Rok akademicki 2012/2013

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA MAGISTERSKA

Leszek Stanisław Śliwa

Komputerowa gra w Go.
Wyzwanie dla metod sztucznej inteligencji.

Opiekun pracy
Prof. dr hab. inż. Jan Zabrodzki

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1960.01.31

Data rozpoczęcia studiów: 2011.09.30

Życiorys

Urodziłem się w 31 stycznia 1960 roku w Warszawie. W 1978 roku ukończyłem szkołę średnią i zdałem maturę. W październiku 1978 roku zostałem studentem jednolitych studiów magisterskich dziennych na Politechnice Warszawskiej, Wydział Elektroniki – Kierunek Telekomunikacja. W 1981 roku rozpocząłem pracę zawodową w Warszawskich Zakładach Radiowych „Rawar". W latach 1988 – 1992 studiowałem na kierunku Informatyka w Instytucie Automatyki i Informatyki Stosowanej. Przez kilka lat prowadziłem własną działalność gospodarczą świadcząc usługi informatyczne. W kolejnych latach pracowałem na stanowiskach: administratora sieci komputerowej, programisty, projektanta, analityka, kierownika zespołu wdrażającego systemy obiegu dokumentów elektronicznych, kierownika projektu. W lutym 2010 roku rozpocząłem Wieczorowe Studia Zawodowe – Informatyka na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Po obronie we wrześniu 2011 roku, kontynuuję naukę na studiach uzupełniających drugiego stopnia w Instytucie Informatyki. Jestem żonaty, mam dwie córki.

.....
Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu ..... 2013 r.

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

STRESZCZENIE

Starożytna orientalna gra w Go od dawna uważana jest za wielkie wyzwanie dla metod Sztucznej Inteligencji. Gra w Go, ze względu na ogromną przestrzeń przeszukiwania, duży współczynnik rozgałęzienia, opóźnione skutki działania oraz trudną do skonstruowania funkcję oceny, uosabia wyzwania stojące przed rzeczywistymi sekwencyjnymi problemami decyzyjnymi. W ostatnich latach komputerowa gra w Go korzysta z nowego paradygmatu bazującego na drzewie przeszukiwania z wykorzystaniem metod Monte-Carlo. Programy komputerowe, w których zaimplementowano drzewo przeszukiwania Monte-Carlo mogą grać na poziomie mistrzowskim i zaczynają kwestionować prymat profesjonalnych graczy. W pracy omówiono i zweryfikowano algorytmy przeszukiwania drzewa Monte-Carlo przy wykorzystaniu zbudowanego programu komputerowego do gry w Go.

Słowa kluczowe:
GRA W GO, MONTE-CARLO, MCTS, UCB, UCT, SZTUCZNA INTELIGENCJA, HEURYSTYKI, AMAF

COMPUTER PROGRAM FOR GO GAME

Ancient oriental game of Go for a long time has long been considered to be a great challenge for the methods of Artificial Intelligence. The game of Go, because of its huge search space, a large branching factor, delayed effects, and difficulty to construct the evaluation function, epitomizes the challenges facing real problems of sequential decision making. In recent years, computer game of Go uses a new paradigm based on a tree search using Monte-Carlo methods. Computer programs that implement MCTS algorithms can play at the level of master and begin to question the primacy of professional players. This thesis discusses and verifies the Monte-Carlo search tree algorithms using a self-build Go game computer program.

Keywords:
GO GAME, MONTE-CARLO, MCTS, UCB, UCT, AI, HEURISTICS, AMAF

Rodzicom poświęcam,

Leszek Stanisław Śliwa

Spis treści

Spis treści ..... **v**

Wykaz rysunków ..... **viii**

Wykaz tabel ..... **ix**

Wykaz algorytmów ..... **ix**

1. Wstęp **1**
1.1. Wprowadzenie 1
1.2. Cel i zakres pracy 2
1.2.1. Algorytm przeszukiwania Monte-Carlo 2
1.2.2. Usprawnienia metody przeszukiwania Monte-Carlo 3
1.2.3. Zrównoleglenie metody przeszukiwania Monte-Carlo 4
1.2.4. Podsumowanie 4
1.3. Układ pracy 4

2. Gra w Go **6**
2.1. Historia gry w Go 6
2.2. Podstawowe pojęcia używane w grze w Go 6
2.2.1. Plansza do gry w Go 6
2.2.2. *Oddechy* i zbijanie kamieni 7
2.2.3. Bloki, łańcuchy, grupy 7
2.2.4. Terytorium 7
2.2.5. Wpływ 7
2.2.6. Koniec gry 8
2.2.7. Obliczanie wyniku gry 8
2.2.8. Reguła *Ko* 8
2.2.9. Szyk kamieni 8
2.3. Zasady gry w Go 9
2.4. Ocena siły gry zawodników 11
2.5. Charakterystyka gry w Go 12
2.6. Gra w Go wyzwaniem dla metod sztucznej inteligencji 13

3. Metoda przeszukiwania Monte-Carlo **15**
3.1. Symulacje Monte-Carlo 15
3.2. Ocena Monte-Carlo 16
3.3. Algorytm przeszukiwania MCTS 17
3.4. Fazy algorytmu MCTS 21
3.4.1. Wybór 21
3.4.2. Wzrost 24
3.4.3. Symulacja 25

3.4.4. Wsteczna propagacja ... 25
3.5. Ostateczny wybór ruchu ... 26
3.6. Zalety i wady algorytmu MCTS ... 26
3.7. Podsumowanie ... 27

4. Usprawnienia algorytmu MCTS ... 28
4.1. Heurystyka AMAF ... 28
4.2. Algorytm UCT-RAVE ... 30
4.3. Wykorzystanie wiedzy w strategii przeglądania drzewa ... 32
4.3.1. Przeglądanie z progresywnym uprzywilejowaniem ... 32
4.3.2. Przeglądanie z progresywnym poszerzaniem ... 33
4.4. Wykorzystanie wiedzy w strategii symulacji ... 34
4.4.1. Algorytm symulacji bazujący na pilności ... 35
4.4.2. Algorytm symulacji zbliżonej do sekwencyjnej ... 36

5. Zrównoleglanie algorytmu MCTS ... 37
5.1. Zrównoleglanie w liściu ... 37
5.2. Zrównoleglenie w wierzchołku ... 38
5.3. Zrównoleglenie w drzewie ... 39
5.4. Znacznik odwiedzin ... 40
5.5. Metody zrównoleglania w GPU ... 40
5.5.1. Zrównoleglenie w wierzchołku i drzewie ... 41
5.5.2. Zrównoleglenie w liściu ... 41
5.5.3. Zrównoleglanie blokowe ... 41
5.6. Podsumowanie ... 42

6. Program komputerowy do gry w Go ... 43
6.1. Algorytmy i struktury danych w polityce drzewa ... 43
6.1.1. Struktury danych dla planszy ... 43
6.1.2. Struktury danych dla łańcuchów ... 45
6.1.3. Kryteria weryfikacji poprawności ruchu ... 48
6.1.4. Wykonywanie ruchów ... 48
6.1.5. Historia i anulowanie ruchów ... 50
6.2. Algorytmy i struktury danych w polityce symulacji ... 50
6.2.1. Tablica łańcuchów ... 51
6.2.2. Wykonywanie losowych ruchów ... 53
6.2.3. Generator liczb pseudolosowych ... 54
6.3. Ocena wyniku gry ... 55
6.3.1. Metody bezpośrednie ... 56
6.3.2. Ograniczenia metod bezpośrednich ... 60
6.3.3. Identyfikacja *martwych* kamieni ... 60
6.4. Implementacja rozwiązania ... 61
6.4.1. Opis rozwiązania ... 61
6.4.2. Konfigurowanie programu ... 62
6.4.3. Metoda testowania ... 63

6.4.4. Testy wydajność symulacji Monte-Carlo 64
6.4.5. Testy siły gry programu 65
6.4.6. Testy wpływu strategii UCT na siłę gry programu 67
6.4.7. Testy kryterium wyboru najlepszego ruchu 67

7. Podsumowanie pracy 70

Bibliografia 71

Dodatek A. Słownik terminów 72

Dodatek B. Wyniki najlepszych programów komputerowych 74

Dodatek C. Opis dołączonej płyty CD 75

Wykaz rysunków

Rysunek 2.1. Ilustracja *martwej* (czarne*) i bezwarunkowo *żywej* (białe) grupy kamieni. 8
Rysunek 2.2. Ilustracja pojęć wpływu i terytorium. 8
Rysunek 2.3. Szyk kamieni. 9
Rysunek 2.4. Przykłady typowych sytuacji ilustrujących zasady gry w Go. 11
Rysunek 2.5. Ocena siły gry zawodników w porządku rosnącym. 11
Rysunek 3.1. Fazy algorytmu MCTS. 18
Rysunek 3.2. Ilustracja kolejnych iteracji algorytmu MCTS. 20
Rysunek 4.1. Przykład rozgrywki i porównanie aktualizacji bazowej z AMAF. 29
Rysunek 4.2. Idea algorytmu UCT-RAVE. 31
Rysunek 4.3. Przykłady wzorców i wag. 33
Rysunek 4.4. Przeglądanie z progresywnym poszerzaniem. 34
Rysunek 5.1. Technika zrównoleglania w liściu algorytmu MCTS. 38
Rysunek 5.2. Technika zrównoleglania w wierzchołku algorytmu MCTS. 38
Rysunek 5.3. Technika zrównoleglania z użyciem jednej sekcji krytycznej. 39
Rysunek 5.4. Technika zrównoleglania z użyciem lokalnych sekcji krytycznych. 39
Rysunek 5.5. Techniki zrównoleglenia: w liściu, w wierzchołku i blokowa (GPU). 42
Rysunek 6.1. Struktura danych dla planszy gry. 44
Rysunek 6.2. Struktura danych dla łańcuchów w polityce drzewa. 45
Rysunek 6.3. Przykład tworzenie łańcucha. 46
Rysunek 6.4. Usunięcie kamienia z łańcucha. 47
Rysunek 6.5. Struktura danych dla łańcuchów w polityce symulacji. 51
Rysunek 6.6. Łączenia łańcuchów w polityce symulacji. 53
Rysunek 6.7. Obliczenie wyniku gry na podstawie terytorium i obszaru. 55
Rysunek 6.8. Działanie algorytmu Bouzy'ego z użyciem operacji morfologicznych. 59
Rysunek 6.9. Wyniki testów wydajności symulacji Monte-Carlo. 64
Rysunek 6.10. Siła gry programu w odniesieniu do programu referencyjnego GNU Go. 66
Rysunek 6.11. Średnia długość gry w turnieju z programem GNU Go. 67
Rysunek 6.12. Wpływ kryterium wyboru najlepszego ruchu na siłę gry programu. 68

Wykaz tabel

Tabela 2.1. Porównanie złożoności dla kilku popularnych gier kombinatorycznych [1]. 13
Tabela 3.1. Kryteria wyboru najlepszego węzła. 26
Tabela 6.1. Wyniki testów wydajności symulacji Monte-Carlo. 65
Tabela 6.2. Wyniki pojedynków z programem referencyjnym GNU Go. 65
Tabela 6.3. Porównanie siły gry programu dla strategii UCT i funkcji oceny Monte-Carlo... 67
Tabela 6.4. Wpływ kryterium wyboru najlepszego ruchu na siłę gry programu. 68
Tabela 7.1. Słownik dziedziny 72
Tabela 7.2. Ranking najlepszych programów na planszy 9 × 9 na serwerze CGO..... 74

Wykaz algorytmów

Algorytm 1.1. Algorytm pierwszy najlepszy. 3
Algorytm 3.1. Pseudokod „płaskiej" strategii MCTS. 19
Algorytm 3.2. Pseudokod MCTS z perspektywy dwóch strategii: drzewa i symulacji. 21
Algorytm 3.3. Pseudokod MCTS ze strategią wyboru UCT. 22
Algorytm 6.1. Iteracja przez wszystkie punkty na planszy. 45
Algorytm 6.2. Pseudokod operacji dodawania i usuwania punktów w łańcuchu. 47
Algorytm 6.3. Pseudokod procedury wykonywania ruchu. 49
Algorytm 6.4. Pseudokod wyznaczenia liczby *oddechów* w polityce symulacji. 52
Algorytm 6.5. Przykład użycia generatora liczb losowych. 54
Algorytm 6.6. Algorytm rozrostu ziarna w wersji z kolejką. 57

1. Wstęp

1.1. Wprowadzenie

W ostatnich dziesięcioleciach dokonano ogromnego postępu w zastosowaniu sztucznej inteligencji w grach komputerowych. W roku 1950, Claude E. Shannon opublikował przełomową pracę o programowaniu komputerowym dla gry w szachy [15], która położyła teoretyczne podstawy dla dalszego rozwoju metod sztucznej inteligencji. W pracy zaproponowano użycie algorytmu mini-max (von Neumann 1928), w którym do oceny stanów końcowych użyto funkcji heurystycznej. Algorytm mini-max oraz jego liczne udoskonalenia − na przykład alfa-beta (Knuth i Moore, 1975) − utworzyły ramy, które na długie lata służyły do budowy programów komputerowych, w szczególności do gry w szachy. Stały postęp w rozwoju programów komputerowych dla gier, zapoczątkowany w roku 1950, osiągnął punkt kulminacyjny w 1997 roku, kiedy program komputerowy działający na super-komputerze DEEP BLUE IBM[1] pokonał arcymistrza Garry Kasparowa. Wydarzenie to stało się kamieniem milowym w pracach nad sztuczną inteligencją.

Jest kilka powodów, dla których konstruowanie komputerowych programów do gry w Go jest interesujące. Przede wszystkim, w odróżnieniu od gry w szachy, komputerowe programy do gry w Go nadal nie dorównują siłą gry ludzkim arcymistrzom. Prowadzone badania mają na celu opracowanie nowych algorytmów, które zastąpią klasyczne metody przeszukiwania drzewa gry. Algorytmy mini-max i alfa-beta, które okazały się skuteczne w przypadków gry w szachy lub warcaby, nie umożliwiają konstruowania efektywnych programów do gry w Go. Algorytmy te nie są w stanie poradzić sobie z ogromnym rozmiarem przestrzeni stanów gry, który jest naturalną konsekwencją kombinatorycznej złożoności gry w Go. Ponadto, gra w Go jest dobrym obiektem badań z uwagi na następujące cechy:

a) Podstawowe zasady gry w Go są proste i można je łatwo zaimplementować.
b) Gra w Go jest popularna. W Go, gra amatorsko kilkadziesiąt milionów ludzi na całym świecie. Kilka tysięcy – gra w Go profesjonalnie.
c) Gra w Go jest trudna do opanowania przez program komputerowy. Nie ma programu komputerowego, który grałby na poziomie mistrzowskim na planszy 19×19.
d) Gra w Go ma ugruntowane pole badawcze. Setki publikacji naukowych poświęconych jest programom komputerowym do gry w Go.

W ostatnich latach, dla gry w Go został opracowany nowy paradygmat sztucznej inteligencji Podejście to, bazujące na symulacji Monte-Carlo, pozwoliło na błyskawiczny postęp w budowie programów i przybliżyło siłę gry programów komputerowych do poziomu mistrzowskiego. W przeciwieństwie do przeszukiwania alfa-beta, algorytmy te są jeszcze we wczesnej fazie rozwoju, zaś dziedzina jest wciąż szeroko otwarta na nowe pomysły. Warto podkreślić zaletę nowego podejścia, które umożliwia uzyskanie dobrych wyników nawet przy braku eksperckiej wiedzy dziedzinowej. Mimo że ten paradygmat został opracowany początkowo dla gry w Go, to nie jest on specyficzny dla tej gry i znajduje zastosowanie w innych dziedzinach.

[1] Pierwszy mecz odbył się w 1996 roku i zakończył się wygraną Kasparowa z wynikiem 4: 2. Ponowny pojedynek w maju 1997 roku zakończył się wynikiem 3½ : 2½ dla komputera.

1.2. Cel i zakres pracy

Celem pracy jest zbudowanie programu do gry w Go, w którym dla rozwiązania problemu przeszukiwania przestrzeni sekwencji użyto algorytm MCTS[2] (ang. Monte Carlo Tree Search).

