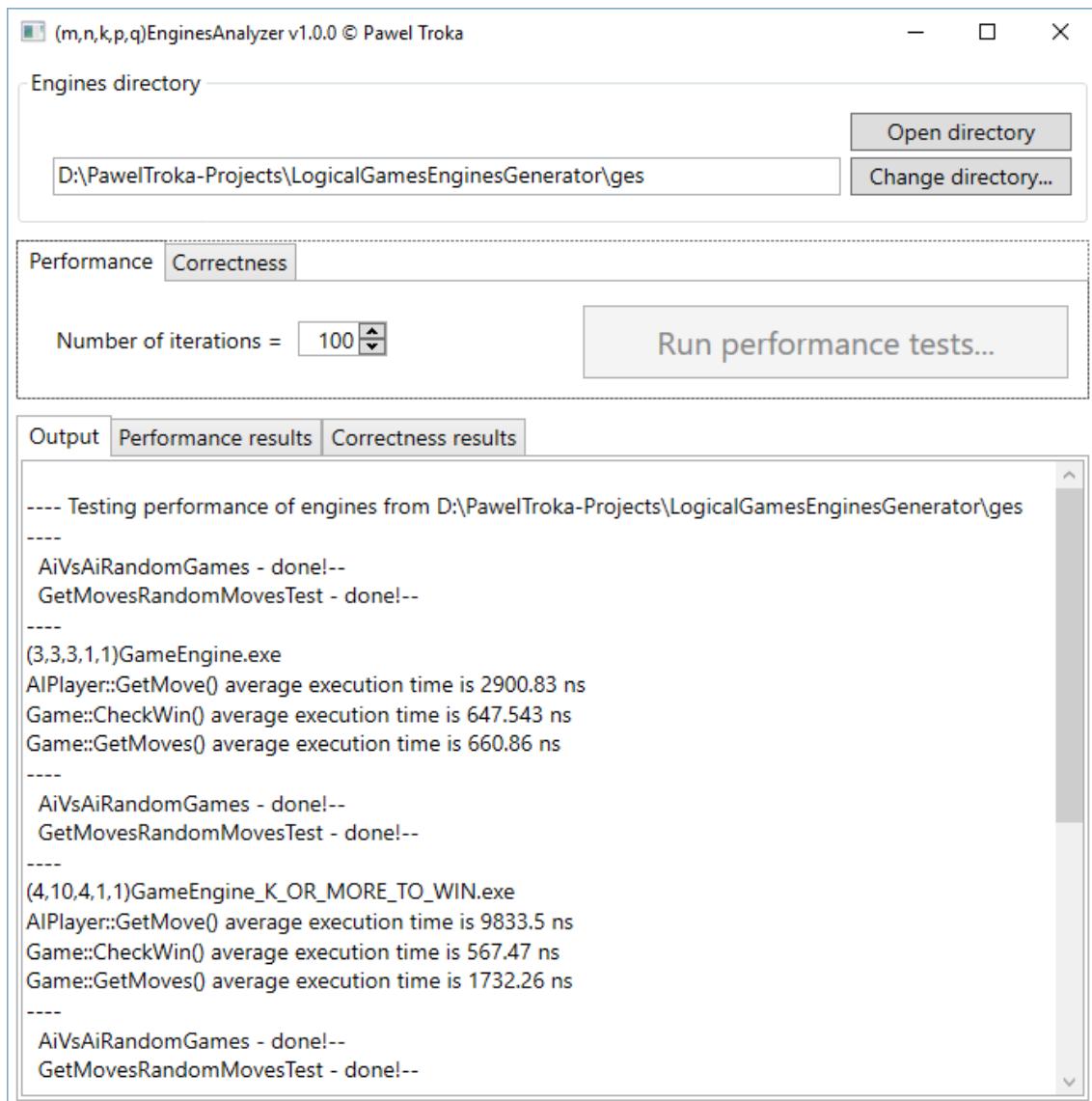
```
        test.Invoke(performanceTests, null);
        _progressHandler.Report($"  {test.Name} - done! --{Environment.NewLine}");
    }

    var pi = engine.GetPerformanceInformation();
    _progressHandler.Report(
        $"----{Environment.NewLine}{engine.EngineName}{Environment.NewLine}{pi}{Environment.NewLine}-----{Environment.NewLine}");
    PerformanceResults.Rows.Add(engine.EngineName,
        pi.AverageAiGetMoveExecution.Value,
        pi.AverageGetMovesExecution.Value, pi.AverageCheckWinExecution.Value);
    }
}
```

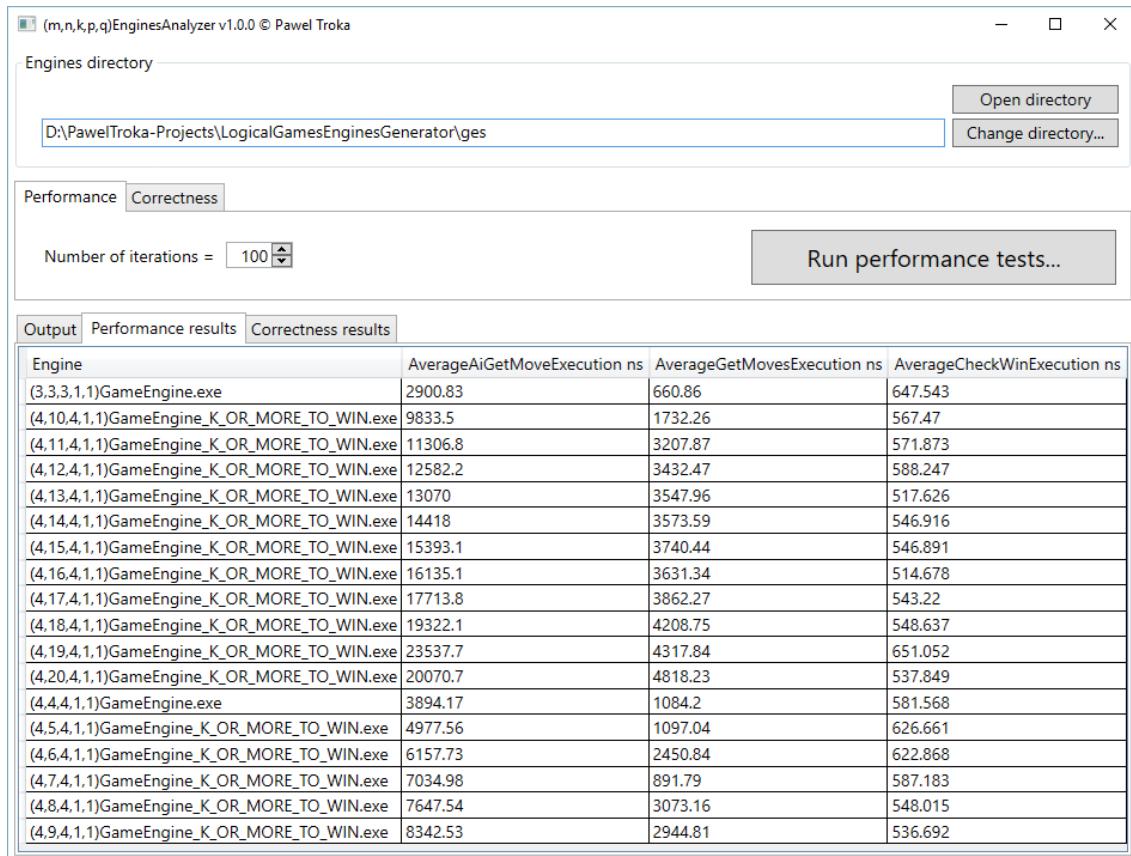
W głównej pętli metody iteruje się po kolejnych ścieżkach do silników i tworzy się dla każdego silnika wrapper (klasa EnginesWrapper opisana w poprzednim rozdziale). Nowo stworzony obiekt EngineWrapper uruchamia silnik, a następnie tworzona jest instancja klasy PerformanceTests na podstawie wrappera i iteracji. Instancja klasy PerformanceTests zwraca metodą GetTests() konkretne testy, które możemy uruchomić i po zakończeniu każdego testu wypisać komunikat do UI. Po zakończeniu wszystkich testów dla danego silnika pobiera się z niego informacje o wydajności, które wpisuje się do UI i dodaje do rezultatów.

Należy również zauważyć, że obiekt klasy EngineWrapper uruchamiający i współpracujący z silnikiem jest tworzony w bloku using, co oznacza to, że po wyjściu z bloku zostanie on sprzątnięty i zostanie m.in. wywołana na nim metoda Dispose() (bo implementuje interfejs IDisposable), która posprząta również natywne zasoby [81] [82] [83] – w tym przypadku wyłączy działający w tle proces silnika. Implementacja Dispose() dla EngineWrapper została opisana w poprzednim rozdziale.

Poniżej pokazano, jak wygląda główne okno aplikacji w trakcie wykonywania testów i orazdatagrid z wynikami testów wydajności po ich zakończeniu.



Rysunek 3.22. Aplikacja (m,n,k,p,q)EnginesAnalyzer w trakcie testowania wydajności silników



Rysunek 3.23. Aplikacja (m,n,k,p,q)EnginesAnalyzer po zakończeniu testowania wydajności silników

Kolejną metodą klasy EnginesTester jest RunCorrectnessTests(), która uruchamia testy poprawności dla silników z danej lokalizacji. Analogicznie, jak w przypadku poprzedniej metody, również tutaj najpierw czyścimy wyniki testów z poprzedniego uruchomienia metody, uzyskujemy ścieżki do wszystkich silników i raportujemy postęp do UI. Różnica jest taka, że tworzymy dwa liczniki – licznik przeprowadzonych testów i licznik testów zakończonych sukcesem.

```

public void RunCorrectnessTests(string enginesDirectory)
{
    CorrectnessResults.Rows.Clear();

    _enginesPaths = Directory.GetFiles(enginesDirectory, @"*.exe");
    _progressHandler.Report(
        $"{Environment.NewLine}---- Testing correctness of engines from
{enginesDirectory} ----{Environment.NewLine}");
    var totalCount = 0;
    var successCount = 0;
    foreach (var path in _enginesPaths)
    {
        using (var engine = new EngineWrapper(path, null, null))
        {
            _progressHandler.Report($"---- Testing engine:
{engine.EngineName}{Environment.NewLine}");
            engine.Run();

            var correctnessTests = new CorrectnessTests(engine);
            var tests = correctnessTests.GetTests().ToArray();

            var row = new List<object>() {engine.EngineName};

```

```

        row.AddRange(Enumerable.Repeat((object)false, tests.Length));
        CorrectnessResults.Rows.Add(row.ToArray());

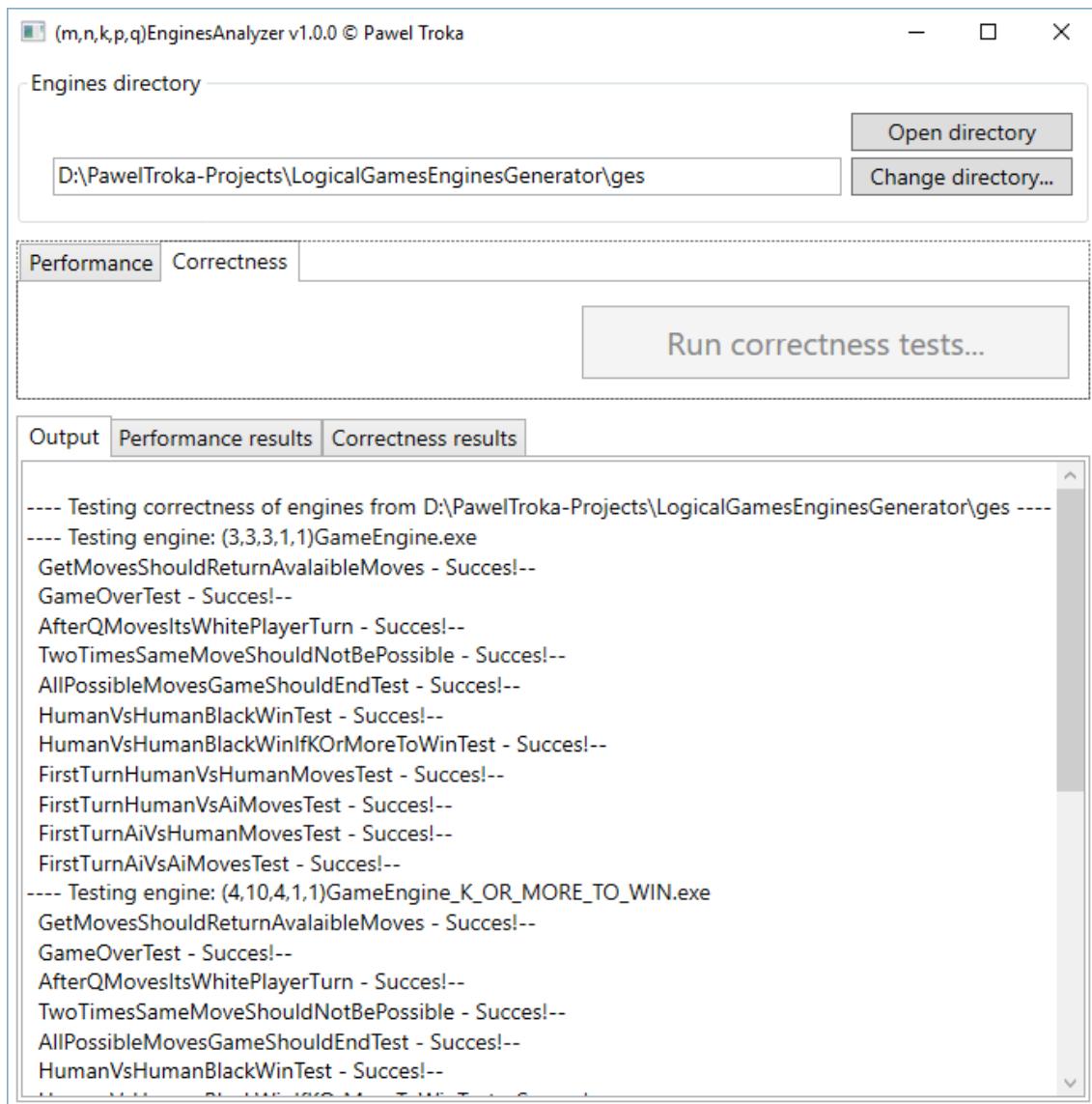
        for (int index = 0; index < tests.Length; index++)
        {
            var correctnessTest = tests[index];
            testCount++;
            var result = (bool)
correctnessTest.Invoke(correctnessTests, null);
            if (result) successCount++;
            _progressHandler.Report(
                $" {correctnessTest.Name} - {(result ? "Success!" :
"Failed...")}{Environment.NewLine}");
            CorrectnessResults.Rows[CorrectnessResults.Rows.Count -
1][index + 1] = result;
        }
    }
    _progressHandler.Report($"{Environment.NewLine}---- Correctness tests
{successCount}/{testCount} succeeded!");
}

```

Iterujemy po wszystkich ścieżkach do silników z danej lokalizacji i dla każdej ścieżki tworzymy wrapper na silnik. Silnik jest uruchamiany, obiekt reprezentujący testy poprawności dla niego tworzony, a UI jest aktualizowane o postęp.

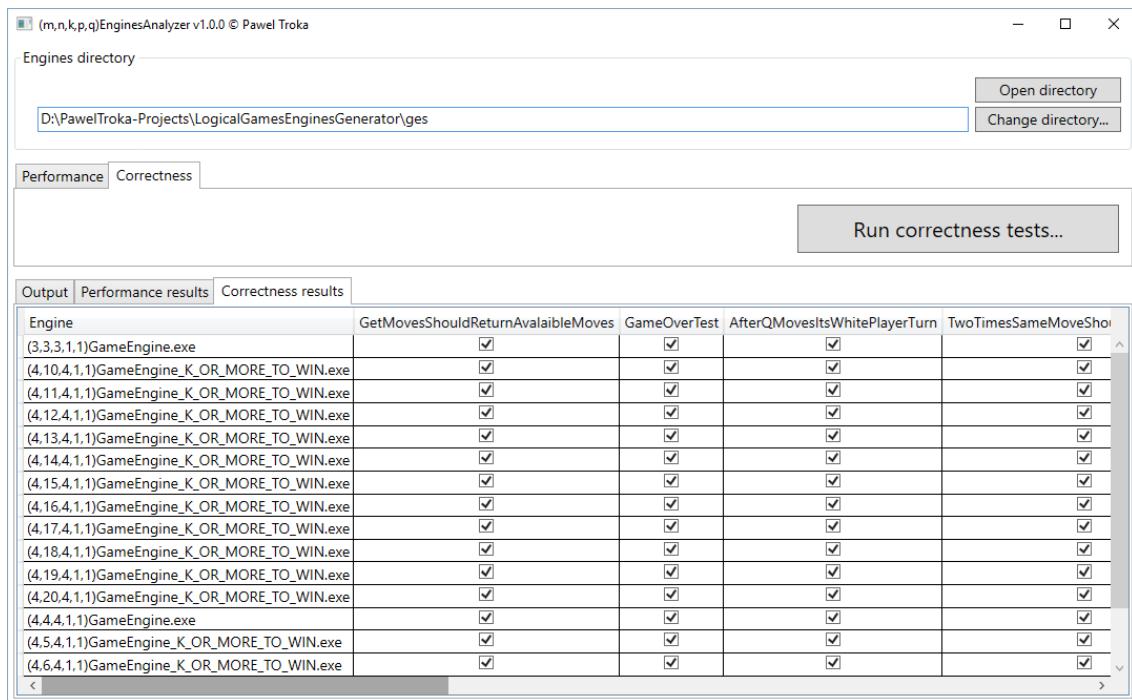
Następnie pojawia się kolejna różnica w stosunku do testów wydajności. Tworzona jest lista zmiennych typu object, będąca tak naprawdę wierszem rezultatów testów poprawności dla danego silnika. Na początku wszystkie rezultaty są false, czyli dany test się nie powiodł. Dopiero teraz iteruje się po wszystkich testach poprawności, uruchamia się je na danym silniku i sprawdza rezultat. Testy poprawności zwracają true w przypadku sukcesu i false w przypadku nieprzejścia testu przez silnik. Rezultaty te są umieszczane w kolejnych kolumnach właśnie dodanego wiersza (poza pierwszą kolumną, w której jest nazwa silnika). Również liczniki przeprowadzonych testów i testów zakończonych powodzeniem są aktualizowane. Różnica w stosunku do testów wydajności wynika z tego, że tam założyliśmy, że wiemy, że są trzy i tylko trzy metryki wydajności i wszystkie testy się uruchamia, a dopiero potem zbiera te rezultaty. Tutaj natomiast nie wiadomo w trakcie komplikacji, ile jest testów, poza tym trzeba iterować po testach i dla każdego wpisać jego rezultat, a nie po prostu je uruchomić.

Oczywiście cały czas każdy wynik wysyłany jest do UI w celu wyświetlania na bieżąco postępu przeprowadzania testów. Poniżej pokazano, jak to wygląda w trakcie pracy aplikacji testującej.



Rysunek 3.24. Aplikacja (m,n,k,p,q)EnginesAnalyzer w trakcie testowania poprawności silników

Na koniec wyświetlany jest komunikat pokazujący, ile testów się powiodło w stosunku do całkowitej ilości wykonanych testów. Poniżej pokazano, jak wygląda datagrid reprezentujący wyniki testów poprawności po ich zakończeniu.



Rysunek 3.25. Aplikacja (m,n,k,p,q)EnginesAnalyzer po zakończeniu testowania poprawności silników

Widok ten może być mniej lub bardziej czytelny – zależy to od liczby testów. W przypadku ostatecznej wersji aplikacji jest ich 11, wymagane jest więc przewinięcie widoku w prawo, aby sprawdzić wszystkie wyniki dla danego silnika. Pomimo to można dosyć czytelnie i łatwo sprawdzić w ten sposób rezultaty dla wszystkich testowanych silników i uruchamianych testów.

### 3.6.2 Klasa CorrectnessTests – testy poprawności

Tak jak wcześniej wspomniano, klasa EnginesTester testująca silniki nie ma tak naprawdę żadnej wiedzy na temat testów. Zna ich sygnaturę, wie, jak je uruchomić, ale to wszystko. Cała logika związana z testami wraz z ich treścią jest zawarta w dedykowanych klasach, takich jak opisywana w tym rozdziale klasa CorrectnessTests, zawierająca testy poprawności.

Poniżej pokazano pola tejże klasy. Jak widać, są to: parametry silnika, silnik reprezentowany przez obiekt klasy EngineWrapper i tablica możliwych dla danego silnika do wykonania ruchów. Wszystkie trzy pola mają spore znaczenie z punktu widzenia testów poprawności, są po prostu potrzebne w niektórych testach bardziej lub mniej.

```
public class CorrectnessTests
{
    private readonly EngineWrapper _engine;
    private readonly EngineParameters _engineParameters;
    private readonly Move[] _possibleMoves;
```

Kolejnym istotnym fragmentem kodu opisywanej klasy jest jej konstruktor. Przyjmuje on sam silnik, który ma być testowany w reprezentacji przez wrapper. Silnik ten jest wstrzykiwany przez

konstruktor, ale również w konstruktorze na podstawie wrappera do silnika inicjowane są kolejne pola. Najpierw wywołaniem metody GetEngineInfo() na obiekcie typu EngineWrapper uzyskuje się podstawowe informacje na temat parametrów silnika, a następnie dzięki tej informacji tworzy się tablice wszystkich możliwych ruchów. Oczywiście można by po prostu wołać GetEngineInfo() wtedy, gdy jest potrzebne w danym teście, i dopiero potem generować listę możliwych ruchów, ale byłoby to bardziej kosztowne obliczeniowo.

```
public CorrectnessTests(EngineWrapper engine)
{
    _engine = engine;
    _engineParameters = _engine.GetEngineInfo();

    var possibleMovesList = new List<Move>();

    for (byte x = 1; x <= _engineParameters.N; x++)
        for (byte y = 1; y <= _engineParameters.M; y++)
            possibleMovesList.Add(new Move(x, y));

    _possibleMoves = possibleMovesList.ToArray();
}
```

Kolejna metoda tej klasy służy pozyskiwaniu sekwencji testów do uruchomienia. Jest generalnie bardzo prosta. Pozyskuje się z pomocą mechanizmu refleksji wszystkie metody klasy i iteruje się po nich, zwracając leniwie (yield return [84]) kolejne metody wtedy i tylko wtedy, gdy spełniają pewne wymagania co do sygnatury. Jako że metody mają testować poprawność silników, muszą nie być abstrakcyjne, nie być konstruktorem i zwracać zmienną typu logicznego (test zakończył się sukcesem lub nie). Poza tym test nie powinien mieć parametrów, gdyż komplikowałoby to jego wykonanie.

```
public static IEnumerable<MethodInfo> GetTests()
{
    var methods = typeof(CorrectnessTests).GetMethods();
    foreach (var methodInfo in methods)
    {
        if (!methodInfo.IsConstructor && !methodInfo.IsAbstract &&
!methodInfo.GetParameters().Any() &&
            methodInfo.ReturnType == typeof(bool))
        {
            yield return methodInfo;
        }
    }
}
```

Metoda ta zwraca jedynie testy poprawności, ale klasa CorrectnessTests zawiera oczywiście też implementacje tych testów. Z uwagi jednak na to, że ich logika jest niezwykle ważna z punktu widzenia analizy poprawnościowej generowanych silników, a mniej z punktu widzenia implementacji, zostały one obszernie opisane wraz z przytoczeniem kodu źródłowego w rozdziale 4.

### 3.6.3 Klasa PerformanceTests – testy wydajności

Analogicznie do poprzednio opisywanej klasy, klasa PerformanceTests przechowuje całą logikę testów, tyle że wydajności. Ze względu jednak na inny charakter testów różni się nieco budową. Poniżej przedstawiono stan klasy – zmienne prywatne i konstruktor. Jak widać, wszelkie zależności są wstrzykiwane przez konstruktor, a jest to silnik i liczba iteracji testów.

```
public class PerformanceTests
{
    private readonly EngineWrapper _engine;
    private readonly long _iterations;

    public PerformanceTests(EngineWrapper engine, long iterations)
    {
        _engine = engine;
        _iterations = iterations;
    }
}
```

Biorąc pod uwagę budowę, klasa ta jest więc prostsza od poprzednio opisywanej.

Podobnie jak w poprzednim przypadku, również tutaj zdefiniowano metodę GetTests() o bardzo podobnej sygnaturze. Również leniwie zwraca ona kolejne metody będące testami. Inny jest jednak warunek, aby metoda została zwrocona jako test. Także nie może mieć parametrów, być konstruktorem i być abstrakcyjna, ale musi tym razem zwracać void zamiast bool. Oznacza to jedynie tyle, że test wydajności generalnie powodzi się zawsze i nie ma sensu zwracać z niego cokolwiek oprócz ewentualnych wyników wydajności. One jednak trzymane są w silniku zgodnie z wcześniejszymi założeniami, że tego typu pomiary są najdokładniejsze, bo nie ma żadnego narzutu.

```
public IEnumerable<MethodInfo> GetTests()
{
    var methods = GetType().GetMethods();
    foreach (var methodInfo in methods)
    {
        if (!methodInfo.IsConstructor && !methodInfo.IsAbstract &&
!methodInfo.GetParameters().Any() &&
            methodInfo.ReturnType == typeof(void))
        {
            yield return methodInfo;
        }
    }
}
```

Analogicznie do poprzedniego przypadku dalsza część klasy to konkretne implementacje testów. Również tutaj z racji, że nie są one cudem techniki programowania, ale są niezwykle istotne z punktu widzenia analizy wydajnościowej generowanych silników, nie zostały one w tym miejscu omówione. Ich opis można znaleźć w rozdziale 4. wraz z obszernym komentarzem i kodem źródłowym.

## **4. ANALIZA WYDAJNOŚCIOWO-POPRAWNOŚCIOWA ZIMPLEMENTOWANEGO GENERATORA SILNIKÓW GIER**

Niniejszy rozdział jest najważniejszy w całej pracy, ponieważ zawiera wszakże realizacja celu pracy – zbadanie wydajności generatorów silników gier logicznych. Aby ułatwić tę realizację, powstała wcześniej opisywana aplikacja testująca generowane silniki pod względem poprawności i wydajności – (m,n,k,p,q)EnginesAnalyzer.

Sama aplikacja to jednak jedynie narzędzie, ważne są konkretne testy wydajności i poprawności opisane w niniejszym rozdziale. Jeszcze ważniejsza jest interpretacja i przedstawienie wyników. Na treść tego rozdziału składać się będzie głównie przedstawienie możliwie obszernych rezultatów wielokrotnych, długich testów. Proces testowania silników trwał odpowiednio długo, ale proces ich przygotowania i interpretacji jeszcze dużej.

