# Streszczenie

W niniejszej pracy celem była implementacja i analiza wydajnościowa generatora silników gier logicznych. Aby jednak go prawdiłowo zaimplementować najpierw należało dokonać dokładnego przeglądu i analizy dostępnych implementacji silników popularnych gier.

Właśnie takiej opisowej analizie wraz z przytaczaniem kluczowych fragmentów kodów źródłowych programów grających poświęcony jest cały pierwszy rozdział. Jest to rozdział typowo opisowy aczkolwiek z racji skupienia tematu na silnikach i wydajności pokuszono się o dokładną analizę kodu kluczowego dla wydajności. Dużo uwagi poświęcono efektywnym reprezentacjom planszy i stanów gry. Wydajne struktury danych używane dla generowania ruchów i algorytmów AI zajęły sporą część opisu. W każdym z analizowanych silników szczególnym zainteresowaniem potraktowano wszelkie optymalizacje sprzętowe. Poszukiwano i jeżeli znaleziono to dokładnio opisano wszelkie wstawki assemblerowe, funkcje „instrinsic” 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 trzy 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 wersję z 2011 roku mistrzowskiego programu MoHex do gry w Hex jak i jedyny otwartoźródłowy silnik Connect-k dla stosunkowo nowej gry Connect-6. Ten ostatni nie zachwycił optymalnymi rozwiązaniami, ale to nawet dobrze, gdyż umożliwił w ten sposób sukces w realizacji części implementacyjnej pracy.

W rozdziale drugim skupiono się na implementacji generatora silników dla gier z rodziny (m,n,k,p,q) (czyli Connect-k, GoMoku, Tic-Tac-Toe również). Celem było zaimplementowanie generatora tworzącego na podstawie zdefiniowanych przez użytkownika parametrów konkretnego silnika do konkretnej gry z rodziny (m,n,k,p,q) a przy tym lepszego niż dostępne rozwiązania. Dzięki wyborowi programu Connect-k w części opisowej można było z powodzeniem napisać generator tak aby generowane silniki były lepsze wydajnościowo i silniejsze pod względem AI niż silnik programu Connect-k. Impelementacja okazała się w pełni sukcesem a generowane silniki poprawne i szybkie. Co ważniejsze generowane silniki wykorzystują udogodnienia nowoczesnych architektur procesorów jak i efektywną reprezentację (kompresję) stanów gry z uwzględnieniem zdefiniowanych parametrów. Dla mniejszej planszy jest np. używana w bitboardzie zmienna o mniejszym zakresie.

W trzecim i ostatnim rozdziale celem była dokładna analiza poprawnościowo-wydajnościowa zaimplementowanego generatora silników gier. W tym celu opracowano oddzielną aplikację, która była w stanie się komunikować z silnikiem po znanym interfejsie. Mając w swojej bazie zakodowane konkretne gry wraz z wynikami i optymalnymi posunięciami aplikacja ta wywołuje silnik dla zdefiniowanych przypadków testowych i sprawdza jakość i czas wygenerowania nowego ruchu. Zdefiniowano w bazie aplikacji testującej ponad XYZ przypadków testowych, testowane silniki zachowują się poprawnie dla każdego z nich. Testy wydajności pozytywnie zaskoczyły autora i jednocześnie utrwaliłi w przekonaniu, że implementacja silnika gry w C/C++ ze wstawkami assemblerowymi to dobry wybór.

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

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

# Abstract

//TODO….

**Keywords**: generator

# Spis treści

[Streszczenie 3](#_Toc455343866)

[Abstract 5](#_Toc455343867)

[Spis treści 6](#_Toc455343868)

[Wykaz ważniejszych oznaczeń i skrótów 8](#_Toc455343869)

[1. Wstęp i cel pracy 9](#_Toc455343870)

[2. Przegląd i analiza dostępnych implementacji silników popularnych gier logicznych 11](#_Toc455343871)

[2.1 Stockfish 13](#_Toc455343872)

[2.1.1 Reprezentacja planszy – bitboard 16](#_Toc455343873)

[2.1.2 Wykorzystanie zaawansowanych instrukcji assemblerowych procesora 17](#_Toc455343874)

[2.1.3 Reprezentacja ruchu i wartości dla funkcji ewaluacji 20](#_Toc455343875)

[2.1.4 Fishtest 22](#_Toc455343876)

[2.2 Mo-Hex 23](#_Toc455343877)

[2.3 MIMHex 23](#_Toc455343878)

[2.4 Connect-k 24](#_Toc455343879)

[2.4.1 Plik shared.h – definicje stałych i typów używanych w całym programie 24](#_Toc455343880)

[2.4.2 Plansza i stany gry 26](#_Toc455343881)

[2.4.3 Plik state.c i funkcje modyfikujące stan 28](#_Toc455343882)

[2.4.4 Sztuczna inteligencja – ogólnie 33](#_Toc455343883)

[2.4.5 Strategia oparta na zagrożeniu 37](#_Toc455343884)

[2.4.6 Sekwencje 37](#_Toc455343885)

[2.4.7 Monte Carlo 37](#_Toc455343886)

[3. Implementacja generatora silników wybranej gry 40](#_Toc455343887)

[3.1 Problem wyboru gry 41](#_Toc455343888)

[3.1.1 Początkowy wybór – szachy 42](#_Toc455343889)

[3.1.2 Wybór ostateczny – Connect6, Gomoku, (m,n,k,p,q) 43](#_Toc455343890)

[3.2 Architektura projektu 43](#_Toc455343891)

[3.3 Generator silników - (m,n,k,p,q)EnginesGenerator 44](#_Toc455343892)

[3.3.1 Podstawowe struktury danych – EngineParameters i EngineScheme 46](#_Toc455343893)

[3.3.2 Schematy silników 51](#_Toc455343894)

[3.3.3 Generacja silników gier logicznych – klasa EnginesGenerator 53](#_Toc455343895)

[3.3.4 Proste interakcje z wygenerowanymi silnikami 59](#_Toc455343896)

[3.3.5 Tryb „batch” 64](#_Toc455343897)

[3.4 Silnik gry w C++ z makrami - (m,n,k,p,q)GameEngine 65](#_Toc455343898)

[3.5 Biblioteka ułatwiająca współpracę z silnikiem – (m,n,k,p,q)EngineWrapper 65](#_Toc455343899)

[3.6 Aplikacja do interfejsu użytkownika - (m,n,k,p,q)GUI 65](#_Toc455343900)

[3.7 Aplikacja do analizy wydajnościowo-poprawnościowej generowanych silników - (m,n,k,p,q)EnginesAnalyzer 65](#_Toc455343901)

[4. Analiza wydajnościowo-poprawnościowa zaimplementowanego generatora silników gier 65](#_Toc455343902)

[5. Podsumowanie 66](#_Toc455343903)

[Wykaz literatury 68](#_Toc455343904)

[Wykaz rysunków 71](#_Toc455343905)

[Wykaz tabel 72](#_Toc455343906)

[Dodatek A: Tytuł dodatku A 73](#_Toc455343907)

# 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

# Wstęp i cel pracy

W początkowych dziejach algorytmiki i komputerów, znacznie bardziej istotną cechą algorytmów była złożoność pamięciowa [1]. Wynikało to z faktu, że dostępne komputery miały mało pamięci operacyjnej, a sama w sobie była ona niezwykle droga. Duża jej ilość jest natomiast niezbędna dla niektórych silników gier logicznych, jako, że muszą one np. generować możliwe posunięcia lub trzymać „drzewo gry” w pamięci. Dokładając do tego niewielką wydajność pierwszych komputerów, pierwsze programy dobrze grające w gry logiczne, jak warcaby czy szachy nie pojawiły się aż do późnych lat siedemdziesiątych [2][3][4].