Każdy problem przeszukiwania przestrzeni sekwencji może być wyrażony w postaci skierowanego grafu, w którym węzły reprezentują stany, natomiast krawędzie – działania. Drzewo przeszukiwania jest grafem acyklicznym zbudowanym z węzłów, które mają zero lub więcej węzłów podrzędnych (potomnych) i co najwyżej jeden węzeł nadrzędny (rodzica). Taki model ułatwia konstruowanie algorytmów rozwiązujących problem przeszukiwania przestrzeni sekwencji. W pracy użyta została terminologia, która jest powszechnie używana w tej kategorii problemów. Pojęcie ruchu jest synonim działania, pozycja synonimem stanu oraz gra − problemu.

Wyróżniamy trzy rodzaje problemów: (1) problemy bez przeciwników (tzw. problemy optymalizacji, gry jednoosobowe), (2) problemy z jednym przeciwnikiem (gry dwuosobowe), oraz (3) problemy z wieloma przeciwnikami (gry wieloosobowe). W grach dla dwóch graczy, gracze mogą konkurować lub współpracować ze sobą. W grach wieloosobowych gracze mogą tworzyć koalicje. Ponadto wyróżniamy problemy deterministyczne i stochastyczne. Gra w Go jest niekooperatywną dwuosobową grą deterministyczną.

Problem optymalizacji sekwencji najczęściej rozwiązywany jest za pomocą algorytmów bazujących na drzewie przeszukiwania. Do powszechnie stosowanych należy zaliczyć algorytm A* (Harta, Nielsona i Raphaela, 1968). Dla gier dwuosobowych, używany jest algorytm mini-max (von Neumann 1928) oraz jego udoskonalona wersja – algorytm alfa-beta (Knuth i Moore, 1975). Istnieje wiele udoskonaleń algorytmu alfa-beta. Jednym z najbardziej udanych jest wariant z iteracyjnym pogłębianiem IDA* (Marsland, 1983). Większość z tych algorytmów korzysta z funkcji oceny pozycji. Funkcja ta oblicza wartość heurystyczną na podstawie stanu gry dla potencjalnych ruchów. W przeciwieństwie do klasycznych algorytmów przeszukiwaniu drzewa, takich jak A* czy alfa-beta, algorytm przeszukiwania Monte-Carlo nie bazuje na funkcji oceny pozycji.

1.2.1. Algorytm przeszukiwania Monte-Carlo

Pierwszym zadaniem pracy było zbudowanie programu komputerowego, w którym zaimplementowano algorytm przeszukiwania Monte-Carlo w wersji podstawowej.

Programy komputerowe do gry w Go budowane do 2005 roku były heterogeniczną kombinacją algorytmu alfa-beta, (który był przez wiele lat standardem przeszukiwania w grach dwuosobowych), systemów eksperckich, heurystyk i wzorców. W algorytmie alfa-beta, w celu zapewnienia zadowalających wyników, wymagana jest dobra funkcji oceny. Im gorsza jest jej jakość, tym bardziej prowadzi to do nieosiągalnego wysiłku obliczeniowego. Dla gry w Go bardzo trudno jest zbudować dobrą funkcję oceny. Na początku stulecia pojawił się pomysł zastąpienia funkcji oceny pozycyjnej, symulacją Monte-Carlo. Pomysł ewoluował do metody nazwanej przeszukiwaniem Monte-Carlo − MCTS. Metoda ta implementuje

[2] Heurystyka podejmowania decyzji w zadaniach sztucznej inteligencji, używana m.in. do wyboru ruchów w grach. Metoda MCTS skupia się na analizie najbardziej obiecujących ruchów, w której rozwoju drzewa wariantów bazuje na losowym próbkowaniu przestrzeni przeszukiwań.

strategię pierwszy najlepszy (pierwszy najtańszy, jeżeli funkcja oceny jest postrzegana jako funkcja kosztu) sterowanej wynikami symulacji Monte-Carlo. Algorytm pierwszy najlepszy 1.1, jest algorytmem zachłannym, który po rozwinięciu stanu, wybiera spośród potomków ten, dla którego suma kosztów i szacowanego kosztu dotarcia jest najniższa.

Algorytm 1.1. Algorytm pierwszy najlepszy.

1. przypisz $i = 0$;
2. rozwiń stan s_i w zbiór stanów S_i;
3. wybierz $s_i + 1$ jako ten stan $s \in S_i$, który jest „najlepszy";
4. przypisz $i = i + 1$;
5. jeżeli s_i jest terminalny, to zwróć sekwencję działań prowadzącą do niego, w przeciwnym razie wróć do kroku 2.

Algorytm MCTS pojawił się w 2006 roku w trzech różnych odmianach: (1) Coulom, (2) Kocsis i Szepesvari oraz (3) Chaslot. Pierwszy wariant został wykorzystany przy budowie programu komputerowego do gry w GO − CRAZY STONE. Program zwyciężył w turnieju gry w Go rozgrywanym na planszy 9×9 podczas jedenastej *Olimpiady komputerowej*. Wariant wprowadzony przez Kocsisa i Szepesvariego, nazwany UCT (ang. Upper Confidence Tree), korzystał z algorytmu górnych przedziałów ufności UCB (ang. Upper Confidence Bounds) (Auer, Cesa-Bianchi, Fischer, 2002). Wariant zaproponowany przez Chaslota bazował na algorytmie OMC (ang. Objective Monte Carlo), którego głównym celem było oszacowanie wartości pozycji podczas gry.

1.2.2. Usprawnienia metody przeszukiwania Monte-Carlo

Drugim zadaniem pracy było usprawnienie metody przeszukiwania Monte-Carlo.

Metoda przeszukiwania Monte-Carlo składa się z dwóch strategii: (1) przeglądania drzewa gry oraz (2) symulacji. Każdą z tych strategii można udoskonalić.

W pierwszej strategii – przeglądania drzewa gry, najważniejszą decyzją jest wybór węzła, którego ocena będzie następnie szacowana na podstawie symulacji Monte-Carlo. Z jednej strony, podczas przeszukiwania powinny być preferowane najbardziej obiecujące ruchy (eksploatacja). Z drugiej strony słabsze posunięcia, z uwagi no to, że ich niskie wyniki mogą być spowodowane zbiegiem okoliczności („nieszczęśliwymi" symulacjami), powinny być nadal badane (eksploracja). W każdym węźle drzewa MCTS, podczas podejmowania decyzji o wyborze węzła potomnego, należy znaleźć równowagę pomiędzy eksploracją i eksploatacją. Zadanie wyboru najlepszego posunięcia może być ułatwione przez zastosowanie wiedzy, która staje się czynnikiem dominującym, gdy liczba symulacji w węźle jest mała oraz traci stopniowo na znaczeniu, gdy liczba przeprowadzonych symulacji wzrasta.

W drugiej strategii – symulacji, w jej podstawowym wariancie wykonywane są losowe ruchy, które przy wykorzystaniu wiedzy można przekształcić w bardziej przemyślane, „silne" pseudo-losowe posunięcia. Wiedza może być zaprojektowany przez eksperta lub utworzona na podstawie uczenia maszynowego. W pracy zbadano różne strategie stosujące wiedzę.

1.2.3. Zrównoleglenie metody przeszukiwania Monte-Carlo

Trzecim zadaniem pracy było zrównoleglenie metody przeszukiwania Monte-Carlo.

Rozwój technologii spowodował, że procesory we współczesnych komputerach (nawet osobistych) zawierają kilka rdzeni. Aby w pełni wykorzystać możliwości współczesnego sprzętu metoda przeszukiwania Monte-Carlo musi umożliwiać obliczenia współbieżne. Obliczenia współbieżne zwiększają liczbę obliczeń w jednostce czasu, co przekłada się na siłę gry programu. Jak to zostanie opisane w dalszej części (rozdział 3), algorytm przeszukiwania Monte-Carlo składa się z czterech kroków, z których każdy może być zrównoleglony za pomocą innej metody.

1.2.4. Podsumowanie

Głównym celem pracy było zbadanie podstawowej wersji algorytmu MCTS, jego udoskonaleń oraz możliwości współbieżnego wykonywania na przykładzie implementacji w komputerowej grze w Go. MCTS jest algorytmem ogólnego zastosowania i może być używany do rozwiązywania różnych problemów. Jednak najbardziej obiecujące wyniki można uzyskać przy przeszukiwaniu przestrzeni sekwencji o wielkiej złożoności kombinatorycznej i dla których trudno jest znaleźć dobrą funkcję heurystyczną. Przykładem jest gra w Go, w której algorytm MCTS pokonał wszystkie dotychczas stosowane techniki.

1.3. Układ pracy

Rozdział 1 zawiera wstęp, cel i zakres pracy.

W rozdziale 2 przedstawiono środowisko testowe – grę w Go. Przedstawiono historię gry, jej zasady, różnorodność odmian gry, podstawowe pojęcia stosowane przez graczy. Omówiono rolę, jaką pełni gra w Go w obszarze sztucznej inteligencji.

W rozdziale 3 opisano metodę przeszukiwania drzewa Monte-Carlo − MCTS. W pierwszej części przedstawiono zagadnienie użycia oceny Monte-Carlo, alternatywy dla funkcji oceny pozycji. To podejście jest obecnie rzadko stosowane, ale stanowi ważny krok w kierunku algorytmu MCTS. Następnie opisano budowę algorytmu MCTS. Algorytm MCTS składa się z czterech kroków: wyboru, rozwoju, symulacji i wstecznej propagacji. W rozdziale przedstawiono dwie główne polityki: (1) przeglądania drzewa i (2) symulacji. Na zakończenie wskazano inne obszary zastosowania dla algorytmu MCTS, różne od gier dwuosobowych.

W rozdziale 4 skoncentrowano się na udoskonaleniach algorytmu MCTS. Omówiono różne metody wykorzystania wiedzy do poprawy efektywności algorytmu i w konsekwencji zwiększenia siły gry programu. Zastosowanie wiedzy możliwe jest zarówno w strategii przeglądania drzewa jak i w strategii symulacji. W strategii przeglądania drzewa omówiono dwie najbardziej popularne heurystyki: UCT oraz RAVE.

W rozdziale 5 przedstawiono sposoby współbieżnego wykonywania algorytmu MCTS: (1) zrównoleglania w liściach, (2) zrównoleglania w korzeniu i (3) zrównoleglania w drzewie. Zwrócono uwagę na techniki zrównoleglenia algorytmu z użyciem procesora graficznego.

W rozdziale 6 scharakteryzowano program komputerowy, który został zbudowany w celu przeprowadzenia badań nad opisanymi w pracy algorytmami. Omówiono architekturę rozwiązania (składającego się z czterech projektów), algorytmy i struktury danych, generator liczb pseudo-losowych, kryteria weryfikacji poprawności ruchu, metodę oceny wyniku gry.

Dokumentacja rozwiązania, kod źródłowy, instrukcja użytkownika oraz wyniki testów zostały umieszczone na dołączonej płycie CD.

Rozdział 7 zawiera podsumowanie całej pracy i wnioski z przeprowadzonych doświadczeń.

Po rozdziale 7 umieszczono wykaz publikacji i wykorzystanych zasobów.

W dodatku A umieszczono krótki słownik dziedziny. W dodatku B przedstawiono historyczne wyniki najlepszych programów komputerowych do gry w Go w pojedynkach przeciwko profesjonalnym graczom. W dodatku C wyszczególniono zawartość załączonej płyty CD.

2. Gra w Go

W tym rozdziale opisano grę w GO. Na początek opisano historię gry i omówiono podstawowe pojęcia używane przez graczy oraz jej podstawowe zasady. Następnie przedstawiono formalną charakterystykę gry. Dalej poświęcono nieco uwagi sposobowi oceny siły gry zawodników. Na zakończenie dokonano krótkiego przeglądu roli jaką pełni gra w Go w dziedzinie sztucznej inteligencji.

2.1. Historia gry w Go

Gra w Go jest jedną z najstarszych gier na świecie, pochodzącą ze starożytnych Chin. Według legendy, chiński cesarz Yao (2337-2258 p.n.e.), poprosił swojego doradcę, aby ten zlecił zaprojektowanie gry, która nauczy cesarskiego syna sztuki wojennej, dyscypliny, koncentracji i równowagi. Pierwsza pisemna informacja o grze znajduje się w historycznym annałach Tso Chuan (IV wieku p.n.e.), odnosząc się do wydarzeń historycznych w 548 roku p.n.e.. Początkowo gra była rozgrywana na planszy składającej się z 17×17 punktów. Od czasów dynastii Tang (618-907 n.e.) pojedynki toczy się na planszach 19×19. Standard 19×19 został ostatecznie wprowadzony w Japonii i Korei na przełomie V i VII wieku n.e. W Japonii, początkowo grano w Go na dworze cesarskim. Gra w Go uzyskała popularność w XIII wieku. W XVII wieku, przy wsparciu rządu japońskiego, powstało kilka powszechnych szkół gry w Go, co spowodowało znaczną poprawę siły gry zawodników.

Pomimo wielkiej popularność w Azji Wschodniej, gra w Go wolno zdobywała popularność w innych regionach świata w odróżnieniu od innych gier azjatyckiego pochodzenia, na przykład gry w szachy. Mimo tego, że o grze pierwsze wzmianki można znaleźć w zachodniej literaturze pochodzącej jeszcze z XVI wieku, gra w Go stała się popularna na Zachodzie dopiero w końcu XIX wieku, kiedy niemiecki naukowiec Korschelt (1880) napisał traktat na temat gry w Go. Na początku XX wieku, grano w Go w Cesarstwie Niemieckim i Austro-węgierskim. Dopiero w drugiej połowie XX wieku, gra w Go stała się popularna w innych krajach zachodnich. Obecnie Międzynarodowa Federacja gry w Go zrzesza 71 krajów członkowskich. Szacuje się, że w Go gra kilkanaście milionów ludzi na całym świecie, w tym kilka tysięcy – profesjonalnie (głównie w Japonii, Chinach i Korei).

2.2. Podstawowe pojęcia używane w grze w Go

W punkcie opisano podstawowe pojęcia używane przez graczy w Go. Zostaną wyjaśnione kolejno: grupy kamieni *żywych* i *martwych*, terytorium, wpływ oraz wzorce.

2.2.1. Plansza do gry w Go

Tradycyjnie w Go gra się kamieniami wykonanymi ze specjalnego łupku oraz muszli małży na planszy (*goban*) wykonanej z drzewa *kaya* rosnącego w prefekturze Miyazaki na wyspie Kiusiu z liniami wykonanymi gorącą laką przy użyciu samurajskiego miecza. Taki sprzęt jest jednak bardzo drogi. Dlatego najczęściej spotyka się kamienie wykonane ze szkła lub tworzyw sztucznych oraz plansze z drewna, materiałów drewnopochodnych, blachy, a nawet tektury. Pole gry ma kształt prostokąta o wymiarach około 396×432 mm z zaznaczonymi 19 liniami pionowymi i 19 liniami poziomymi tworzącymi łącznie z narożnikami i liniami brzegowymi siatkę składająca się z 361 węzłów. W pracy, węzeł siatki (miejsce), w którym

można położyć kamień będzie nazywane **punktem**. Początkujący gracze zwykle zaczynają od mniejszej planszy, o wymiarach 13×13 (169 punktów), a nawet 9×9 (81 punktów). Rozgrywka polega na stawianiu kamieni w punktach. Gracze stawiają kamienie na przemian, zaczynając od gracza grającego czarnymi. Raz postawiony kamień nie jest przesuwany – jeżeli nie zostanie zbity, to pozostaje na swoim miejscu do końca gry.

2.2.2. *Oddechy* i zbijanie kamieni

Wolne punkty sąsiadujące z kamieniem nazywany są *oddechami*. Kamień przeciwnika znajdujący się w jednym z sąsiadujących punktów zabiera mu jeden *oddech*. Kamień, któremu zabrano wszystkie *oddechy* zostaje zbity. Przeciwnik zdejmuje go z planszy i kamień zostaje „wzięty do niewoli".