Z racji charakteru niniejszego rozdziału będzie on przede wszystkim bogaty w wyniki. Wykresy prezentujące rezultaty testów dominują, chociaż równie ważne są same testy i ich opisy. Starano się, aby ta część pracy w pełni zrealizowała cele pracy związane z analizą wydajnościowo-poprawnościową generowanych silników i generatora, co zakończyło się sukcesem. Poprawność generatora jest sprawdzana jako poprawność generowanych przez niego silników. Jeżeli chodzi o wydajność, to oddziennie jest sprawdzana wydajność samego generatora, a oddziennie wydajność generowanych silników.

### **4.1 Platforma testowa**

Platformą testową był prawie 6-letni notebook osobisty Asus G73Jh. Użyty został przede wszystkim z powodu dostępności – jako notebook autora jest zawsze gotowy do użycia. Jego wiek i stosunkowo niska jak na dzisiejsze czasy wydajność to dodatkowy plus – pomiary dają większe czasy, więc błąd pomiarowy powinien być mniejszy.

Parametry sprzętowe i systemowe przedstawiono poniżej. Należy mieć na uwadze, że wszelkie testy robione były pod minimalnym obciążeniem platformy testowej ze strony systemu operacyjnego i innych aplikacji – większość, jeżeli nie całość, mocy obliczeniowej była więc dostępna dla testów.

**Tabela 4.1.** Konfiguracja sprzętowa platformy testowej

Procesor	Intel® Core™ i7-720QM (6M Cache, 1.60-2.80 GHz) (4 rdzenie, 8 wątków) [78]
Chipset	Mobile Intel HM55 Express
Pamięć RAM	2 x 4 GB DDR3 SDRAM 1066 MHz
Karta Graficzna	ATI Mobility™ Radeon® HD5870 z pamięcią 1GB GDDR5 DX11
Dysk twardy	2 x 500 GB HDD 7200 rpm Serial ATA-150 ST9500420AS

**Tabela 4.2.** Konfiguracja systemowa platformy testowej

System operacyjny	Microsoft Windows 10 Build 10586
Platforma uruchomieniowa	Microsoft .NET Framework 4.6.1

Kompilator C#	Roslyn RTM 1
Kompilator C++	Microsoft Visual C++ 2015 Update 2 (v14.0.23918.0)
IDE	Microsoft Visual Studio Enterprise 2015 Update 2

W konfiguracji oprogramowania kierowano się generalnie zasadą, że im nowsze, tym lepsze. Z małym wyjątkiem – w trakcie pisania pracy pojawił się „Update 3” do Microsoft Visual Studio. Premiera miała miejsce 27 czerwca 2016 roku [80], praca była jednak już wtedy na tyle zaawansowana, że w obawie przed naruszeniem całego środowiska wstrzymano się z aktualizacją.

## 4.2 Testy wydajności generatora

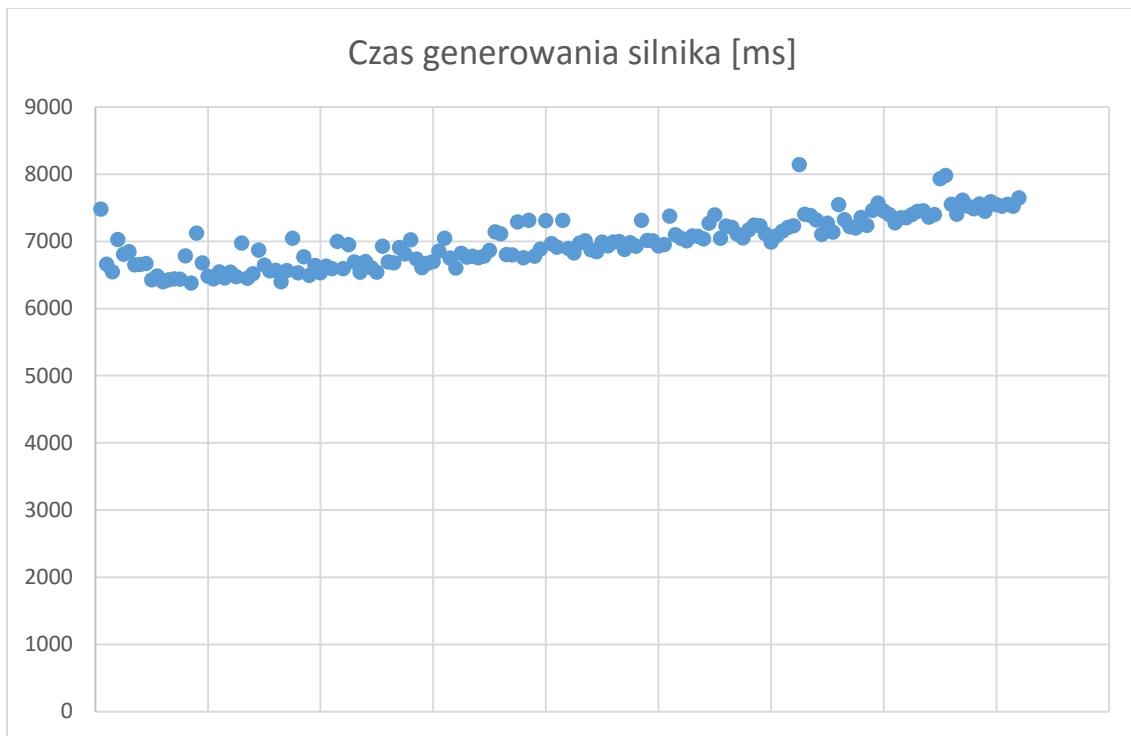
Generator sam w sobie (a nie jego wyjście – silniki) był testowany pod względem wydajności. W trybie „batch” przy zaznaczeniu na przycisku opcji „measure performance” przed każdym generowaniem silnika czyszczona jest lokalizacja wynikowa, więc kompilator C++ musi od nowa wykonać całą operację. Czas takiego pełnego generowania silnika był mierzony zgodnie z kodem opisany w rozdziale 3.3

W niniejszym rozdziale istotne jest jedynie to, że metodyka pomiarów jest poprawna. I chociaż mierzenie wydajności generatora to niestety głównie pomiary wydajności Microsoft MSBuild i kompilatora Microsoft Visual C++, to jednak pomiary były wykonywane w ten sposób, aby zmierzyć pełen czas generacji – wraz z ewentualnym dodatkiem od samego generatora. Rzeczywiście czasy generacji były trochę większe niż czasy samej komplikacji, ale bardzo nieznacznie. Dalej zaprezentowane zostaną kolejne wyniki pomiarowe.

### 4.2.1 Wyniki wydajności generatora

Początkowo rozdział ten zawierał obszerne wyniki wydajności generatora w zależności od kolejnych wartości parametrów m,n,k,p,q. Z racji jednak, że testy generatora, to głównie testy kompilatora, a komplikowany dla różnych parametrów kod nie różni się za bardzo, postanowiono przedstawić tutaj zebrane ze wszystkich przeprowadzonych 164 procesów generowania silników dane liczbowe poddane odpowiedniej obróbce statystycznej.

Poniżej przedstawiono wykres ze wszystkimi 164 czasami generowania silników. Trzeba pamiętać, że są to wartości dla różnych parametrów, więc naturalnie powinny się nieco różnić.



Rysunek 4.1. Czasy generowania silników

Wykres ten daje mniej więcej obraz rozrzutu wartości. Już po wstępnych jego oględzinach widać, że czasy te są do siebie bardzo zbliżone i praktycznie wszystkie silniki zostały wygenerowane w czasie 6 do 8 sekund. Dokładniejsze dane statystyczne przedstawiono w tabeli 4.3 poniżej.

**Tabela 4.3.** Dane statystyczne dla procesu generowania silników

Średnia	Mediania	Odchylenie	Minimum	Maksimum
6998.195	6988.933	363.6507	6379.989	8141.848

Średnio silnik generowany jest więc w 7 sekund z niewielkim odchyleniem standardowym na poziomie poniżej pół sekundy. Mediania jest bardzo zbliżona do średniej, a minimum i maksimum znajdują się w odległości odpowiednio około dwóch i około trzech odchyleń standardowych.

### 4.3 Testy poprawności generowanych silników

Aby przekonać się, czy generowane silniki są w pełni poprawne, konieczne były odpowiednie testy poprawności. Te zaimplementowane i opisane tutaj to niezbędny zestaw wyczerpujący sprawdzający poprawność silników pod każdym względem. Należy pamiętać, że zostały one zaprojektowane tak, żeby przede wszystkim udowodnić poprawność silników, tzn. często np. jakiś test nigdy się nie kończy, jeżeli silnik działa błędnie. Oczywiście, jeżeli wszystkie testy przechodzą poprawnie, to silnik działa poprawnie.

Testowanie to przypomina trochę testy jednostkowe. I rzeczywiście, autor wzorował się tutaj w pewnym zakresie na tej idei, chociaż są one przeprowadzane z nieco wyższego poziomu, bo aplikacja testująca komunikuje się z silnikiem, który jest działającym w tle procesem. Nadal

jednak mamy wejście i poprawne (przewidywane) wyjście, które silnik musi dla danego wejścia wyemitować, aby poprawnie przejść dany test.

#### 4.3.1 GetMovesShouldReturnAvailableMoves

Pierwszym testem poprawności jest weryfikacja, czy silnik zwraca prawidłową listę dostępnych posunięć. Na podstawie parametrów silnika można łatwo ustalić listę wszystkich dostępnych posunięć przed rozpoczęciem gry. Następnie, jeżeli decydujemy o każdym posunięciu, możemy również usuwać dane posunięcie z naszej listy. Po każdym takim ruchu możemy weryfikować, czy lista dostępnych posunięć zwracana przez silnik jest identyczna z naszą listą (asercja SequenceEqual czy podobna metoda z Linq).

Wartość false, czyli niezaliczenie testu, jest zwracana dla jakiegokolwiek niezgodności między listami. Ruchy pobierane są synchronicznie, a gra dwóch graczy – ludzi, aby można było kontrolować oba ruchy. Nie ma znaczenia, kto ma ile ruchów – liczą się tylko ruchy wykonane i wolne. Test bardzo dobrze sprawdza poprawność działania komendy silnika „getmoves”.

#### 4.3.2 AllPossibleMovesGameShouldEndTest

Kolejny test ma bardzo proste zadanie – sprawdzać, czy po wykonaniu m·n ruchów (czyli zapełnienia całej planszy) gra się skończy. Gra oczywiście może się też skończyć wcześniej (wygraną któregoś z graczy), ale nie jest to tutaj istotne. Test ma jedynie sprawdzić, czy dla każdego z silników gra się kończy po maksymalnie m·n ruchach.

Kod jest bardzo prosty. Rozpoczynamy grę synchroniczną dwóch graczy – ludzi. W tablicy \_possibleMoves mamy wygenerowane w konstruktorze na podstawie parametrów silnika wszystkie możliwe ruchy. Wysyłamy je po kolej, nie przejmując się graczami i turami. Istotne jest jedynie to, że po wysłaniu wszystkich możliwych ruchów gra musi się skończyć – w pesymistycznym przypadku remisem. Zwracamy więc wartość logiczną, oznaczającą, czy stan gry jest stanem gry zakończonej (wygrana jednego z graczy lub remis).

#### 4.3.3 GameOverTest

W bardzo podobnej modzie powstał test GameOverTest. Również on sprawdza, czy gra się kiedyś skończy, ale robi to dla nieco innej rozgrywki, jest też prostszy. Rozgrywa grę AI kontra AI i sprawdza, czy gra kiedykolwiek osiągnie stan końca gry. Jeżeli nie, to będzie niestety czekać w nieskończoność (w przyszłości wymaga to usprawnienia poprzez wprowadzenie timeout). Jeżeli tak, towróci prawdę.

```
public bool GameOverTest()
{
    _engine.StopAsync();
    _engine.StartGame(GameType.TwoAIs);

    while (!_engine.GetGameStateSync().IsGameOver())
    {
    }
    return true;
}
```

Widać więc, że jest to bardzo prosty test, ale sprawdza bardzo ważną rzecz – czy w przypadku gry dwóch AI gra się skończy. Gdyby był błąd w AI i wybierałoby ono ciągle niepoprawne ruchy, to by do tego nie doszło. Testuje on więc w pewnym sensie także poprawność AI dla danego silnika.

#### 4.3.4 TwoTimesSameMoveShouldNotBePossible

Zostając w kategorii testów prostych, ale użytecznych, warto sprawdzić, czy silnik prawidłowo rozpoznaje dopuszczalność ruchów. W szczególności, czy zabroni wykonania dwa razy tego samego ruchu.

```
public bool TwoTimesSameMoveShouldNotBePossible()
{
    _engine.StartGame(GameType.TwoHumans);
    _engine.StopAsync();

    var firstMove = new Move(1, 1) {Player = Player.Black};
    _engine.MakeMove(firstMove);
    if (_engine.GetMoveSync() != firstMove)
        return false;

    _engine.MakeMove(firstMove);
    return _engine.GetMoveSync() == null;
}
```

Rozpoczynamy grę człowiek kontra człowiek i wykonujemy ruch w pole 1,1. Sprawdzamy, czy silnik rzeczywiście to odnotował – jeżeli nie, to już jest błąd w silniku. Następnie wykonujemy znowu ten sam ruch. Silnik powinien zwrócić „invalid move”, które zostanie sparsowane jako null. Sprawdza się więc, czy zwrócony ruch jest nullem – wtedy i tylko wtedy silnik działa prawidłowo i nie dopuszcza do duplikowania ruchów.

#### 4.3.5 AfterQMovesItsWhitePlayerTurn

Kolejny test został zaprojektowany do dwóch celów. Po pierwsze ma sprawdzać mechanizm tur, a po drugie poprawność działania komendy „getplayer”, służącej do zwieracania obecnego gracza.

```
public bool AfterQMovesItsWhitePlayerTurn()
{
    _engine.StartGame(GameType.TwoHumans);
    _engine.StopAsync();

    //black turn
    for (ulong i = 0; i < _engineParameters.Q; i++)
    {
        if (_engine.GetCurrentPlayer() != Player.Black)
            return false;
        _engine.MakeMove(_possibleMoves[i]);
    }

    //white turn
    for (ulong i = 0; i < _engineParameters.P; i++)
```

```

    {
        if (_engine.GetCurrentPlayer() != Player.White)
            return false;
        _engine.MakeMove(_possibleMoves[_engineParameters.Q + i]);
    }
    //should be again black turn
    return _engine.GetCurrentPlayer() == Player.Black;
}

```

Rozpoczynana jest synchroniczna gra człowiek kontra człowiek. Najpierw sprawdzane jest, czy podczas pierwszych q ruchów, ruch ma gracza czarnego. Następnie sprawdza się, czy podczas następnych p ruchów, ruch ma gracza białego. Na koniec ruch powinien należeć znowu do gracza czarnego. Po drodze wykonuje się tylko poprawne i niepowtarzalne ruchy. Jeżeli kiedykolwiek po drodze tura gracza się nie zgadza, to zwracana jest nieprawda. W innym wypadku test przechodzi poprawnie, a silnik ma prawidłowo działający mechanizm tur i komendę „getplayer”.

#### 4.3.6 FirstTurnHumanVsHumanMovesTest

Badanie pierwszej tury ma również swój dedykowany test. W przypadku gry człowiek kontra człowiek warto zweryfikować, czy w pierwszych p+q ruchach silnik prawidłowo rozpoznaje wysyłane ruchy i do którego gracza dany ruch przynależy. Oczywiście pierwsze q ruchów należy do czarnego, a następne p ruchów do białego. Można to jednak sprawnie weryfikować, korzystając z przeciążonego operatora równości dla klasy Move.

```

public bool FirstTurnHumanVsHumanMovesTest()
{
    _engine.StartGame(GameType.TwoHumans);
    _engine.StopAsync();

    for (ulong i = 0; i < _engineParameters.Q + _engineParameters.P; i++)
    {
        var move = new Move(_possibleMoves[i].X, _possibleMoves[i].Y)
        {
            Player = i < _engineParameters.Q ? Player.Black :
Player.White
        };
        _engine.MakeMove(move);
        var returnedMove = _engine.GetMoveSync();
        if (move != returnedMove)
            return false;
    }

    return true;
}

```

Jak widać, tworzony jest unikalny ruch z dobrze dopasowanym kolorem. Następnie dokonuje się tego ruchu komendą „makemove” na silniku (oczywiście przez wrapper). Potem pobiera się ruch z silnika (odpowiedź) i sprawdza się, czy jest równy wykonanemu ruchowi (wraz z kolorem gracza!). Tylko wtedy, gdy wszystkie ruchy były równe, test się powiodł.

#### 4.3.7 FirstTurnHumanVsAiMovesTest

Pozostając w tematyce pierwszej tury, można sprawdzić, czy również AI zachowa się poprawnie, odpowiadając na ruchy człowieka. Sprawa jest podobna do tej z poprzedniego testu – trzeba sprawdzić, czy ruchy są dobrze rozpoznawane, przypisywane dobrym graczom, oraz dodatkowo, czy AI wykonuje prawidłowe ruchy zgodne ze swoją specyfikacją.

Jako że AI bazuje tutaj na algorytmie ai\_adjacent z silnika Connect-k, powinno wykonywać ruchy, które są sąsiadujące z obecnymi już na planszy ruchami. Właśnie to jest dodatkowo weryfikowane przez ten test – czy świeżo wykonany ruch AI jest sąsiadujący do jakiegokolwiek wcześniej wykonanego ruchu.

```
public bool FirstTurnHumanVsAiMovesTest()
{
    _engine.StartGame(GameType.BlackHumanVsWhiteAi);
    _engine.StopAsync();

    var movesMade = new List<Move>();

    //human's turn
    for (ulong i = 0; i < _engineParameters.Q; i++)
    {
        var move = new Move(_possibleMoves[i].X, _possibleMoves[i].Y)
{Player = Player.Black};
        _engine.MakeMove(move);
        var returnedMove = _engine.GetMoveSync();
        if (move != returnedMove)
            return false;
        movesMade.Add(returnedMove);
    }

    //ai's turn
    for (ulong i = 0; i < _engineParameters.P; i++)
    {
        var aiMove = _engine.GetMoveSync();
        if (!movesMade.Any(m => m.IsAdjacent(aiMove)))
            return false;
        movesMade.Add(aiMove);
    }
    return true;
}
```

Jak widać, poza standardowym sprawdzaniem, czy ruch zwracany przez silnik i wykonany przez gracza-człowieka jest poprawny, w turze AI występuje sprawdzanie, czy ruch wykonany właśnie przez AI jest sąsiadujący do już wykonanych ruchów. Jeżeli chociaż raz nie będzie, to test zakończy się porażką.

#### 4.3.8 FirstTurnAiVsHumanMovesTest i FirstTurnAiVsAiMovesTest

Bardzo podobnym testem do poprzedniego jest test, w którym to AI jest graczem czarnym, a człowiek białym. Różnica jest więc tylko w kolejności. Sprawdzamy więc, czy AI wykonuje ruchy sąsiadujące z innymi (ale jako że zaczyna, to pierwszy ruch jest wyłączony z tego wymogu, bo nie ma z czym sąsiadować), a potem, czy ruchy człowieka są prawidłowo rozpoznawane.

Właściwie konstrukcja jest analogiczna jak w poprzednim wypadku z zamienieniem tylko kolejności i zrezygnowaniem z warunku sąsiedniości dla pierwszego ruchu AI w pierwszej turze.

Pozostając w tej samej metodyce badania pierwszej tury, należy jeszcze sprawdzić, czy silnik zachowuje się w niej prawidłowo również dla gry AI kontra AI. W tym celu w teście „FirstTurnAiVsAiMovesTest” są pobierane po prostu kolejne ruchy wykonane przez AI i sprawdzane jest, czy sąsiadują przynajmniej z jednym ruchem już wykonanym (wyłączony z tego warunku jest oczywiście pierwszy ruch w całej rozgrywce).

Z braku lepszej weryfikacji poprawności AI sprawdzane jest, czy ruchy spełniają warunek sąsiadowania – czyli założenie projektowe tego AI.

#### 4.3.9 HumanVsHumanBlackWinTest

Kolejny test jest już znacznie bardziej złożony. Wynika to jednak z tego, że ma sprawdzać znacznie bardziej skomplikowaną rzecz – warunek wygranej. Czy jeżeli gracz ustawi równe k swoich pionów pod rząd, to wygra? Zakładając, że drugi gracz będzie w tym czasie realizować strategię przegrywającą, to powinien.

Nie jest to niestety tak proste. Ciężko jest (a być może nawet jest to nieosiągalne) wyznaczyć taką sekwencję ruchów, aby była ona zawsze przegrywająca niezależnie od wartości m,n,k,p,q. Z tego też powodu zdecydowano się tutaj badać wygraną gracza czarnego (który z reguły ma częściej wygrywającą strategię) i stawiać graczem białym ruchy możliwie zmniejszające prawdopodobieństwo wygranej szybciej niż gracz czarny.

```
public bool HumanVsHumanBlackWinTest()
{
    _engine.StartGame(GameType.TwoHumans);

    if (_engineParameters.K < Math.Max(_engineParameters.M,
    _engineParameters.N) ||
        Math.Min(_engineParameters.M, _engineParameters.N) < 3)
        //engine doesn't allow winning plus we assume m*n<=3k so min(m,n)
        must be at least 3
        return true;

    var blackWiningMoves = WiningMovesFromEnd(_engineParameters.K,
    Player.Black);

    var whitePossibleMoves = new List<Move>();

    foreach (var possibleMove in _possibleMoves)
    {
        if (!blackWiningMoves.Any(m => m.X == possibleMove.X && m.Y ==
        possibleMove.Y))
        {
            whitePossibleMoves.Add(possibleMove);
            whitePossibleMoves.Last().Player = Player.White;
        }
    }
    _engine.StopAsync();
    var whiteMove = 0;
    for (ulong blackMove = 0; blackMove < _engineParameters.K;
    blackMove++)
    {
```

```

        while (_engine.GetCurrentPlayer() != Player.Black)
    {
        _engine.MakeMove(whitePossibleMoves[2*whiteMove]);
        var retMove = _engine.GetMoveSync();
        if (retMove != whitePossibleMoves[2*whiteMove])
            return false;
        whiteMove++;
    }

    _engine.MakeMove(blackWiningMoves[(int) blackMove]);
    if (_engine.GetMoveSync() != blackWiningMoves[(int) blackMove])
        return false;
}
return _engine.GetGameStateSync() == GameState.WinnerIsBlack;
}

```

Po pierwsze dla pewnych silników nie da się wygrać. Taka sytuacja istnieje choćby w przypadku, gdy  $\max(m,n)$  jest mniejsze od  $k$  – czyli nie da się ustawić na planszy  $k$  pionów pod rzęd. Dla takich silników po prostu zwracamy, że przeszły test poprawnie.

Następnie są generowane metodą `WiningMovesFromEnd` k ruchów dających łącznie wygraną w k ruchach graczu czarnemu od końca planszy (czyli od prawego dolnego rogu). Kolejnym krokiem jest generowanie ruchów gracza białego, które są po prostu możliwymi ruchami od początku planszy (lewy, górny róg), takimi, które nie należą do wygrywającej sekwencji gracza czarnego.

W kolejnym kroku jest już rozgrywana gra właściwa. Czarny gracz ma łatwe zadanie – po prostu stawia kolejne wygrywające ruchy ze swojej wygenerowanej sekwencji. Biały gracz z kolei stawia co drugi z dostępnych mu ruchów tak, aby miał strategię przegrywającą – żeby później niż czarny gracz ustawił k swoich pionów pod rzęd.

Po każdym ruchu sprawdzane jest, czy silnik poprawnie go rozpoznał i został on wykonany – jeżeli nie, to oczywiście test kończy się porażką. Na koniec gra powinna znajdować się w stanie wygranej przez gracza czarnego – jeżeli tak jest, to test zakończył się sukcesem.

#### 4.3.10 HumanVsHumanBlackWinIfKOrMoreToWinTest

O ile poprzedni test był skomplikowany, bo miał za zadanie przetestować prawidłowe przyznanie wygranej, o tyle ten jest jeszcze bardziej skomplikowany, bo ma sprawdzić poprawność warunku wygranej. Gracz czarny powinien wygrać grę w tym teście, ale tylko jeżeli silnik ma warunek wygranej dopuszczający k lub więcej pionów pod rzęd. Jeżeli silnik dopuszcza jako wygraną tylko dokładnie k pod rzęd pionów, to gracz czarny nie wygra, gdyż w tym teście ustawia k+1 pionów pod rzęd.

Test ma sens tylko dla silników, których  $\max(m,n)$  jest większe lub równe  $k+1$ , gdyż musimy ustawić  $k+1$  pionów pod rzęd.

Podobnie jak w poprzednim teście, pozyskujemy  $k+1$  (poprzednio  $k$ ) od tyłu ruchów wygrywających dla gracza czarnego. Tym razem jednak pozyskujemy też  $k+1$  ruchów wygrywających od przodu dla gracza białego. Dodatkowo dla gracza białego tworzymy listę unikalnych ruchów z listy ruchów dostępnych, które jednak nie są wygrywające.

Dla gracza czarnego wykonujemy wszystkie wygrywające ruchy, oprócz ruchu z indeksem 1. Mamy w ten sposób k-1 ruchów pod rząd, w tym samym rzędzie dalej puste pole i ruch czarny. Wystarczy więc wstawić ruch czarny w puste pole, aby otrzymać k+1 ruchów pod rząd.

W turach gracza białego dodajemy z kolei kolejne wygrywające ruchy tak długo, aż zostanie tylko jeden brakujący – wtedy dodajemy ruchy nie wygrywające.

Uzyskuje się więc sytuację, w której graczowi czarnemu brakuje jeden ruch z listy ruchów wygrywających, aby zbudował k+1 ruchów pod rząd, a graczowi białemu brakuje jeden ruch z listy ruchów wygrywających, aby zbudował k ruchów pod rząd. Po drodze sprawdza się poprawność rozpoznawania i uznawania ruchów przez silnik.

Następnie wykonuje się brakujący ruch gracza czarnego. Teraz, jeżeli silnik dopuszczał k i więcej pionów pod rząd jako sytuację wygrywającą, gra się powinna kończyć wygraną gracza czarnego. Jeżeli natomiast silnik wymagał dokładnie k pionów pod rząd i nie więcej, to gracz biały wykonuje swój k-ty ruch i powinien wygrać. Jeżeli żadna z dwóch sytuacji nie nastąpi, będzie to oznaczać, że silnik nie przeszedł testu.