Dzisiaj, dzięki dużej, taniej i łatwo dostępnej pamięci operacyjnej, wysokiej wydajności procesorów i rozwojowi algorytmiki mamy do czynienia z prawdziwym rozkwitem dziedziny zajmującej się programami grającymi. Rozwój jest tym bardziej podsycany poprzez nieustanną rywalizację 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].

Przy całej tej rywalizacji i rozwoju oczywiste jest, że kolejne generacje silników gier będą coraz bardziej wydajne i coraz lepiej wykorzystujące zasoby sprzętowe. Jedną z możliwości dalszego zwiększania wydajności jest stosowanie optymalizacji sprzętowych – wykorzystywanie konkretnych funkcji architektury procesora, o ile są dostępne. Często wymaga to umieszczania wstawek assemblerowych w kodzie źródłowym programu grającego.

Trzeba jednak przyznać nam, jako gatunkowi ludzkiemu niemały sukces na tym polu, dzięki wspomnianemu szybkiemu rozwojowi wydajności procesorów, pojemności pamięci i efektywności algorytmów udało się w 2007 roku rozwiązać wariant angielski Warcabów [10]. Rozwiązanie jest określane jako *słabe*, tj algorytm zaimplementowany w programie Chinook nie mna gwarancji wygranej z każdym przeciwnikie, ale jest zagwarantowane, że nigdy nie przegra. Tak czy owak, robi to ogromne wrażenie i jest wspólnym osiągnięciem algorytmiki i inżynierii komputerowej – wyznacznikiem naszego postępu jako ludzkość. Algorytmika, która korzysta z dobrodziejstw inżynierii komputerowej, możnaby rzecz „algorytmy zoptymalizowane pod sprzęt” to właśnie bardzo istotna część tej pracy dyplomowej jak i nowoczesnych programów grających.

W niniejszej pracy, z racji złożoności tematyki i trudności implementacji wydajnych silników gier logicznych poświecono cały pierwszy rozdział na przegląd i analizę dostępnych implementacji silników popularnych gier logicznych. Przegląd ten przede wszystkim ma za zadanie nakreślić ogólny trend w rozwoju silników gier i co ważniejsze, umożliwić autorowi implementację generatora silników dla wybranej gry logicznej w rozdziale drugim.

Głównym 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 jest napisać taki generator, 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, jak już wspomniano praca ma zasadniczo trzy części, w pewnym sensie są więc trzy cele do zrealizowania w pracy. Część pierwsza jest czysto opisowa, skupia się na przeglądzie i analizie dostępnych silników gier logicznych, jej celem jest poznanie trendów implementacyjnych współczesnych programów grających. Część druga jest czysto implementacyjna, jej celem jest powstanie generatora wydajnych i poprawnych silników wybranej gry logicznej. Część trzecia ma najbardziej charakter analizy i testów, jej celem jest zbadanie wydajności zaimplementowanego generatora i jego dogłębne przetestowanie (głównie poprzez testowanie silników przez niego generowanych.

# Przegląd 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 mądrych wyborów, tj. wybrać nowoczesne i aktywnie rozwijane silniki (aby uwzględnić trend ich rozwoju), ale też 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 najlepsze do analizy będą programy najlepsze w swojej dziedzinie, interesujące są przecież 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 | *brak (sezon niedokończony)* | |
| 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 dwa najlepsze programy grające w szachy na dzień dzisiejszy to Stockfish oraz Komodo. Przydałoby się przeanalizować w niniejszym rozdziale oba. Niestety Komodo, który jest prawdopodobnie najlepszym silnikiem szachowym na świecie [25] [26], jest także komercyjny i jego kod źródłowy jest zamknięty. Można jedynie kupić obecną wersję wykonywalną [27] lub poprzednią w wyprzedaży [28], dwie wersje wstecz są natomiast dostępne na licencji freeware [29]. Tak czy owak jako, że 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 jednym z kluczowych silników poddanych analizie.

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

|  |  |  |
| --- | --- | --- |
| Miejsce | Nazwa programu | Punkty |
| **1** | **Zen** | 9.5 |
| **2** | **Abakus** | 7.5 |
| **3** | **CGI** | 7.0 |
| 4 | Nomitan | 4.0 |
| 5 | MC\_ark | 2.0 |
| 6 | Wingo | 0.0 |

Program Zen w 2015 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 też w wersjach 13x13 i 19x19. Byłby więc doskonałym kandydatem na analizę, niestety program został wydany komercyjnie [12]. Autor – japoński programista Yoji Ojima po raz pierwszy w opublikował go jako komercyjny program w roku 2009 i kolejne jego wersje w latach następnych. Najnowsza wersja 6 pochodzi z 2016 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 być 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 ułatwi analizę.

Ostatnią już grą dla jakiej postanowiono dokonać przeglądu i analizy zaimplementowanych i dostępnych silników jest Connect6. Przyczyną jest pewna fascynacja autora zagadnieniem gier (n,m,k,p,q), gdzie Connect6 jest grą (∞,∞,6,2,1) [34] [35]. Istnieją też inne, niezwiązane bezpośrednio z jakością dostępnych silników tej gry, przyczyny, które zostaną ujawnione w dalszej części pracy.

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

|  |  |  |
| --- | --- | --- |
| Miejsce | Nazwa | Punkty |
| 2 | Floating Cloud | 6 |
| 1 | **Explorer** | 6 |
| **3** | **USTB1** | 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 w internecie. Ani w formie plików wykonywalnych ani tym bardziej w formie kodu źródłowego. Po dosyć dogłębnych poszukiwaniach programów grających w Connect6 udało się znaleźć jeden dobrze napisany, nazwany „Connect-k”. Jego autor twierdzi, że jedynym innym dostępnym w internecie programem grający w Connect6 jest NCTU6 autorstwa dr-a I-Chen Wu (pomysłodawcy Connect6) [37]. NCTU6 zazwyczaj wygrywa z Connect-k ale jest za to o zamkniętym kodzie i w języku Chińskim [37] [38], odpada więc uwzględnienie go w analizie. Autor tej 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 dogłębnie zanalizowany.

## 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 Linscott, Tord Romstad i internetową społczeność od 2004 roku i od ponad siedmiu lat pod własną nazwą (wcześniej Glaurung) [46], składający się z ponad 12000 linii kodu, zajmujący notorycznie pierwsze lub drugie miejsce na świecie w rankingu siły silników szachowych CCLR [47] [48] [49] Stockfish jest prawdopodobnie najlepszym do analizy silnikiem ze względu na cele pracy. Aby pokazać jego złożoność i zaawansowanie technologiczne i algorytmiczne poniżej zebrano niektóre jego cechy i wykorzystywane przez niego techniki:

* wykorzystanie kilku reprezentacji szachownicy i przełączanie między nimi
  + bitboard z mapowaniem LERF
  + „magiczny” bitboard z podejściem wyszukanym (ang. „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 dobrodziejstw 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 z techniki i możliwości programu grającego, techniki, które często są bardzo złożone i trudne pojęciowo. Jednakże z punktu widzenia celu tej pracy, większość z tych technik nie ma aż takiego wielkiego znaczenia, gdyż dotyczy 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 częśti tego podrozdziału skupimy się w dużej mierze na reprezentacji planszy, stanów gry i innych danych dotyczących rozgrywki a mniej na wyszukanych technikach AI.



Rysunek 2.1. Graf zależności w programie Stockfish

### 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 rejrestry w procesorach 64-bitowych. Przechowywanie więc planszy w 64-bitowej zmiennej na 64-bitowym procesorze poza korzyściami pamięciowymi oznacza możliwość wykonania niektórych operacji w jednym cyklu zegara – czyli najszybciej jak się da [50]. Sposób mapowania planszy na 64-bitową zmienna 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 pojedyńczy bitboard nie wystarczy to repreentacji 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, szkoczek).

Typ Biboard 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.

//TODO: magic bitboard? Inne reprezentacje?