2.2.3. Bloki, łańcuchy, grupy

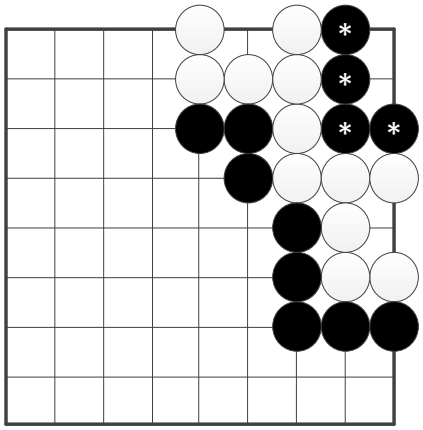
Kamienie tego samego koloru, znajdujące się w punktach, które można połączyć prostopadłymi liniami, tworzą łańcuch (często też nazywany blokiem). Kamienie w łańcuchu mają wspólne *oddechy*. Łańcuch można zbić zabierając mu wszystkie *oddechy*. Uogólnieniem pojęcia łańcucha jest grupa. Grupa jest luźno połączonym zestawem łańcuchów (bloków) jednego koloru, które zazwyczaj kontrolują jeden spójny obszar. Z definicji, każdy łańcuch (blok) jest również grupą. Grupa jest *żywa*, jeśli nie może być zbita przez przeciwnika. Grupa nazywa się bezwarunkowo *żywa*, jeśli nie może być zbita przez przeciwnika nawet, jeżeli będzie on rezygnował z prawa do wykonania ruchów. Grupa, która nie może uniknąć zbicia uważana jest za *martwą*. Ustalenie statusu grupy (bezwarunkowo *żywa*, *żywa* albo *martwa*) jest kluczowe dla przebiegu gry w Go.

2.2.4. Terytorium

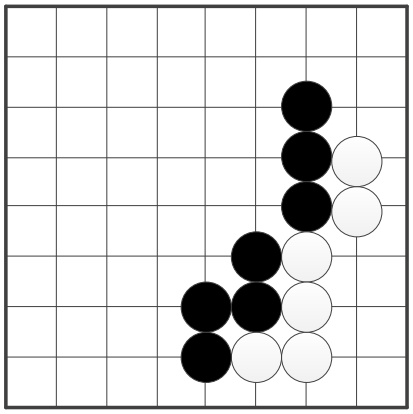
Terytorium nazywany jest obszar otoczony przez żywą grupę, wewnątrz którego przeciwnik nie może utworzyć swojej *żywej* grupy. Obliczenie terytorium jest bardziej złożone niż określenie statusu łańcucha lub grupy. W programach komputerowych stosowanych jest kilka alternatywnych rozwiązań. Na przykład, Muller (1997) używa kombinacji przeszukiwania i statycznych reguł, natomiast Bouzy (2003) stosuje matematyczną morfologię (dział matematyki bazujący na teorii zbiorów). Van der Werf, Van den Herik i Uiterwijk (2006) wytrenowali wielowarstwowy perceptron potrafiący przewidywać potencjalne terytorium. Na rysunku 2.1 zilustrowano pojęcie *martwej* i bezwarunkowo *żywej* grupy kamieni. Grupa białych kamieni nigdy nie może być zbita, dlatego mówi się, że jest bezwarunkowo *żywa*. Czarne nie mogą zapobiec zbiciu czarnej grupy (oznaczonej *), dlatego grupa czarnych kamieni (oznaczona *) jest *martwa*.

2.2.5. Wpływ

Gracz ma wpływ w regionie, jeśli jest bardzo prawdopodobne, że jest on zdolny do utworzenia kontrolowanego przez siebie terytorium. Pojęcie to jest trudniejsze do oceny przez komputer niż terytorium, ponieważ wpływ zależy od konkretnej sytuacji na planszy. W Go, często istnieje kompromis między wpływem i terytorium. Gracz próbuje kontrolować jak największe terytorium, oponent stara się jak najbardziej ograniczyć jego wpływ, grając na zewnątrz terytorium gracza. Przykładem takiego kompromisu widać na rysunku 2.2. Białe kontrolują terytorium w rogu planszy, natomiast czarne mają wpływ na zewnątrz tego obszaru.



Rysunek 2.1. Ilustracja *martwej* (czarne*) i bezwarunkowo *żywej* (białe) grupy kamieni.



Rysunek 2.2. Ilustracja pojęć wpływu i terytorium.

2.2.6. Koniec gry

Gracz, który nie widzi ruchów, które mogą przynieść mu korzyść, możesz zrezygnować z ruchu (spasować). W tym momencie przeciwnik może wykonać ruch, na który gracz może odpowiedzieć. Gra kończy się, jeżeli obaj gracze zrezygnują z ruchu.

2.2.7. Obliczanie wyniku gry

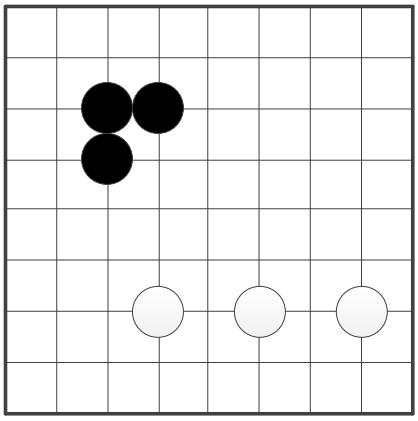
Martwe grupy usuwa się z planszy i dołącza do jeńców (nie ma potrzeby wypełniania wszystkich ich *oddechów*). Następnie liczy się wszystkie puste punkty tworzące terytoria i dodaje liczbę jeńców. Wygrywa gracz z większą liczbą punktów. Zgodnie z regułami chińskimi nie uwzględnia się jeńców, natomiast dodaje się do siebie liczbę kamieni tworzących *żywe* grupy oraz puste punkty, które są przez te kamienie otoczone. W większości przypadków otrzymuje się ten sam wynik.

2.2.8. Reguła *Ko*

Reguła *ko* zabrania graczom natychmiastowego „odbicia" kamienia, przeciwnik musi najpierw zagrać w innym punkcie planszy. Ta reguła obowiązuje jedynie, gdy zbijamy i „odbijamy" tylko jeden kamień. W niektórych zasadach istnieje też reguła *Super Ko* zabraniająca powtórzenia tej samej pozycji wszystkich kamieni na całej planszy. Została ona wprowadzona, aby uniemożliwić nieskończone powtarzanie sytuacji w pozycjach, których nie obejmuje zwykła reguła *Ko*.

2.2.9. Szyk kamieni

Szyk jest lokalną konfiguracją kamieni (wzorcem), którego celem jest uzyskanie przewagi taktycznej na oponentem. Szyk jest ważny zarówno dla ludzi jak dla programów komputerów. Na rysunku 2.3 porównano dwa szyki, których siła „bojowa" jest różnie oceniana przez ekspertów. Klin (szyk trójkątny) uważany jest za mało efektywny. Szyk liniowy w ocenie ekspertów ma dużą siłę „bojową".



Rysunek 2.3. Szyk kamieni.

2.3. Zasady gry w Go

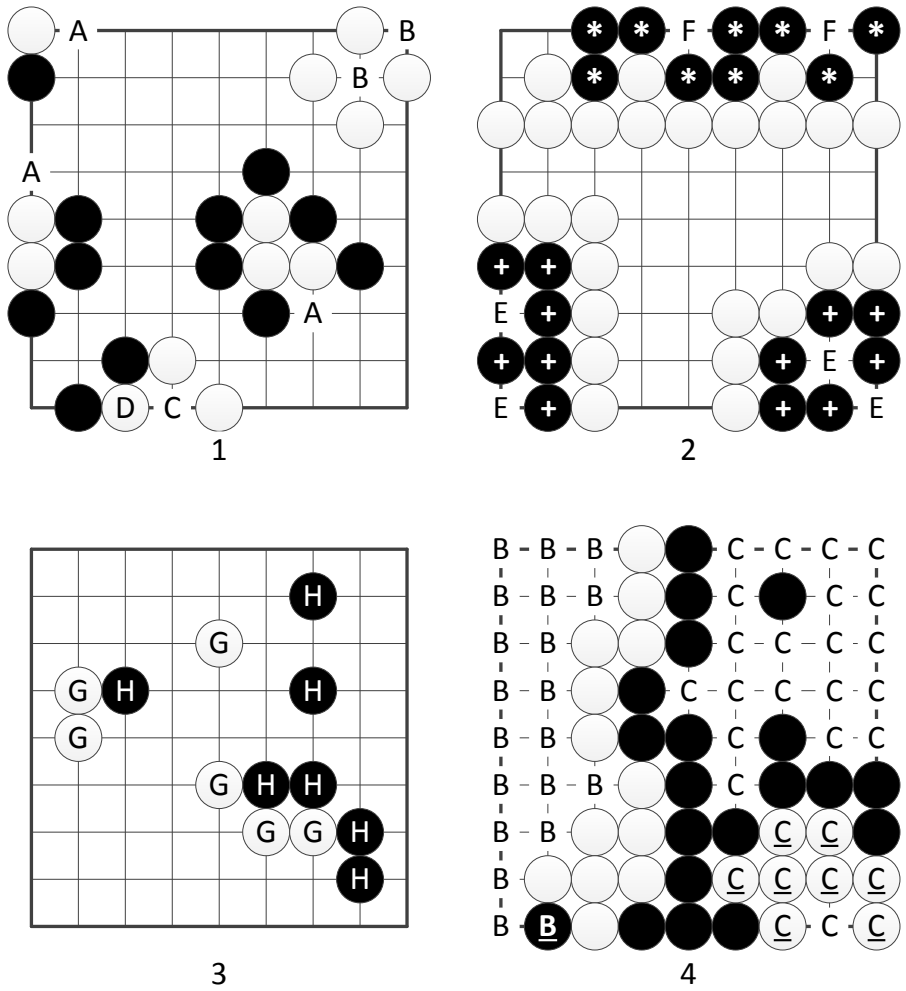
Aktualnie istnieje szereg odmian gry w Go (np. chińska, japońska, amerykańska). Chociaż wszystkie główne odmiany reguł są zgodne na poziomie ogólnej idei jak gra ma przebiegać, to jednak istnieje kilka subtelnych różnic. Podstawowa różnica pomiędzy zestawami reguł polega na użyciu innych metod punktacji. Warto jednak dodać, że pomimo różnych metod punktacji, wyniki rzadko różnią się więcej niż o jeden punkt. Oznacza to, że implementacja wybranej metody punktacji w programie komputerowym nie ma wpływu na wynik badań nad zastosowanymi algorytmami. Dwie najbardziej popularne metody punktacji to zestawy reguł japońskich i chińskich. Japońskie zasady punktacji są uważane przez niektórych graczy za bardziej interesujące niż chińskie. Jednak japońskie zasady są dość trudne (a w opinii niektórych ekspertów nawet za niemożliwe) do realizacji w programie komputerowym ze względu na szereg niejasności i niespójności występujących w oficjalnych tekstach. Chińskie zasady punktacji, często też są niespójne, jednak w znacznie mniejszym stopniu. Naturalnym wyborem dla konstruktorów programów komputerowych są chińskie przepisy i właśnie te przepisy obowiązują w turniejach komputerowych gry w Go. Wyjaśnienie wszystkich zasad w szczegółach jest poza zakresem pracy. Bardzie szczegółowe zasady można znaleźć w pracy [9]. Poniżej przedstawiony jest podstawowy zestaw zasad.

1. Najczęściej plansza do gry w Go składa się z 19×19 punktów. Pojedynki są też rozgrywane na mniejszych planszach np. 13×13 lub 9×9 punktów. Na początku gry plansza jest pusta.
2. W grze bierze udział dwóch graczy, jeden grający kamieniami czarnymi, drugi - białymi.
3. Czarne rozpoczynają grę. Gracze wykonują ruchy naprzemiennie. Białe za rozpoczynanie jako drugie otrzymują rekompensatę nazwaną *komi*.
4. Ruch polega na umieszczeniu jednego kamienia własnego koloru w pustym punkcie planszy.
5. Puste punkty znajdujące się w bezpośrednim sąsiedztwie kamienia nazywane są *oddechami* (ang. liberties). Kamień lub grupa kamieni, która nie ma *oddechów* jest zbijana i usuwane z planszy. Kamienie lub grupy kamieni mające tylko jeden *oddech* są w stanie *atari*. Nie wolno położyć kamienia w punkcie, w którym będzie miał zero *oddechów*, chyba, że spowoduje to zero *oddechów* w grupie kamieni przeciwnika. W takim przypadku kamienie przeciwnika są zbijane i usuwane z planszy.

6. Nie można wykonać ruchu, który jest powtórzeniem poprzednio zajmowanej pozycji na planszy. Sytuacja, w której może wystąpić powtórzenie ruchu nazywane jest *ko*.
7. Terytorium gracza składa się z wszystkich punktów zajętych lub otoczonych przez kamienie w jego kolorze. Zbiór sąsiadujących pustych punktów całkowicie otoczonych przez kamienie w jednym kolorze nazywany jest „okiem" (ang. eye). Naturalną konsekwencją zasad jest to, że blok z parą oczu nie może być nigdy zdobyty przez przeciwnika. Bloki, które nie mogą być nigdy zdobyte nazywane są *żywymi* (ang. alive). Bloki, które z całą pewnością zostaną zdobyte nazywane są *martwymi* (ang. dead). Określenie czy grupa jest *martwa* czy *żywa* należy do fundamentalnych zasad strategii gry w Go.
8. Gracz może zrezygnować z ruchu (przekazać swoją kolej) – „spasować".
9. Dwa kolejne rezygnacje z ruchu (pass) kończą grę. *Martwe* bloki kamieni są usuwane z planszy.
10. Grę wygrywa gracz, który kontroluje większe terytorium (po wcześniejszym dodaniu *komi* dla białych).

Typowe pojęcia stosowane przez graczy do opisu stanu gry zilustrowano na rysunku 2.4: (1) stan *atari,* (2) *żywe* i *martwe* bloki, (3) grupy kamieni, (4) koniec gry.

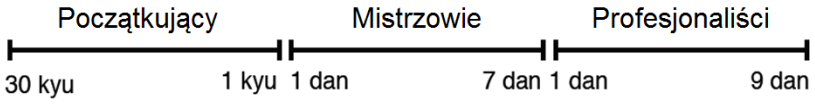
1. Białe kamienie są w stanie *atari* i mogą być zbite, jeżeli czarne zagrają w miejscach oznaczonych literą A. Literą B oznaczone punkty, w których czarne nie mogą wykonać ruchu, ponieważ czarny kamień nie miałby „oddechu". Czarne mogą jednak zagrać w punkcie C tak, aby zbić kamień znajdujący się w węźle D. Białe nie mogą odzyskać kamienia grając od razu w punkcie D, ponieważ nie mogą wykonać ruchu, który jest powtórzeniem poprzednio zajmowanej pozycji na planszy (reguła *ko*).
2. Punkty E są *oczami* w kolorze czarnym. Punkty F są fałszywymi *oczami*. Białe mogą umieścić tam swoje kamienie i zbić grupę czarnych kamieni. Grupy czarnych kamieni na dole planszy są bezwarunkowo *żywe*, ponieważ nigdy nie będą mogły być przechwycone przez białe. Grupy czarnych kamieni oznaczone * są *martwe*.
3. Białe kamienie G i czarne H nie kontrolują terytorium za wyjątkiem punktów, które same zajmują.
4. Przykład planszy po zakończeniu gry. *Martwe* kamienie C i B są usuwane z planszy. Terytoria kontrolowane przez graczy, punkty C i B oraz wszystkie pozostałe kamienie czarne i białe liczone są dla każdego z graczy. Jeśli wartość rekompensaty (*komi*) wynosi 6.5 punktu, to pojedynek wygrywają czarne różnicą 8.5 punktu.



Rysunek 2.4. Przykłady typowych sytuacji ilustrujących zasady gry w Go.
(1) Stan *atari* A oraz C i następnie zabroniony ruch D (reguła *ko*).
(2) *Oczy* E, fałszywe *oczy* F, kamienie bezwarunkowo *żywe* (+) i *martwe* (*).
(3) Terytorium składające się wyłącznie z kamieni G i H.
(4) Ocena wyniku gry: terytoria B (białe) i C (czarne) oraz *martwe* kamienie B i C.

2.4. Ocena siły gry zawodników

Siła gry zawodników grający w Go oceniana jest w skali trzystopniowej: początkujący (*kyu*), mistrzowie (*dan*) oraz profesjonaliści. Stopnie początkujące są w kolejności malejącej siły, natomiast stopnie mistrzowskie są w porządku rosnącym. Na poziomie amatorskim, różnica w stopniu odpowiada liczbie kamieni uprzywilejowanych (ang. handicap), które słabszy gracz może umieścić na planszy przed rozpoczęciem gry. Ma to na celu stworzenie lepszych warunków słabszemu przeciwnikowi w celu wyrównania szans. Różnica pomiędzy 1 *kyu* i 1 *dan* zwykle jest rekompensowana jednym kamieniem. Na rysunku 2.5 przedstawiono ocenę siły gry zawodników w porządku rosnącym (od lewej do prawej).