#### **4.4 Wyniki poprawności generowanych silników**

Przetestowano 164 silniki o różnych parametrach. Jako że testów poprawności jest 11, łącznie dało to 1804 rezultatów testów. Wszystkie wyniki testów okazały się sukcesem. Poniżej wymieniono wszystkie uruchamiane na tych silnikach testy

- a- GetMovesShouldReturnAvalaibleMoves
- b- GameOverTest
- c- AfterQMovesItsWhitePlayerTurn
- d- TwoTimesSameMoveShouldNotBePossible
- e- AllPossibleMovesGameShouldEndTest
- f- HumanVsHumanBlackWinTest
- g- HumanVsHumanBlackWinIfKOrMoreToWinTest
- h- FirstTurnHumanVsHumanMovesTest
- i- FirstTurnHumanVsAiMovesTest
- j- FirstTurnAiVsHumanMovesTest
- k- FirstTurnAiVsAiMovesTest

Po zakończeniu wszystkich testów, aplikacja (m,n,k,p,q)EnginesAnalyzer wypisała komunikat podsumowujący testy wszystkich silników. Przedstawiono go poniżej.

---- Correctness tests 1804/1804 succeeded!

Komunikat oznacza oczywiście 100% sukcesu w testach silników, czyli że wszystkie 164 wygenerowane silniki o bardzo różnych przecież parametrach działają w pełni poprawnie. Testy były na tyle dokładne, że wszelkie złe działanie silników powinno zostać wykryte. Mamy więc sporą gwarancję, że generowane silniki rzeczywiście są poprawne.

#### 4.5 Testy wydajności generowanych silników

Podstawą dla zaprojektowania odpowiednich testów wydajności są wcześniej przyjęte metryki wydajności. Dla przypomnienia są to: średni czas wygenerowania ruchu przez AI, średni czas sprawdzenia, czy spełniony został warunek wygranej, i średni czas wygenerowania listy dostępnych posunięć. Tak naprawdę w każdej rozgrywce z udziałem AI już po pierwszym ruchu AI dwie pierwsze wartości mają przynajmniej jeden punkt pomiarowy. Wystarczy więc rozegrać odpowiednio dużo, odpowiednio długich rozgrywek z udziałem AI, aby uzyskać dobrą statystykę dla dwóch pierwszych metryk. Tak właśnie zrobiono, projektując pierwszy test wydajności.

```
public void AiVsAiRandomGames()
{
    _engine.StopAsync();
    for (long i = 0; i < _iterations; i++)
    {
        _engine.StartGame(GameType.TwoAIs);
        while (!_engine.GetGameStateSync().IsGameOver())
        {
            Thread.Sleep(20);
        }
        _engine.ClearMessageQueue();
    }
}
```

Rozpoczynamy grę AI kontra AI i tak naprawdę czekamy, aż się ona skończy. Nawet dla niewielkich plansz powinno to oznaczać przynajmniej kilka ruchów obu graczy AI, a co za tym idzie, przynajmniej kilka punktów pomiarowych dla pierwszej i drugiej metryki. Należy jeszcze dodać, że test jest uruchamiany w wielu iteracjach, więc mamy bardzo dużo danych pomiarowych. Przy dużych planszach jest ich jeszcze więcej, bo więcej ruchów jest potrzebnych, aby zakończyć grę. W przypadku silników z dużymi planszami, można więc użyć mniejszej liczby iteracji – czas i tak będzie odpowiednio dłuższy.

Poprzedni test przy niewielkiej ilości prostego kodu zapewniał dużą ilość danych pomiarowych dla wyznaczenia średniego czasu generowania ruchu przez AI i średniego czasu sprawdzenia warunku wygranej. Nie sprawdzał on jednak wcale trzeciej metryki wydajnościowej – średniego czasu wygenerowania listy dostępnych posunięć. Właśnie taki jest cel kolejnego testu.

```
public void GetMovesRandomMovesTest()
{
    _engine.StopAsync();
    _engine.StartGame(GameType.TwoHumans);
    for (long i = 0; i < _iterations; i++)
    {
        var moves = _engine.GetMovesSync();
        _engine.ClearMessageQueue();
    }
}
```

Analogicznie do poprzedniego przypadku, test polega na wykonywaniu pewnej operacji określoną ilość razy, aby uzyskać wartość średnią metryki wydajnościowej ze stosunkowo małym błędem pomiarowym. Dlatego też w pętli tyle razy, ile wynosi pole `_iterations`, pobierana jest lista dostępnych posunięć. Aby nie kurczyła się do listy pustej, rozgrywka jest typu człowiek kontra człowieek. W ten sposób tak naprawdę za każdym razem zwracana jest taka sama lista, ale też za każdym razem musi ona być na nowo wyznaczona przez silnik, w którym pojawi się kolejny pomiar czasu operacji.

#### 4.6 Wyniki testów wydajności

W celu uzyskania dobrej statystki informującej nas o zależnościach wydajnościowych i samej wydajności silników przeprowadzono odpowiednio dużo testów – dla każdego ze 164 wygenerowanych silników z planszą poniżej rozmiaru 250x250 było to aż 1000 iteracji w każdym teście. Dla silników z planszą o rozmiarach 250x250 zastosowano co prawda jedynie 10 iteracji, ale te pomiary i tak kosztowały prawie dwie doby nieprzerwanej pracy platformy testowej, ponieważ gry na tak dużej planszy trwają odpowiednio dłużej.

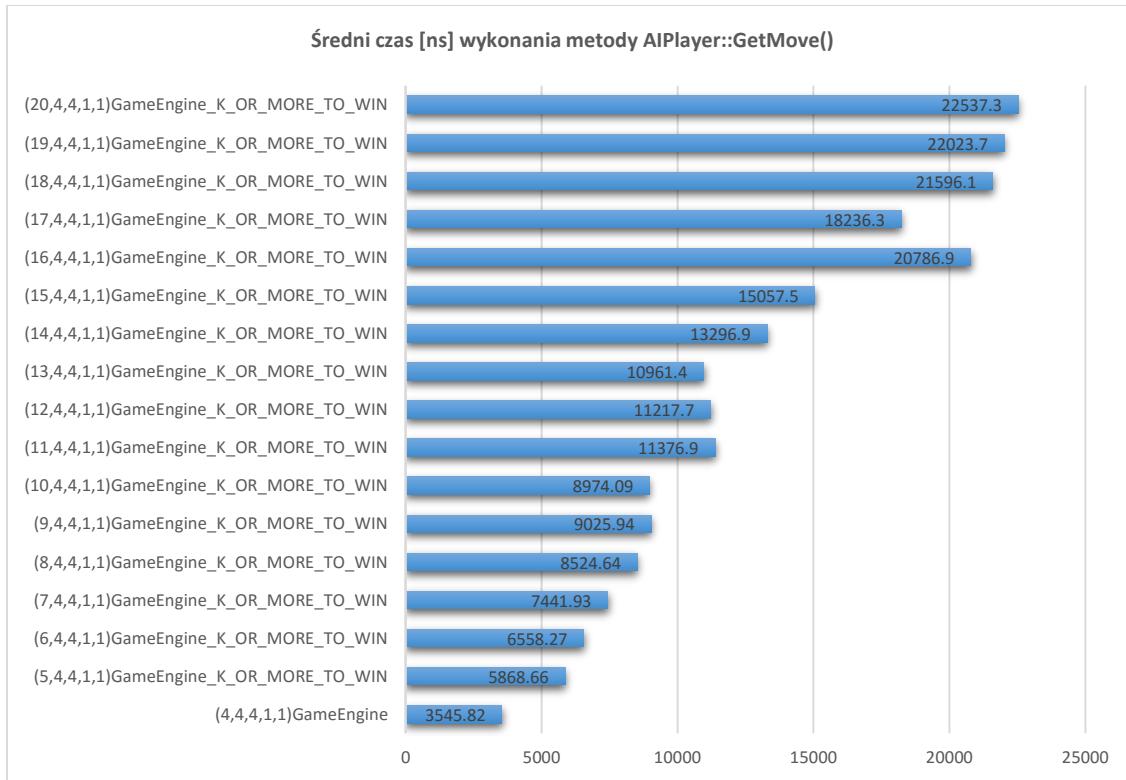
Podobnie jak w przypadku testów wydajnościowych generatora, również tutaj wyniki zostały podzielone na pewne grupy. Są np. testy wydajności silników przy rosnącym k i zachowaniu stałosci innych parametrów. Sens tego jest oczywiście taki, żeby przebadać zależności wydajności silników od konkretnego parametru.

Wyniki, jeżeli podejrzewano, że reprezentują pewną zależność matematyczną, przebadano metodą regresji liniowej lub kwadratowej. Tam gdzie taka sytuacja miała miejsce umieszczono w treści pod wykresem wyników wartość współczynnika  $R^2$  wraz z komentarzem co dana wartość może oznaczać. Współczynnik  $R^2$  opisuje stopień dopasowania modelu do danych pomiarowych, nazywany jest również współczynnikiem determinacji [87][88]. Jest on znormalizowany do przedziału od 0 do 1 a im bliższy jedności tym oznacza lepsze dopasowanie modelu do danych. Stawiając jakąś hipotezę – np. że dane mają charakter liniowy, pożądane jest aby wartość  $R^2$  dla modelu reprezentującego tę hipotezę była jak najbliższa 1.0. Przykładowo wartość 0.9 lub więcej uważa się za bardzo dobre dopasowanie. Z kolei wartości poniżej 0.5 oznaczają dopasowanie niezadowalające.

##### 4.6.1 Wydajność silników a rozmiar planszy

Najbardziej oczywistą zależnością powinna być zależność wydajności od rozmiaru planszy – parametrów m i n. Wszystkie trzy operacje powinny trwać dłużej dla planszy większej. Wygenerowanie ruchu przez AI powinno trwać dłużej, ponieważ AI „skacze” po całej planszy, szukając wolnego miejsca siedzącego z miejscem zajętym. Im większa plansza, tym więcej wolnych miejsc, ale znacznie czężej znaleźć wolne miejsce, które sąsiaduje z jakimś innym. W wygenerowaniu listy wolnych ruchów trzeba po prostu sprawdzić wszystkie pola planszy – więc oczywiste jest to, że czas ten powinien rosnąć wraz ze wzrostem planszy. Inaczej sytuacja może wyglądać ze sprawdzeniem warunku wygranej – zliczamy ilości pionów pod rząd z danego miejsca, gdzie właśnie dokonano ruchu.

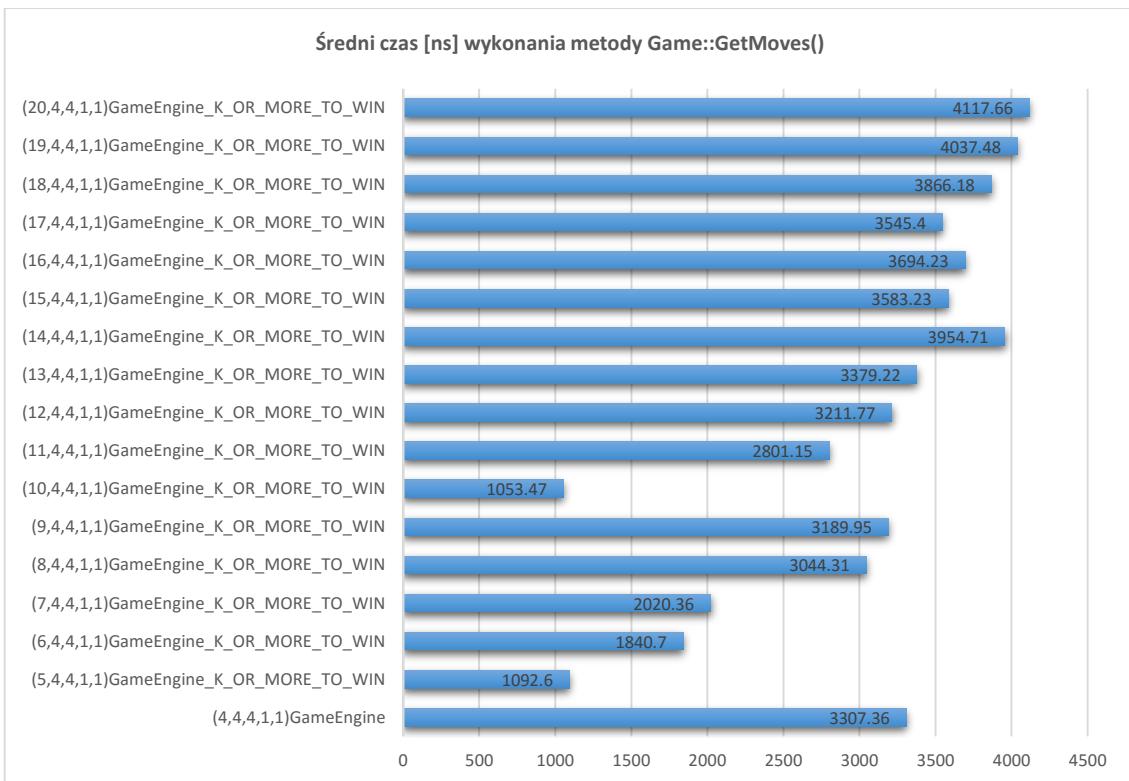
Testy zaczynają się od zbadania zależności dla parametru m w przedziale 5,...,20. Pozostałe parametry są stałe i wynoszą odpowiednio n=4=k=4, p=q=1. Bada się więc tylko zależność od m.



Rysunek 4.2. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru m

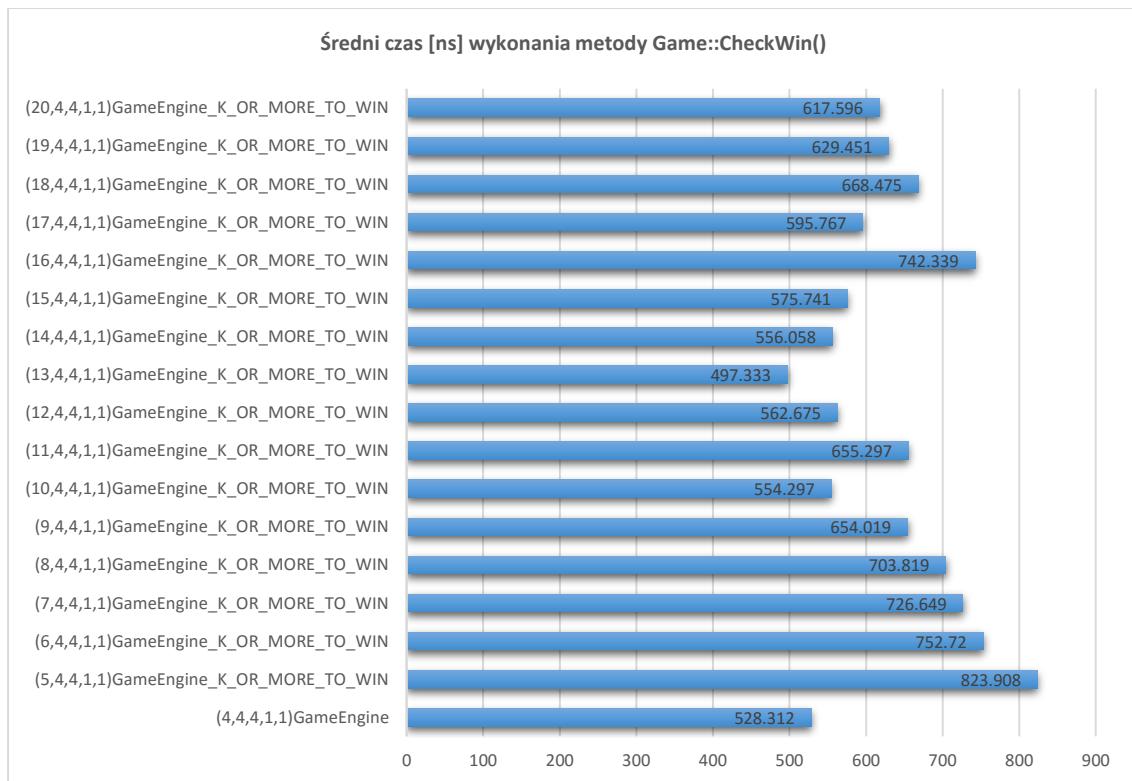
Zależność dla generowania ruchu przez AI, poza małymi odchyleniami, spełnia bardzo dobrze oczekiwania teoretyczne. Wzrost czasu wraz ze wzrostem parametru m jest pomiędzy liniowym ( $R^2 = 0.9399$ ) a kwadratowym ( $R^2 = 0.9266$ ).

Dzieje się tak, dlatego, bo chociaż w metodzie AI::GetMove przechodzona jest cała plansza ze sprawdzeniem IsAdjacent dla każdego pola tylko raz to jednak wywołanie IsAdjacent nie musi się zawsze wykonać w czasie stałym. IsAdjacent wywołuje Board::CountPieces we wszystkie 8 strony od pola. CountPieces z kolei zlicza wystąpienia tego samego (w tym przypadku pustego) koloru tak długo aż się skończy plansza lub napotka inny kolor. W pesymistycznym przypadku IsAdjacent za pomocą CountPieces może sprawdzić kolor pola  $m+n+2(m^2+n^2)^{1/2}$  razy. To z kolei jako, że jest wywoływane w pętli może dawać kwadratową zależność od m. Dla mniej zapełnionej planszy AI::GetMove powinno zatem wykonywać się dłużej.



Rysunek 4.3. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m

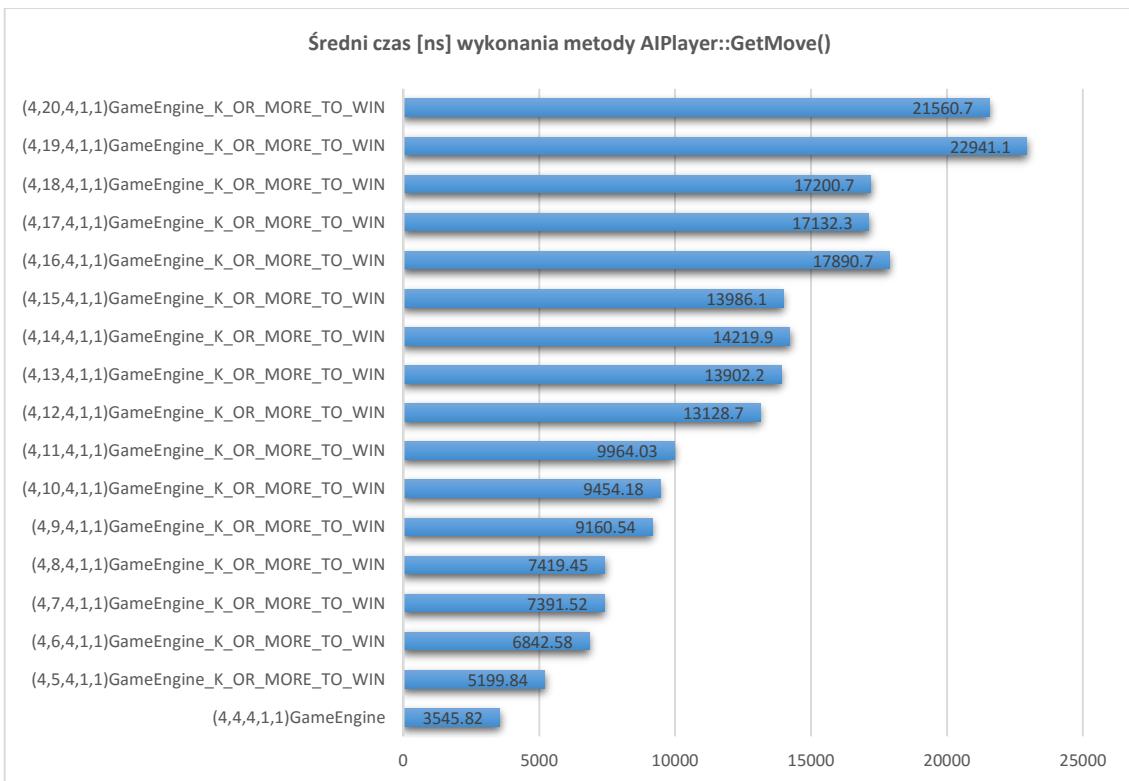
W przypadku metody zwracającej wolne ruchy, zależność jest już trochę bardziej „zaszumiona”. Jako, że metoda jedynie iteruje po planszy i dla każdego pola sprawdza w stałym czasie jego kolor (odczytanie wartości odpowiedniego bitu lub odpowiedniej wartości w tablicy) jej czas wykonania powinien zależeć liniowo od m. Nieśmiało potwierdza to współczynnik korelacji liniowej R<sup>2</sup> wynoszący 0.5257. Był może kompilator stosuje tutaj jakąś nieznaną optymalizację, lub daje o sobie znać dokładność wspominanego wcześniej API QueryPerformanceCounter [76].



Rysunek 4.4. Zależność średniego sprawdzenia warunku wygranej od wartości parametru m

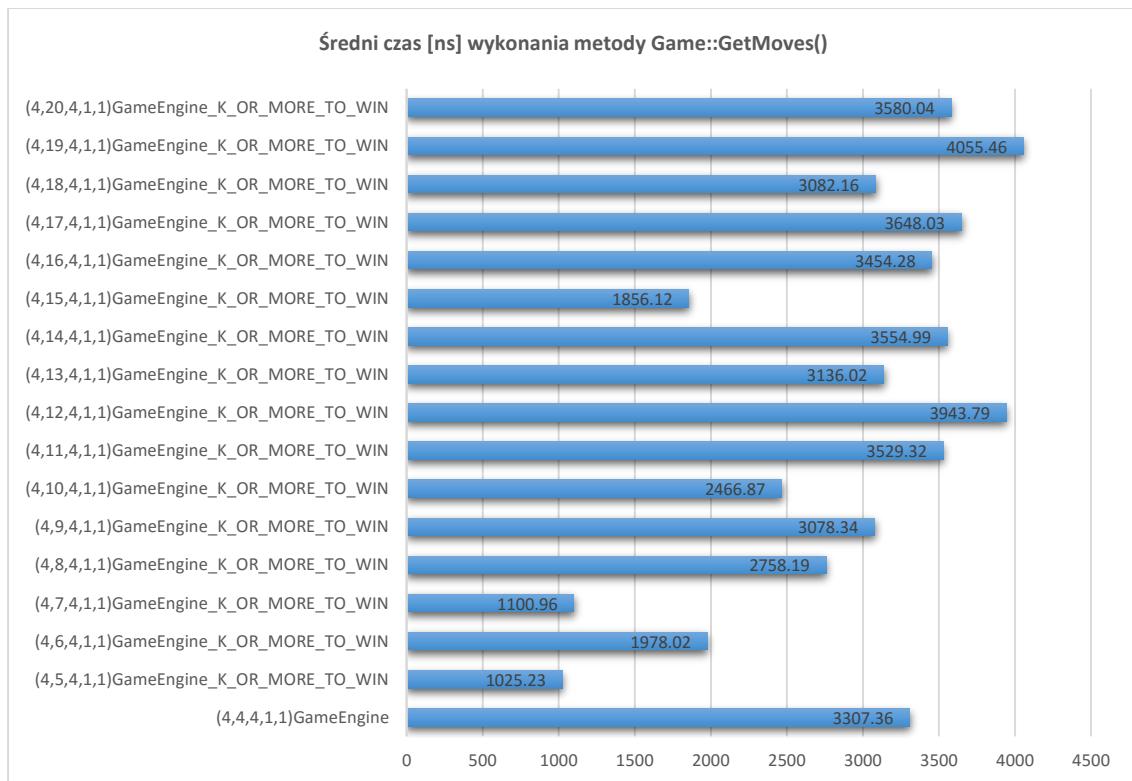
Z kolei sprawdzenie warunku wygranej, zgodnie z przewidywaniami raczej nie wykazuje żadnej zależności. Należy jednak zauważać, jak niewielkie są to czasy – paręset nanosekund, więc zbliżamy się do błędu pomiarowego samej metody pomiarowej, opisywanego wcześniej w rozdziale o silnikach. Z drugiej jednak strony należy to też uznać za powód do dumy z powodu kodu natywnego – gdyby silnik był napisany w technologii zarządzalnej, czasy te byłyby przynajmniej o rząd wielkości większe.

Pora sprawdzić zależność od parametru n. Oczywiście powinna ona być dokładnie taka sama z racji symetrii dwóch parametrów opisujących rozmiar planszy. Testy jednak pozwolą nam to udowodnić empirycznie.



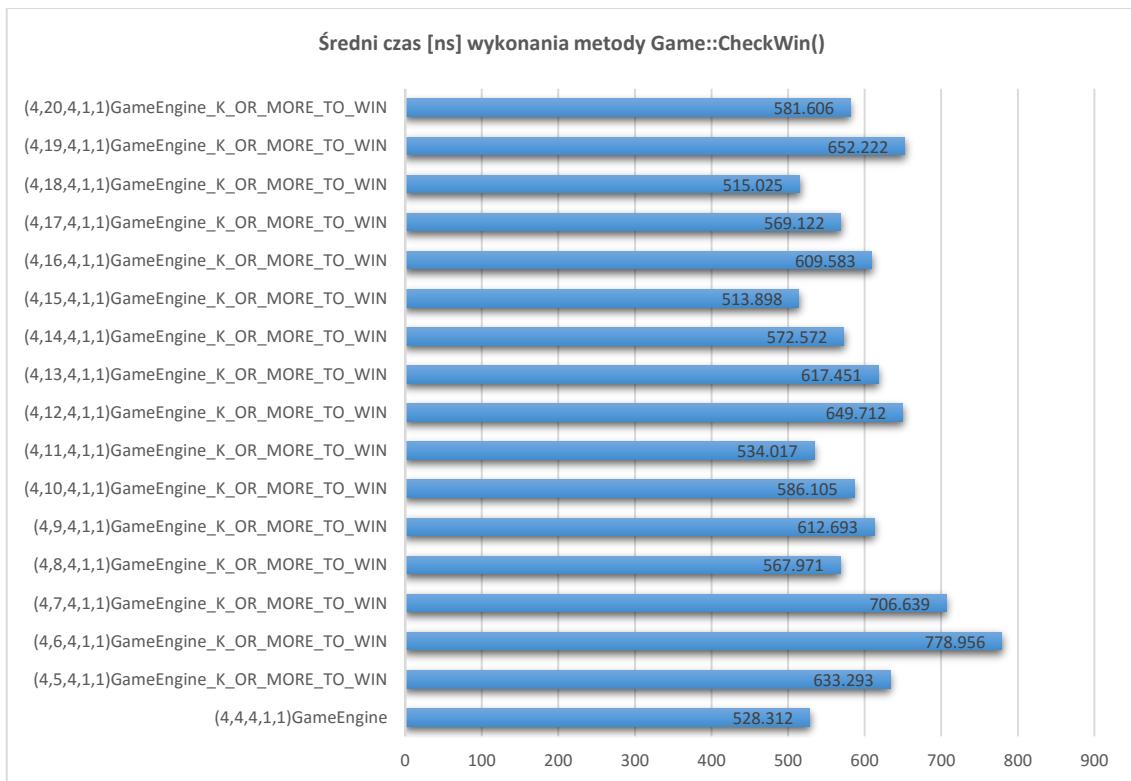
Rysunek 4.5. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru n

Zależność, dokładnie taka, jakiej można by się spodziewać. Niewielkie szумy mogą być spowodowane czynnikami, nad którymi nie mamy kontroli. Dla korelacji liniowej współczynnik  $R^2$  wyniósł 0.9588, natomiast dla korelacji kwadratowej 0.9553.