### Wykorzystanie zaawansowanych instrukcji assemblerowych procesora

W celach wydajnościowych niektóre silniki gier logicznych wykorzystują wstawki kodu assemblerowego. Jednocześnie nowoczesne acrhitektury 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 „instrinsic” są oczywiście tak implementowane przez współczesne kompilatory z przyczyn wydajnościowych. Opisywane w tym podrodziale funkcje, są właśnie funkcjami tego typu.

Wspomniane wcześniej użycie bitboardu ma jeszcze jedną zaletę, często np. w funkcjach ewalucji (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 instrukcje, to przy każdym zliczaniu w funkcjach ewaluujących wystąpieniu konkretnego typu figury zostanie na bitboardzie wywołana instrukcja procesora POPCNT, która zliczy liczbę bitów ustawionych na 1 w danej zmiennej – czyli ilość typu figuy 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 <nmmintrin.h> // Intel and Microsoft header for \_mm\_popcnt\_u64()

#endif

Funkcja popcount w pliku bitboard.h jest inline’owana w celach wydajnościowych. Jak widać poniżej, jeżeli nie jest zdefiniowana stała USE\_POPCNT (którą można zdefiniować jako flagę kompilacji) to wykorzystywana jest implementacja programowa. W innym wypadku zależnie od używanego kompilatora wywoływana jest funkcja wywołująca bezpośrednio instrukcje na procesorze.

inline int popcount(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 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 przede wszystkim związanych z ewaluacją. Autor znalazł 19 wywołań funkcji popcount w projekcie, każde o dużym znaczeniu dl programu grającego. Nie należy więc marginalizować znaczenia szybkiego wykonania tej funkcji dzięki instrukcjom assemblerowym procesora.

Inną funkcją zaimplementowaną podobnie jak popcount jest w Stockfishu funkcja pext. Zdefiniowana w rozszerzeniu BMI2 zbioru instrukcji dla architektury procesorów x86 umożliwia zrównolegloną ekstrakcję bitów [57]. Funkcja przydaje się w reprezentacji „magiczny bitboard”. Gdzie po ustawieniu 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, jedyny zysk z tej funkcji jest jeżeli przy zdefiniowanej fladze do kompilatora USE\_PEXT korzysta się z funkcji wywołującej bezpośrednio instrukcję procesora.

#if defined(USE\_PEXT)

# include <immintrin.h> // Header for \_pext\_u64() intrinsic

# define pext(b, m) \_pext\_u64(b, m)

#else

# define pext(b, m) (0)

#endif

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

#if !defined(NO\_PREFETCH) && (defined(\_\_INTEL\_COMPILER) || defined(\_MSC\_VER))

# include <xmmintrin.h> // Intel and Microsoft header for \_mm\_prefetch()

#endif

Jak widać poniżej jeżeli stała powodujące dla komilatora Intela lub Microsoftu niezałączanie pliku nagłówkowego xmmintrin.h jest zadeklarowana, to wywołanie funkcji prefetch nie robi nic.

#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 jeszcze bliżej sprzętu 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 hashowaniu talic transpozycji w przeszukiwaniu jak i hashowaniu 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]);

Jeszcze dwie kolejnymi funkcjami wywołującymi zależnie od kompilatora i platformy bezpośrednie instrukcję procesora są funckje 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

Jak widać podobnie jak popcount nie są tylko funkcjami optymalizującymi, ale są realnie potrzebne do obliczeń – więc w przypadku, gdy na kompilowanej platformie nie jest dostępna konkretna instrukcja procesora to korzysta się z implementacji programowej, która tutaj nie zostanie już omówiona z powodu trywialności. Instrukcja clzll procesora jest dostępna jednak na większości architektur sprzętowych [58].

### 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 pomyslnie umieszczone bez zwiększania rozmiarów z 16-bitów, dzięki faktu, że każdy ruch będzie miał inną pozycję docelową niż źródłową (w szachach nie można dokonać ruchu w miejscu), więc wszystkie wartości Move i 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, gdzie wartości dla figur z sufixem Mg (mid-game) są dla środkowej fazy gry a te z Eg (end-game) są 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. Kolejna optymalizacja, która co prawda wymaga wołania specjalnych funkcja (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);

}

### 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 zauważyć, że większość rozwiązań opisywanych przy okazji analizy tego programu grającego występuje też w innych tego rodzaju programach. To niekoniecznie 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 Stockfisha 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 to 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 wyniki 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 Stockfisha to w dużej mierze modyfikowanie wartości różnych parametrów (np. wartości figur-położen-faz gry dla funkcji ewaluacyjnej) 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.



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.



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

## Mo-Hex

Ramadan kokokdad, dashjaf afjkashfjka i fala moko. Ragodago miko miko regajlsafjas, aksgsa jkla. Jakaakaka moranranin polo mdiasdoas.

Ja kaokkka hjfksahfja fsjdkhfas fsdkjfhsdjk vdsjvhds, hgjkdsgds, hgjdkshgjskdhgjksdhgjkasd i jakaa ahkgjsdhjkghasdjkgsadgds. Moja fhjadkfhsjdkfhui fhdsiufhdsui i jok. Hjhfahfsa hjafsaj i jaaa, hjaksfhkjsa jvskld.

## MIMHex

Ramadan kokokdad, dashjaf afjkashfjka i fala moko. Ragodago miko miko regajlsafjas, aksgsa jkla. Jakaakaka moranranin polo mdiasdoas.

Ja kaokkka hjfksahfja fsjdkhfas fsdkjfhsdjk vdsjvhds, hgjkdsgds, hgjdkshgjskdhgjksdhgjkasd i jakaa ahkgjsdhjkghasdjkgsadgds. Moja fhjadkfhsjdkfhui fhdsiufhdsui i jok. Hjhfahfsa hjafsaj i jaaa, hjaksfhkjsa jvskld.

## 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 opartę o metodę Monte Carlo i sekwencje, Gabe Emerson i Erik Shimshock pracowali nad interfejsem i częścią silnika gry natomiast Michael Levin poza rysowaniem i strategii AI opartej o zagrożenie (Threat-Based Strategy) 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 analizowna 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 odpowiadzialny za grafikę i interfejs jest mieszany z kodem odpowiedzialnym za logikę ale za to AI jest dosyć dobrze wyodrębnione i podzielone na 3 wspomniane wcześniej zaimplementowane w programie metody.

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

Analizę należy zacząć od pliku nagłówkowego który jest załączany dyrektyrwą include w prawie każdym innym pliku z kodem. To tutaj znajdziemy definicję podstawowych stałych jak i ograniczeń dla nich. Poniżej przedstawiono doyć 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 dosyć 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 typu pól. Jeżeli pole na planszy gry jest puste to przyjmuje wartość PIECE\_NONE, jeżeli zajętę przez pierwszego lub drugiego gracza to odpowiednio PIECE\_BLACK i PIECE\_WHITE. Ciekawostką jest fakt wprowadzenia w reprezentacji pól definicji pola błędnego PIECE\_ERROR, używane ono jest 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 zarządzalnych jak C# nie ma standardowego sposobu wykrywania wyjścia poza granicę tablicy [41] [42] czy też innych tego typu błędów – tablica to po prostu wskażnik, co może prowdzić do wielu błędów. Wprowadzając definicję pola błędnego PIECE\_ERROR przynajmniej część z tych błędów wyjścia indeksu poza normalną planszę możemy uniknąć. Inne zadeklarowane stałe w typ enumie to m.in. znaczniki używane przez AI do określania np. ryzyka w strategii operatej 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 spokojnie 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 wagi pozycji czy posunięc. 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 zdefniowanych 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 piste – nie jest to całkiem trywialne, ponieważ pole ejst pste 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 kolejen 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 tek 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.