Rysunek 2.5. Ocena siły gry zawodników w porządku rosnącym.

W Go do obliczania relatywnej siły gry zawodników stosowana jest metoda punktacji *Elo*. Ranking *Elo* pochodzi od nazwiska Arpada Elo – amerykańskiego naukowca węgierskiego pochodzenia, którego prace ukształtowały system rankingowy bazujący na naukowych podstawach. Ranking *Elo* był pierwotnie zaprojektowany dla gry w szachy, ale współcześnie jest wykorzystywany w wielu grach m.in. Go. Różnica w ocenie między dwoma graczami służy jako prognoza wyniku meczu. Jeśli dwóch graczy, którzy mają zbliżony ranking, będzie grało przeciwko sobie, to oczekuje się, że obaj zdobędą taką samą liczbę zwycięstw. Gracz, którey ma ranking większy o 100 punktów niż jego przeciwnik wygra zgodnie z prognozami 64% gier. Jeśli różnica wynosi 200 punktów, to oczekiwana liczba wygranych partii wynosi 76%. W pierwotnej wersji rankingu, przyjmowano, że wyniki graczy przyjmują rozkład normalny. Współcześnie często używany jest rozkład logistyczny.

Większość współczesnych programów do gry w Go toczy pojedynki na serwerze CGOS (ang. Computer Go Server). Na tym serwerze rozgrywane są gry błyskawiczne z limitem czasu 5 minut dla planszy 9×9 i z limitem 20 minut dla planszy 19×19. Ranking *Elo* jest dla każdego programu na bieżąco aktualizowany. Skala *Elo* na serwerze CGOS używa rozkładu logistycznego z prawdopodobieństwem zwycięstwa programu A nad programem B, wyrażonym formułą 2.1:

$$P(A|B) = 1/\left(1 + 10^{\frac{\mu_B - \mu_A}{400}}\right) \tag{2.1}$$

gdzie μ_A i μ_B są rankingami *Elo* dla programów A i B. W tej skali różnica 200 punktów oznacza 76% wygranych partii, 500 punktów – 95%.

2.5. Charakterystyka gry w Go

Gra w Go może być formalnie scharakteryzowana jako gra: *turowa (sekwencyjna), dwuosobowa, o sumie zerowej, deterministyczna, partyzancka, z pełną informacją, dyskretna*. W grze turowej (sekwencyjnej) gracze wykonują ruchy naprzemiennie. W grze o sumie zerowej zyski lub straty jednego z graczy są równoważone przez zyski lub straty pozostałych graczy. W grze dwuosobowej, jeśli jeden z graczy wygrywa, to drugi przegrywa. Gra jest deterministyczna, jeśli żaden ruch nie jest losowy. Gra jest partyzancka, gdy potencjalne ruchy nie są takie same dla wszystkich graczy. W grze z pełną informacją, informacja o aktualnym stanie gry jest dostępna dla wszystkich graczy. Gra jest dyskretna, jeżeli wykonywane ruchy są dyskretne (w odróżnieniu od ruchów wykonywanych w czasie rzeczywistym). Gry dla dwóch graczy, o sumie zerowej, z pełną informacją, deterministyczne, dyskretne i sekwencyjne są nazywane grami *kombinatorycznymi*.

Liczba stanów gry w Go jest szacowana[3] na 10^{171} i rozmiar drzewa gry (liczba wszystkich możliwych gier równa sumie liczby liści w drzewach, których wierzchołkiem jest stan początkowy gry) na 10^{360}. Rozmiar drzewa gry jest większy od liczby stanów, ponieważ ten sam stan gry może być osiągnięty w wyniku innej sekwencji ruchów. Średni współczynnik rozgałęzienia drzewa gry (średnia liczba możliwych ruchów, zgodnych z regułami gry, w danym stanie) jest znacznie większy niż w grze w szachy: 250 do 35. W tabeli 2.1. porównano złożoność stanów gry i drzewa gry dla kilku popularnych gier kombinatorycznych. Z uwagi na te charakterystyki, gra w Go jest uważana przez wielu ekspertów za jedną z najtrudniejszych gier planszowych. Na obecnym etapie rozwoju znalezienie, w skończonym czasie, optymalnych posunięć w grach na planszach 19×19 punktów, czy nawet 9×9 przy użyciu algorytmów siłowych, rozwiązujących problemy

[3] Złożoność przestrzeni stanów gry − Tromp i Farnebäck, 2006. Złożoność drzewa gry – Allis, 1994.

przez weryfikację i ocenę wszystkich możliwych wariantów postępowania jest niemożliwe. Aktualnie plansza składająca się z 5×5 jest największą, dla której zostały sprawdzone wszystkie możliwe ruchy graczy i na której czarne, które rozpoczynają grę, zawsze wygrywają.

Tabela 2.1. Porównanie złożoności dla kilku popularnych gier kombinatorycznych [1].

Gra	Liczba pól na planszy	Złożoność stanów gry	Złożoność drzewa gry	Średnia liczba ruchów w grze	Aktualny stan problemu
„Kółko i krzyżyk”	9	10^3	10^5	9	Rozwiązanie dokładne do znalezienia bez użycia komputera
Warcaby	64	10^{20}	10^{31}	10	Rozwiązanie dokładne znalezione w 1997
Szachy	64	10^{50}	10^{123}	80	Siła gry programu komputerowego $>$ siła gry arcymistrzów
Go	361 (19×19)	10^{171}	10^{360}	150	Siła gry programu komputerowego $\ll$ siła gry arcymistrzów

2.6. Gra w Go wyzwaniem dla metod sztucznej inteligencji

Gry umysłowe były przedmiotem badań sztucznej inteligencji od początku jej istnienia. Jedne z pierwszych algorytmów, które przeszukiwały i uczyły się, zostały opracowane dla gry w szachy (Shannon, 1950; Turing, 1953) oraz warcaby (Samuel, 1959). Pierwszy program komputerowy do gry w Go został zbudowany dla komputera IBM 704 (1962). Siła gry programu była mała i nigdy nie wygrał on pojedynku z człowiekiem. Pierwszy program komputerowy, któremu udało się pokonać początkującego gracza został zaprojektowana przez Zobrista w 1970. Pierwsze programy bazowały na heurystycznych funkcjach oceny. Głównym ich składnikiem była ocena wpływu każdego kamienia, który „promieniuje” na sąsiednie pola planszy. Programy te, z uwagi na niewielką moc obliczeniową komputerów, nie były w stanie wykonywać głębokiego przeszukiwania.

W latach siedemdziesiątych, najlepszym programem do gry w Go był INTERIM2 (Reitman i Wilcox, 1988). W programie zaprogramowanym w języku LISP zastosowano system ekspercki, który podejmował decyzje na podstawie abstrakcyjnej reprezentacji gry. W latach osiemdziesiątych najlepsze programy bazowały na połączeniu przeszukiwania, heurystyk oraz systemów eksperckich. Pod koniec lat osiemdziesiątych pojawiła się nowa grupa programów, w których wykorzystano techniki rozpoznawania obrazów (Boon, 1990). W kolejnych latach próbowano (bez powodzenia) zastosować inne techniki sztucznej inteligencji, takie jak uczenie ze wzmacnianiem (Schraudolph, Dayan, Sejnowski, 1993) czy sieci neuronowe (Richards, Moriarty, Miikkulainen, 1998). Zwycięskimi programami zostały programy hybrydowe, w których zastosowano algorytmy będące połączeniem przeszukiwania, heurystyk, systemów eksperckich i rozpoznawania obrazów. Wszystkie programy wymagały głębokiej wiedzy eksperckiej.

W latach 90. programy konkurowały ze sobą w turniejach Ing Cup[4] i Fost Cup[5]. Mistrzowskimi programami były THE MANY FACES OF GO (Fotland), GO INTELLECT (Chen), GO++ (Reiss), GOLIATH (Boon), JIMMY (Shu-Jim Yen) i najlepszy HANDTALK nazwany następnie GOEMATE (Chen Zhuxing). Największym osiągnięciem programu HANDTALK było zwycięstwo z 9. letnim przeciwnikiem grającym z siłą 6-dan amatorów w partii przy 11. kamieniach handicapowych (co odpowiada sile gry 5-kyu). Można zauważyć, że programy komputerowe nie mogły rywalizować z profesjonalnymi mistrzami.

Metody Monte-Carlo w programie komputerowym po raz pierwszy zastosowano w 1993 roku (Brügmann). Metody Monte-Carlo zostały użyte do konstruowania funkcji oceny pozycji. Pomysł nie wzbudził jednak większego zainteresowania i został zapomniany na całą dekadę. Na początku stulecia Bouzy i Helmstetter połączyli funkcję oceny Monte-Carlo z przeszukiwaniem i w celu weryfikacji na ile skuteczne jest nowe podejście skonstruowali program komputerowy INDIGO. Program zdobył trzy brązowe medale na olimpiadach komputerowych w 2004, 2005 i 2006 roku. Ich pionierskie badania były inspiracją dla skonstruowania algorytmu MCTS (Coulom, 2006; Kocsis i Szepesvari, 2006; Chaslot 2006). W 2007 roku rozpoczęła się dominacja programów korzystających z algorytmu MCTS.

Gwałtowny postęp jaki dokonał się dzięki zastosowaniu algorytmu MCTS spowodował, że niektórzy naukowcy przewidują, że już za dziesięć[6] lub dwadzieścia[7] lat programy komputerowe do gry w Go będą grały lepiej niż arcymistrzowie.

[4] Pula nagród 250 tys. USD
[5] Pula nagród 2 mln. JPY
[6] Jaap van den Herik, profesor w Centre for Creative Computing, Uniwersytet w Tilburgu, Holandia
[7] David Fotland, autor programu komputerowego MANY FACES OF GO

3. Metoda przeszukiwania Monte-Carlo

W tym rozdziale omówiono metodę przeszukiwania MCTS (ang. Monte-Carlo Tree Search). Metoda MCTS w różnych wariantach została zaproponowana w 2006 roku przez trzy niezależnie pracujące zespoły badaczy: (1) Coulom; (2) Kocsis i Szepesvari; (3) Chaslot. Coulom zbudował program komputerowy do gry w Go – CRAZY STONE, który zwyciężył w Komputerowej Olimpiadzie, w pojedynkach rozgrywanych na planszy 9×9. Wariant wprowadzony przez Kocsisa i Szepesvariego (2006), nazwany UCT (ang. Upper Confidence bounds applied to Trees), co można przetłumaczyć, jako górna granica przedziału ufności dla drzew przeszukiwania, bazował na algorytmie górnej granicy przedziału ufności UCB (ang. Upper Confidence Bounds). Wariant zaproponowany przez Chaslota został określony przez autora akronimem OMC (ang. Objective Monte-Carlo).

Metoda MCTS jest zachłanną strategią przeszukiwania zgodnie z zasadą pierwszy najlepszy, w której wybór najlepszej sekwencji prowadzącej do zwycięstwa w grze jest dokonywany na podstawie symulacji Monte-Carlo. W przeciwieństwie do klasycznych algorytmów przeszukiwania drzew, takich jak alfa-beta oraz A*, MCTS nie wymaga budowy heurystycznej pozycyjnej funkcji oceny. Metoda MCTS jest szczególnie interesującą strategią przeszukiwania przestrzeni w problemach, dla których budowa funkcje oceny pozycji jest trudna albo bardzo czasochłonna.

Metoda MCTS składa się z dwóch silnie połączonych części: stosunkowo płytkiego *przeglądania drzewa* oraz głębokich *symulacji gier*. Struktura drzewa określa pierwsze ruchy w symulowanych grach. Wyniki gier symulacji wpływają na kształt drzewa. W metodzie MCTS można wyróżnić cztery główne fazy: (1) w fazie *wyboru* drzewo jest przeglądane od korzenia do liścia; (2) w fazie *rozwoju* (ekspansji, rozwoju, rozbudowy, wzrostu) do drzewa dodawany jest jeden węzeł, (3) podczas fazy *symulacji* wykonywane są ruchy w symulowanej grze; (4) w fazie *propagacji wstecznej* wynik symulowanej gry jest wstecznie propagowany, przez wszystkie wcześniej odwiedzone węzły drzewa.

Ten rozdział jest zorganizowany w następujący sposób. Na wstępie zostały omówione wcześniejsze badania na temat korzystania z oceny Monte-Carlo będącej alternatywą dla heurystycznej funkcji oceny pozycji. To podejście jest aktualnie rzadko używane, ale jest ważnym krokiem w kierunku metody MCTS. Następnie szczegółowo opisano fazy algorytmu MCTS. Kolejno, zaprezentowano różne strategie dla każdej z faz MCTS. Dalej omówiono, strategie wyboru najlepszego ruchu podczas rzeczywistego pojedynku. Na zakończenie przytoczono przykłady zastosowania MCTS w innych dziedzinach.

3.1. Symulacje Monte-Carlo

Metodą Monte-Carlo, w szerokim rozumieniu, nazywamy każdą technikę wykorzystującą liczby losowe do rozwiązania problemu. Według węższej definicji Haltona (1970) metoda Monte-Carlo jest to metoda reprezentująca rozwiązanie problemu w postaci parametru pewnej hipotetycznej populacji i używająca sekwencji liczb losowych do skonstruowania próbki losowej danej populacji, z której można otrzymać statystyczne oszacowanie tego parametru.

Metody Monte-Carlo mają swoje korzenie w fizyce statystycznej, gdzie zostały wykorzystane do uzyskania przybliżonej wartości całek trudnych do obliczenia metodami analitycznymi i od tego czasu są używane w wielu dziedzinach.

Aktualnie symulacja Monte-Carlo jest jedną z najczęściej stosowanych metod symulacji. Warto pamiętać, że metoda została wymyślona i zastosowania przez Stanisław Ulama. Symulacje Monte-Carlo składają się z sekwencji losowych działań podejmowanych na podstawie sekwencji liczb losowych. Liczby losowe są z definicji nieprzewidywalne a więc i niereprodukowane. Ciągi liczb prawdziwie losowych mogą być wygenerowane z użyciem generatorów fizycznych: (1) mechanicznych np. losowanie z urny, rzut kostką lub monetą, ruletka oraz (2) bazujących na losowych procesach fizycznych np. szumie termicznym urządzeń elektronicznych, rozpadzie promieniotwórczym, promieniowaniu kosmicznym. W praktycznych zastosowaniach do symulacji Monte-Carlo nie stosuje się generatorów fizycznych ponieważ są zbyt wolne (jak na potrzeby współczesnych technik obliczeniowych), mają dużą niestabilność właściwości losowych z uwagi na wpływ otoczenia oraz nie mają powtarzalności sekwencji co utrudnia procesy testowania oprogramowania oraz weryfikację wyników. Ciekawym pomysłem są zapisane sekwencje liczb losowych np. G. Marsaglia CD-ROM (1995) zawierający 4.8×10^9 liczb. Najczęściej do symulacji wykorzystywane są generatory, w których liczby generowane są według ścisłej zależności matematycznej. Właściwości statystyczne liczb pseudolosowych są bliskie właściwościom liczb prawdziwie losowych. Dokładność wyników uzyskanych tą metodą jest zależna od liczby symulacji i jakości użytego generatora liczb pseudolosowych. Obecnie, symulacje Monte-Carlo znajdują zastosowanie w wielu dziedzinach, na przykład w chemii, biologii, ekonomii i finansach.

3.2. Ocena Monte-Carlo

W programach komputerowych ocena Monte-Carlo MCE (ang. Monte-Carlo Evaluation) została po raz pierwszy użyta w grach w Othello (Reversi), „kółko i krzyżyk" oraz szachy (Abramson, 1990). W programie do gry w Go, po raz pierwszy zastosowano ocenę Monte-Carlo w 1993 (Brügmann). W latach dziewięćdziesiątych, ocena Monte-Carlo była wykorzystywana w wielu programach komputerowych do gier losowych: brydż (1998), poker (1999) oraz scrabble (2002).