Rysunek 4.6. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru n

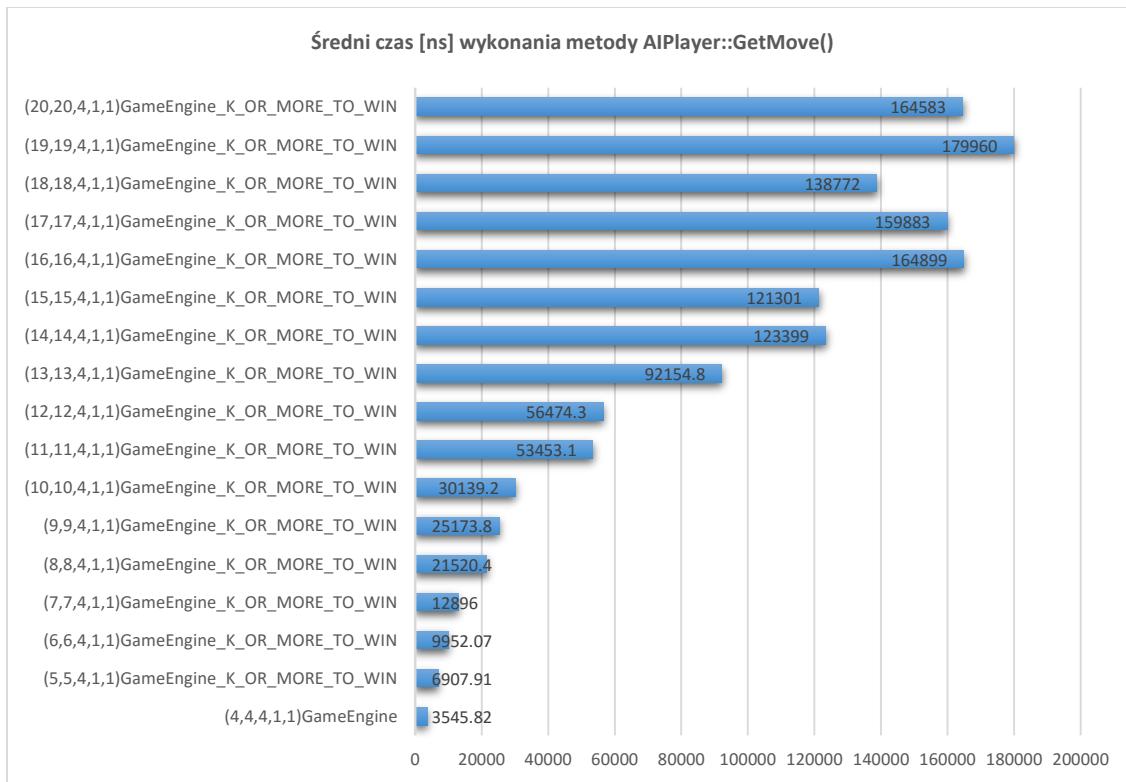
Zaskakuje trochę czas wykonania silnika z najmniejszą planszą, poza tym trend nie jest tak klarowny jak w przypadku generowania ruchów przez AI. Ale jednocześnie jest to identyczna sytuacja jak w przypadku badania zależności od parametru m. Niestety korelacja liniowa jest tutaj jeszcze słabsza niż w przypadku parametru m – współczynnik  $R^2 = 0.3394$ .



Rysunek 4.7. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru n

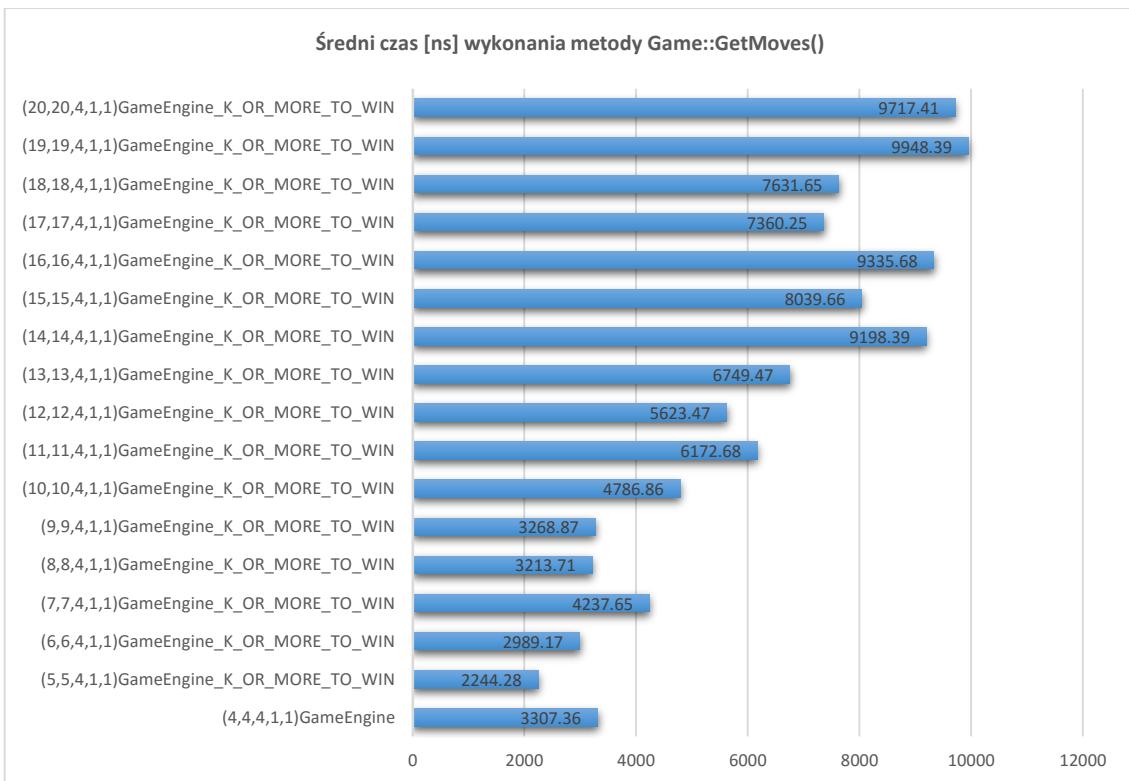
CheckWin bez zmian – bardzo szybkie wykonanie, ale bez widocznej jakiejkolwiek zależności dla rozmiaru planszy.

Kolejnym pomysłem jest sprawdzenie metryk wydajnościowych dla jednoczesnego wzrostu m i n.



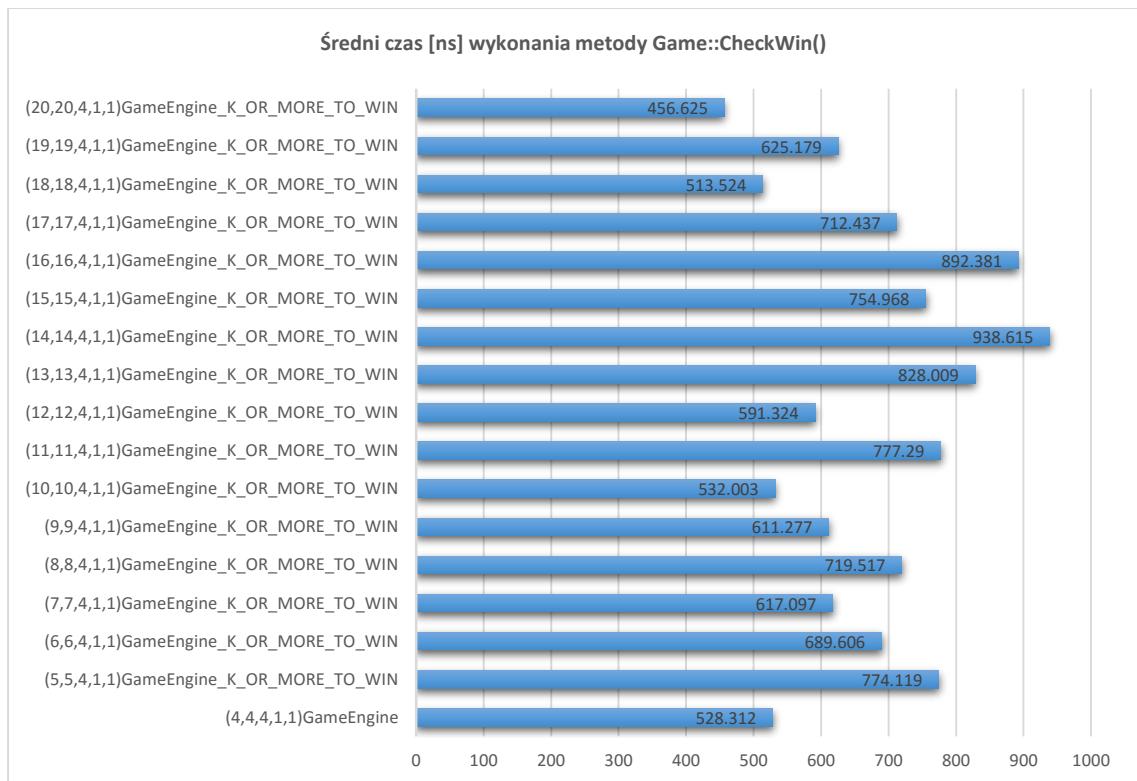
Rysunek 4.8. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru m i n

Zgodnie z teorią i wiedzą z poprzednich analiz, jednoczesny wzrost m i n daje kwadratowy ( $R^2 = 0.9576$ ) wzrost czasu wygenerowania ruchów przez AI. Widoczne są jednak dosyć duże odchylenia od tego trendu dla niektórych silników (zwłaszcza większych), które trudno w tej chwili wytłumaczyć i mogą być efektem błędów pomiarowych. Możliwe też, że wynikają one z momentami większej niż liniowej zależności czasu wykonania tej metody dla wzrostu m lub n.



Rysunek 4.9. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m i n

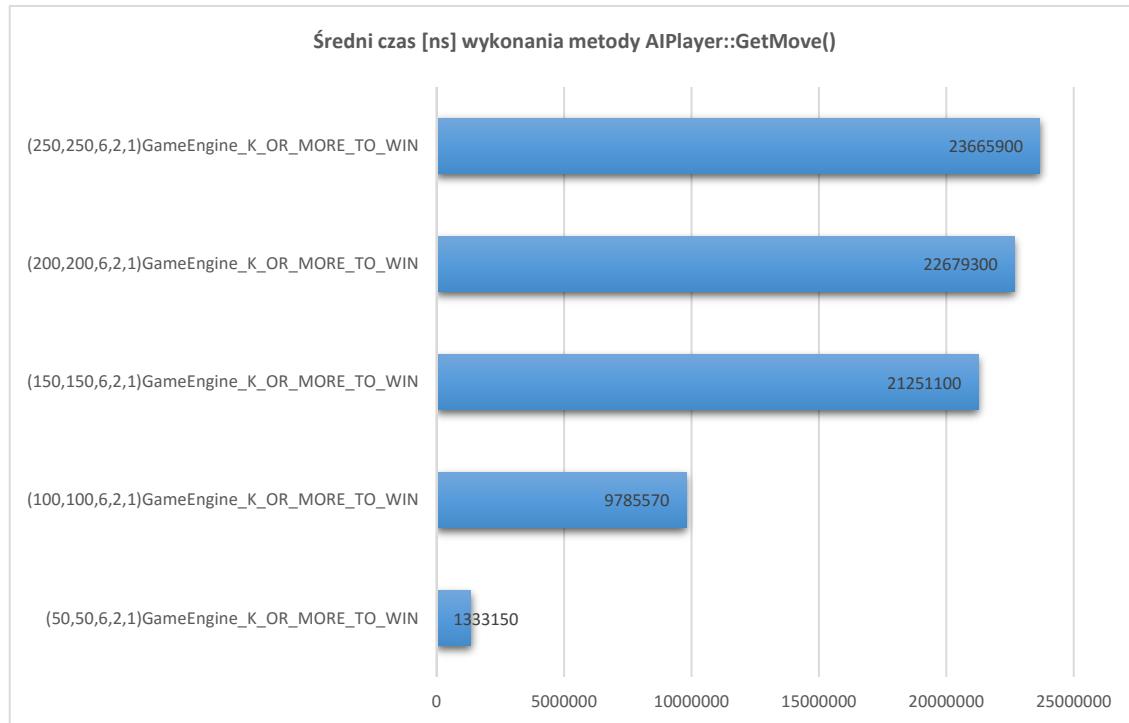
Generowanie listy wolnych posunięć również przejawia kwadratowy wzrost czasu wykonania wraz ze wzrostem m i n. Współczynnik korelacji kwadratowej  $R^2$  wynosi 0.7622, widać poprawę w zgodności z oczekiwaniemi teoretycznymi w stosunku do badania wzrostu czasu wykonania tej metryki wraz z jednym z parametrów m,n.



Rysunek 4.10. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru m i n

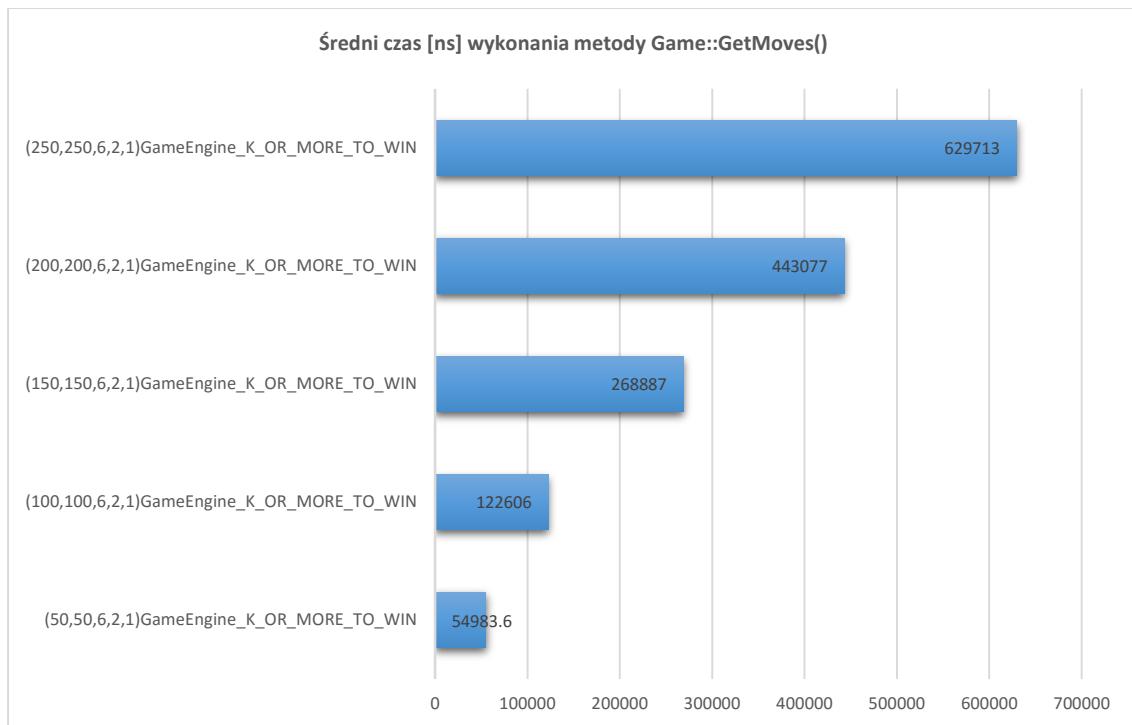
Czas sprawdzenia warunku wygranej bez zmian – brak jakiegokolwiek zależności explicite od rozmiaru planszy.

Należy jeszcze sprawdzić zależność wydajności silników od rozmiaru planszy dla plansz dużych – w przedziale od 50x50 do 250x250.



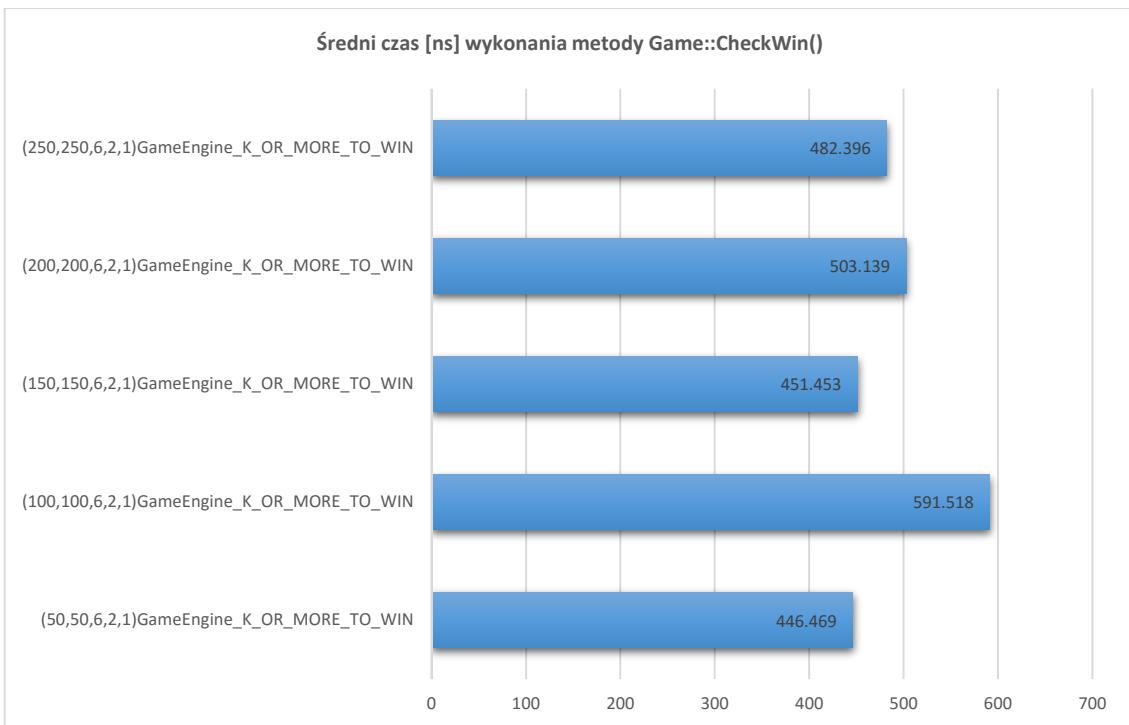
Rysunek 4.11. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametrów m i n dla dużej planszy

Przy dwukrotnym zwiększeniu m i n otrzymujemy około siedmiokrotny wzrost czasu wykonania metody, a w pozostałych przypadkach wzrost jest mniejszy. Zależność czasu wykonania `AIPlayer::GetMove()` pozostaje w przybliżeniu kwadratowa ( $R^2 = 0.8989$ ), ale wydaje się, że jakiś inny efekt odgrywa tutaj rolę. I owszem – przy małych planszach ilość ruchów wewnętrznie wygenerowanych przez AI, jako spełniających reguły sąsiedztwa, jest mała, a co za tym idzie, przydzielany na nie wektor nie musi (albo musi rzadko) alokować nowej pamięci. W przypadku dużej planszy, ruchów jest naprawdę sporo (przy założeniu małego k), więc częściej jest realokowana pamięć i dużo większa jej ilość jest potrzebna. To wszystko kosztuje dodatkową wydajność.



Rysunek 4.12. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m i n dla dużej planszy

GetMoves() tak naprawdę udowadnia nam wcześniejszą obserwację. Tutaj pamięć jest alokowana jedynie raz na początku wywołania metody (bo wiemy dokładnie, ile ruchów jest wolnych, na podstawie tego, ile ruchów zostało już wykonanych). A potem cała operacja jest po prostu przejęciem po wszystkich polach planszy, sprawdzeniem, czy są wolne i umieszczeniem ich współrzędnych w tablicy pod kolejnym indeksem. Czas wykonania zależy więc liniowo od m i liniowo od n. Przy takich dużych wartościach czasowych maleje wpływ względny dokładności QPC i dlatego współczynnik R<sup>2</sup> dla korelacji kwadratowej wynosi tutaj aż 0.9889.



Rysunek 4.13. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru m i n dla dużej planszy

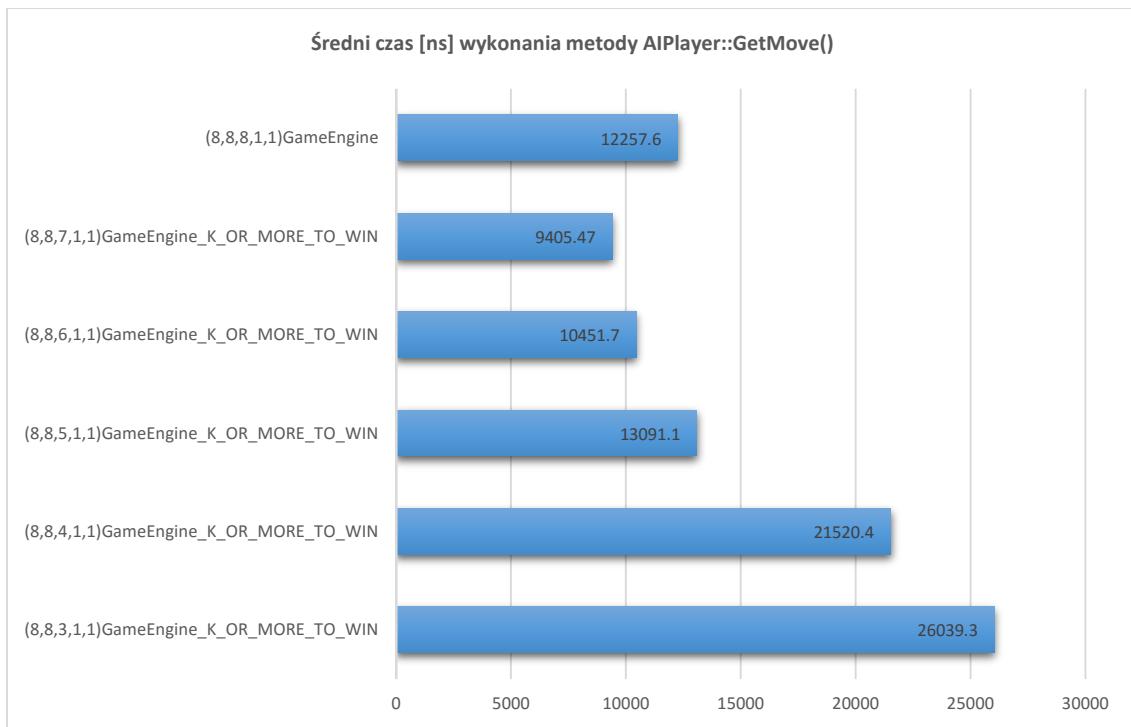
Sprawdzenie warunku wygranej jest generalnie bez zmian – nie zależy od rozmiaru planszy nawet przy bardzo dużych zmianach typu 50x50 -> 250x250.

Wnioski: czas generowania posunięć przez AI oraz czas generowania listy wolnych posunięć zależą liniowo od m i liniowo od n, a sprawdzenie warunku wygranej jest niezależne od m i n.

#### 4.6.2 Wydajność silników a parametr k

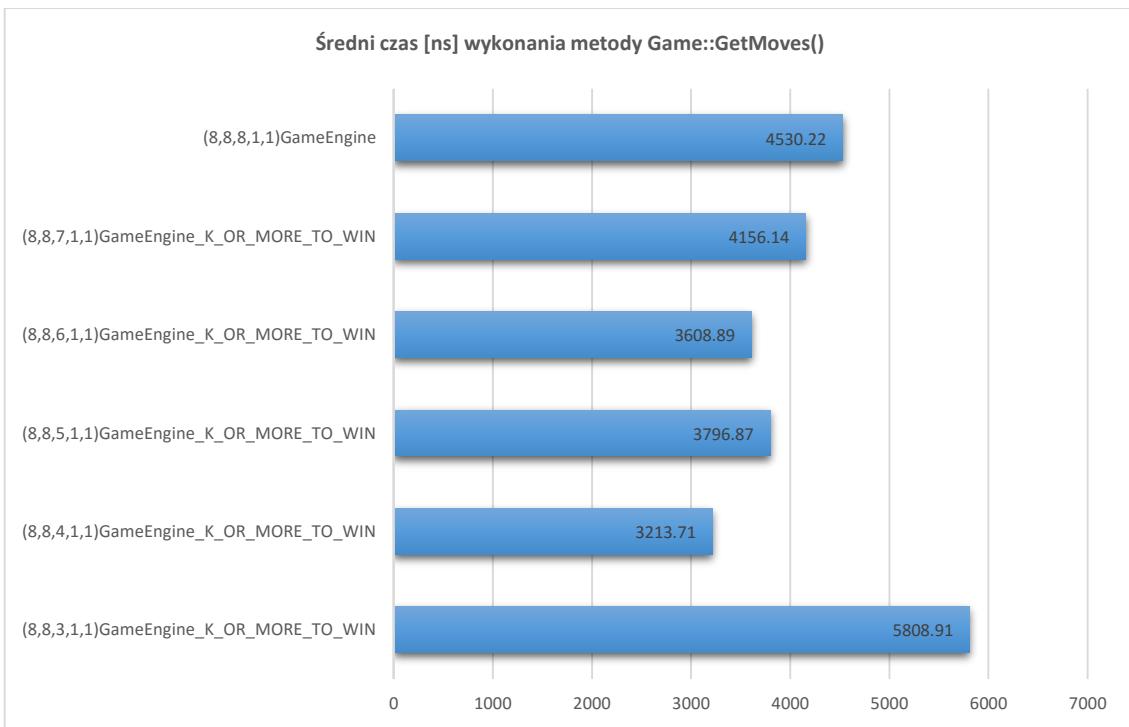
Znacznie ciekawszą zależnością do zbadania jest zależność od ilości potrzebnych pionów tego samego koloru pod rząd do wygranej. Parametr k jest używany m.in. w sprawdzaniu warunku wygranej – przymierza się ilość znalezionych pionów tego samego koloru pod rząd do jego wartości.