### 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 jest także używana przez wszystkie algorytmu AI do reprezentacji wszystkich stanów w drzewie gry itp. Struktura trzyma m.in. wskażnik 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 allokować /deallokować 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 preceeding 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. \*/

Z punktu widzenia optymalności silnika niezwykle ważne jest jak trzymane są konkretne pola planszy. Tutaj jest to zwykła tablica o nazwie data, która trzyma wartości PIECE, które jak wiemy z poprzedniego podrozdziału są tak naprawdę typu int. Reprezentacja ta nie jest więcj 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 słówkę extern – tab 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.

### 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 podrodziale pliku nagłówkowego. Znajdują się to 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 przedtsawionych 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 z kolei jest 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);

}

Funkcji 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(AllocChain \*\*root, gsize mem)

{

AllocChain \*ac = \*root, \*ac\_next;

while (ac) {

ac\_next = ac->next;

g\_slice\_free1(mem, ac);

ac = ac\_next;

}

\*root = NULL;

}

Wreszczie 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 wywowołania funkcji count\_pieces z współrzędnymi x,y i kolejnymi czterema kierunkami od tych współrzędnych z rosnącymi i malającymi 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 zauważyć 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 zwaracamy prawdę. W innymi 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 sprawdzenia funkcją piece\_at typu pola na które dokonano ruchu. Typ możnaby po prostu przekazać do funkcji, co zwiększyłoby odrobinę 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 cełgo 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;

return i;

}

Funkcja jest dosyć trywialna. Zliczamy wszystkie wystąpienia pola typu type od miejsca na planszy o współrzędnych x,y w kierunku zdefniowanym 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.

Kolejne funkcje służą do reprezentacji współrzędnych w formie alfanumerycznej – np. b23 to inaczej (1,23). Mogłoby to być przydatne do komunikacji z GUI w formie ustandaryzowanego standardu takiego jak UCI w przypadku szachów [45] i w opisywanym wcześniej Stockfish’u. Niestety gra Connect-6 jest stosunkowo młoda i nie powstały jeszcze ani ustandaryzowane protokoły komunikacji ani tym bardziej z nich korzystające, niezależne od silnika programy GUI [35]. W programie jest to jednak wykorzystywane do generaowania i wczytywania ruchów z plików tekstowych, funkcje są wywoływane w ten sposób w pliku projektu file.c Poniżej funkcja zamieniająca współrzędną na literkę.

/\* Convert a boord coordinate to alpha representation \*/

const char \*bcoord\_to\_alpha(BCOORD x)

{

static char buf[2][32];

static int which;

int i, divisor = 26;

which = !which;

for (i = 0; i < sizeof (buf[which]) - 1; i++) {

div\_t result;

result = div(x, divisor);

buf[which][i] = 'a' + result.rem \* 26 / divisor;

if (i)

buf[which][i]--;

x -= result.rem;

if (!x)

break;

divisor \*= 26;

}

buf[which][i + 1] = 0;

return g\_strreverse(buf[which]);

}

// Get a string representation of board x/y coordinates (d7, h16, etc)

const char \*bcoords\_to\_string(BCOORD x, BCOORD y)

{

static char buf[2][32];

static int which;

which = !which;

g\_snprintf(buf[which], sizeof (buf[which]), "%s%d",

bcoord\_to\_alpha(x), board\_size - y);

return buf[which];

}

/\* Convert a string representation to coordinates \*/

void string\_to\_bcoords(const char \*str, BCOORD \*x, BCOORD \*y)

{

\*x = 0;

\*y = 0;

while (\*str && \*str >= 'a' && \*str <= 'z') {

\*x \*= 26;

\*x += \*str - 'a';

str++;

}

while (\*str && \*str >= '0' && \*str <= '9') {

\*y \*= 10;

\*y += \*str - '0';

str++;

}

if (\*y)

\*y = board\_size - \*y;

}

Kolejne powyżej zaprezentowane funkcje to konwersje współrzędnych na łańcuchy znaków i vice versa.

Z ciekawostek program trzyma całą historię rozgrywki wraz z opcjami powrotu do konkretnego miejsca.

static GPtrArray \*history = NULL;

Board \*board\_at(unsigned int move)

{

if (move >= history->len)

return NULL;

return (Board\*)g\_ptr\_array\_index(history, move);

}

void go\_to\_move(unsigned int move)

{

Board \*board2;

if (!history)

history = g\_ptr\_array\_sized\_new(32);

if (move > history->len)

move = history->len;

if (move == history->len) {

board2 = board\_new();

if (board)

board\_copy(board, board2);

g\_ptr\_array\_add(history, board2);

board2->parent = board;

} else

board2 = (Board\*)g\_ptr\_array\_index(history, move);

board = board2;

move\_no = move;

if (move\_no > move\_last)

move\_last = move\_no;

}

void clear\_history(unsigned int from)

{

int i;

if (!history)

return;

if (from >= history->len) {

g\_warning("Attempted to clear future history");

return;

}

for (i = from; i < history->len; i++)

board\_free(g\_ptr\_array\_index(history, i));

g\_ptr\_array\_remove\_range(history, from, history->len - from);

move\_last = from;

}

### 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:

* 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 są trywialne, więc nie warte omawiania. Algorytmy ai\_windows i ai\_priority nie są z bliżej nieznanego powodu nie są uwzględnione w interfejsie użytkownika (być może są jeszcze niedokończone lub występują w nich błędy), ich analiza nie ma więc również większego sensu. Nawet jeżeli byłby sens omawiać wszystkie to należy zauwżyć, że niniejsza praca nie powinna za bardzo skupiać się na AI – tylko na silnikach programów grających i ich implementacji co wraz z AI ma wpływ na wydajność całego programu grającego ale w tej pracy ineteresujący jest wpływ implementacji silnika programu grającego a nie jego AI na wydajność. Ostatecznie zadecydowano więc, że zostaną omówione jedynie trzy z tych algorytmów w dalszej części pracy: ai\_threats, ai\_sequences, ai\_monte\_carlo.

Zanim dokonana zostanie analiza wspomnianych algorytmów, należy opisać nieco 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 struktry 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. Istotnie 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 tutaj już zostana pominięte, jako, że są dosyć podobne do analogicznych funkcji dla planszy. Dwie podobne do siebie funkcje, które służą do dodawania lub podłączania 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 uważny czytelnik może zauważyć 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. Pierwszy 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 zauważyć, że struktura AIMoves może przechowywać maksymalnie tyl ruchów ile jest pól na planszy, co jednej strony jest oczywiste, z drugiej jeszcze bardziej upadabnia ją w koncepcji do strukutry planszy.

W przeszukiwaniu DFS spore znaczenie ma też funkcja zwracająca najlepszy możliwocy 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 wywowałanie funkcji sortującej aimoves\_sort.

void aimoves\_sort(AIMoves \*moves)

{

qsort(moves->data, moves->len, sizeof (AIMove), aimoves\_compare);

}

Jak widać 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 poniżej zaprezentowana 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(n2) w przypadku tablicy już posortowanej, autorzy na wszelki wypadek najpierw „przetasowują” tablicę ruchów a potem wysyłają do sortowania. Również tutaj możnaby się pokusić o drobną optymalizację i np. w funkcji sort sprawdzać czy tablica już jest posortowana.

W dalszej części kodu mamy również metody duplikujące, wyszukające (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 z punktu widzenia analizy wydajnościowej ani z punktu widzenia trudności.

W usuwaniu konkretnego indeksu z polaga data struktury AIMoves posłużono się dorbną 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;

}

}

}

Drobna optymalizacja ale dla algorytmów AI znaczy wiele, 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 kosztowane, że algorytm stałby się bezużyteczny.

### Strategia oparta na zagrożeniu

//TODO: Do zrobienia lub zrezygnować.

### Sekwencje

//TODO: Do zrobienia lub zrezygnować.

### 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 genracji 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. Ruch dla którego największa liczba gier losowych jest wygranych dla obecnego gracza zostaje wybrany.