Podstawowa wersja oceny Monte-Carlo działa w następujący sposób: ocena pozycji P jest szacowana na podstawie wykonanych symulacji. Symulacja rozgrywki polega na wykonywaniu (pseudo-) losowych ruchów aż do zakończenia gry. W praktyce oznacza to, że liczba ruchów w każdej grze musi być ograniczona. Dodatkowo wprowadzana jest reguła zabraniająca umieszczania kamieni w *oczach* własnej formacji. Każda symulacja i daje na wyjściu wektor wypłat W_i dla każdego z graczy. Ocena pozycji P po n symulacjach $E_n(P)$ równa jest średniej wszystkich uzyskanych wyników $E_n(P) = \frac{1}{n}\sum W_i$.

Metodę można opisać w czterech krokach:

1. Rozpoczynając od zadanej pozycji rozegraj pseudo-losową grę. Podczas gry pamiętaj o regule *ko*, aby nie dopuścić do zapętlenia się gry.
2. Na zakończenie gry oblicz wynik (dla każdego z graczy zsumuj: liczba kamieni, kontrolowane terytorium oraz dla białych dodatkowe punkty *komi*).
3. Zapamiętaj wynik gry i jeżeli czas jeszcze się nie skończył, to kontynuuj od kroku 1.
4. Dla oceny pozycji użyj wartości średniej wszystkich wyników symulowanych rozgrywek.

Program komputerowy, korzystający z tej metody na planszy 9×9 punktów, przy zaledwie kilkunastu symulacjach na ruch, gra z siłą porównywalną do początkującego amatora. Warto zwrócić uwagę, że metoda nie korzysta z wiedzy dziedzinowej.

Właściwością oceny Monte-Carlo jest to, że jeżeli wartości wygranych są ograniczone, to ocena pozycji $E_n(P)$ jest zbieżna do stałej wartości, gdy n dąży do ∞. Oznaczmy $E_\infty(P)$ jako granicę $E_n(P)$, gdy n dąży do ∞ i niech σ będzie odchyleniem standardowym wypłaty. Ponadto na podstawie centralnego twierdzenia granicznego CTG można stwierdzić, że zmienna losowa $E_n(P)$ zbiega do rozkładu normalnego z wartością średnią równą $E_\infty(P)$ i odchyleniem standardowym równym $\sigma/\sqrt{n}$. Dla dostatecznie dużego n można użyć następującego przybliżenia: $E_n(P) = E_\infty(P) + X$, gdzie X jest zmienną losowa o rozkładzie normalnym ze średnią wartością równą 0 i odchyleniem standardowym równym $\sigma/\sqrt{n}$.

Można zadać pytanie czy ocena Monte-Carlo jest porównywalna z wartością funkcji oceny pozycji, tzn. czy program mający nieograniczone zasoby, działający według algorytmu alfa-beta i używający oceny Monte-Carlo będzie równie skuteczny jak program używający funkcji oceny pozycji. Kolejnym pytaniem jest ile w praktyce należy przeprowadzić symulacji rozgrywek, aby otrzymany wynik był dostatecznie bliski wartości $E_\infty(P)$? Czy liczba tych symulacji jest do osiągnięcia podczas rzeczywistej gry?

Przeprowadzone badania (Bouzy i Helmstetter, 2003) wykazały, że dla gry w Go, dla liczby symulacji w na poziomie kilku tysięcy, wartość oceny Monte-Carlo zbliżona jest do wartości funkcji oceny pozycji.

Metoda przeszukiwania Monte-Carlo nie korzysta wprost z oceny Monte-Carlo. Koncentruje się na polepszeniu jej jakości. W metodzie przeszukiwania Monte-Carlo badane są najbardziej obiecujące obszary. Mechanizm ten opisano szczegółowo w następnym punkcie.

3.3. Algorytm przeszukiwania MCTS

Do pomysłu oceny Monte-Carlo powrócono w 2006 roku. Pomimo tego, że funkcja oceny bazująca na symulacjach Monte-Carlo była interesująca, to jednak siła gry programów komputerowych była na poziomie początkujących amatorów. Okazało się, że zwiększanie liczby symulacji wzmacniało siłę gry programu jedynie do pewnego poziomu, po czym następowało „nasycenie”. Wtedy pojawiła się koncepcja, aby oceniać tylko te posunięcia, które mogą zapewnić lepsze rezultaty. To podejście może być zapisane w czterech krokach:

1. Rozpocznij od wierzchołka, który reprezentuje bieżący stan gry.
2. Przeszukuj drzewo zgodnie z algorytmem pierwszy najlepszy od wierzchołka w kierunku liści.
3. Jeżeli doszedłeś do liścia, który „odwiedziło” już kilka symulacji, to rozwiń liść dodając węzeł potomny.
4. Wykonaj symulację Monte-Carlo i za pomocą wstecznej propagacji zaktualizuj wartości węzłów w drzewie o wynik symulacji.

Rezultatem działania tego algorytmu jest drzewo asymetryczne (niektóre z gałęzi są bardziej rozwinięte niż inne) budowane w pamięci programu.

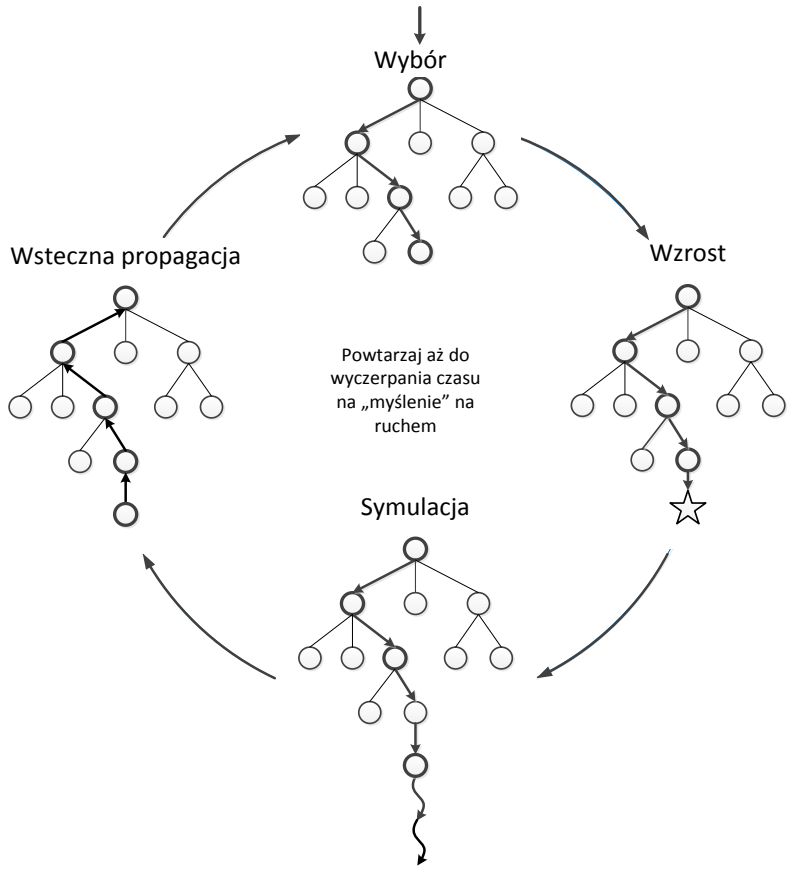
Strategia MCTS jest zachłannym algorytmem dokonującym wyboru zgodnie z zasadą pierwszy-najlepszy i którego użycie nie wymaga konstruowania funkcji oceny pozycji. Bazuje on na losowej eksploracji obszaru przeszukiwania. Korzystając z wyników z poprzednich poszukiwań, algorytm stopniowo buduje drzewo gry w pamięci i sukcesywnie

z coraz większą dokładnością szacuje wartości oceny dla najbardziej obiecujących ruchów. MCTS znajduje zastosowanie do rozwiązywania problemów, w których spełnione są trzy następujące warunki: (1) wygrane w grach są ograniczone, (2) zasady gry są znane (pełna informacja), oraz (3) symulacje można zakończyć się w „rozsądnym" czasie (czas trwania gier jest ograniczony).

Podstawowa struktura algorytmu MCTS jest następująca. Każdy węzeł i reprezentuje jedną pozycję tzw. stan gry. Węzeł zawiera, co najmniej dwie informacje: (1) obecną wartość pozycji w_i, którą zazwyczaj jest średnia z wyników gier symulacji, które odwiedziły ten węzeł oraz i (2) n_i liczba odwiedzin tej pozycji podczas przeglądania drzewa. Działanie algorytmu MCTS zwykle rozpoczyna się od drzewa zawierającego tylko węzeł główny.

Algorytm MCTS składa się z czterech faz, powtarzanych aż do wyczerpania się „czasu myślenia" nad ruchem. Są to następujące fazy. (1) W fazie *wyboru* drzewo jest przeglądane od korzenia do liścia, w którym można wybrać węzeł potomny, który nie jest jeszcze częścią drzewa. (2) Następnie, w etapie *rozwoju* do drzewa dodawany jest węzeł potomny liścia. (3) W kolejne fazie *symulacji* prowadzona jest rozgrywka z losowo wybranymi ruchami do chwili zakończenia symulowanej gry. Wynik symulowanej gry równy jest $+1$ w przypadku wygranej czarnych, 0 w przypadku remisu oraz -1 w przypadku wygranej białych. (4) W etapie *propagacji wstecznej*, wartość wyniku jest propagowana wstecz aż do korzenia, przez wszystkie wcześniej odwiedzone węzły. Ostatecznie wykonywany jest ruch, który odpowiada węzłowi potomnemu wierzchołka drzewa z największą liczbą odwiedzin. Symbolicznie cztery kroki algorytmu MCTS przedstawiono na rysunku 3.1.

Warto wspomnieć, że w literaturze krok symulacji jest alternatywnie nazywany *rozgrywką*.



Rysunek 3.1. Fazy algorytmu MCTS.

Algorytm 3.1 przedstawia pseudokod „płaskiej” strategii MCTS. W pseudokodzie, $\boldsymbol{T}$ jest zbiorem wszystkich węzłów drzewa przeszukiwania, $\boldsymbol{N}$ bieżącym węzłem. Funkcja **WYBIERZ** zwraca jeden węzeł potomny dla węzła $\boldsymbol{N}$. Funkcja **ROZWIN** dołącza jeden węzeł do drzewa i zwraca nowy węzeł. Funkcja **ROZEGRAJ** rozgrywa symulowaną partię od stanu opowiadającemu nowo dodanemu węzłowi i zwraca wynik gry $\boldsymbol{R}$, którego wartość należy do zbioru $\{-1, 0, +1\}$. Procedura **WSTECZNIE_PROPAGUJ** aktualizuje wartości węzłów zgodnie z wynikiem symulowanej rozgrywki. Funkcja **RODZIC** zwraca węzeł macierzysty dla węzła $\boldsymbol{N}$. N_c jest zbiorem węzłów potomnych węzła $\boldsymbol{N}$.

Algorytm 3.1. Pseudokod „płaskiej” strategii MCTS.

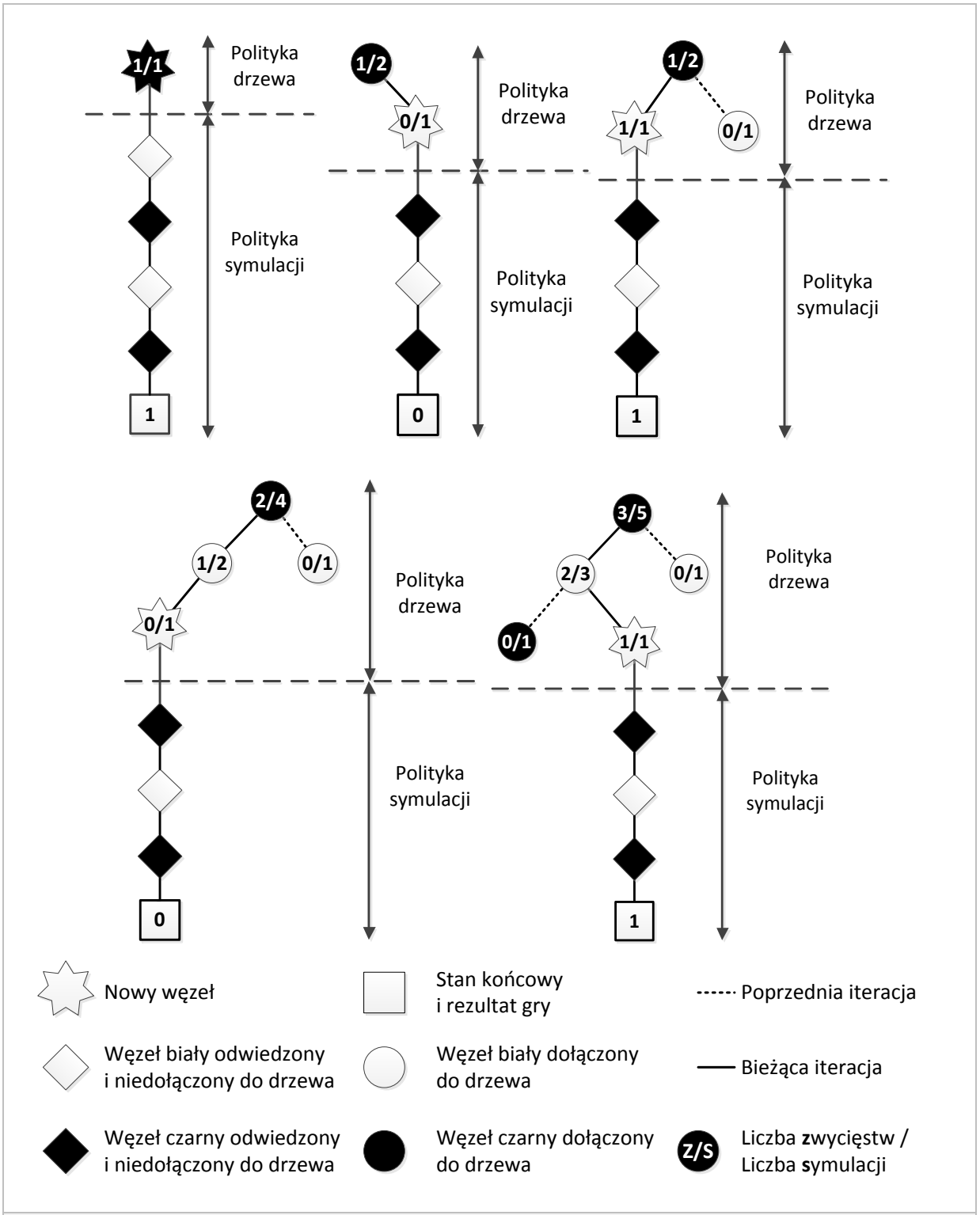
```
DANE WEJŚCIOWE:  WEZEL_GLOWNY
WYNIK:           NAJLEPSZY_RUCH

WHILE (JEST_CZAS) DO
        BIEZACY_WEZEL ← WEZEL_GLOWNY

        /* DRZEWO JEST PRZEGLADANE */
        WHILE (BIEZACY_WEZEL_JEST_W_ DRZEWIE_T) DO
                OSTATNI_WEZEL ← BIEZACY_WEZEL
                BIEZACY_WEZEL ← WYBIERZ(BIEZACY_WEZEL)
        END

        /* WEZEL JEST DOŁACZANY DO DRZEWA T */
        OSTATNI_WEZEL ← ROZWIN(OSTATNI_WEZEL)

        /* SYMULACJA ROZGRYWKI */
        R ← ROZEGRAJ(OSTATNI_WEZEL)

        /* WYNIK ROZGRYWKI JEST WSTECZNIE PROPAGOWANY */
        BIEZACY_WEZEL ← LAST_NODE
        WHILE (BIEZACY_WEZEL_JEST_W_DRZEWIE_T ) DO
                WSTECZNIE_PROPAGUJ(BIEZACY_WEZEL, R)
                BIEZACY_WEZEL ← RODZIC(BIEZACY_WEZEL)
        END
END
RETURN NAJLEPSZY_RUCH ← argmax_{N∈N_c}(WEZEL_GLOWNY)
```

Cztery kroki algorytmu MCTS mogą być zgrupowane w **dwie** odrębne *polityki*:

1) **Politykę drzewa** polegająca na wyborze liścia z węzłów już zawartych w drzewie wyszukiwania (kroki wyboru i rozwoju).
2) **Politykę symulacji** polegającą na rozgrywce z danego stanu aż do stanu końcowego w celu oszacowania wartości ruchu oraz na propagacji wstecznej aktualizującej statystyki węzłów.