Testy zaczynają się od sprawdzenia zależności wydajnościowej silników dla parametru k na stosunkowo małej planszy 8x8 (jest to też największa plansza będąca bitboardem). Mniejszych plansz nie testujemy, ponieważ ograniczyłoby to bardzo mocno zakres wartości k, który tutaj jest od 3 do 8.



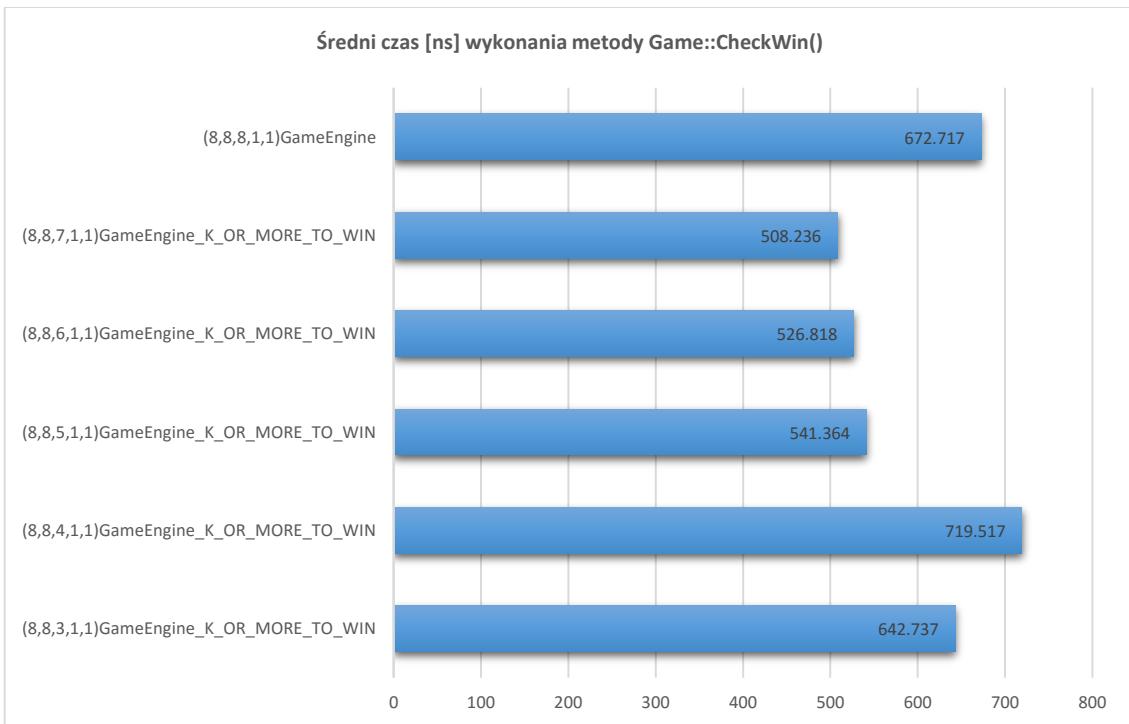
Rysunek 4.14. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla małej planszy

Wydaje się, że wraz ze wzrostem k maleje średni czas wykonania metody AIPlayer::GetMove(). Może się tak dziać, ponieważ większe k oznacza, że gra będzie trwać dłużej – jeżeli będzie dłużej trwać, to plansza będzie bardziej załoczona. Bardziej załoczona plansza oznacza generalnie, że trudniej znaleźć wolne miejsce sąsiadujące z miejscem zajętym, a w konsekwencji: rzadziej dodaje się kolejny ruch do wektora i rzadziej zachodzi alokacja dodatkowej pamięci.



Rysunek 4.15. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla małej planszy

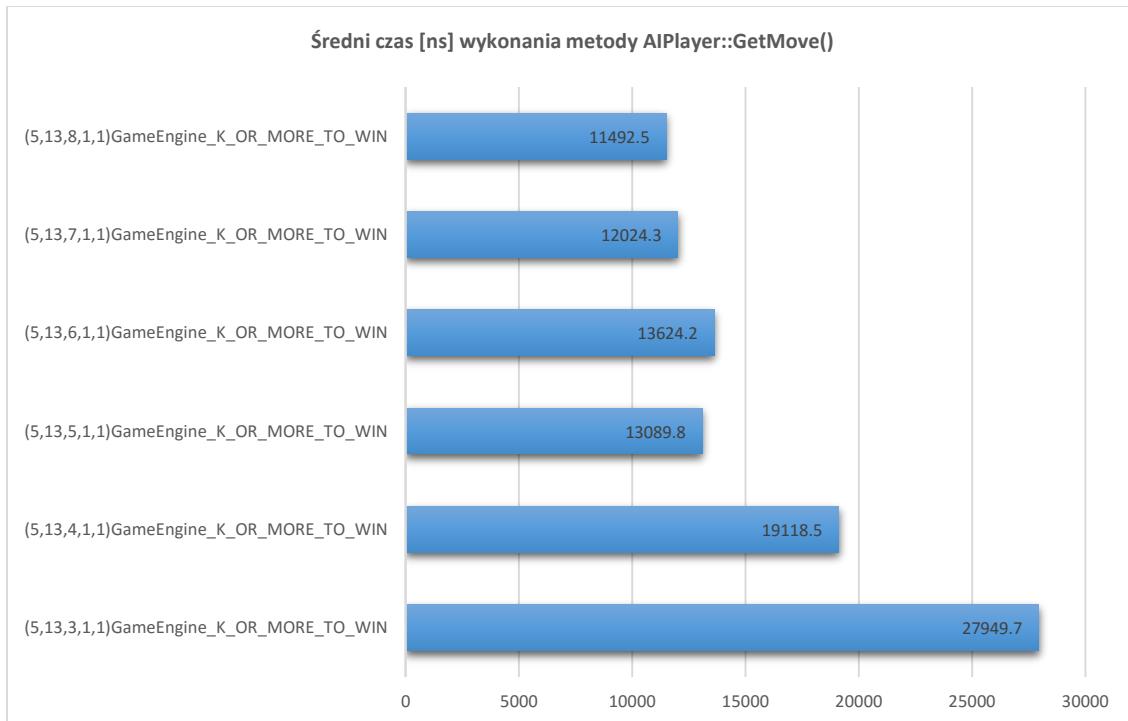
Dokładnie odwrotny trend jest widoczny dla wygenerowania wolnych posunięć. Im większe k, tym dłużej trwa generowanie listy wolnych posunięć. Póki co jednak trudno wyjaśnić jego przyczynę.



Rysunek 4.16. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla małej planszy

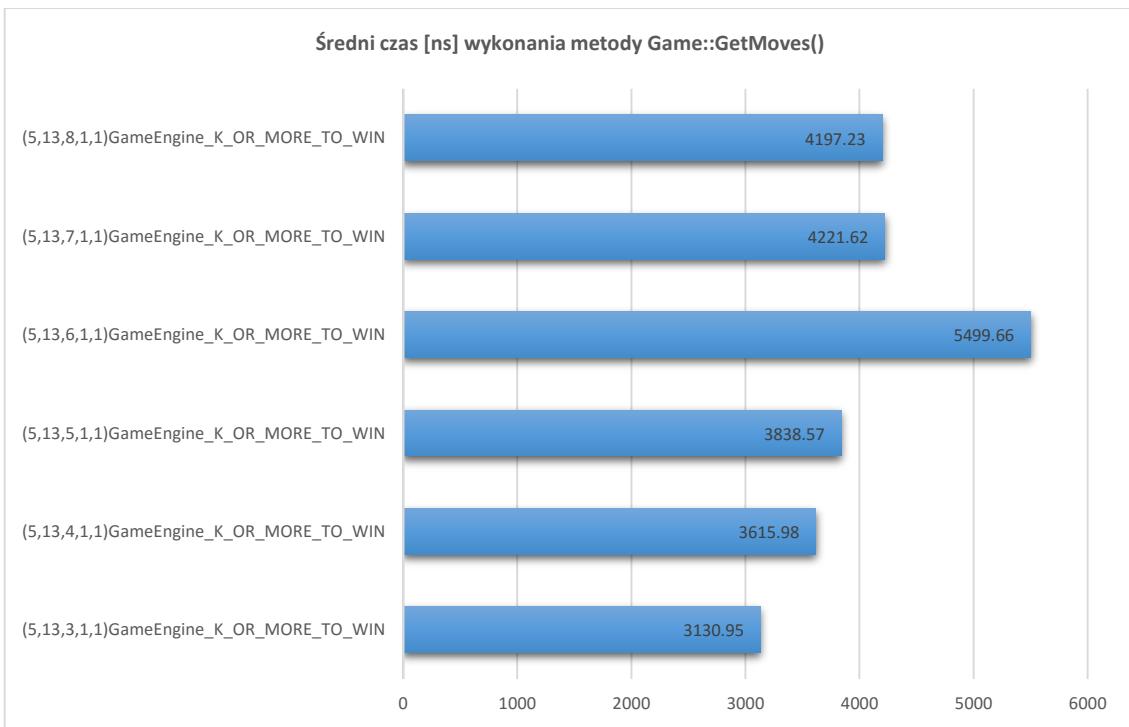
CheckWin wydaje się nie wykazywać żadnej zależności – zlicza przecież zawsze tyle, ile jest pionów pod rząd, i dopiero potem porównuje tę liczbę z k. Więc k nie powinno mieć istotnego wpływu na wykonanie tej metody.

Kolejna seria testów to sprawdzenie zależności od k z identycznymi parametrami, ale na planszy 5x13 – jest to stosunkowo mała plansza, ale już powyżej bitboardu.



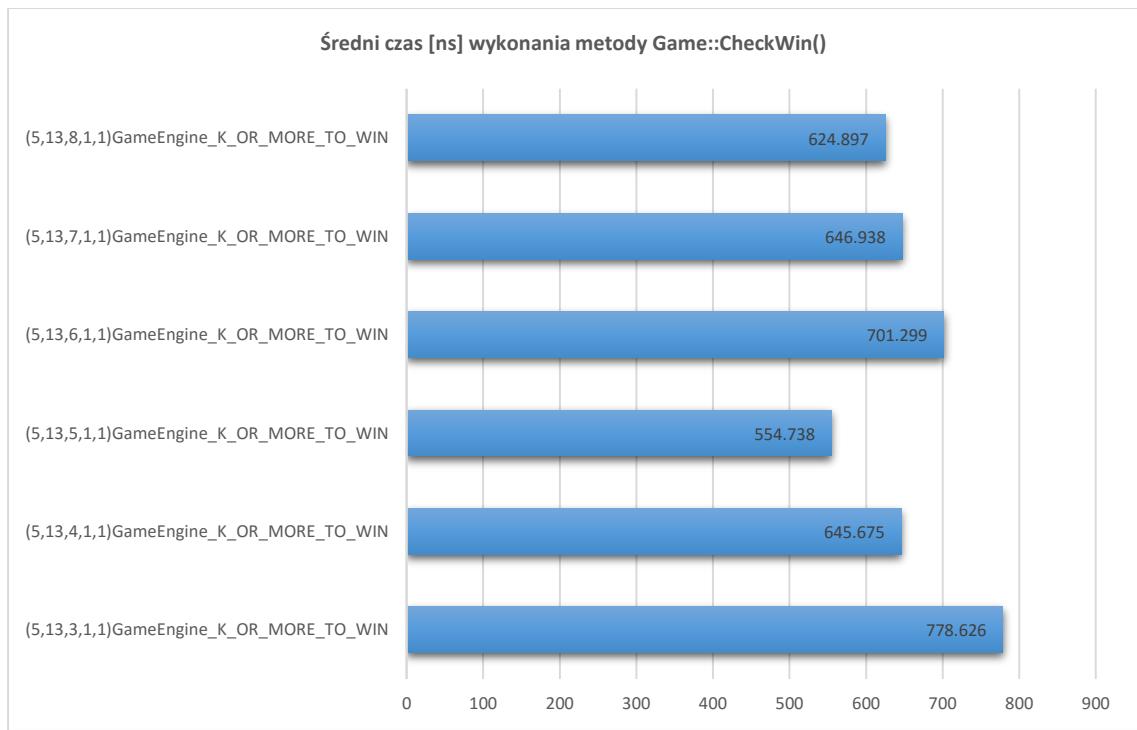
Rysunek 4.17. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla średniej planszy

Widać trend bardzo podobny jak w przypadku planszy 8x8 – im większa k, tym średnio AI generuje szybciej ruch. Przyczyna tutaj jest ta sama, co w planszy 8x8 – bardziej zatłoczona plansza, mniej alokacji pamięci na możliwe posunięcia AI.



Rysunek 4.18. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla średniej planszy

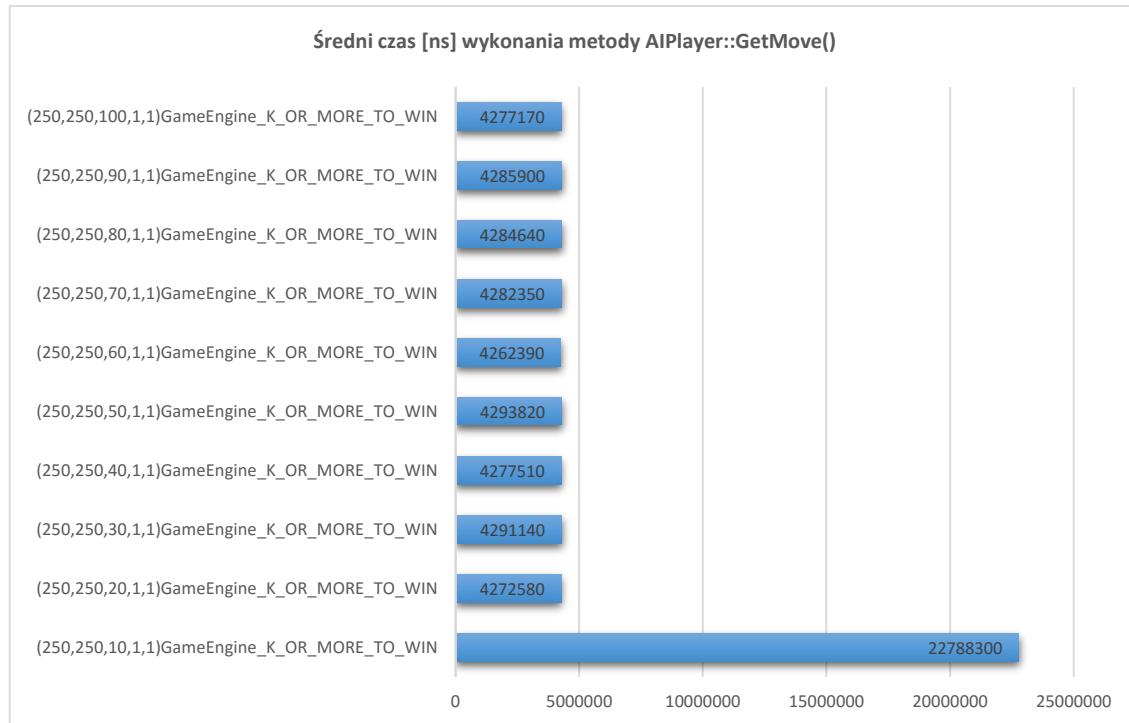
GetMoves() także ma ten sam trend co w planszy 8x8. Widać wyraźnie, że większe k oznacza wolniejsze wykonanie metody GetMoves(). Nadal trudno to wytlumaczyć, może to być dziełem przypadku. Ciekawym pomysłem na wyjaśnienie tego zjawiska jest istnienie jakiejś optymalizacji stosowanej przez kompilator, która istotnie zmienia kod – dopóki jednak nie ma na to żadnych dowodów, pozostaje to jedynie domysłem.



Rysunek 4.19. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla średniej planszy

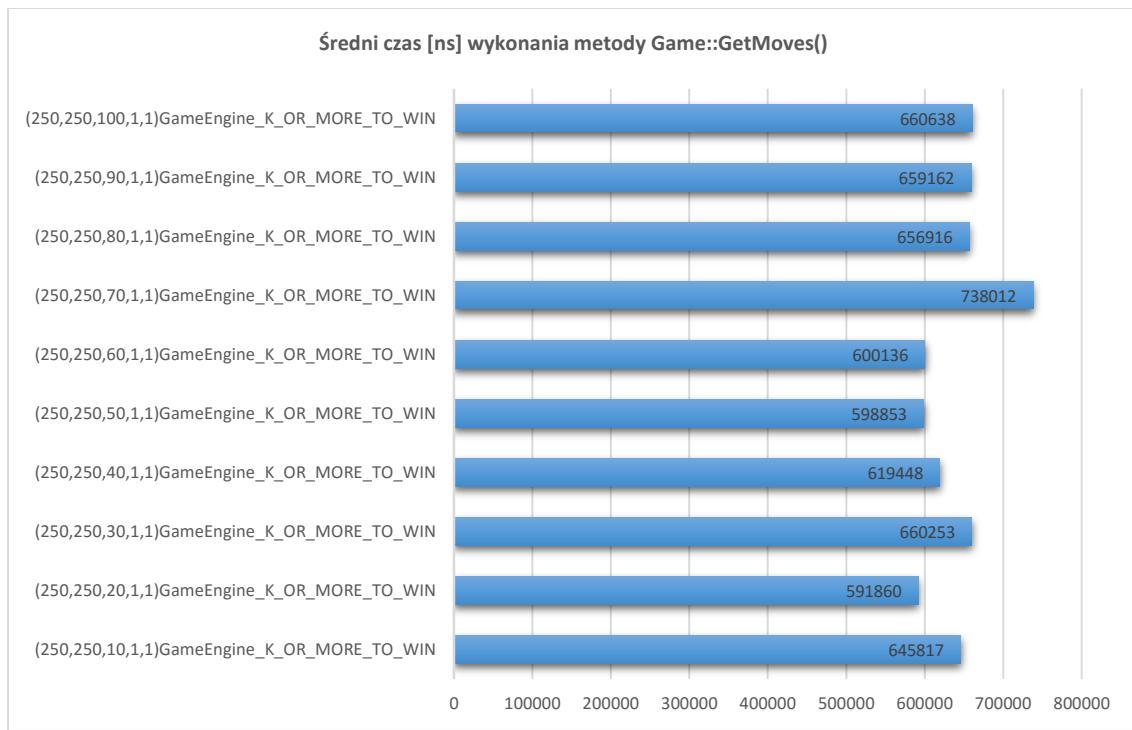
Mimo że te wyniki wydają się lekko sugerować wzrost czasu wykonania metody CheckWin() wraz z malejącym k, to raczej nie ma tutaj zależności.

Pozostaje sprawdzić zależność od k dla naprawdę dużej planszy 250x250. Pozwala to również na zbadanie zupełnie innego przedziału wartości k – od 10 do 100 ze wzrostem co 10.



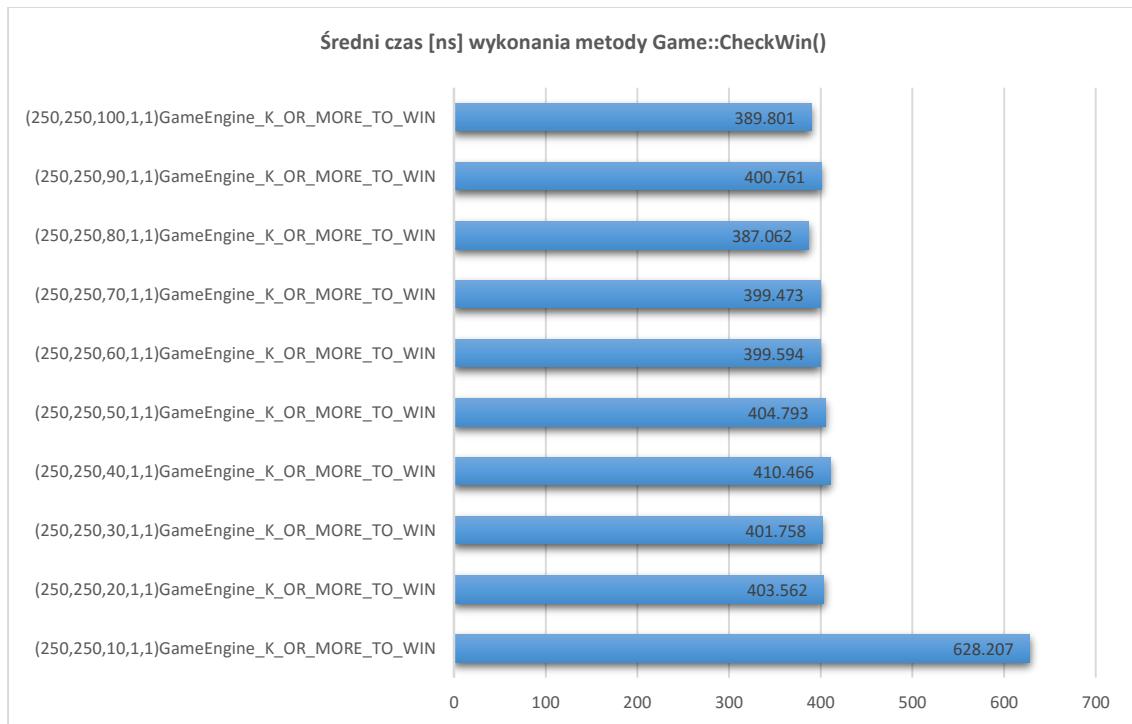
Rysunek 4.20. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla dużej planszy

Wyniki wydają się być bardzo dziwne. Były one jednak wielokrotnie powtarzane i zawsze otrzymywano podobny rezultat. Prawdopodobnie przeskok w czasie trwania gry z k=10 na k=20 jest tak duży, że nie można go nazywać ilościowym, tylko jakościowym. Dla k=10 gra się kończy w miarę szybko, a w związku z tym ma zupełnie inny charakter. Dla k=20 i więcej gra trwa nieporównywalnie długo (dowodzi to także korzystania z aplikacji testującej, która wyniki dla tej grupy silników zwraca szybko dla k=10, a bardzo powoli dla innych k). W nieporównywalnie dłuższej grze dla większych k, AI musi zdecydowanie szybciej znajdować puste pole sąsiadujące z polem zajętym. Dla tak dużej planszy nie ma znaczenia, czy k=20, czy 100, po prostu przy takich rozmiarach gra trwa długo, plansza ma w różnych miejscach piony, do których można dostawić inne piony. Coś, co wydaje się dziwnym trendem, jest w istocie normalną sytuacją, gdy mówimy o tak różnych czasach trwania pojedynczej rozgrywki. 10 pionów pod rząd ustawić jest jeszcze względnie łatwo i gra trwa krótko, 20 już bardzo ciężko i gra trwa zazwyczaj do remisu, czyli wykonania 250 ruchów.



Rysunek 4.21. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla dużej planszy

Wygenerowanie listy dostępnych posunięć nie zależy zbytnio od k przy tak dużej planszy. Zresztą nie powinno – i tak trzeba całą ją przejść.



Rysunek 4.22. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla dużej planszy

Ciekawostką jest, że trend dla sprawdzenia warunku wygranej wygląda podobnie do trendu dla wygenerowania ruchu przez AI. Przyczyną może być oczywiście zupełnie inny rodzaj rozgrywki dla k=10 i k=20 i więcej, ale jako że nie ma dobrego uzasadnienia na to, że długość rozgrywki powinna mieć wpływ na czas sprawdzenia warunku wygranej, to wydaje się, że jest to zbieg okoliczności. Poza tym różnice na poziomie kilku czy nawet kilkuset nanosekund są nadal bardzo niewielkimi wartościami.

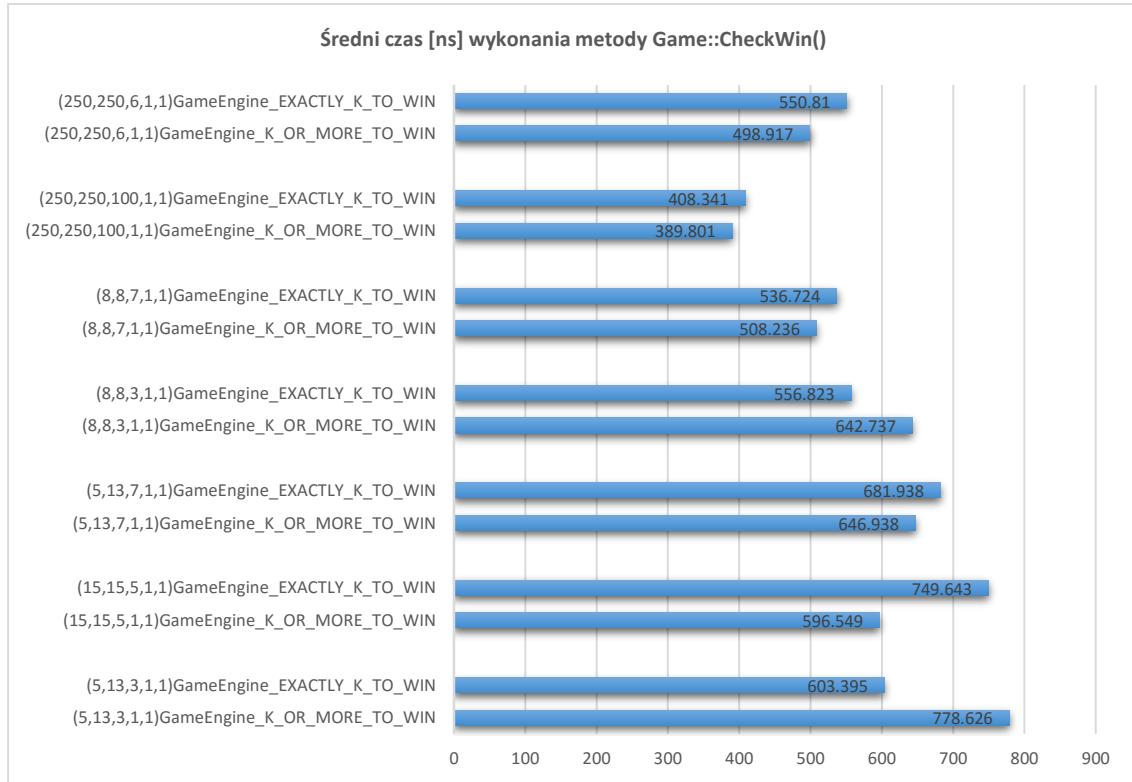
Wnioski:

- czas wygenerowania ruchu przez AI zależy w sposób uwikłany od k
  - maleje wraz z długością rozgrywki, która zależy od k (widoczne szczególnie dla dużych planszy)
  - maleje wraz ze stopniem załoczenia planszy (który zależy od k), bo bardziej załoczona plansza to mniej kosztownych czasowo alokacji pamięci (widoczne szczególnie dla małych planszy)
  - można więc powiedzieć, że generalnie maleje ze wzrostem k, ale zależność ta ma charakter bardzo złożony
- czas wygenerowania listy dostępnych posunięć raczej nie zależy od k
- czas sprawdzenia warunku wygranej raczej nie zależy od k.