Monte Carlo generuje w praktyce silne posunięcia ale spędza sporo czasu obliczeniowego wybierając ruch. To oczywiście wynika z faktu, że 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 przedstawiono obrazek planszy z programu, gdzie po uruchomieniu metody Monte Carlo dla czterech ruchów wygenerowanych z algorytmu „sekwencji” zostanie wybrany tak naprawdę ten sam ruch (kolor pomarańczowy - wybrany, bordowy to analizowane), który wybrałaby sekwencja.



Rysunek 2.6. Monte Carlo

Poniżej przedstawiono kod metody ai\_monte\_carlo, odpowiedzialną za zwrócenie MONTE\_N lub mniej (zdefiniowane na 10) najlepszych według metody ruchów. Jak widać metoda alokuje pamięc na nową planszę, pozyskuje wstępnie dobre ruchu 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 uitility struktury moves przechowuje ogólną liczbę wygranych gier losowcyh przez wszystkie ruchu.

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 odpowiedziala na rozegranie gry losowej na zmodyfikowanej o jeden, nowy – analizowany ruch.

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 przeciwnika zwracana jest po prostu wartość 0 – czyli AI nie wygrało. Ruchy stawiane przez graczy są losowe, ponieważ ze wszystkich pustych pól na planszy zwracacanych przez metodę empty\_cells wybieramy za każdym razem to o losowym indeksie.

# Implementacja generatora silników wybranej gry

Na podstawie całej zdobytej z analizy silników popularnych gier wiedzy należy teraz zaimplementować generator, który zależnie od zdefiniowanych parametrów będzie generować silnik do takiej sparametryzowanej gry. W uproszczeniu ideę możnaby 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 przez kompilator
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 mnieszej 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 z skomplilowanym 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

Jak widać to całkiem sporo pracy, trzy aplikacje z interfejsem graficznym, cztery programy, z czego jeden wymagający wysokiej wydajności. Oczywiście z punktu widzenia celów pracy punkty 5-6 nie są konieczne, ale jednocześnie mało satysfakcjonujące byłoby napisanie generatora silników gier logicznych bez możliwości zagrania w jedną z tych parametryzowanych gier.

## Problem wyboru gry

Pierwsza decyzja jaką należy podjąć przed rozpoczęciem implementacji to wybów konkretnej gry logicznej. Możliwości jet oczywiście wiele, ale najpierw należy się zastanowić co wynika z takiego lub innego wyboru. Oczywiście wybór gry w tym rozdziale nie ma generalnie żadnego wpływu na rozdział wcześniejszy. Przegląd i analiza silników gier logicznych - rozdział opisowy jest niezależny od problemu wyboru gry. Jednakże wybranie gry, której silnik podlegał analizie w poprzednim rozdziale mogłoby ułatwić implementację.

Inna 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 jedna 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 generaor 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 nieco 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 wskazać jednoznacznego zwycięzce porównania. Dlatego też w kolejnych podrozdziałach zostaną przedstawione dwa wyboru, każdy z innego etapu pracy.

### Początkowy wybór – szachy

Po rozważaniu za i przeciw oraz przeglądaniu internetu podjęto decyzje i wyborze gry szachy. Wydaje się to dosyć solidny wybór, przemawiają za nim różne argumenty zaprezenowane poniżej:

* bardzo dobre wsparcie ze strony społeczności,
* pełno 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 9 takich parametrów, szybko jednak okazało się, że jest ich nawet więcej. Przykładowe parametry to:

* liczba pól w formacie n x m (czyli zarówno n jak i m, szachownica nie musi być kwadratowa)
* kto ma pierwszy ruch
* ile ruchów jest wykonywanych po pierwszym ruchu
* **ilość figur każdego typu** (szczególnie jeśli jakiegoś typu figury się nie pojawią (możliwość optymalizacji pamięci na stan gry))
* rozstawienie początkowe bierek
* zasady promocji
* zasady wygranej
* zasady posunięć

Po wstępnym przemyśleniu parametrów, nadszedł czas na rozpoczęcie implementacji. Na sam początek nalezało przemyśleć architekturę całego projektu. Wykorzystano w tym przemyślenia 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, co zdecydowanie upraszcza życie, gdyż nie ma potrzeby pisania własnego programu do interfejsu graficznego gry.

Niestety w trakcie implementacji pojawiły się różne problemy. Implementacja generatora silników szachowych okazała się nie lada wyzwaniem i wraz z upływającym czasem na przygotowanie pracy autor postanowił zmienić koncepcję. Postanowiono, że najlepszą opcją będzie zmiana gry na prostszą koncepcyjnie i z mniejszą konkurencją. Autor uważa, że w szachach jest zbyt duża konkurencja ze strony Stockfisha aby można było zrealizować w ramach pracy magisterskiej silnik lepszy.

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

Z powodu analizy programu Connect-k i zauważenia jego wad, niedociągnięć, zadecydowano o realizacji generator silników dla gier z rodziny (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 a w dodatku lepszego i porównania tego z analizowanym wcześniej silnikiem.

## 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 wywoływaniuniu kompilatora C++ zamiast zamieniania „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. I podpatrując rozwiązania z Stockfisha generowane silniki zrealizowano jako aplikacje konsolowe, gdzie komunikacja opiera się na założonym protokole tekstowym.

//obrazki i architektura



Rysunek 3.2. "Code map" dla solucji

Projekt powstał w IDE Microsoft Visual Studio 2015 Ultimate. Oprócz przyzwaczajeń autora, dużym plusem z punktu widzenia projektu jest wsparcie wielu technologii – tutaj przydatne okazały się jezyki 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 całkiem niezłe [63][64].

## 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. 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 przemyślanie w XAML-u natomiast model (Model) jest spójny i w miarę prosty. Jedynym zadaniem na przyszłość jest więc napisanie warstwy pośredniczącej (ViewModel), póki co jednak ze względu na rozmiary aplikacji autor uważa, że byłby to „overkill”.

### Podstawowe struktury danych – EngineParameters i EngineScheme

//TODO: wyrzuć do rozdziału o EngineWrapper

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.

EngineParameters implementuje więc 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 zobrazować nieco idee przechowywwanych 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 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/własności w której dokonano wywołania.

W związku z tym, że jeżeli jest 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 oznacz, 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 jaim 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 co będzie widoczne 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 framewoka .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 z zgodnie z wyrażeniem kolejne jego grupy przechwytujące 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\*(EXACTLY\_K\_TO\_WIN|K\_OR\_MORE\_TO\_WIN)?",

RegexOptions.IgnoreCase | RegexOptions.Compiled);

Osoba zaznajomiona ze składnią wyrażeń regularnych nie powinna mieć większych problemów ze zrozumieniem tego wyrażenia. Dopuszczamy wszędzie dowolną ilość spacji ale wymagamy kolejne parametrey m,n,k,p,q wymienione w nawiasie i po sobie. Warunek wygranej może być oddzielony od nawiasów z liczbami być oddzielony połogą „\_”. Dopuszcza się tylko dwie jego wartości i słusznie bo nie za bardzo miałaby sens sytuacja gdyby do wygranej mogło doprowadzić k lub mniej kamieni pod rząd. Dodatkowo wyrażenie jest skompilowane (co oznacza trochę dłuższe uruchamianie programu ale szybsze działanie) i niezależne od wielkości liter.

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 ineteresujące 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 dotyczącyc implementacji interfejsu IEquatable. Druga wersja metody – porównanie z object jest specjalnie dla kolekcji z czasów .NET 1.1 i starszych – czyli sprzed ery generyków. Kolekcje tamte 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 hashowanych, takich jak słownik czy tablica hashująca potrzebna jest jeszcze metoda GetHashCode.Wyegenrowana przez ReSharper metoda przeciążona może wyglądać całkiem ciekawie 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 interesujących nas polach i xor-ujemy ich hashe. 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 zauważyć, że wyliczenie haszkodu 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 haszkod może mieć dowolną wartość (jedynie ważne żeby nie dochodziło do zbyt częstych kolizji).