Na rysunku 4.2 pokazano dwie polityki algorytmu MCTS: (1) drzewa i (2) symulacji. Na kolejnych diagramach pokazano rezultat iteracji algorytmu z zaznaczeniem granicy obu polityk.

Algorytm 3.2 przedstawia pseudokod MCTS w ujęciu dwóch polityk: drzewa i symulacji. Węzeł v_k odpowiada stanowi s_k, który jest węzłem końcowym (nie zawsze liściem) osiągniętym w fazie strategii drzewa (funkcja POLITYKA_DRZEWA). $\boldsymbol{R}$ jest wynikiem rozgrywki rozpoczętej w stanie s_k i rozegranej zgodnie ze strategią symulacji (funkcja POLITYKA_SYMULACJI). W chwili, gdy czas przeznaczony na „myślenie” nad ruchem zostanie wyczerpany, przeszukiwanie zostaje przerwane. Wynikiem algorytmu jest działanie, które prowadzi od węzła głównego v_0 (WEZEL_GLOWNY) do „najlepszego” węzła potomnego (funkcja NAJLEPSZY_POTOMEK). Stosowane kryteria wyboru „najlepszego” węzła potomnego zostały opisane w dalszej części.



Rysunek 3.2. Ilustracja kolejnych iteracji algorytmu MCTS.

Warto zwrócić uwagę, że najlepszy ruch wykonywany ostatecznie w rzeczywistej grze jest wybierany na podstawie dużej liczby symulowanych gier, w których w większości przypadków działania są podejmowane losowo z rozkładem równomiernym. Innymi słowy, nawet, jeśli iteracyjny proces jest realizowany przez dłuższy czas, to wybrany ruch może nie być optymalny. Dokładność symulacji Monte-Carlo może być poprawiona poprzez wybór węzłów w zależności od prawdopodobieństwa, że są one „lepsze” niż aktualnie znaleziony „najlepszy” ruch (funkcja NAJLEPSZY_POTOMEK).

Algorytm 3.2. Pseudokod MCTS z perspektywy dwóch strategii: drzewa i symulacji.

```
DANE WEJŚCIOWE:  WEZEL_GLOWNY
WYNIK:           NAJLEPSZY_RUCH

WHILE (JEST_CZAS) DO

        WEZEL_KONCOWY ← POLITYKA_DRZEWA(WEZEL_GLOWNY)

        R ← POLITYKA_SYMULACJI(WEZEL_ KONCOWY)

        WSTECZNA_PROPAGACJA(WEZEL_ KONCOWY, R)
END
RETURN NAJLEPSZY_RUCH ← NAJLEPSZY_POTOMEK(WEZEL_GLOWNY)
```

3.4. Fazy algorytmu MCTS

W tym punkcie zostaną przedstawione **cztery** podstawowe kroki algorytmu MCTS: *wybór*, *rozwój*, *symulacja* i *wsteczna propagacja*.

3.4.1. Wybór

Najistotniejsza różnica pomiędzy oceną Monte-Carlo i algorytmem MCTS znajduje się strategii stosowanej w kroku wyboru. Od strategii wyboru zależy efektywność całego algorytmu.

Zostało opracowanych wiele algorytmów, które realizują strategię wyboru. Celem tych strategii jest zastąpienie równomiernego badania wszystkich ruchów (stosowanego w „klasycznym" algorytmie MCTS) selektywnym próbkowaniem. Selektywne próbkowanie redukuje współczynnik rozgałęzienia drzewa i w efekcie umożliwia przeprowadzenie badania w skończonym czasie. Najczęściej omawianym w literaturze i stosowanym w praktyce (z uwagi na prostą implementację) jest algorytm UCT. Algorytm UCT i jego odmiana UCB1-TUNED zostały wyprowadzone z algorytmu MAB. W dalszej części zostaną omówione cztery strategie wyboru: UCT (Kocsis i Szepesvari), OMC (Chaslot), PPBM (Coulom) i UCB1-TUNED (Gelly i Wang). Warto podkreślić, że wszystkie zaprezentowane strategie są niezależne od gry i nie korzystają z wiedzy dziedzinowej.

Algorytm UCB

Algorytm UCB (ang. Upper Confidence Bound) nie jest metodą przeszukiwania drzewa. Celem algorytmu UCB jest wybór, na podstawie losowych prób, opcji zapewniającej największą wygraną. Oszacowanie wygranej dla każdej z opcji składa się z wartości średniej wygranych oraz czynnika, którego wartość maleje wraz ze wzrostem liczby prób. Algorytm wybiera tę opcję, której szacowana wartość wygranej jest największa. W istocie UCB korzysta z badań nad problem z teorii gier – MAB (ang. Multi-Armed Bandit). W problemie MAB rozważa się maszynę hazardową wyposażoną w N ramion. W każdym kroku, gracz może wybrać jedno z N ramion urządzenia. Celem gracza jest maksymalizacja nagrody.

Algorytm UCT

Strategia wyboru w algorytmie UCT (ang. Upper Confidence bound to Trees) bazuje na algorytmie UCB. Algorytm jest często opisywany z użyciem taksonomii algorytmu MAB. Każdy węzeł N w drzewie (stan gry) razem z węzłami potomnymi N_i jest odpowiednikiem „wielorękiego bandyty". Węzły potomne N_i są rękami „bandyty". W ten sposób, całe drzewo gry złożone z N węzłów jest postrzegane, jako N „wielorękich bandytów".

Strategia UCT została opracowana przez Kocsisa i Szepesvari'ego. Strategia ta jest łatwa w implementacji i jest wykorzystywana w wielu programach (m.in. została zastosowana w programie komputerowym MANGO). UCT działa w następujący sposób:

1. Graj każde z „ramion" jeden raz.
2. Graj to „ramię", które maksymalizuje formułę $v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}$.

Innymi słowy, jeżeli I jest zbiorem osiągalnych węzłów od aktualnego węzła p, to UCT wybiera $k - ty$ węzeł potomny węzła p, jeżeli spełnia on formułę 3.1:

$$k \in argmax_{i \in I}\left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}\right) \tag{3.1}$$

gdzie $argmax$ oznacza *argument maksimum*, to znaczy wartość argumentu, dla którego funkcja osiąga maksimum[8]. v_i jest wartością węzła i, n_i jest liczbą odwiedzin węzła i, n_p jest liczbą odwiedzin węzła p i C jest współczynnikiem, który jest dobierany eksperymentalnie. Pierwotnie autorzy algorytmu zaproponowali $C = 2/\sqrt{2}$.

Jak można zinterpretować „działanie" formuły 3.1? Załóżmy, że dla węzła zostały już przeprowadzone symulacje. Pierwszy składnik v_i wyznacza najlepszy możliwy węzeł w analizowanym drzewie (eksploatacja), podczas gdy drugi z nich decyduje o ewentualnym dalszym badaniu drzewa (eksploracja). Jeżeli jeden z węzłów był rzadko odwiedzany, to wartość drugiego czynnika jest duża i tym samym rośnie prawdopodobieństwo, że ten węzeł zostanie wybrany do dalszych badań. Parametr C reguluje proporcję pomiędzy eksploatacją i eksploracją. Uważa się, że strategia UCT wyjątkowo dobrze zachowuje równowagę pomiędzy eksploracją i eksploatacją. UCT odwzorowuje logarytmiczną funkcję żalu (żal jest oczekiwaną stratą po n grach spowodowany faktem, że algorytm nie zawsze wybierał „ramię bandyty" w sposób optymalny). Dla porównania, funkcja żalu dla prostej strategii korzystającej z oceny Monte-Carlo jest liniowa.

Pseudokod implementacji UCT pokazano w algorytmie 3.3. W pseudokodzie pominięto funkcje wstecznej propagacji z uwagi na jej oczywistą implementację.

Algorytm 3.3. Pseudokod MCTS ze strategią wyboru UCT.

```
FUNCTION PRZESZUKIWANIE_UCT(s_0)
   Utwórz nowy węzeł główny v_0 ze stanem s_0
   WHILE jest czas DO
      v_k ← POLITYKA_DRZEWA(v_0)
      R ← POLITYKA_SYMULACJI(s(v_k))
      WSTECZNA_PROPAGACJA(v_k;R)
   END
RETURN a(NAJLEPSZY_RUCH(v_0; 0))
```

[8] $argmax_x\, f(x) \in \{\, x \mid \forall y : f(y) \leq f(x) \,\}$

```
FUNCTION POLITYKA_DRZEWA(v)
   WHILE v nie jest węzłem końcowym DO
      IF v nie jest całkowicie rozwinięty THEN
         RETURN ROZWIN(v)
      ELSE
         v ← NAJLEPSZY_POTOMEK(v;Cp)
      END
   END
RETURN v

FUNCTION ROZWIN(v)
   Wybierz ruch a ∈ A(s(v)), który nie był jeszcze sprawdzany
   Dodaj nowy węzeł v0 do v
      podstaw s(v') = f(s(v); a)
      oraz a(v') = a
RETURN v'

FUNCTION NAJLEPSZY_ POTOMEK(v,c)
RETURN argmax_{i∈I}(v_i + C × √(ln n_p/n_i))

FUNCTION POLITYKA_SYMULACJI(s)
   WHILE s nie jest stanem końcowym DO
      Wybierz losowo a ∈ A(s)
      s ← f(s,a)
   END
RETURN rezultat R dla stanu końcowego s
```

Algorytm OMC

Algorytm OMC (ang. Objective Monte-Carlo) został zaproponowany przez Chaslota. Algorytm składa się z dwóch elementów. Pierwszym jest *funkcja pilności*, która określa pilność U(i) dla ruchu. Drugim jest *funkcja sprawiedliwości*, która decyduje, który z potencjalnych ruchów powinien być wykonany. Wartość funkcji uczciwości jest proporcjonalna do funkcji pilności. Funkcja pilności w algorytmie OMC wyrażona jest formułą 3.2:

$$U(i) = erfc(\frac{x_o - x_{oi}}{\sqrt{2}\sigma_i}) \tag{3.2}$$

gdzie $erfc(.)$ jest uzupełniającą funkcją błędu[9], v_0 jest wartością najlepszego ruchu oraz v_i i σ_i są wartością i odchyleniem standardowym potencjalnego ruchu. Ideą tej formuły jest to, aby wartość funkcji $U(i)$ była proporcjonalna do prawdopodobieństwa i żeby potencjalny ruch był lepszy niż aktualny najlepszy ruch. Następnie wybierany jest potomny węzeł według następującej zasady: wybierz węzeł $\boldsymbol{i}$, który maksymalizuje wartość funkcji uczciwości f_i wyrażoną formułą 3.3:

$$f_i = \frac{n_p \times U(i)}{n_i \times \sum_{j \in S_i} U(j)} \tag{3.3}$$

gdzie n_i jest liczbą odwiedzin węzła i, p jest liczbą odwiedzin węzła p oraz S_i jest zbiorem zawierającym węzły sąsiadujące z węzłem i.

[9] *Funkcja błędu Gaussa* $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt;$ $\qquad erfc(x) \equiv 1 - erc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt;$

Algorytm PBBM

W 2006 Coulom zaproponował algorytm PBBM (ang. Probability to be Better than Best Move). Podobnie jak OMC, algorytm PBBM składa się z funkcji pilności i funkcji sprawiedliwości. Funkcja pilności $U(i)$ jest proporcjonalna do prawdopodobieństwa, że potencjalny ruch jest lepszy niż aktualnie najlepszy ruch. W odróżnieniu od OMC, algorytm PBBM bierze pod uwagę odchylenie standardowe najlepszego ruchu. Funkcja pilności wyrażona jest formułą 3.4:

$$U(i) = \exp(-2.4 \times \frac{v_o - v_i}{\sqrt{2(\sigma_o^2 + \sigma_i^2)}}) \tag{3.4}$$

gdzie v_o i σ_0 są wartością i standardowym odchyleniem **najlepszego** ruchu, v_i i σ_i są wartością i odchyleniem standardowym **potencjalnego** ruchu.

Algorytm UCB1-TUNED

Gelly i Wang (2006) zaproponowali wariant nazwany przez autorów UCB1-TUNED. W strategii UCB1-TUNED wybierany jest węzeł potomny ***k***, jeżeli spełnia on formułę 3.5:

$$k \in argmax_{i \in I}\left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i} \times min\left\{\frac{1}{4}, V_i(n_i)\right\}}\right) \tag{3.5}$$

gdzie funkcja V_i jest szacowaną górna granicą wariancji v_i i określona jest wzorem 3.6:

$$V_n(n_i) = \left(\frac{1}{n_i}\sum_{t=1}^{n_i} R_{i,t,j}^2 - v_i^2 + \sqrt{\frac{2\ln n_p}{n_i}}\right) \tag{3.6}$$

gdzie $R_{i,t,j}$ jest wypłatą uzyskaną w węźle ***i*** przez gracza ***j***. Wymaganie, aby każde z „ramion" było grane jeden raz jest ograniczeniem, szczególnie wtedy gdy dobre „ramię" zostanie odkryte na początku. Dlatego zaproponowano użycie globalnej stałej FPU (ang. First Play Urgency), której wartość empirycznie ustalono na 1.1. Wartość „ramienia" jest określona według formuły 3.7:

$$W_i = \begin{cases} FPU\ dla\ n_i = 0 \\ v_i\ dla\ n_i > 0 \end{cases} \tag{3.7}$$

W ten sposób dobre „ramię" może być grane kilka razy z rzędu nawet, jeżeli inne nie były jeszcze grane ani razu. Technika ta była stosowana we wczesnych implementacjach algorytmu UCB. Aktualnie używane są bardziej zaawansowane metody polegające na oszacowaniu początkowej wartości węzła.

3.4.2. Wzrost

W kroku wzrostu do drzewa MCTS dodawane są nowe węzły. Ponieważ w większości zastosowań całe drzewo gry nie może być przechowywane w pamięci, strategia wzrostu decyduje, czy do danego węzła ***L*** będą dołączone nowe węzły potomne. Najprostszą strategię wzrostu można opisać w następujący sposób:

Dla każdej kolejnej symulowanej rozgrywki dodawany jest jeden nowy węzeł. Węzeł ten odpowiada pierwszej pozycji napotkanej podczas przeglądania drzewa, która nie została jeszcze dołączona do drzewa gry.

Ta prosta strategia jest wydajna, łatwa do wykonania i nie zajmuje zbyt dużo pamięci. Możliwe też są inne strategie. Na przykład, można rozwijać drzewo do pewnej głębokości (np. 2 lub 3 warstwy) przed rozpoczęciem przeszukiwania. Możliwe jest również dodanie wszystkich węzłów potomnych do węzła macierzystego dopiero, gdy liczba symulacji

przebiegających przez ten węzeł przekroczy zadany próg $\boldsymbol{T}$. Jednak ta strategia jest w praktyce możliwa do zaimplementowania tylko wtedy, gdy dostępna jest pamięć o dużej pojemności. W przeciwieństwie do tej strategii, można zabronić dodawania węzła potomnego dopóki liczba symulacji przechodzących przez węzeł macierzysty nie przekroczy progu T. To podejście pozwala na zaoszczędzenie pamięci, ale jednocześnie nieznacznie zmniejsza siłę gry programu. Ogólnie, wpływ przytoczonych strategii na siłę gry programu jest mały. Strategia dodawania tylko jednego węzła w jednej rozgrywce jest w większości przypadków wystarczająca.

3.4.3. Symulacja

Symulacja (zwana także rozgrywką) jest krokiem, w którym program gra sam ze sobą aż do końca gry. Symulacja może polegać na wykonywaniu losowych ruchów lub lepiej pseudo-losowych zależnych od przyjętej strategii symulacji. Zastosowanie odpowiedniej strategii symulacji może znacząco poprawić siłę gry programu. Głównym pomysłem jest wykonywanie korzystnych ruchów zgodnie z wiedzę heurystyczną.