#### 4.6.3 Wydajność silników a warunek wygranej

Warunek wygranej, czyli czy wymaga się dokładnie k pod rząd, czy może dopuszcza się również więcej, został poddany testom dla różnych wartości innych parametrów. W związku z tym część wyników pomiarowych bardzo od siebie odstawała (np. plansza 20x20 kontra plansza 250x250) i nie można ich pokazać na wspólnym wykresie.

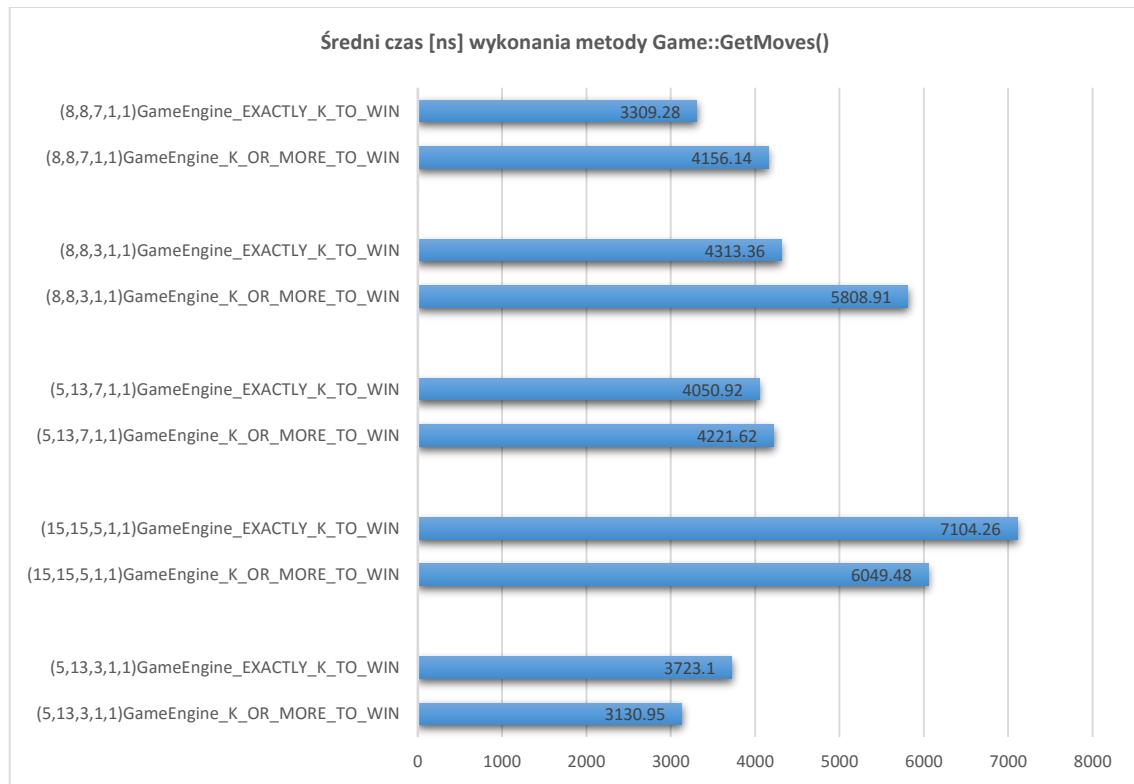
Dlatego też pierwszym testem jest czas sprawdzenia warunku wygranej, który właściwie nie zależy od innych parametrów i można różne silniki pokazać razem.



Rysunek 4.23. Zależność średniego czasu sprawdzenia warunku wygranej od warunku wygranej

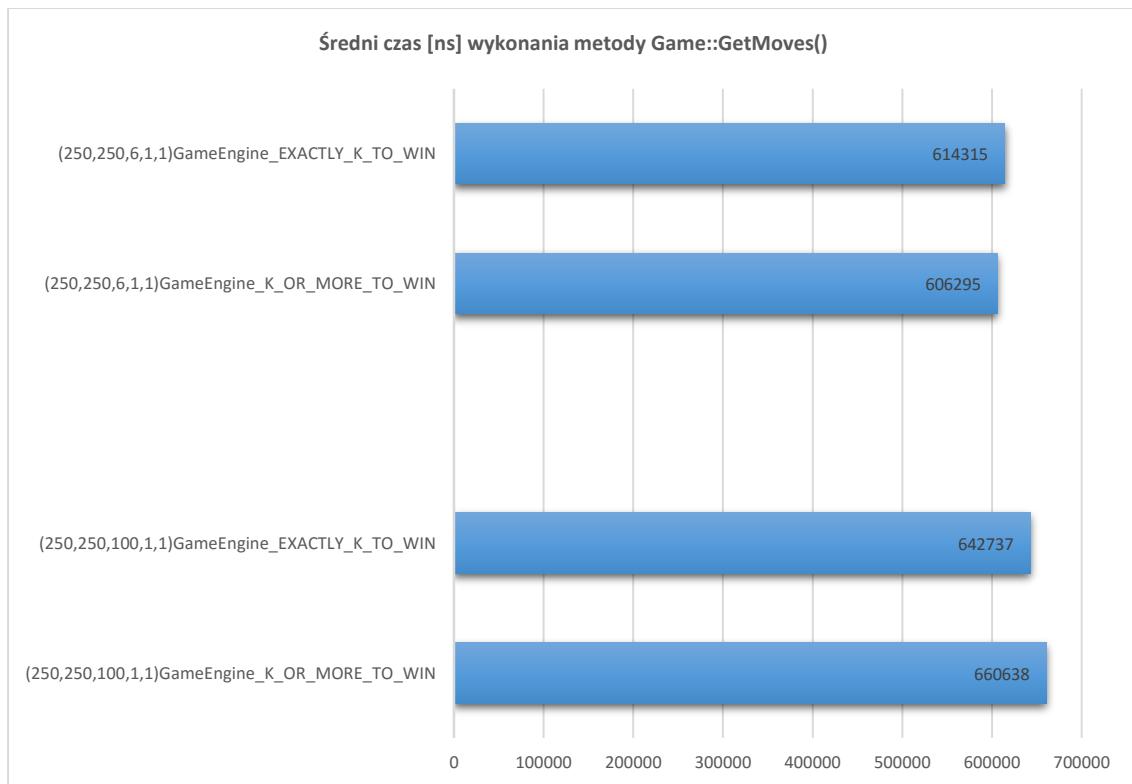
W przypadku bardzo dużych plansz wydaje się, że EXACTLY\_K\_TO\_WIN ma niższą wydajność (tym bardziej gdy k jest małe). Dla małych plansz z bardzo małym k jest dokładnie odwrotnie. W pozostałych przypadkach również szybszymi silnikami są te z K\_OR\_MORE\_TO\_WIN. Oczywiście różnice te są bardzo niewielkie, poniżej nawet wartości błędu pomiarowego, więc pewności nie ma. Mogą one wynikać z optymalizacji kompilatora lub zachowania samego kodu asemblerowego.

Wyniki metody GetMoves() musiały już być przedstawione na dwóch oddzielnych wykresach – jeden dla planszy małych i średnich, a drugi dla dużych. Na początek wyniki dla małych.



Rysunek 4.24. Zależność średniego czasu wygenerowania listy dostępnych posunięć od warunku wygranej dla małej i średniej planszy

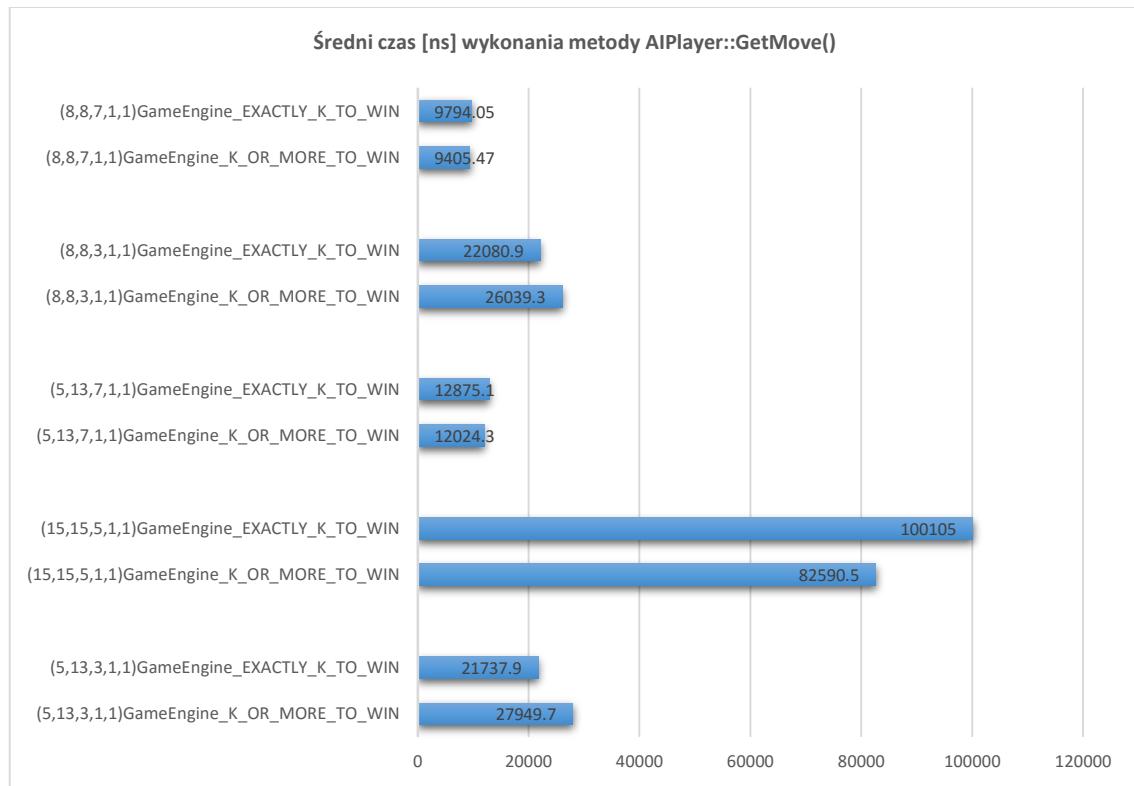
Trudno zinterpretować te rezultaty. Różnice na pewno są bardzo niewielkie, jednak wydaje się, że przynajmniej w przypadku bitboardu silniki z EXACTLY\_K\_TO\_WIN generują zauważalnie szybciej listę wolnych posunięć. Tym bardziej jednak brak jakichkolwiek podstaw teoretycznych tłumaczących to zjawisko.



Rysunek 4.25. Zależność średniego czasu wygenerowania listy dostępnych posunięć od warunku wygranej dla dużej planszy

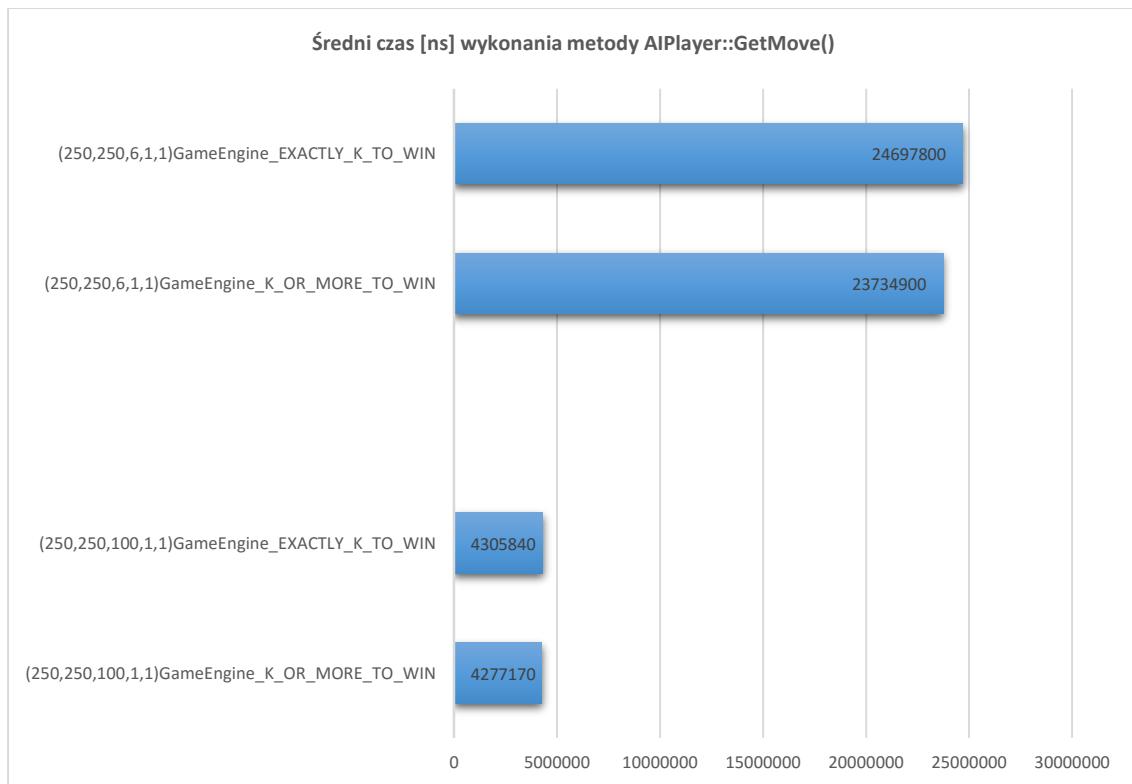
W silnikach z większymi planszami widać wyraźnie, że różnica jest bardzo niewielka.

Ostatnią testowaną dla tej zależności metryką jest AIPlayer::GetMove(), najpierw silniki małe i średnie.



Rysunek 4.26. Zależność średniego czasu wygenerowania ruchu przez AI od warunku wygranej dla małej i średniej planszy

Utrwala się trend, że w bitboardzie szybsze są silniki z EXACTLY\_K\_TO\_WIN, a w przypadku, gdy plansza jest przechowywana jako tablica – K\_OR\_MORE\_TO\_WIN.



Rysunek 4.27. Zależność średniego czasu wygenerowania ruchu przez AI od warunku wygranej dla dużej planszy

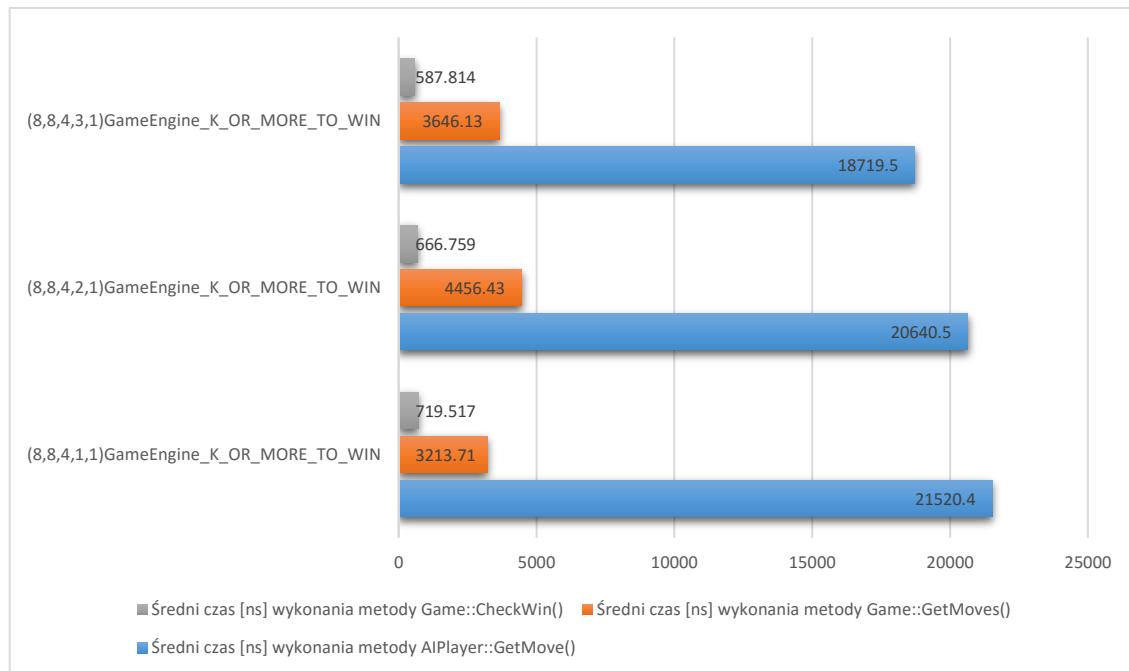
Większe plansze po raz kolejny pokazują, że wszelkie różnice mogą być przypadkowe.

Wnioski: warunek wygranej ma generalnie pomijalny wpływ na wydajność, ale wydaje się, że szybsze są silniki z EXACTLY\_K\_TO\_WIN dla reprezentacji bitboard i odwrotnie w przypadku reprezentacji planszy przez tablicę.

#### 4.6.4 Wydajność silników a parametry $p$ i $q$

Ciekawszą, a być może równie ciekawą jak zależność wydajności silników od parametru  $k$ , jest zależność od parametrów  $p$  i  $q$ . Parametr  $q$  jest używany właściwie tylko raz – w pierwszym ruchu, pierwszego gracza. Parametr  $p$  natomiast w każdym kolejnym ruchu. Mają one pośredni wpływ na załoczenie planszy – duże ich wartości (zwłaszcza  $p$ ) oznaczają sporo zajętych pól.

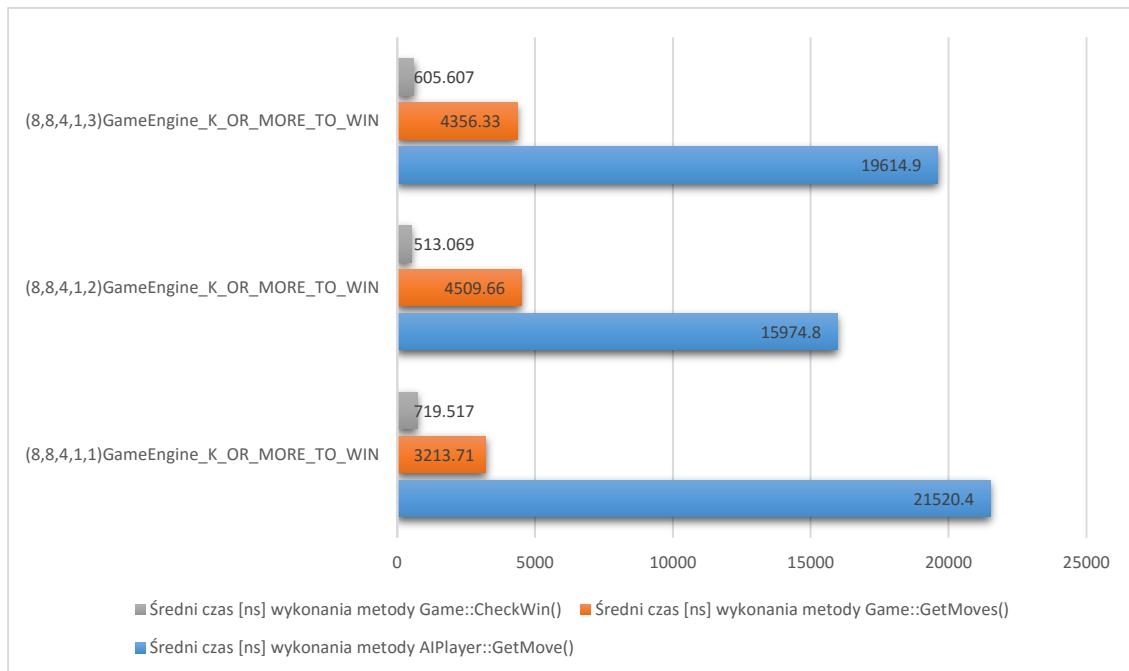
Na początek warto sprawdzić, jak zachowa się wydajność silników dla małej planszy z małym k i rosnącym p.



Rysunek 4.28. Wydajność silników w zależności od wartości parametru p dla małej planszy i małego k

Z racji małego k również p musi mieć niewielką wartość – dlatego zmienia się od 1 do 3. Wydaje się, że im większe p, tym większa wydajność silnika – przynajmniej w przypadku generowania ruchu przez AI i sprawdzania warunku wygranej. Może to być spowodowane stopniem zajętości planszy.

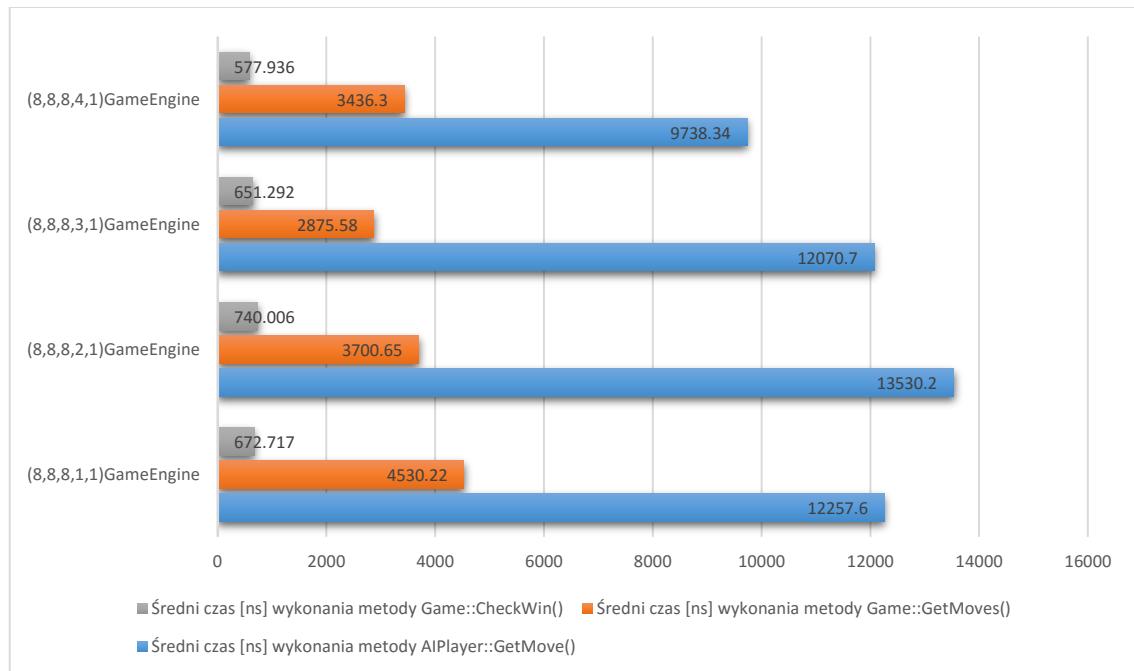
Następne w kolejce jest sprawdzenie dla wszystkich takich samych parametrów wzrostu q przy założeniu stałego p=1.



Rysunek 4.29. Wydajność silników w zależności od wartości parametru q dla małej planszy i małego k

Parametr q wydaje się nie mieć wpływu na wydajność silników, przynajmniej dla takich wartości pozostałych parametrów.

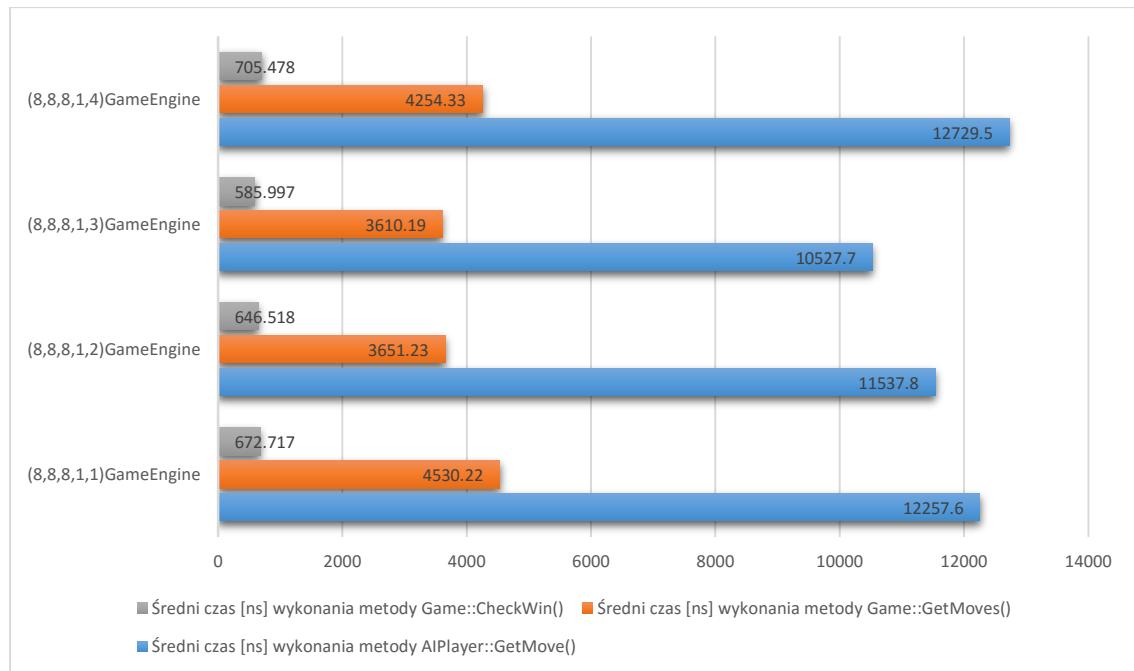
Teraz czas sprawdzić silniki z planszą w takim samym rozmiarze 8x8, ale z k=8, co umożliwia większy zakres wartości parametrów p i q od 1 do 4. Najpierw zależność od p przy założeniu stałego q.



Rysunek 4.30. Wydajność silników w zależności od wartości parametru p dla małej planszy i dużego k

Różnice w wydajności silników dla różnych wartości parametru p są bardzo niewielkie. Raczej wydaje się, że wydajność rośnie wraz z jego wzrostem – szczególnie widać to dla generowania listy dostępnych posunięć.