### Schematy silników

Poza umożliwieniem użytkownikowi wygenerowania silnika z dowolnymi, zdefiniowanymi przez niego, parametrami warto było umożliwić 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. Sam enum oczywiście by nic nie znaczył, bo nie może mieć żadnej logiki. Wykorzystano jednak metody rozszerzające dostępne w C# aby rozszerzyć go o logikę a konkretnie o jedną metodę pozwalającą na pozyskanie wartości parametrów z 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 postawie konkretnej wartości schematu tworzy nowy obiekt klasy EngineParameters (klasa opisana w podrozdziale opisującym bibliotekę (m,n,k,p,q)EngineWrapper) z odpowiednimi wartościami parametrów. Jak widać w przypadku kółka i krzyżyka nie musimy ustawiać warunku wygranej, gdyż oba są równoznaczne – bo k=m=n. Standard i freestyle Gomoku z kolei różnią się tylko warunkiem wygranej. Schematy można łatwo ładować w interefejsie 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.

### Generacja silników gier logicznych – klasa EnginesGenerator

Zakładając, że mamy zdefiniowane parametry do silnika (lub wczytaliśmy schemat) i poprawną ściezkę 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 zczytywane z UI i uruchamiany jest awaitowany Task z wywołaniem metody GenerateEngine na obiekcie \_generator.

Poniżej zaprezentowano stan generatora w trakcie pracy po nacinięciu przycisku „Generate”. Należy zauważyć, że cały „output” z wywoływanego kompilatora jest wyświetlany na bieżąco w interfejsie użytkownika – widać np. flagi przekazane do kompilatora. Szczególną uwagę należy zwrócić na wartości takie 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 kompilowanego 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 technicznego 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={engineParameters.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 trafiać generowane silniki (pozostawienie go powodowało błędy). Potem ustalamy nazwę assembly generowanego silnika – jest to po prostu wywołanie metody ToString na parameterach silnika. Scieżka 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 komilacji 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.

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.

Wywołujemy 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 trochę 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 0 bo trzeba go progromować, bo się zmienai i treścią – bo 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 jako content dla projektu generatora. Takie rozwiązanie generuje bardzo niewiele problemów a sprawdza się bardzo dobrze i jest wygodne. Jedyny wymóg to dodawanie nowo powstałych plików z projektu C++ dodatkowo jako content do projektu generatora.



Rysunek 3.7. Pliki silników załączone zarówno jako oddzielny projekt jak i content dla generatora

Do argumentów wysyłanych do procesu MSBuild wysyłane są również flagi związane z budową lub wyczyszczenie projektu (zależnie od przycisku wybranego przez użytkownika). Zawsze jest to release – ze względu na potrzebę jak najwyżej wydajności generowanych silników. Również ze względów wydajnościowych ale także z powodu trzymania dla pewnych rozmiarów planszy w zmiennej 64bitowej tryb kompilacji 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.

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



Rysunek 3.8. Generacja silnika zakończona

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



Rysunek 3.9. Lokalizacja z wygenerowanymi silnikami

dsadasdasdasdasdasdsadas

Ddasddsadsadsa

### 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 wyegenerowanymi silnikami, że z założenia jest maksymalnie prosta. Oznacza to, że nie jest używana klasa EngineWrapper z bibliotetki (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 3.10. 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 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 3.11. 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 3.12. Wysłanie komendy newgame black ai white ai do silnika

Na rysunku powyżej pokazano zachowanie się silnika dla komendy „newgame black ai white ai”. Wybraną ją, gdyż powoduje automatyczne rozpoczęcie i w skończonym czasie zakończenie gry – jest grą komputer kontra komputer więc nie wymaga dalszej interakcji ze strony użytkownika. Obsługa kliknięcia jest bardzo prosta – jest to wywołanie na silniku urchomionym 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 możnaby się spodziewać. Tak naprawdę znając implementację lub specyfikację silnika możemy bez problemy 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 3.13. Wywołanie komendy "printboard" po zakończonej rozgrywce

W związku z tym, że kluczowa w całej interakcji jest klasa ProcessInBackground

Ddasddsadsadsa

### Tryb „batch”

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

Rozwiązaniem 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 3.14. Tryb "Batch generation" generatora

Zadanie użytkownika sprowadza się tutaj to 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 3.15. Dodanie nowego silnika do kolejki zadań generatora

W kodzie źródłowym MainWindow operacja jest bardzo prosta – sprowadza się do zczytania wartości parametrów z UI, stworzenia na ich podstawie obiektu klasy EngineParameters i dodaniu go do kontrolki ListBox. Obiekty klasy EngineParameters są reprezentowane w ListBoxie jako swpje 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);

}

Gdzie metoda GetEngineParametersFromUI robi dokładnie to czego możnaby 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) winingConditionComboBox.SelectedIndex

};

return engineParameters;

}

Po zdefiniowaniu wszystkich 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 program w trakcie generacji silników.



Rysunek 3.16. 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 od 0 – czyli przed wywołaniem build na kompilatorze wołane jest clean. W innym wypadku najdłużej trwałaby generacja pierwszego silnika a potem kompilator używałby częśći funkcji już skompilowanych. 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 clean na projekcie oraz uruchamiany jest stoper. Wyczyszczenie wyników kompilacji projektu przed kolejną kompilacją jest niezbędnę aby poprawnie zmierzyć prawdziwą wydajność generatora – w innym wypadku kompilator mógłby użyć części skompilowanych funkcji z poprzedniej kompilacji. 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 3.17. 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 i

## 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 kompilację silnika kluczowy jest jak najlepiej opisany, jak najbardziej optymalny kod w tym projekcie. Jest to o tyle wyzwanie również z tego powodu, że autor nie ma zbyt dużego doświadczenia z językiem C++, jest zdecydowanie fanem C# i środowiska .NET.

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

### Plik Types.h – definicje i makra

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

#ifdef \_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 podobnym sposobem potraktowano stałą EXACTLY\_K\_TO\_WIN – jeżeli nie jest zdefiniowana to definiujemy K\_OR\_MORE\_TO\_WIN.

Jako, że wewnętrzenie 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 zadecydować czy w ogóle istnieje typ prymitywny zmiennej, który pomieści bitboard. W innym wypadku używamy 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 żeby w razie czego łatwo było zmniejszyć lub zwiększyć zakres używanego typu dla konkretnych przypadków. Np. coord jest zawsze używany dla współrzędnych – więc hipotetycznie jeżeli byłaby potrzeba umożliwienia planszy większych niż 255 w jednym z dwóch wymiarów to trzebaby zmienić typ coord z uint8\_t na uint16\_t.

### Klasa Board – reprezentacja planszy

Stan klasy Board odpowiedzialnej za reprezentację planszy to właściwie dwie zmienne – blackPieces przechowująca informacje o obecności lub jej braku czarnego piona w danym mijescu (1 lub 0 – wartość logiczna true lub false) i analogicznie 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)

Zdeifniowano także funckję inline opisujące dwa wymiary planszy – są one troszeczkę bardziej czytelne niż M i N a dzięki słowu 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 sprawdzenie czy dane pole jest puste. Przedstawiona poniżej funkcja planszy robi dokładnie to. 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 nową możliwością C++ jest oznaczenie metody jako const. Oznacza to, że taka metoda nie zmienia stanu wewnętrznego obiektu klasy – tj. 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 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 1. Dzieje się tak dlatego ponieważ CHECK\_BIT zeruje wszystkie boty oprócz tego, który znajduje się na ineteresującej 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 poniżej zaprezentowana 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∙M∙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 piona o określonym kolorze. Korzysta ona z wspomnianych wcześniej makr SET\_BIT i POSITION. Dla uproszczenia nie przyjmuje koloru ale flagę czy pion jest czarny (nie przewiduje się sytuacja 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

}

Ostatnia metoda, 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 sekwencje może być zapisywany we wskaźniku breakingColor o ile nie jest on równy nullptr. Jak uważny czytelnik mógłby zauważyć implementacja metody jest inspirowana metodą o podobnej nazwie z wcześniej opisywanego silnika Connect-k.

uint16\_t Board::CountPieces(char x, char y, Color color, char dx, char dy, Color\* breakingColor) const

{

arrayIndex\_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)

// ReSharper disable once CppLocalVariableMightNotBeInitialized

\*breakingColor = currentColor;

return count;

}

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

//TODO: Lorem ipsum dolor sit amet, eu cum mentitum percipit hendrerit, brute tacimates convenire id vim. An pro graece tractatos, in postea oporteat platonem est. Illum tibique mnesarchum eam ea, ex sed aliquid volumus, eam sumo consequat at. Ea ludus laoreet tacimates qui, te nam omnesque platonem. Mel euismod iracundia persecuti te.