Strategia symulacji jest przedmiotem dwóch kompromisów. Pierwszym z nich jest kompromis pomiędzy poszukiwaniem i wiedzą. Dodawanie wiedzy do strategii symulacji zwiększa siłę gry. Symulowane gry stają się bardziej dokładne, a ich wyniki bardziej wiarygodne. Jednakże, jeśli wiedza heurystyczna jest zbyt kosztowna obliczeniowo, to liczba symulacji na sekundę może się nadmiernie zmniejszyć − drzewo MCTS będzie płytkie i siła gry spadnie. Drugi kompromis dotyczy proporcji pomiędzy eksploracją i eksploatacją. Jeśli strategia jest zbyt stochastyczna (np. zbyt dużo losowości), to eksploracja zaczyna być czynnikiem dominującym. Ruchy wykonywane w symulowanych grach są często słabe, co czyni symulacje mało realnymi i słabnie siła programu. Z drugiej strony, jeżeli strategia jest zbyt deterministyczna (co oznacza, że ruch dla danej pozycji jest prawie zawsze taki sam), wówczas eksploatacja staje się czynnikiem dominującym. W efekcie siła programu również maleje.

Ze względu na te dwa kompromisy, opracowanie skutecznej strategii symulacji jest trudnym problemem. Istnieją dwa sposoby oceny jakości strategii symulacji. Określamy strategię symulacji A, jako *lepszą* niż strategia symulacji B, jeśli A wygra więcej pojedynków z B, gdy oba programy grają przeciwko sobie (tj. bez użycia MCTS). Określamy, że strategia symulacji A jest *MCTS-lepsza* niż strategii symulacji B, jeśli program MCTS stosujący strategię A wygra więcej pojedynków z tym samym programem MCTS i stosującym strategię B. Celem projektowania strategii symulacji jest skonstruowanie strategii MCTS-*lepszej*. Bouzy i Chaslot (2006) pokazali, że jest możliwe, aby strategia symulacji była *lepsza* nie będąc jednocześnie MCTS-*lepszą*. Zostało to również zauważone i opisane przez Gelly'ego i Silvera (2007).

3.4.4. Wsteczna propagacja

Wsteczna propagacja jest krokiem, w którym *wynik* symulowanej gry k jest propagowany od liścia L w kierunku wierzchołka przez wszystkie węzły, przez które przechodził algorytm podczas symulacji. Na przykład, dla dwuosobowej gry o sumie zerowej (np. Go), wynik jest dodatni ($R_k = +1$), jeżeli wygrały czarne i ujemny ($R_k = -1$) jeżeli białe. Wynik gry zakończonej remisem jest równy zero ($R_k = 0$). Strategia propagacji wstecznej aktualizuje wartości v_L w węźle. Najbardziej popularną i najbardziej skuteczną strategią jest średnia arytmetyczna wyników wszystkich gier symulacji, które odwiedziły węzeł $v_L = (\sum_k R_k)/n_L$. Po raz pierwszy strategia średniej został użyta przez Kocsisa i Szepesvariego w 2006 roku i od tej pory jest stosowana w większości programów.

3.5. Ostateczny wybór ruchu

Po przeprowadzeniu symulacji − ruch, który zostanie wykonany w rzeczywistej grze odpowiada najlepszemu węzłowi potomnemu wierzchołka drzewa gry. Najlepszy węzeł spośród węzłów potomnych może zostać wybrany na kilka sposobów. Kryteria wyboru zostały opisane w tabeli 3.1.

Tabela 3.1. Kryteria wyboru najlepszego węzła.

Węzeł	Kryterium wyboru
Maksymalny	Węzeł mający największą wartość.
Silny	Węzeł mający największą liczbę odwiedzin.
Maksymalny i silny	Węzeł mający największą wartość oraz mający największą liczbę odwiedzin. Jeżeli nie ma takiego węzła, to wykonywane są dalsze symulacje aż do jego znalezienia.
Bezpieczny	Węzeł mający maksymalną dolna granicę ufności, tzn. mający maksymalną wartość formuły $v + A/\sqrt{n}$, gdzie A jest stałą dobieraną doświadczalnie, v jest wartością węzła, n liczbą odwiedzin.

3.6. Zalety i wady algorytmu MCTS

Na zakończenie warto podsumować zalety i wady algorytmu MCTS.

Zalety:

a) Algorytm nie wymaga wiedzy o danej dziedzinie, aby podejmować odpowiednie decyzje. Nie jest wymagana funkcja oceny pozycji, której budowa może być trudna (gra w Go).
b) Algorytm może być wykonywany dowolną liczbę iteracji i może być przerwany praktycznie w dowolnej chwili np. po wyczerpaniu limitu czasu. W przypadku algorytmu alfa-beta podobny efekt można uzyskać za pomocą iteracyjnego pogłębiania[10].
c) Im dłużej działa algorytm tym lepsze znajduje rozwiązania. Sterując czasem przeznaczonym na ruch, można wpływać na jakość rozwiązania. Algorytm gwarantuje uzyskanie rozwiązania (niekoniecznie optymalnego).
d) Algorytm pozwala na rozwiązywanie problemów o dużej złożoności kombinatorycznej, które nie poddają się klasycznym algorytmom.
e) Algorytm płynnie dostosowuje się do zmian oceny pozycji. W początkowym „okresie życia" węzła, eksploracja jest dominująca i każdy z węzłów potomnych (ramion „bandyty") grany jest wiele razy. Jednak wraz ze wzrostem liczby odwiedzin, każdy z węzłów grany jest coraz rzadziej. Często tylko jeden raz. Sąsiednie węzły lepiej „rokujące" są częściej wybierane do symulacji. Wynik symulacji zbiega do wyniku algorytmu alfa-beta (po dostatecznie długim czasie).

[10] Iteracyjne pogłębianie polega na przeszukiwaniu w głąb, ale jedynie do założonej głębokości. W kolejnych wywołaniach przeszukiwania, głębokość jest powiększana aż do znalezienia stanu terminalnego.

f) Algorytm automatycznie buduje asymetryczne drzewo. Gałęzie lepiej „rokujące” są silniej rozbudowane i drzewo w tym obszarze jest głębsze. Tam, gdzie zagrania wydają się mniej interesujące, drzewo jest płytkie. Większość nakładu obliczeniowego poświęcana jest badaniu obiecujących ruchów.
g) Algorytm można efektywnie zrównoleglić w wielordzeniowych procesorach CPU (ang. Central Processing Unit), klastrach i procesorach graficznych GPU (ang. Graphics Processing Unit). Metody zrównoleglenia algorytmu MCTS opisane zostały w rozdziale 5.

Wady:

a) Słaba taktyka gry. Algorytm nie sprawdza najlepszych ruchów w otoczeniu wierzchołka, który odpowiada bieżącemu stanowi gry.
b) Algorytm jest wolniejszy niż alfa-beta. Strategia wyboru wymaga dużych nakładów obliczeniowych. Dla każdego odwiedzanego węzła (od wierzchołka do liścia) algorytm musi obliczyć i porównać wartość funkcji oceny dla każdego z węzłów potomnych.
c) Całe drzewo gry musi być przechowywane w pamięci.
d) Niepewność, co do jakości znalezionego rozwiązania. Jest to największa wada w stosunku do algorytmów „klasycznych”, które znajdują dokładne rozwiązanie.

3.7. Podsumowanie

W tym rozdziale przedstawione zostały ogólne ramy dla algorytmu przeszukiwania Monte-Carlo (MCTS). Algorytm MCTS jest zachłanną strategią przeszukiwania korzystającą z algorytmu pierwszy-najlepszy, która nie wymaga funkcji oceny, w przeciwieństwie do algorytmu alfa-beta. Algorytm MCTS bazuje na losowej eksploracji obszaru wyszukiwania. Korzystając z wyników poprzednich poszukiwań, MCTS stopniowo buduje drzewo gry w pamięci i sukcesywnie coraz dokładniej szacuje wartości najbardziej obiecujących ruchów.

Omówiliśmy cztery główne kroki strategii MCTS: wybór, wzrost, symulację i wsteczną propagację. W fazie wyboru najczęściej stosowany jest algorytm UCT, ponieważ jest efektywny i łatwy do wdrożenia. Algorytm UCT w swojej podstawowej postaci nie bierze pod uwagę wiedzy dziedzinowej, której dodanie mogłyby zwiększyć jego efektywność. W kolejnym kroku, stosowana jest prosta i skuteczna strategia, która rozwija drzewo dołączając jeden węzeł w każdej symulacji.

Następny krok − symulacja, jest najtrudniejszą fazą algorytmu MCTS. Dla strategii symulacji, należy znaleźć optymalne proporcje pomiędzy przeszukiwaniem (eksploracją) i aktualnie posiadaną wiedzą (eksploatacją). Wreszcie, w kroku propagacji wstecznej najefektywniejszą strategią jest zapisywanie średniej wyników wszystkich gier symulacji wykonanych z danego węzła.

Optymalizacja algorytmów w krokach wyboru i symulacji może mieć największy wpływ na poprawę efektywności przeszukiwania i wzrost siły gry programu.

4. Usprawnienia algorytmu MCTS

Heurystyka AMAF

W algorytmie MCTS, w kroku wstecznej propagacji, aktualizowane są wartości tylko tych węzłów, przez które prowadziła ścieżka przeszukiwania drzewa w fazie wyboru. W rezultacie proces poszukiwania najlepszych ruchów jest wolny, co jest istotną wadą algorytmu. Heurystyka AMAF (All-Moves-As-First) bazuje na spostrzeżeniu, że wartości ruchów mogą być częściowo niezależne od kolejności, w jakiej są wykonywane. Innymi słowy, ocena ruchu jest niezależna od kontekstu. Takie podejście pozwala na przeprowadzenie większej liczby symulacji w tym samym czasie i w efekcie prowadzi do wzrostu efektywności algorytmu. Jednym z takich istotnych usprawnień algorytmu MCTS, które korzysta z heurystyki AMAF jest algorytm UCT-RAVE. Algorytm UCT-RAVE został omówiony w punkcie 4.2.

Wykorzystanie wiedzy dziedzinowej

Algorytm MCTS i jego warianty nie korzystają z wiedzy domenowej[11] (wyjątkiem jest przestrzeganie prostych zasad gry np. reguły *ko*). Dodanie wiedzy może znacząco poprawić siłę gry programu. Z uwagi na architekturę algorytmu MCTS naturalnym pomysłem jest zastosowanie wiedzy w strategiach: (1) przeglądania drzewa oraz (2) symulacji.

Wiedza w strategii przeglądania drzewa użyta stosunkowo blisko wierzchołka, powinna pomóc przy filtracji słabych ruchów.

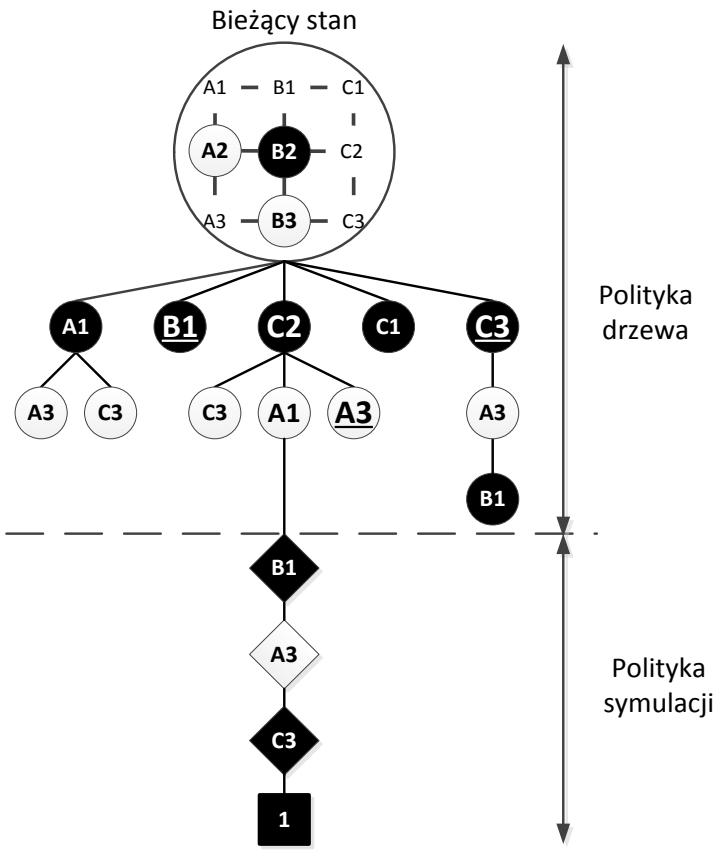
W strategii symulacji, użycie wiedzy może poprawić jakość rozgrywek i zmniejszyć ich liczbę niezbędną do oszacowania wartość pozycji. Rozgrywki korzystające z wiedzy, nazywane są „ciężkimi” (ang. heavy), w odróżnieniu od „lekkich” (ang. light) rozgrywek składających się wyłącznie z losowych ruchów.

4.1. Heurystyka AMAF

W „klasycznym” algorytmie MCTS, jeżeli symulowana gra zakończy się wygraną, to przyjmuje się założenie, że tylko ruch, który rozpoczął rozgrywkę jest dobrze rokujący. W konsekwencji, podczas wstecznej propagacji aktualizowane są wartości węzłów: liścia, w którym zaczęła się symulacja, wszystkich węzłów nadrzędnych i ostatecznie − wierzchołka.

W heurystyce AMAF przyjmuje się założenie, że jeżeli gra zakończyła się wygraną, to wszystkie ruchy w grze były „dobre”. Heurystykę AMAF można opisać w następujący sposób. Załóżmy, że rozgrywka rozpoczyna się w stanie $\boldsymbol{s}$ od wykonania ruchu $\boldsymbol{a}$. Jeżeli dla różnej kolejności wykonywanych ruchów (przez tego samego gracza) wynik gry będzie taki sam, to każdy z ruchów może otrzymać taka ocenę jak ruch $\boldsymbol{a}$. **Z perspektywy AMAF wszystkie rozgrywki wykonane z danego wierzchołka są wykonane równocześnie**. Rozumowanie to bazuje na przeświadczeniu, że ocena ruchu zależy od pozycji a nie od kolejności ruchów, które doprowadziły do bieżącego stanu. Jest to uproszczenie, ponieważ w rzeczywistości kolejność wykonywanych ruchów przez gracza zależy w dużej mierze od ruchów oponenta. Przykład rozgrywki i porównanie aktualizacji bazowej z aktualizacją AMAF zilustrowano na rysunku 4.1.

[11] Przez wiedzę należy rozumieć informację dostarczoną przez eksperta lub zdobytą w procesie uczenia maszynowego.



Rysunek 4.1. Przykład rozgrywki i porównanie aktualizacji bazowej z AMAF.

Bieżący stan gry (wierzchołek drzewa gry) wyznaczony jest przez czarny kamień B2 oraz dwa kamienie białe A2 i B3. W wyniku kilku poprzednich iteracji algorytmu MCTS zostało zbudowane drzewo gry (polityka drzewa). Kolejny ruch (ze stanu bieżącego) ma być wykonany przez czarne. Algorytm poszukując optymalnego zagrania, analizował możliwe ruchy czarnych: A1, B1, C2, C1, C3. Na potencjalny ruch czarnych C2, algorytm sprawdzał możliwe odpowiedzi białych: C3, A1, A3 itd.

W gałęzi wierzchołek – C2 – A1, od liścia A1 **rozpoczyna się symulowana rozgrywka** (polityka symulacji): B1, A3, C3, zakończona zwycięstwem czarnych.

Aktualizacja bazowa (np. UCT) w kroku wstecznej propagacji zmieni wartości tylko w węzłach: A1, C2 oraz wierzchołku.

Aktualizacja AMAF na podstawie ruchu (w kolejności wstępującej):

- C3 czarne w symulacji – zmiana wartości w węźle C3 czarne w drzewie;
- A3 białe w symulacji – zmiana wartości w węźle A3 białe w drzewie;
- B1 czarne w symulacji – zmiana wartości w węźle B1 czarne w drzewie;
- A1 i C2 jak dla symulacji bazowej.

Warto zwrócić uwagę, że ruch C3 czarne w symulacji NIE zmieni wartości w dwóch białych węzłach C3 w drzewie z powodu niezgodności koloru. Ruch A3 białe w symulacji NIE zmieni wartości w dwóch białych węzłach A3 ponieważ nie sąsiadują z węzłami gałęzi, dla której przeprowadzona została symulacja. Ruch B1 czarne w symulacji NIE zmieni wartości w B1 czarne, ponieważ B1 czarne nie sąsiadują z gałęzią, dla której przeprowadzona została symulacja.