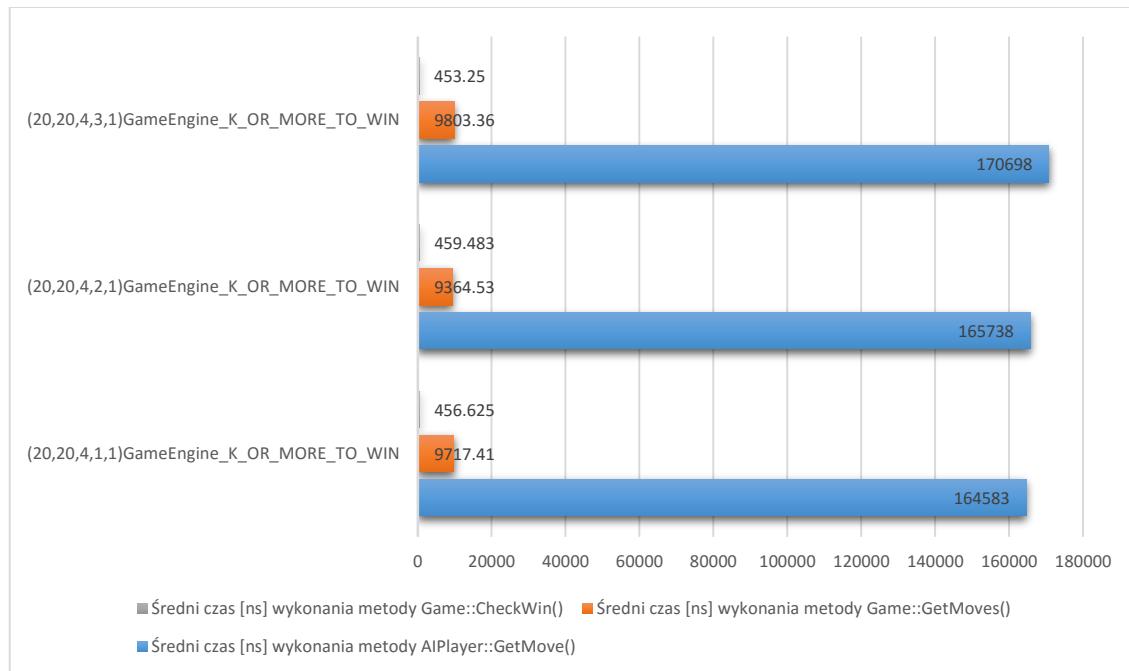
Pora sprawdzić to samo, ale dla parametru q.



Rysunek 4.31. Wydajność silników w zależności od wartości parametru q dla małej planszy i dużego k

Parametr q nie wydaje się mieć istotnego wpływu na wydajność. Słusznie, bo nie ma też żadnych przesłanek teoretycznych ku temu, przecież parametr q jest używany jedynie raz przez dany silnik w danej rozgrywce.

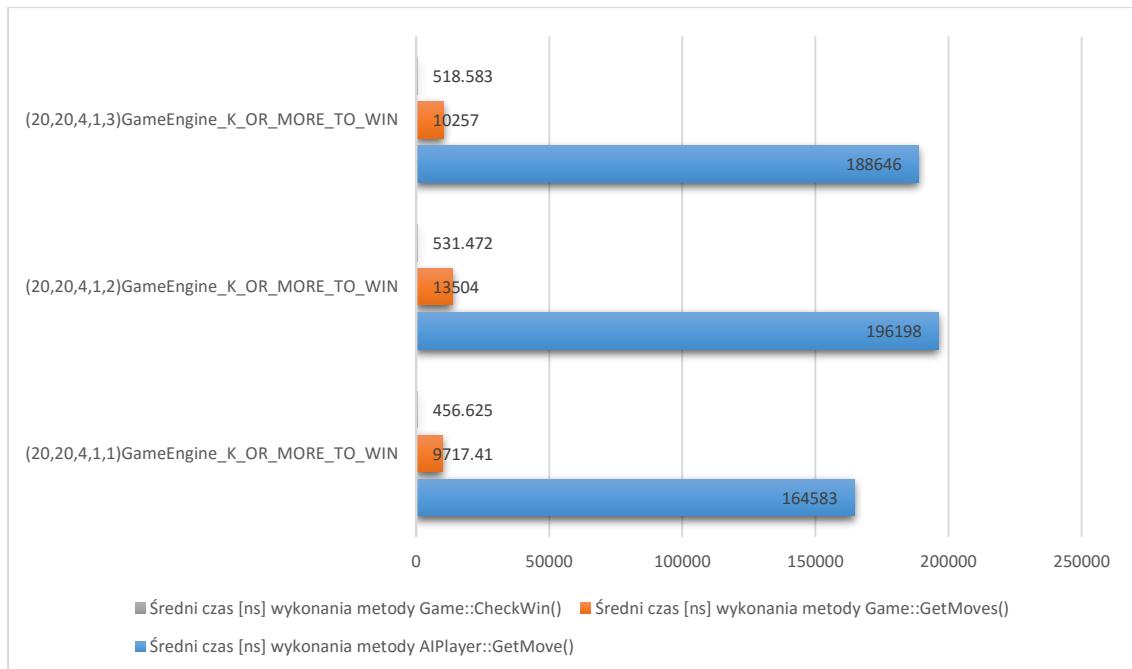
Kolejnym krokiem jest zbadanie wydajności silników w zależności od parametrów p i q dla planszy o rozmiarze 20x20. Najpierw zależność od p przy zachowaniu stałego q i niskiej wartości k.



Rysunek 4.32. Wydajność silników w zależności od wartości parametru p dla średniej planszy i małego k

W przypadku CheckWin() i GetMoves() p raczej nie ma istotnego wpływu na wydajność. Z kolei dla metody AIPlayer::GetMove() wydajność wydaje się maleć wraz ze wzrostem p, zupełnie odwrotnie niż było dla planszy 8x8.

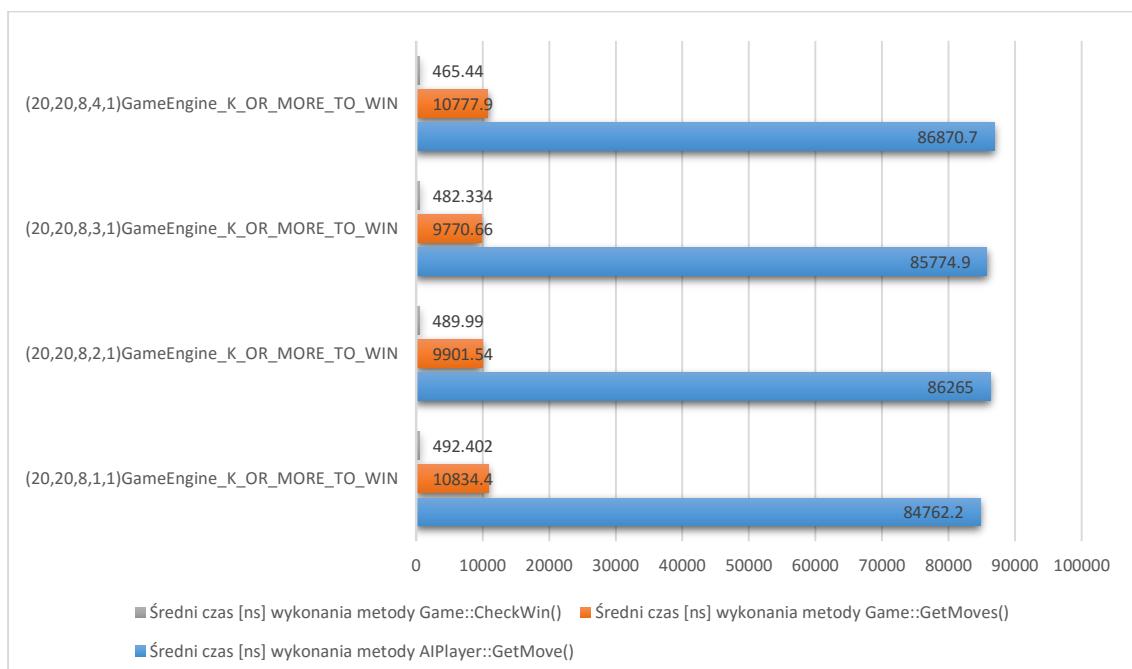
Czas na zależność od q przy stałym p=1.



Rysunek 4.33. Wydajność silników w zależności od wartości parametru q dla średniej planszy i małego k

Wartość q nie ma wpływu na wydajność także dla planszy o średnim rozmiarze.

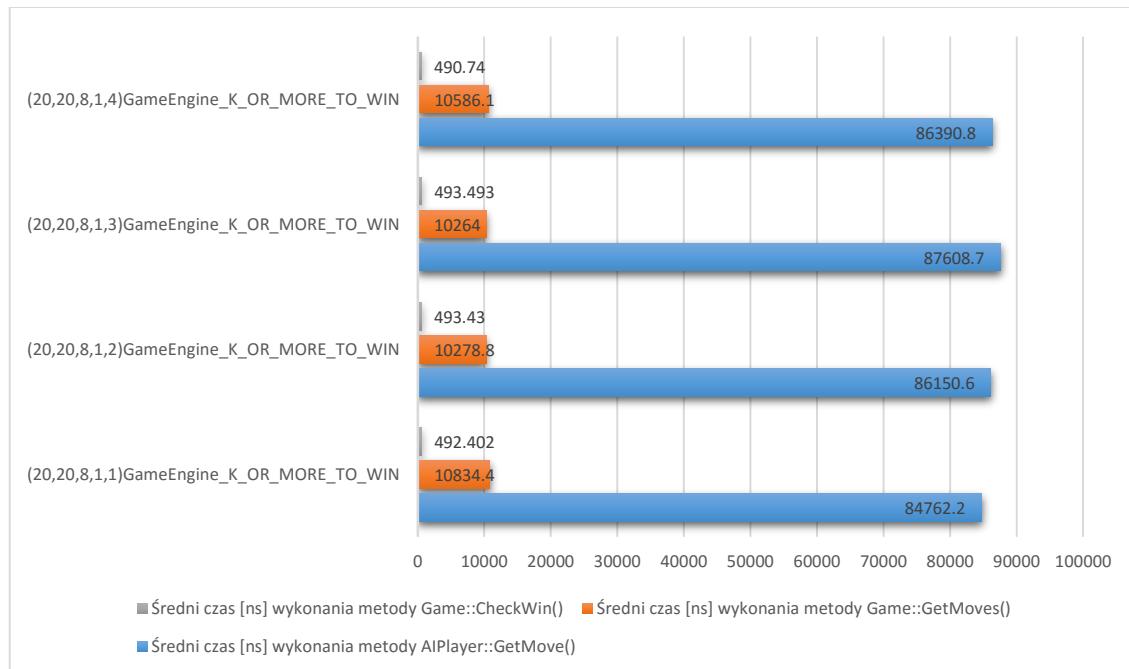
Trzeba również sprawdzić, czy trend będzie się powtarzał, gdy k będzie stosunkowo duże jak na planszę 20x20, czyli będzie wynosić 8. Poniżej zależność od p.



Rysunek 4.34. Wydajność silników w zależności od wartości parametru p dla średniej planszy i dużego k

Różnice są na tyle niewielkie, że nie można przypisywać im jakiegoś znaczenia. Możliwe jednak, że wraz ze wzrostem  $p$  nieznacznie wydłuża się czas odpowiedzi AI również i w tym wypadku.

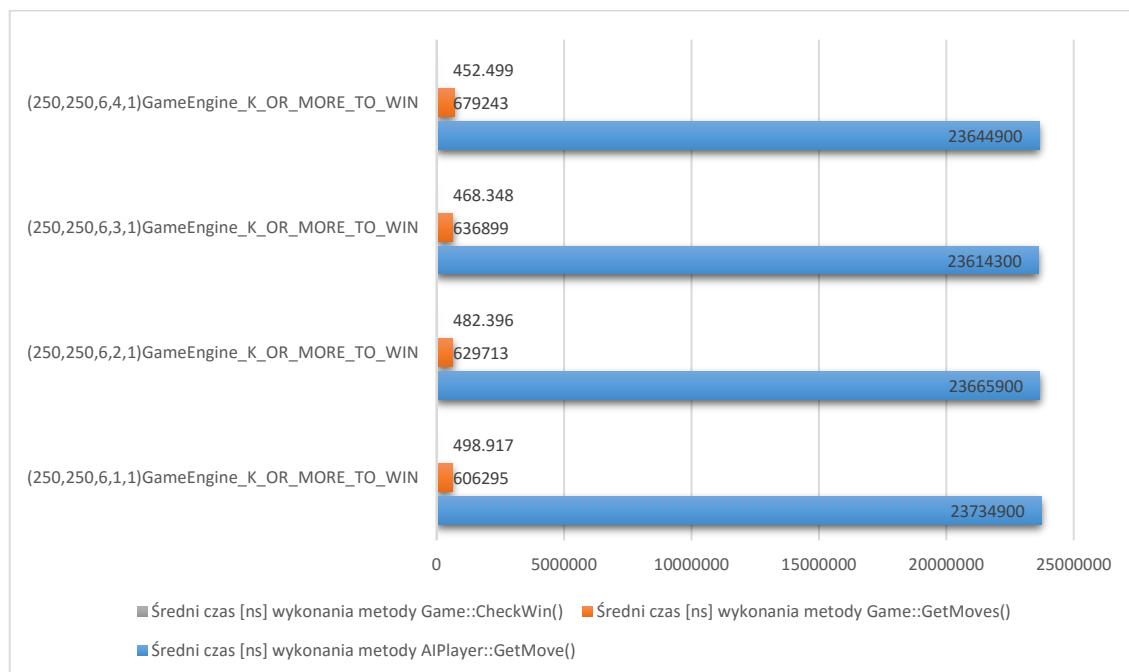
Sprawdzenie zależności od  $q$  w analogicznych warunkach pokazano poniżej.



Rysunek 4.35. Wydajność silników w zależności od wartości parametru  $q$  dla średniej planszy i dużego  $k$

Zmiana tego parametru również w tych warunkach nie ma wpływu na wydajność.

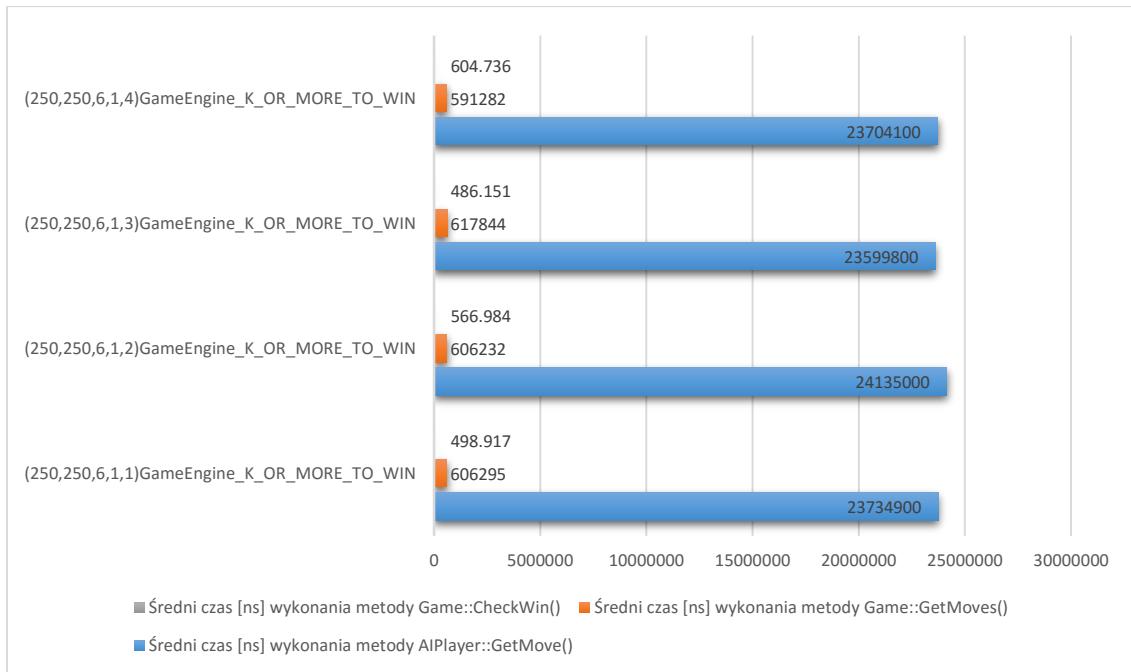
Następnie w schemacie testowym jest tzw. „waga ciężka”, czyli sprawdzenie zależności od  $p$  i  $q$  dla silników z planszami 250x250. Najpierw oczywiście mała wartość  $k$  i  $p$ .



Rysunek 4.36. Wydajność silników w zależności od wartości parametru  $p$  dla dużej planszy i małego  $k$

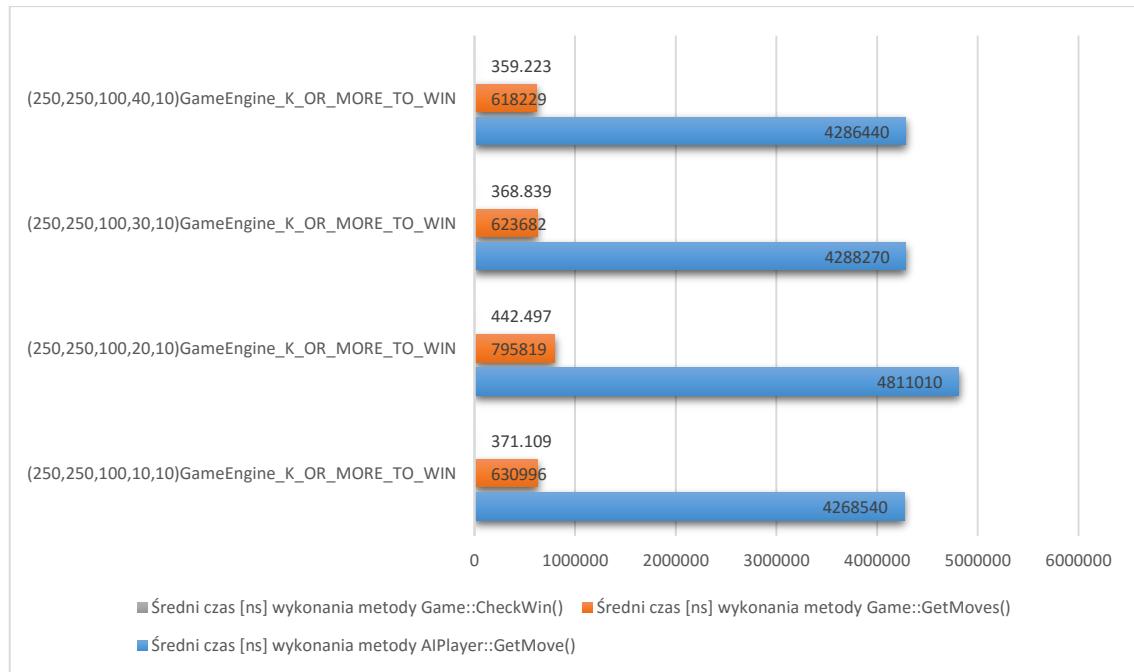
Dla takich parametrów różnice wydajnościowe wynikające z różnych wartości p są praktycznie zaniedbywalne. Widać jednak delikatny wzrost czasu wykonania procedury GetMoves() wraz ze wzrostem wartości p.

Czas sprawdzić te zależności dla q.



Rysunek 4.37. Wydajność silników w zależności od wartości parametru q dla dużej planszy i małego k  
Standardowo parametr q nie ma wpływu na wydajność jakiekolwiek z metryk.

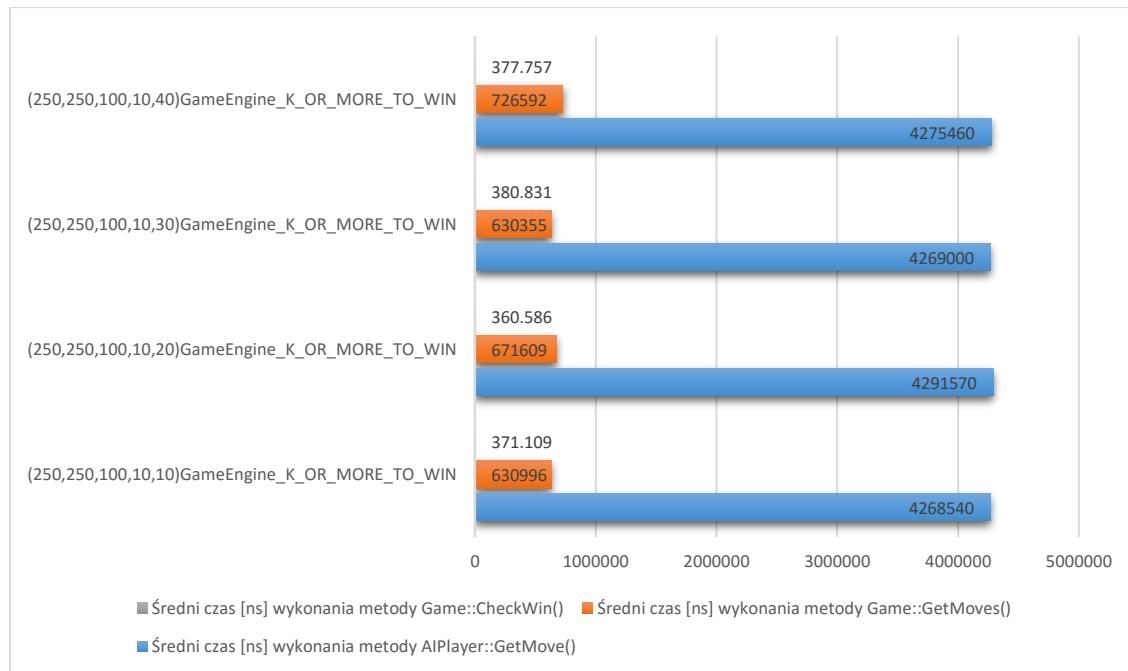
Kolejna seria testów będzie dla planszy o tym samym rozmiarze 250x250, ale dla dużych wartości k. Tak duża plansza umożliwia duże wartości k, co w efekcie pozwala na przetestowanie dużych wartości p i q. Zdecydowano się na sprawdzenie wartości od 10 do 40 ze wzrostem co 10. Najpierw zależność dla p.



Rysunek 4.38. Wydajność silników w zależności od wartości parametru p dla dużej planszy i dużego k

Dla tak dużej planszy, przy dużych wartościach parametru k, nie widać generalnie żadnych sensownych zależności metryk od wartości p.

Należy sprawdzić, czy metryki wydajnościowe nie zależą również od parametru q.



Rysunek 4.39. Wydajność silników w zależności od wartości parametru q dla dużej planszy i dużego k

Podobnie jest w przypadku q, nawet gdyby była jakaś niewielka zależność, to czasy te są tak duże, że po prostu tego nie widać.

Wnioski, co do zależności wydajności silników od parametrów p i q, są następujące:

- wydajność generowanych silników nie zależy od wartości q,
- im większy iloczyn m i n silnika, tym wpływ p na wydajność jest mniejszy
- dla silników z małą wartością iloczynu m i n wydajność rośnie wraz ze wzrostem p, efekt ten jest najbardziej zauważalny dla stosunkowo dużych wartości k

## 5. PODSUMOWANIE

Zdefiniowane na początku cele pracy udało się w pełni zrealizować. Implementacja generatora silników gier logicznych okazała się wyzwaniem, ale ostatecznie działa on w pełni poprawnie, a generowane silniki wykorzystują zalety nowoczesnych architektur procesorów oraz efektywną reprezentację stanów gry w pamięci. Zbadanie wydajności i poprawności dzięki zaimplementowaniu specjalnego programu testującego generowane silniki zostało zrealizowane z nawiązką. W dodatku jest możliwe łatwe dopisywanie przypadków testowych, które ma analizować aplikacja-tester. Obszerny przegląd i analiza dostępnych silników popularnych gier logicznych umożliwiły poznanie najnowszych trendów w implementacjach programów grających i zdobycie niezbędnej wiedzy do implementacji generatora.

Wybór gier z rodziny (m,n,k,p,q) umożliwił bogactwo parametrów do modyfikacji w generatorze. Program z pewnością pozwoliłby badaczom na lepsze poznanie gier z tej rodziny i zastanowienie się nad ich rozwiązywalnością i wydajnością zależnie od zdefiniowanych parametrów. Jedynie pozostaje niedosyt w zaimplementowanych algorytmach sztucznej inteligencji, umieszczanych w generowanych silnikach. Wydaje się, że należałoby generować AI z uwzględnieniem parametrów generowanych silników. Np. AI mogłoby zmieniać zachowanie dla konkretnych wartości m,n,k,p,q, szczególnie gdy wystąpią wartości reprezentujące grę już rozwiązana. Z pewnością AI to obszar dający duże możliwości do dalszego rozwoju generatora.

Dzięki wielowarstwowej, przemyślanej architekturze projekt umożliwia łatwą rozszerzalność o nowe testy wydajnościowe i poprawnościowe oraz silniki. W celu dodania nowego testu wystarczy dopisać nową metodę w odpowiedniej klasie a aplikacja sama wczyta i wykona ten test. Jest to duże pole do dalszych działań, które w przyszłości wraz z ujednoliconym protokołem komunikacji z silnikami mogłoby przerodzić część testującą projektu w prawdziwy framework testowy, taki jak Fishtest, tyle że dla gier z rodziny m,n,k,p,q.

W dalszych pracach można by także pokusić się o większy stopień skomplikowania parametrów w generowanych silnikach. Na przykład sensownym pomysłem byłoby umożliwienie generacji silników z nieskończonym parametrem m lub n. Oczywiście wymagałoby to dostosowania aplikacji GUI, ale z pewnością nie jest to niemożliwe, a badania nad takimi grami mogłyby być niezwykle interesujące. Idąc dalej, można by definiować niektóre parametry jako zależne od tury gry itp.

Jakkolwiek by na to nie patrzeć, możliwości dalszego rozwoju samego generatora, a nawet całej pracy, jest wiele. Jednym z prostszych zadań, które należałoby w przyszłości wykonać, byłoby stworzenie bardziej ustandaryzowanego protokołu łączności pomiędzy programem grającym a GUI dla gier z rodziny (m,n,k,p,q), tak jak UCI dla szachów. Obecne podejście ze znanym interfejsem wydaje się mieć pewne wady, np. interfejs nie jest nigdzie publicznie wystawiony.

Należy też podkreślić, że chociaż praca nie wyczerpała tematu, to jednak omówiono w niej najważniejsze wątki w podjętej tematyce. Widać też, że chociaż niektóre cechy silników są wspólne niezależnie od gry, to jednak niezwykle ważna jest w implementacji tzw. wiedza domenowa. Wiedza ta może być bardzo obszerna, zwłaszcza w przypadku złożonych gier

logicznych, jak choćby szachy czy Go. Niemniej jednak autor, nie posiadając wcześniej zbyt dużej wiedzy z tego zakresu, opanował przynajmniej jej część na potrzeby opracowania założonego projektu, więc nie jest to nieosiągalne.