## Aplikacja do interfejsu użytkownika - (m,n,k,p,q)GUI

//TODO: Lorem ipsum dolor sit amet, eu cum mentitum percipit hendrerit, brute tacimates convenire id vim. An pro graece tractatos, in postea oporteat platonem est. Illum tibique mnesarchum eam ea, ex sed aliquid volumus, eam sumo consequat at. Ea ludus laoreet tacimates qui, te nam omnesque platonem. Mel euismod iracundia persecuti te.

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

//TODO: Lorem ipsum dolor sit amet, eu cum mentitum percipit hendrerit, brute tacimates convenire id vim. An pro graece tractatos, in postea oporteat platonem est. Illum tibique mnesarchum eam ea, ex sed aliquid volumus, eam sumo consequat at. Ea ludus laoreet tacimates qui, te nam omnesque platonem. Mel euismod iracundia persecuti te.

# Analiza wydajnościowo-poprawnościowa zaimplementowanego generatora silników gier

//TODO Lorem ipsum dolor sit amet, eu cum mentitum percipit hendrerit, brute tacimates convenire id vim. An pro graece tractatos, in postea oporteat platonem est. Illum tibique mnesarchum eam ea, ex sed aliquid volumus, eam sumo consequat at. Ea ludus laoreet tacimates qui, te nam omnesque platonem. Mel euismod iracundia persecuti te.

# Podsumowanie

Zdefiniowane na początku cele pracy udało się zrealizować z sporym zapasem. Impelementacja 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 grającyh i zdobycie niezbędnej wiedzy do implementacji generatora.

Wybór gier z rodziny (m,n,k,p,q) umożliwł bogactwo parametrów do modyfikacji w generatorze. Program z pewnością umożliwiłby badaczom lepsze poznanie gier z tej rodziny i zastanowienie się nad ich rozwiązywalnością zależnie od zdefiniowanych parametrów. Jedynie pozostaje niedosyt w zaimplementowanych algorytmach sztucznej inteligencji umieszczanych w generowanych silnikach. Wydaje się, że należaloby generować AI bardziej z uwzględnieniem parametrów. Z pewnością AI to spore pole do dalszego rozwoju generatora.

W dalszych pracach możnaby także pokusić się o większy stopień skomplikowania parametrów w generowanych silnikach. Np. 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 nat takimi grami mogłoby być niezwykle ciekawe. Idąc dalej możnaby 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 nalezałoby w przyszłości zrobić byłoby stworzenie 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 praca chociaż nie wyczerpała tematu to jednak dzięki swojej objętości podjęła najważniejsze wątki w tematyce. Widać też, że chociaż niektóre cechy silników są wspólne niezależnie od gry to jednak niezwykle ważna jest w implementacji tzw wiedza domenowa. Wiedza to może być bardzo obszerna, zwłascza w przypadku złożonych gier logicznych jak choćby szachy czy Go. Niemniej jednak autor nie mając wcześniej zbyt dużej wiedzy w tematyce opanował przynajmniej część tej wiedzy na potrzeby pracy, więc nie jest to nieosiągalne.