Przyszłości silników gier logicznych należy też upatrywać w eksperymentowaniu z różnymi parametrami (ang. „tuning parameters” lub „tuning values”) i nieustannym testowaniu rezultatów. Świeśnie się tutaj sprawdza rozproszone środowisko testowe, takie jak Fishtest używany przez Stockfisha. Kto wie, może następne wielkie odkrycie w tematyce rozwiązywania gier zostanie właśnie dokonane podczas takich niewinnych eksperymentów i nie zostanie przeoczone, ponieważ ktoś postanowi je przetestować w podobnym rozproszonym środowisku testowym.

## WYKAZ LITERATURY

1. Łagodne wprowadzenie do analizy algorytmów, Marek Kubale, Wydawnictwo Politechniki Gdańskiej, 2016, ISBN: 978-83-7348-652-2
2. [http://en.wikipedia.org/wiki/Computer\\_chess](http://en.wikipedia.org/wiki/Computer_chess), (data dostępu 07.06.2016)
3. [http://en.wikipedia.org/wiki/Human–computer\\_chess\\_matches](http://en.wikipedia.org/wiki/Human–computer_chess_matches), (data dostępu 07.06.2016)
4. <http://www.fierz.ch/history.htm>, (data dostępu 08.06.2016)
5. [https://en.wikipedia.org/wiki/Top\\_Chess\\_Engine\\_Championship](https://en.wikipedia.org/wiki/Top_Chess_Engine_Championship), (data dostępu 09.06.2016)
6. [https://en.wikipedia.org/wiki/International\\_Computer\\_Games\\_Association](https://en.wikipedia.org/wiki/International_Computer_Games_Association), (data dostępu 09.06.2016)
7. [https://en.wikipedia.org/wiki/Computer\\_Olympiad](https://en.wikipedia.org/wiki/Computer_Olympiad), (data dostępu 10.06.2016)
8. [https://icga.leidenuniv.nl/?page\\_id=1112](https://icga.leidenuniv.nl/?page_id=1112), (data dostępu 10.06.2016)
9. Zasady XIX Olimpiady Gier Komputerowych ICGA <https://icga.leidenuniv.nl/wp-content/uploads/2016/03/Rules-ICGA-events-2016-3.pdf>, (data dostępu 11.06.2016)
10. [https://en.wikipedia.org/wiki/Solved\\_game](https://en.wikipedia.org/wiki/Solved_game), (data dostępu 11.06.2016)
11. [https://icga.leidenuniv.nl/?page\\_id=1315](https://icga.leidenuniv.nl/?page_id=1315), (data dostępu 11.06.2016)
12. <http://senseis.xmp.net/?ZenGoProgram>, (data dostępu 11.06.2016)
13. <https://book.mynavi.jp/tencho6/>, (data dostępu 11.06.2016)
14. <https://boardgamegeek.com/thread/609440/mohex-available-download>, (data dostępu 13.06.2016)
15. <http://benzene.sourceforge.net>, (data dostępu 13.06.2016)
16. <https://github.com/lukaszlew/MiMHex>, (data dostępu 13.06.2016)
17. <https://github.com/ala/MiMHex>, (data dostępu 13.06.2016)
18. <https://github.com/kdudzik/MiMHex>, (data dostępu 13.06.2016)
19. <https://github.com/krzysiocrash/MiMHex>, (data dostępu 13.06.2016)
20. <https://github.com/theolol/MiMHex>, (data dostępu 13.06.2016)
21. <https://github.com/bartoszborkowski/mimhex>, (data dostępu 13.06.2016)
22. <https://github.com/krzysiocrash/patterns>, (data dostępu 13.06.2016)
23. <https://github.com/qelo/MiMHex>, (data dostępu 13.06.2016)
24. <https://github.com/jakubpawlewicz/MiMHex>, (data dostępu 13.06.2016)
25. [https://en.wikipedia.org/wiki/Komodo\\_\(chess\)](https://en.wikipedia.org/wiki/Komodo_(chess)), (data dostępu 14.06.2016)
26. <https://komodochess.com>, (data dostępu 14.06.2016)
27. <https://komodochess.com/Komodo10-50a.htm>, (data dostępu 14.06.2016)
28. <https://komodochess.com/Komodo9-43a.htm>, (data dostępu 14.06.2016)
29. <https://komodochess.com/pub/komodo-8.zip>, (data dostępu 14.06.2016)
30. <https://stockfishchess.org>, (data dostępu 14.04.2016)
31. <https://github.com/official-stockfish/Stockfish>, (data dostępu 14.06.2016)
32. <https://en.wikipedia.org/wiki/M,n,k-game>, (data dostępu 16.06.2016)
33. <https://en.wikipedia.org/wiki/Tic-tac-toe>, (data dostępu 16.06.2016)
34. <https://en.wikipedia.org/wiki/Gomoku>, (data dostępu 16.06.2016)
35. <https://en.wikipedia.org/wiki/Connect6>, (data dostępu 16.06.2016)

36. A New Family of k-in-a-row Games, I-Chen Wu and Dei-Yen Huang, Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan  
<http://www.connect6.org/k-in-a-row.pdf>, (data dostępu 16.06.2016)
37. <http://risujin.org/connectk/>, (data dostępu 16.06.2016)
38. <http://www.connect6.org>, (data dostępu 16.06.2016)
39. <https://bitbucket.org/BadRobot/connect6/src>, (data dostępu 16.06.2016)
40. Kod źródłowy programu Connect-k, najnowsza wersja stabilna v2.0, czerwiec 2007  
<http://risujin.org/pub/connectk/connectk-2.0.tar.gz>, (data dostępu 16.06.2016)
41. <http://stackoverflow.com/questions/671703/array-index-out-of-bound-in-c/671709#671709>, (data dostępu 18.06.2016)
42. [https://msdn.microsoft.com/en-US/library/system.indexoutofrangeexception\(v=vs.110\).aspx](https://msdn.microsoft.com/en-US/library/system.indexoutofrangeexception(v=vs.110).aspx), (data dostępu 18.06.2016)
43. <https://en.wikipedia.org/wiki/Bitboard>, (data dostępu 19.06.2016)
44. [https://en.wikipedia.org/wiki/Bit\\_field](https://en.wikipedia.org/wiki/Bit_field), (data dostępu 19.06.2016)
45. [https://en.wikipedia.org/wiki/Universal\\_Chess\\_Interface](https://en.wikipedia.org/wiki/Universal_Chess_Interface), (data dostępu 20.06.2016)
46. [https://en.wikipedia.org/wiki/Stockfish\\_\(chess\)](https://en.wikipedia.org/wiki/Stockfish_(chess)), (data dostępu 20.06.2016)
47. <http://www.computerchess.org.uk/ccrl/404/>, (data dostępu 20.06.2016)
48. <http://www.computerchess.org.uk/ccrl/4040/>, (data dostępu 20.06.2016)
49. <http://www.computerchess.org.uk/ccrl/404FRC/>, (data dostępu 20.06.2016)
50. [http://en.wikipedia.org/wiki/64-bit\\_computing](http://en.wikipedia.org/wiki/64-bit_computing), (data dostępu 20.06.2016)
51. [https://en.wikipedia.org/wiki/SSE4#POPCNT\\_and\\_LZCNT](https://en.wikipedia.org/wiki/SSE4#POPCNT_and_LZCNT), (data dostępu 20.06.2016)
52. <https://chessprogramming.wikispaces.com/Stockfish>, (data dostępu 22.06.2016)
53. <https://chessprogramming.wikispaces.com/Square+Mapping+Considerations#LittleEndianRankFileMapping>, (data dostępu 22.06.2016)
54. Wykład "How do modern chess engines work?" – Daylen Yang, University of California, Berkeley  
<https://www.youtube.com/watch?v=pUyURF1Tqvg>, (data dostępu 22.06.2016)
55. <http://www.talkchess.com/forum/viewtopic.php?start=0&t=50220>, (data dostępu 23.06.2016)
56. <https://chessdailynews.com/im-erik-kislik-analyzes-the-tcec-superfinal-in-depth/>, (data dostępu 23.06.2016)
57. [https://en.wikipedia.org/wiki/Bit\\_Manipulation\\_Instruction\\_Sets](https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets), (data dostępu 24.06.2016)
58. [https://en.wikipedia.org/wiki/Find\\_first\\_set](https://en.wikipedia.org/wiki/Find_first_set), (data dostępu 24.06.2016)
59. [https://en.wikipedia.org/wiki/Intrinsic\\_function](https://en.wikipedia.org/wiki/Intrinsic_function), (data dostępu 24.06.2016)
60. [http://computerchess.org.uk/ccrl/404/cgi/compare\\_engines.cgi?family=Stockfish&print=Rating+list&print=Results+table&print=LOS+table&print=Ponder+hit+table&print=Eval+difference+table&print=Comopp+gamenumber+table&print=Overlap+table&print=Score+with+common+opponents](http://computerchess.org.uk/ccrl/404/cgi/compare_engines.cgi?family=Stockfish&print=Rating+list&print=Results+table&print=LOS+table&print=Ponder+hit+table&print=Eval+difference+table&print=Comopp+gamenumber+table&print=Overlap+table&print=Score+with+common+opponents), (data dostępu 25.06.2016)
61. <http://fastgm.de/60+0.60%20-%20E5450.html>, (data dostępu 25.06.2016)
62. <http://tests.stockfishchess.org/tests>, (data dostępu 25.06.2016)
63. <https://msdn.microsoft.com/en-US/library/hh567368.aspx>, (data dostępu 26.06.2016)
64. [http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support), (data dostępu 29.06.2016)
65. <http://stackoverflow.com/questions/871405/why-do-i-need-an-ioc-container-as-opposed-to-straightforward-di-code/1532254#1532254>, (data dostępu 01.07.2016)
66. <https://msdn.microsoft.com/en-US/library/dn961160.aspx>, (data dostępu 06.07.2016)

67. <https://msdn.microsoft.com/en-us/library/a569z7k8.aspx>, (data dostępu 06.07.2016)
68. <https://msdn.microsoft.com/en-US/library/74b4xzyw.aspx>, (data dostępu 06.07.2016)
69. [https://en.wikipedia.org/wiki/Inline\\_function](https://en.wikipedia.org/wiki/Inline_function), (data dostępu 12.07.2016)
70. <http://stackoverflow.com/questions/751681/meaning-of-const-last-in-a-c-method-declaration/751783#751783>, (data dostępu 15.07.2016)
71. <http://en.cppreference.com/w/cpp/language/cv>, (data dostępu 15.07.2016)
72. <http://www.highprogrammer.com/alan/rants/mutable.html>, (data dostępu 15.07.2016)
73. <http://stackoverflow.com/questions/1640258/need-a-fast-random-generator-for-c/3747462#3747462>, (data dostępu 15.07.2016)
74. <https://msdn.microsoft.com/en-US/library/4h2h0ktk.aspx>, (data dostępu 19.07.2016)
75. <http://www.possibility.com/Cpp/const.html>, (data dostępu 19.07.2016)
76. <https://msdn.microsoft.com/en-us/library/hh874757.aspx>, (data dostępu 20.07.2016)
77. [https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx), (data dostępu 20.07.2016)
78. [http://ark.intel.com/pl/products/43122/Intel-Core-i7-720QM-Processor-6M-Cache-1\\_60-GHz](http://ark.intel.com/pl/products/43122/Intel-Core-i7-720QM-Processor-6M-Cache-1_60-GHz), (data dostępu 25.07.2016)
79. <https://msdn.microsoft.com/en-us/library/dn986595.aspx>, (data dostępu 27.07.2016)
80. <https://blogs.msdn.microsoft.com/visualstudio/2016/06/27/visual-studio-2015-update-3-and-net-core-1-0-available-now/>, (data dostępu 30.07.2016)
81. <https://msdn.microsoft.com/en-us/library/system.idisposable.aspx>, (data dostępu 02.08.2016)
82. <https://msdn.microsoft.com/en-us/library/yh598w02.aspx> (data dostępu 02.08.2016)
83. <http://stackoverflow.com/questions/7485075/c-sharp-how-to-implement-dispose-method/7485263#7485263> (data dostępu 02.08.2016)
84. <https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx> (data dostępu 02.08.2016)
85. [https://en.wikipedia.org/wiki/Platform\\_Invocation\\_Services](https://en.wikipedia.org/wiki/Platform_Invocation_Services) (data dostępu 23.09.2016)
86. [https://en.wikipedia.org/wiki/File:Closeup\\_of\\_a\\_Connect\\_6\\_game.jpg](https://en.wikipedia.org/wiki/File:Closeup_of_a_Connect_6_game.jpg) (data dostępu 24.09.2016)
87. [https://pl.wikipedia.org/wiki/Współczynnik\\_determinacji](https://pl.wikipedia.org/wiki/Współczynnik_determinacji) (data dostępu 29.09.2016)
88. [https://en.wikipedia.org/wiki/Coefficient\\_of\\_determination](https://en.wikipedia.org/wiki/Coefficient_of_determination) (data dostępu 29.09.2016)

## WYKAZ RYSUNKÓW

Rysunek 2.1. Graf zależności w programie Stockfish.....	14
Rysunek 2.2. Mapowanie szachownicy na bitboard Little-Endian Rank-File [53] .....	15
Rysunek 2.3. Uruchomione obecnie testy na Fishteście [62] .....	22
Rysunek 2.4. Porównanie zmian wprowadzanych przez testowaną wersję [62].....	22
Rysunek 2.5. Graf zależności w programie Connect-k.....	23
Rysunek 2.6. Przykładowa rozgrywka w Connect-6 .....	24
Rysunek 2.7. Monte Carlo.....	36
Rysunek 3.1. Architektura generatora silników szachowych .....	41
Rysunek 3.2. "Code map" dla solucji zawierającej wszystkie projekty .....	42
Rysunek 3.3. Interfejs użytkownika aplikacji (m,n,k,p,q)EnginesGenerator .....	43
Rysunek 3.4. "Code map" dla projektu (m,n,k,p,q)EnginesGenerator.....	44
Rysunek 3.5. Załadowanie do interfejsu schematu silnika .....	46
Rysunek 3.6. Generator w trakcie pracy .....	48
Rysunek 3.7. Dodatkowa zmienna preprocesora \$(AdditionalPreprocessorDefinitions) .....	50
Rysunek 3.8. Pliki silników załączone zarówno jako oddzielny projekt, jak i content dla generatora .....	52
Rysunek 3.9. Optymalizacje kompilatora zastosowane w projekcie .....	53
Rysunek 3.10. Generacja silnika zakończona .....	54
Rysunek 3.11. Lokalizacja z wygenerowanymi silnikami.....	55
Rysunek 3.12. Uruchomienie ostatnio wygenerowanego silnika.....	56
Rysunek 3.13. Podpowiedzi komend do wysłania do silnika .....	58
Rysunek 3.14. Wysłanie komendy newgame black ai white ai do silnika .....	59
Rysunek 3.15. Wywołanie komendy "printboard" po zakończonej rozgrywce .....	61
Rysunek 3.16. Tryb "Batch generation" generatora .....	62
Rysunek 3.17. Dodanie nowego silnika do kolejki zadań generatora .....	63
Rysunek 3.18. Program w trakcie pracy w trybie "batch" .....	65
Rysunek 3.19. Stan programu po wygenerowaniu silników w trybie "batch" wraz z informacją o wydajności generatora .....	67
Rysunek 3.20. Aplikacja (m,n,k,p,q)EnginesAnalyzer tuż po uruchomieniu.....	105
Rysunek 3.21. Graf zależności w programie (m,n,k,p,q)EnginesAnalyzer .....	106
Rysunek 3.22. Aplikacja (m,n,k,p,q)EnginesAnalyzer w trakcie testowania wydajności silników .....	109
Rysunek 3.23. Aplikacja (m,n,k,p,q)EnginesAnalyzer po zakończeniu testowania wydajności silników.....	110
Rysunek 3.24. Aplikacja (m,n,k,p,q)EnginesAnalyzer w trakcie testowania poprawności silników .....	112
Rysunek 3.25. Aplikacja (m,n,k,p,q)EnginesAnalyzer po zakończeniu testowania poprawności silników.....	113
Rysunek 4.1. Czasy generowania silników .....	118

Rysunek 4.2. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru m.....	128
Rysunek 4.3. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m .....	129
Rysunek 4.4. Zależność średniego sprawdzenia warunku wygranej od wartości parametru m .....	130
Rysunek 4.5. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru n.....	131
Rysunek 4.6. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru n .....	132
Rysunek 4.7. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru n.....	133
Rysunek 4.8. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru m i n.....	134
Rysunek 4.9. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m i n .....	135
Rysunek 4.10. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru m i n .....	136
Rysunek 4.11. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametrów m i n dla dużej planszy.....	137
Rysunek 4.12. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru m i n dla dużej planszy .....	138
Rysunek 4.13. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru m i n dla dużej planszy .....	139
Rysunek 4.14. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla małej planszy .....	140
Rysunek 4.15. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla małej planszy .....	141
Rysunek 4.16. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla małej planszy .....	141
Rysunek 4.17. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla średniej planszy .....	142
Rysunek 4.18. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla średniej planszy .....	143
Rysunek 4.19. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla średniej planszy .....	144
Rysunek 4.20. Zależność średniego czasu wygenerowania ruchu przez AI od wartości parametru k dla dużej planszy .....	145
Rysunek 4.21. Zależność średniego czasu wygenerowania listy dostępnych posunięć od wartości parametru k dla dużej planszy .....	146

Rysunek 4.22. Zależność średniego czasu sprawdzenia warunku wygranej od wartości parametru k dla dużej planszy .....	146
Rysunek 4.23. Zależność średniego czasu sprawdzenia warunku wygranej od warunku wygranej .....	148
Rysunek 4.24. Zależność średniego czasu wygenerowania listy dostępnych posunięć od warunku wygranej dla małej i średniej planszy .....	149
Rysunek 4.25. Zależność średniego czasu wygenerowania listy dostępnych posunięć od warunku wygranej dla dużej planszy .....	150
Rysunek 4.26. Zależność średniego czasu wygenerowania ruchu przez AI od warunku wygranej dla małej i średniej planszy .....	151
Rysunek 4.27. Zależność średniego czasu wygenerowania ruchu przez AI od warunku wygranej dla dużej planszy.....	152
Rysunek 4.28. Wydajność silników w zależności od wartości parametru p dla małej planszy i małego k.....	153
Rysunek 4.29. Wydajność silników w zależności od wartości parametru q dla małej planszy i małego k.....	154
Rysunek 4.30. Wydajność silników w zależności od wartości parametru p dla małej planszy i dużego k.....	155
Rysunek 4.31. Wydajność silników w zależności od wartości parametru q dla małej planszy i dużego k.....	156
Rysunek 4.32. Wydajność silników w zależności od wartości parametru p dla średniej planszy i małego k.....	157
Rysunek 4.33. Wydajność silników w zależności od wartości parametru q dla średniej planszy i małego k.....	158
Rysunek 4.34. Wydajność silników w zależności od wartości parametru p dla średniej planszy i dużego k.....	158
Rysunek 4.35. Wydajność silników w zależności od wartości parametru q dla średniej planszy i dużego k.....	159
Rysunek 4.36. Wydajność silników w zależności od wartości parametru p dla dużej planszy i małego k.....	159
Rysunek 4.37. Wydajność silników w zależności od wartości parametru q dla dużej planszy i małego k.....	160
Rysunek 4.38. Wydajność silników w zależności od wartości parametru p dla dużej planszy i dużego k.....	161
Rysunek 4.39. Wydajność silników w zależności od wartości parametru q dla dużej planszy i dużego k.....	162

## WYKAZ TABEL

<b>Tabela 2.1.</b> Wyniki turnieju "Top Chess Engine Championship" [5] .....	10
<b>Tabela 2.2.</b> Wyniki w grze Go 9x9 [11] .....	11
<b>Tabela 2.3.</b> Wyniki w grze Hex 11x11 [11] .....	11
<b>Tabela 2.4.</b> Wyniki w grze Hex 13x13 [11] .....	11
<b>Tabela 2.5.</b> Wyniki w grze Connect6 [11] .....	12
<b>Tabela 3.1.</b> Zalety i wady – gry proste, złożone, znane i mniej znane .....	39
<b>Tabela 4.1.</b> Konfiguracja sprzętowa platformy testowej .....	116
<b>Tabela 4.2.</b> Konfiguracja systemowa platformy testowej.....	116
<b>Tabela 4.3.</b> Dane statystyczne dla procesu generowania silników .....	118

## Dodatek A: Zawartość załączonej płyty CD

Poniżej przedstawiono zawartość załączonej do pracy płyty CD. Należy mieć na uwadze, że lista ta nie zawiera wszystkich plików zamieszczonych na płycie, ani też wszystkich folderów. Ma ona na celu jedynie pokazanie struktury płyty i umożliwienie łatwej nawigacji w plikach powiązanych z niniejszą pracą magisterską.

- LogicalGamesEnginesGenerator
    - Pliki solucji LogicalGamesEnginesGenerator zawierającej wszystkie projekty, nadrzedna lokacja dla wszystkich plików i folderów powiązanych z pracą magisterską
  - LogicalGamesEnginesGenerator\m,n,k,p,q)EnginesAnalyzer
    - Kod źródłowy i pliki projektowe aplikacji do analizy poprawnościowo-wydajnościowej generowanych silników napisanej w C#, .NET, WPF
  - LogicalGamesEnginesGenerator\m,n,k,p,q)EnginesGenerator
    - Kod źródłowy i pliki projektowe generatora silników gier logicznych z rodziny (m,n,k,p,q) napisanego w C#, .NET, WPF
  - LogicalGamesEnginesGenerator\m,n,k,p,q)EnginesGenerator\m,n,k,p,q)GameEngine
    - Kod źródłowy i pliki projektowe parametryzowanego silnika gier z rodziny (m,n,k,p,q) napisanego w C++
  - LogicalGamesEnginesGenerator\m,n,k,p,q)EngineWrapper
    - Kod źródłowy i pliki projektowe biblioteki .NET ułatwiającej współpracę w środowisku .NET z generowanymi silnikami
  - LogicalGamesEnginesGenerator\LogicalGamesEnginesGenerator.sln
    - Solucja dla wszystkich autorskich projektów zrealizowanych w ramach pracy magisterskiej
  - LogicalGamesEnginesGenerator\docs\ul>  - Dokumenty tekstowe w różnych formatach powiązane w jakiś sposób z pracą magisterską
- LogicalGamesEnginesGenerator\docs\articles\ul>- Artykuły naukowe powiązane z tematyką pracy
- LogicalGamesEnginesGenerator\docs\formalities\ul>- Dokumenty powiązane z wymaganiami formalnymi dotyczącymi pracy
- LogicalGamesEnginesGenerator\docs\seminar\ul>- Dokumenty powiązane z seminarium dyplomowym magisterskim
- LogicalGamesEnginesGenerator\docs\thesis\ul>- Dokumenty powiązane bezpośrednio z pracą, takie jak praca magisterska, strona tytułowa, oświadczenie i wymagania merytoryczne odnośnie pracy
- LogicalGamesEnginesGenerator\docs\thesis\MasterThesis-pl.docx
  - Niniejsza praca magisterska w formacie docx, napisana w języku polskim
- LogicalGamesEnginesGenerator\engines\

- Kody źródłowe wszystkich silników gier logicznych opisywanych w pierwszym rozdziale pracy
  - Zawiera również kody źródłowe silników, które służyły jedynie jako inspiracja lub źródło wiedzy dla autora, ale nie zostały opisane w pracy
  - Może zawierać pliki wykonywalne (.exe) niektórych silników
- LogicalGamesEnginesGenerator\engines\benzene-benzene-7b1f7f7ecc2a9a83f7948c8e34409f5462eaa33b
  - Kod źródłowy pakietu benzen i silnika MoHex w wersji z listopada 2011 roku
- LogicalGamesEnginesGenerator\engines\connectk-2.0
  - Kod źródłowy silnika Connect-k
- LogicalGamesEnginesGenerator\engines\MiMHex-master
  - Kod źródłowy silnika MiMHex w najnowszej, publicznie dostępnej wersji
- LogicalGamesEnginesGenerator\engines\Stockfish-master
  - Kod źródłowy silnika Stockfish w najnowszej dostępnej wersji w chwili pisania pracy
- LogicalGamesEnginesGenerator\GeneratedEngines
  - Wygenerowane przez (m,n,k,p,q)EnginesGenerator silniki w formie plików wykonywalnych (.exe)
  - Używane w analizie poprawnościowo-wydajnościowej przez aplikację testującą (m,n,k,p,q)EnginesAnalyzer do uzyskania wyników dotyczących poprawności i wydajności generowanych silników
- LogicalGamesEnginesGenerator\packages
  - Pakiety NuGet używane w projektach
- LogicalGamesEnginesGenerator\Release v1.0.0\
  - Skompilowane (wykonywalne) wersje autorskich projektów
  - Programy gotowe do uruchomienia pod warunkiem zainstalowanego środowiska .NET Framework w wersji przynajmniej 4.6.1
- LogicalGamesEnginesGenerator\Release v1.0.0\ (m,n,k,p,q)EnginesAnalyzer\
  - Skompilowana wersja 1.0.0 programu (m,n,k,p,q)EnginesAnalyzer
  - Oficjalna wersja aplikacji używana na potrzeby pracy magisterskiej
- **LogicalGamesEnginesGenerator\Release v1.0.0\ (m,n,k,p,q)EnginesAnalyzer\ (m,n,k,p,q)EnginesAnalyzer.exe**
  - Plik wykonywalny aplikacji (m,n,k,p,q)EnginesAnalyzer
  - Uruchamia aplikację do analizy poprawnościowo-wydajnościowej generowanych silników gier logicznych
- **LogicalGamesEnginesGenerator\Release v1.0.0\ (m,n,k,p,q)EnginesGenerator\ (m,n,k,p,q)EnginesGenerator.exe**
  - Plik wykonywalny aplikacji (m,n,k,p,q)EnginesGenerator
  - Uruchamia generator silników gier logicznych
- LogicalGamesEnginesGenerator\Release v1.0.0\ (m,n,k,p,q)EnginesGenerator\

- Skompilowana wersja 1.0.0 programu (m,n,k,p,q)EnginesGenerator
- Oficjalna wersja aplikacji używana na potrzeby pracy magisterskiej
- LogicalGamesEnginesGenerator\Release v1.0.0\benchmarkEnginesList.bin
  - Możliwa do wczytania w programie (m,n,k,p,q)EnginesGenerator lista parametrów silników do wygenerowania na potrzeby powtórzenia wszystkich testów generatora przeprowadzonych w pracy