Przyszłość silników gier logicznych należy też upatrywać w eksperymentowaniu z różnymi parametrami (ang. „tuning parameters” lub „tuning values”) i nieustannym testowaniu rezultatów. Świetnie 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>, (data dostępu 02.04.2016)
3. <http://en.wikipedia.org/wiki/Human–computer_chess_matches>, (data dostępu 02.04.2016)
4. <http://www.fierz.ch/history.htm>, (data dostępu 02.04.2016)
5. <https://en.wikipedia.org/wiki/Top_Chess_Engine_Championship>, (data dostępu 02.04.2016)
6. <https://en.wikipedia.org/wiki/International_Computer_Games_Association>, (data dostępu 02.04.2016)
7. <https://en.wikipedia.org/wiki/Computer_Olympiad>, (data dostępu 02.04.2016)
8. <https://icga.leidenuniv.nl/?page_id=1112>, (data dostępu 02.04.2016)
9. Zasady XIX Olimpiady Gier Komputerowych ICGA  
   <https://icga.leidenuniv.nl/wp-content/uploads/2016/03/Rules-ICGA-events-2016-3.pdf>, (data dostępu 02.04.2016)
10. <https://en.wikipedia.org/wiki/Solved_game>, (data dostępu 02.04.2016)
11. <https://icga.leidenuniv.nl/?page_id=1315>, (data dostępu 02.04.2016)
12. <http://senseis.xmp.net/?ZenGoProgram>, (data dostępu 02.04.2016)
13. <https://book.mynavi.jp/tencho6/>, (data dostępu 02.04.2016)
14. <https://boardgamegeek.com/thread/609440/mohex-available-download>, (data dostępu 02.04.2016)
15. <http://benzene.sourceforge.net>, (data dostępu 02.04.2016)
16. <https://github.com/lukaszlew/MiMHex>, (data dostępu 02.04.2016)
17. <https://github.com/ala/MiMHex>, (data dostępu 02.04.2016)
18. <https://github.com/kdudzik/MiMHex>, (data dostępu 02.04.2016)
19. <https://github.com/krzysiocrash/MiMHex>, (data dostępu 02.04.2016)
20. <https://github.com/theolol/MiMHex>, (data dostępu 02.04.2016)
21. <https://github.com/bartoszborkowski/mimhex>, (data dostępu 02.04.2016)
22. <https://github.com/krzysiocrash/patterns>, (data dostępu 02.04.2016)
23. <https://github.com/qelo/MiMHex>, (data dostępu 02.04.2016)
24. <https://github.com/jakubpawlewicz/MiMHex>, (data dostępu 02.04.2016)
25. <https://en.wikipedia.org/wiki/Komodo_(chess)> , (data dostępu 02.04.2016)
26. <https://komodochess.com>, (data dostępu 02.04.2016)
27. <https://komodochess.com/Komodo10-50a.htm>, (data dostępu 02.04.2016)
28. <https://komodochess.com/Komodo9-43a.htm>, (data dostępu 02.04.2016)
29. <https://komodochess.com/pub/komodo-8.zip>, (data dostępu 02.04.2016)
30. <https://stockfishchess.org>, (data dostępu 02.04.2016)
31. <https://github.com/official-stockfish/Stockfish>, (data dostępu 02.04.2016)
32. <https://en.wikipedia.org/wiki/M,n,k-game>, (data dostępu 02.04.2016)
33. <https://en.wikipedia.org/wiki/Tic-tac-toe>, (data dostępu 02.04.2016)
34. <https://en.wikipedia.org/wiki/Gomoku>, (data dostępu 02.04.2016)
35. <https://en.wikipedia.org/wiki/Connect6>, (data dostępu 02.04.2016)
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 02.04.2016)
37. <http://risujin.org/connectk/>, (data dostępu 02.04.2016)
38. <http://www.connect6.org>, (data dostępu 02.04.2016)
39. <https://bitbucket.org/BadRobot/connect6/src>, (data dostępu 02.04.2016)
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 02.04.2016)
41. <http://stackoverflow.com/questions/671703/array-index-out-of-bound-in-c/671709#671709>, (data dostępu 02.04.2016)
42. <https://msdn.microsoft.com/en-US/library/system.indexoutofrangeexception(v=vs.110).aspx>, (data dostępu 02.04.2016)
43. <https://en.wikipedia.org/wiki/Bitboard>, (data dostępu 02.04.2016)
44. <https://en.wikipedia.org/wiki/Bit_field>, (data dostępu 02.04.2016)
45. <https://en.wikipedia.org/wiki/Universal_Chess_Interface>, (data dostępu 02.04.2016)
46. <https://en.wikipedia.org/wiki/Stockfish_(chess)>, (data dostępu 02.04.2016)
47. <http://www.computerchess.org.uk/ccrl/404/>, (data dostępu 02.04.2016)
48. <http://www.computerchess.org.uk/ccrl/4040/>, (data dostępu 02.04.2016)
49. <http://www.computerchess.org.uk/ccrl/404FRC/>, (data dostępu 02.04.2016)
50. <http://en.wikipedia.org/wiki/64-bit_computing>, (data dostępu 02.04.2016)
51. <https://en.wikipedia.org/wiki/SSE4#POPCNT_and_LZCNT>, (data dostępu 02.04.2016)
52. <https://chessprogramming.wikispaces.com/Stockfish>, (data dostępu 02.04.2016)
53. <https://chessprogramming.wikispaces.com/Square+Mapping+Considerations#LittleEndianRankFileMapping>, (data dostępu 02.04.2016)
54. Wykład “How do modern chess engines work?” - Daylen Yang, University of California, Berkeley  
    <https://www.youtube.com/watch?v=pUyURF1Tqvg>, (data dostępu 02.04.2016)
55. <http://www.talkchess.com/forum/viewtopic.php?start=0&t=50220>, (data dostępu 02.04.2016)
56. <https://chessdailynews.com/im-erik-kislik-analyzes-the-tcec-superfinal-in-depth/>, (data dostępu 02.04.2016)
57. <https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets>, (data dostępu 02.04.2016)
58. <https://en.wikipedia.org/wiki/Find_first_set>, (data dostępu 02.04.2016)
59. <https://en.wikipedia.org/wiki/Intrinsic_function>, (data dostępu 02.04.2016)
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+gamenum+table&print=Overlap+table&print=Score+with+common+opponents>
61. <http://fastgm.de/60+0.60%20-%20E5450.html>
62. <http://tests.stockfishchess.org/tests>
63. <https://msdn.microsoft.com/en-US/library/hh567368.aspx>
64. <http://en.cppreference.com/w/cpp/compiler_support>
65. <http://stackoverflow.com/questions/871405/why-do-i-need-an-ioc-container-as-opposed-to-straightforward-di-code/1532254#1532254>
66. <https://msdn.microsoft.com/en-US/library/dn961160.aspx>
67. <https://msdn.microsoft.com/en-us/library/a569z7k8.aspx>
68. <https://msdn.microsoft.com/en-US/library/74b4xzyw.aspx>
69. <https://en.wikipedia.org/wiki/Inline_function>
70. <http://stackoverflow.com/questions/751681/meaning-of-const-last-in-a-c-method-declaration/751783#751783>
71. <http://webdocs.cs.ualberta.ca/~hayward/hex/#MoHex>, (data dostępu 02.04.2016)
72. <http://www.trmph.com/hexwiki/MoHex.html>, (data dostępu 02.04.2016)
73. Kod źródłowy programu MoHex w ramach pakietu benzen, wersja v0.9, wrzesień 2010  
    <http://downloads.sourceforge.net/project/benzene/benzene/0.9/benzene-0.9.0.tar.gz>
74. Kod źródłowy programu MoHex w ramach pakietu benzen, wersja rozwojowa, listopad 2011  
    <https://sourceforge.net/p/benzene/benzene/ci/master/tree/src/mohex/>
75. Dokumentacja programu MoHex w ramach pakietu benzen  
    <http://benzene.sourceforge.net/benzene-doc/html/mohexmainpage.html>
76. <http://www.game-ai-forum.org/icga-tournaments/program.php?id=555>, (data dostępu 02.04.2016)
77. <http://webdocs.cs.ualberta.ca/~hayward/publications.html>, (data dostępu 02.04.2016)
78. B. Arneson, R. Hayward, P. Henderson: MoHex Wins Hex Tournament (14th Computer Olympiad 2009 Pamplona), ICGA Journal 32-2 (June 2009) 114-116  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/rptPamplona.pdf>, (data dostępu 02.04.2016)
79. B. Arneson, R. Hayward, P. Henderson: MoHex Wins Hex Tournament (16th Computer Olympiad 2010 Kanazawa) ICGA Journal 33-3 (Sept 2010) 181-187  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/rptKanazawa.pdf>, (data dostępu 02.04.2016)
80. R. Hayward: MoHex Wins Hex Tournament, ICGA Journal 35-2 (June 2012) 124-127  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/rptTilburgrevised.pdf>, (data dostępu 02.04.2016)
81. S. Huang, B. Arneson, R. Hayward, M. Mueller, J. Pawlewicz: MoHex 2.0: a pattern-based MCTS Hex player, Proc. Computers and Games CG2013, Springer LNCS 8427 (2014) 60-71  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/m2.pdf>, (data dostępu 02.04.2016)
82. R Hayward, B Arneson, S-C Huang, J Pawlewicz: MoHex Wins Hex Tournament, ICGA Journal 36-3 (Sept 2013) 180-183  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/rptYokohama.pdf>, (data dostępu 02.04.2016)
83. R Hayward, J Pawlewicz, K Takada, T Van der Valk: MoHex Wins Hex Tournaments, ICGA Journal, 2015  
    <http://webdocs.cs.ualberta.ca/~hayward/papers/rptLeiden15.pdf>, (data dostępu 02.04.2016)

# Wykaz rysunków

[Rysunek 2.1. Graf zależności w programie Stockfish 14](#_Toc453636531)

[Rysunek 2.2. Mapowanie szachownicy na bitboard Little-Endian Rank-File [53] 15](#_Toc453636532)

[Rysunek 2.3. Uruchomione obecnie testy na Fishteście [62] 22](#_Toc453636533)

[Rysunek 2.4. Porównanie zmian wprowadzanych przez testowaną wersję [62] 22](#_Toc453636534)

[Rysunek 2.5. Graf zależności w programie Connect-k 23](#_Toc453636535)

[Rysunek 2.6. Monte Carlo 37](#_Toc453636536)

# Wykaz tabel

[**Tabela 2.1.** Wyniki turnieju "Top Chess Engine Championship" [5] 10](#_Toc453636476)

[**Tabela 2.2.** Wyniki w grze Go 9x9 [11] 11](#_Toc453636477)

[**Tabela 2.3.** Wyniki w grze Hex 11x11 [11] 11](#_Toc453636478)

[**Tabela 2.4.** Wyniki w grze Hex 13x13 [11] 11](#_Toc453636479)

[**Tabela 2.5.** Wyniki w grze Connect6 [11] 12](#_Toc453636480)

Dodatek A: Tytuł dodatku A

Ramadan kokokdad, dashjaf afjkashfjka i fala moko. Ragodago miko miko regajlsafjas, aksgsa jkla. Jakaakaka moranranin polo mdiasdoas.

Ja kaokkka hjfksahfja fsjdkhfas fsdkjfhsdjk vdsjvhds, hgjkdsgds, hgjdkshgjskdhgjksdhgjkasd i jakaa ahkgjsdhjkghasdjkgsadgds. Moja fhjadkfhsjdkfhui fhdsiufhdsui i jok. Hjhfahfsa hjafsaj i jaaa, hjaksfhkjsa jvskld.