Heurystyka AMAF ma kilka wariantów (Helmbold i Parker-Wood).

Heurystyka **α-AMAF** łączy wartość oceny ruchu otrzymaną w aktualizacji bazowej (np. UCT) z aktualizacją AMAF. Każdy węzeł drzewa ma dwa liczniki: (1) aktualizacji bazowej C_B i (2) aktualizacji AMAF C_{AMAF}. Wartość ruchu określona jest formułą 4.1:

$$C_{\alpha-AMAF} = \propto \times C_{AMAF} + (1-\propto) \times C_B \tag{4.1}$$

Dla $\propto = 1$ formuła odpowiada algorytmowi AMAF, dla $\propto = 0$ − algorytmowi UCT.

Heurystyka **m-pierwszych**. Po zakończeniu rozgrywki aktualizowane jest tylko m-pierwszych węzłów (pozostałe węzły są pomijane). Jeżeli $m = 0$ heurystyka odpowiada algorytmowi bazowemu, dla dostatecznie dużego m − algorytmowi AMAF.

Heurystyka **z odcięciem**. Dla pierwszy k rozgrywek aktualizowane są wartości węzłów według AMAF. Kolejne rozgrywki aktualizują wartości według algorytmu bazowego. Celem jest szybkie ustawienie początkowych wartości liczników i następnie poszukiwanie „dokładnej" wartości za pomocą metody bazowej. Jeżeli $k = 0$, to wariant odpowiada algorytmowi bazowemu i dla dostatecznie dużego k – algorytmowi AMAF.

Heurystyka **RAVE** (ang. Rapid Action Value Estimation). Celem heurystyki jest (podobnie jak heurystyki z odcięciem) szybkie ustawienie początkowych wartości. Wariant bazuje na heurystyce α-AMAF. W α-AMAF, współczynnik α był wspólny dla całego drzewa, w RAVE − każdy z węzłów ma indywidualny współczynnik α, którego wartość początkowa równa jest jeden i maleje do zera wraz z liczbą symulacji przechodzących przez ten wierzchołek. Dynamika zmian współczynnika α kontrolowana jest za pomocą formuły 4.2:

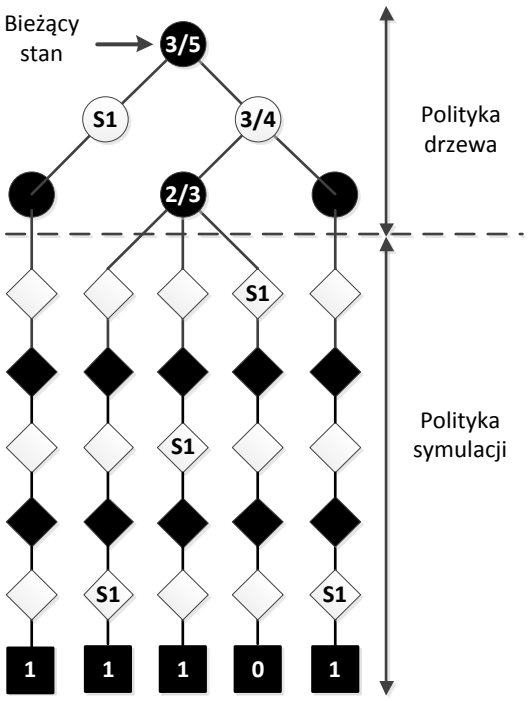
$$\propto = \max\left(\frac{P-N_S}{P}, 0\right) \tag{4.2}$$

gdzie N_s jest liczbą symulacji przechodzących przez węzeł i P empirycznie dobieranym parametrem. Jeżeli $N_s > P$, to $\propto = 0$.

4.2. Algorytm UCT-RAVE

Heurystyka AMAF w programie do gry w Go została po raz pierwszy zaproponowana przez Brügmanna (1993). Gelly i Silver użyli AMAF do budowy algorytmu UCT-RAVE.

Algorytm UCT-RAVE jest rozszerzeniem heurystyki AMAF w ramach UCT. Algorytm MCTS w celu dobrego oszacowania wartości ruchu wymaga przeprowadzenia dużej liczby symulacji. Ocena każdego ruchu przeprowadzana jest niezależnie. Algorytm RAVE pozwala na szybkie oszacowanie wartości każdego ruchu na podstawie uogólnienia rezultatów z równoległych rozgrywek. Zilustrowano to na rysunku 4.2. Algorytm RAVE uwzględnia wszystkie wystąpienia $\boldsymbol{S1}$ w rozgrywkach, w sumie w pięciu − czterech wygranych i jednej przegranej. Wartość ruchu jest współdzielona przez wszystkie poddrzewa wchodzące w skład drzewa gry. Innymi słowy, jeżeli bieżący stan jest korzeniem dla kilku poddrzew, to wartość węzła $\boldsymbol{S1}$ we wszystkich poddrzewach będzie taka sama. Normalnie, jeżeli algorytm MCTS wykonuje iterację od zadanego wierzchołka (bieżącego stanu gry), to jedynie wartości w węzłach, przez które przechodził algorytm są aktualizowane.



Rysunek 4.2. Idea algorytmu UCT-RAVE.

Algorytm UCT-RAVE może zostać opisany w następujący sposób:

- W każdym węźle oprócz statystyk UCT, przechowywane są statystyki RAVE: wartość i liczba odwiedzin.
- Dla każdej iteracji rozpoczynanej w danym wierzchołku zaktualizuj statystyki RAVE dla wszystkich węzłów potomnych odwiedzonych podczas iteracji.
- Podczas wyboru działania użyj obu statystyk UCT i RAVE w taki sposób, że jeżeli węzeł był odwiedzany małą liczbę razy, to dominującą staje się ocena RAVE. Jeżeli liczba odwiedzin rośnie, to dominującą staje się ocena UCT. Zilustrowano to na rysunku **Błąd! Nie można odnaleźć źródła odwołania.**. Połączona estymata UCT i RAVE jest opisana formułą 4.3:

$$Q_{UCT-RAVE}(s,a) = \beta \times Q_{RAVE}(s,a) + (1-\beta) \times Q_{UCT}(s,a) \tag{4.3}$$

$$\beta = \sqrt{\frac{k}{3n(s)+k}}$$

gdzie $\boldsymbol{s}$ jest stanem, $\boldsymbol{a}$ wykonywanym działaniem (ruchem), $\boldsymbol{\beta}$ współczynnikiem wagowym, którego wartość maleje od 1 do 0 wraz ze wzrostem liczby odwiedzin, $\boldsymbol{k}$ nazywany jest współczynnikiem równoważności. Q_{UCT} jest wartością oceny UCT określoną formułą (3.1). $Q_{RAVE}(s,a)$ określony jest formułą 4.4, której konstrukcja przypomina formułę UCT:

$$Q_{RAVE}(s,a) = \overline{X_{RAVE,a}} + \mathrm{C} \times \sqrt{\frac{\ln m(s)}{m(s,a)}} \tag{4.4}$$

gdzie $\overline{X_{RAVE,a}}$ jest średnią wartością RAVE ruchu $\boldsymbol{a}$ w węźle $\boldsymbol{s}$, $\boldsymbol{m(s)}$ jest liczbą odwiedzin węzła $\boldsymbol{s}$ przez RAVE, $\boldsymbol{m(s,a)}$ jest liczbą odwiedzin węzła $\boldsymbol{s}$ przez RAVE będących wynikiem ruchu $\boldsymbol{a}$. C jest współczynnikiem, który jest dobierany eksperymentalnie.

4.3. Wykorzystanie wiedzy w strategii przeglądania drzewa

W tym punkcie zostaną opisane dwa algorytmy przeglądania drzewa wariantów korzystające z wiedzy. Pierwszy – przeglądanie z progresywnym uprzywilejowaniem (ang. progressive bias) został zaproponowany przez Chaslota (2010). Drugi – przeglądanie z progresywnym poszerzaniem (ang. progressive widening) jest autorstwa Couloma (2007).

4.3.1. Przeglądanie z progresywnym uprzywilejowaniem

Celem algorytmu progresywnego uprzywilejowania jest sterowanie wyborem ścieżki przeglądania drzewa wariantów na podstawie funkcji heurystycznej (korzystającej z wiedzy). Wpływ funkcji heurystycznej na wybór ścieżki jest największy na początku działania algorytmu i stopniowo maleje wraz ze wzrostem liczby symulacji. Strategia uprzywilejowania została opisana na przykładzie algorytmu UCT, ale może być użyta zarówno w bazowej wersji algorytmu MCTS jak i innych jego udoskonalonych wariantach.

Dla przypomnienia, algorytm UCT wybiera $\boldsymbol{k-ty}$ węzeł potomny węzła $\boldsymbol{p}$, jeżeli spełnia on formułę 3.1 (punkt 3.4. „Fazy algorytmu MCTS"). Do bazowej formuły zostaje dodany czynnik $\boldsymbol{f(n_i)}$, którego wartość zależy od wiedzy:

$$k \in argmax_{i \in I}\left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + f(n_i)\right) \tag{4.5}$$

gdzie $argmax$ oznacza wartość argumentu, dla którego funkcja osiąga maksimum; v_i jest wartością węzła i; n_i jest liczbą odwiedzin węzła i; n_p jest liczbą odwiedzin węzła p i C jest współczynnikiem dobieranym eksperymentalnie. Funkcja heurystyczna korzystająca z wiedzy $\boldsymbol{f(n_i)}$ może być skonstruowana na wiele sposobów. Istotne jest, aby funkcja była malejąca dla rosnącej liczby symulacji. Formuła 4.6 została użyta w programie MANGO

$$f(n_i) = \frac{H_i}{n_i+1} \tag{4.6}$$

formuła 4.7 w programie MOGO

$$f(n_i) = \frac{H_i}{\ln(n_i+2)} \tag{4.7}$$

gdzie $\boldsymbol{H_i}$ reprezentuje wiedzę w stanie $\boldsymbol{i}$. Wartość $\boldsymbol{H_i}$ zależy od wielu czynników decydujących o tym czy potencjalny ruch jest silny czy słaby. Poniżej opisano niektóre z nich.

Heurystyka zbitych kamieni. Ocena ruchu zależy od liczby zbitych kamieni przeciwnika. Ruch, który może doprowadzić do zbicia ma najwyższą wagę.

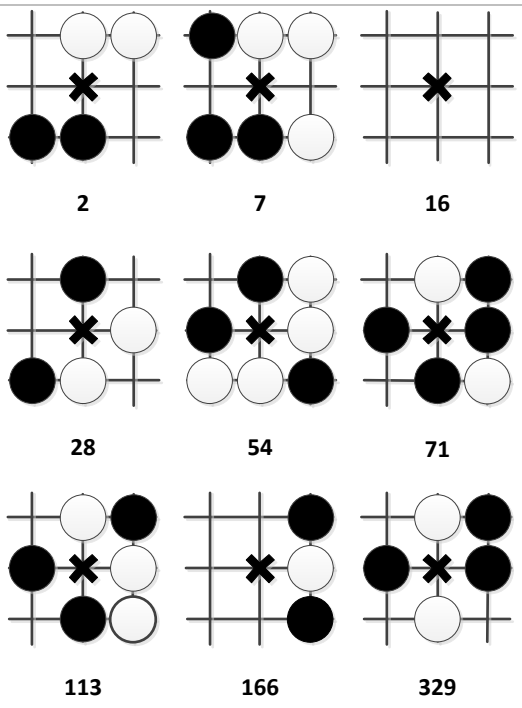
Heurystyka kamieni, które unikną zbicia. Ocena ruchu zależy od liczby kamieni, aktualnie znajdujących się w *atari*, które unikną zbicia przez przeciwnika.

Heurystyka bliskości (ang. proximity). Jak daleko znajduje się wybrany punkt od poprzedniego ruchu? Do pomiaru odległości może być stosowana miara Euklidesowa albo Manhattan.

Heurystyka dążąca do walki i zdobywania terytorium. W heurystyce ukierunkowanej na walkę i zdobywanie terytorium wykonywane są ruchy w miejscach sąsiadujących z grupami kamieni przeciwnika mających jak najmniejszą liczbę oddechów.

Heurystyka korzystająca ze wzorców. Heurystyka korzystająca z „dobrych" wzorców korzysta z bazy wzorców składających się z fragmentu planszy o rozmiarze 9×9, z zaznaczonymi polami wolnymi i zajmowanymi przez kamienie czarne lub białe. Każdemu potencjalnemu ruchowi, wykonywanemu w centralnym polu wzorca, przypisana jest waga. Heurystyka wybiera ruch mający największą wagę. Baza wzorców może być opracowana przez eksperta lub pozyskana na drodze maszynowego uczenia.

Na rysunku 4.3 pokazano przykładowe wzorce razem z ich wagami. Waga wzorca odpowiada sile ruchu wykonanego w węźle oznaczonym krzyżykiem.



Rysunek 4.3. Przykłady wzorców i wag.

Dla każdego potencjalnego ruchu należy obliczyć sumaryczną wartość heurystyk. Chaslot zaproponował formułę 4.8:

$$H_i = (V_i^{ce} + V_i^p) \times \sum_k \frac{1}{(2 \times d_{k,i})^{\propto k}} \tag{4.8}$$

gdzie V_i^{ce} jest sumą heurystyk zbicia i uniknięcia zbicia, V_i^p wartością wzorca. Trzeci czynnik jest sumą odległości w mierze euklidesowej dla wszystkich wcześniej wykonanych ruchów. Współczynnik $\propto_k = 1.25 + k/2$ został wyznaczony eksperymentalnie.

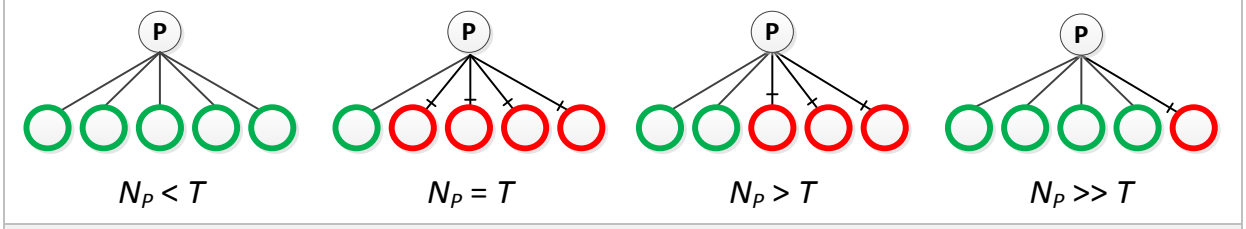
4.3.2. Przeglądanie z progresywnym poszerzaniem

Działanie algorytmu przeglądania z progresywnym poszerzaniem polega na początkowej redukcji stopnia rozgałęzienia drzewa wariantów i następnie na jego stopniowym zwiększaniu.

Redukcja stopnia rozgałęzienia drzewa. Po przekroczeniu przez liczbę symulacji N_p w węźle $\boldsymbol{P}$ zadanego progu $\boldsymbol{T}$, algorytm blokuje dostęp (dla kolejnych symulacji) do tych węzłów potomnych, dla których wartość funkcji heurystyki jest mniejsza od zadanego progu H_T. Niech k_{init} oznacza liczbę węzłów potomnych, które nie zostały zablokowane. Więzły te mają największą wartość funkcji heurystycznej.

Stopniowe zwiększanie rozgałęzienia drzewa. Stopniowo, wraz ze wzrostem liczby symulacji, na podstawie wartości funkcji heurystycznej, zdejmowana jest blokada dla kolejnych węzłów potomnych.

Działanie algorytmu zostało symbolicznie przedstawione na rysunku 4.4.



Rysunek 4.4. Przeglądanie z progresywnym poszerzaniem.

Dla liczby symulacji mniejszej od progu T, $\boldsymbol{N_P < T}$, wszystkie węzły potomne węzła P są dostępne dla symulacji.

Dla $\boldsymbol{N_P = T}$ wszystkie węzły potomne mające wartość funkcji heurystyki mniejszą od progu H_T są blokowane i stają się niedostępne dla symulacji.

Dla $\boldsymbol{N_P > T}$ węzły potomne są progresywnie odblokowywane i stają się ponownie dostępne dla symulacji.

Dla $\boldsymbol{N_P \gg T}$ większość węzłów potomnych jest ponownie dostępnych dla symulacji. Węzły słabo rokujące pozostają zablokowane.

W programie MANGO po przekroczenia progu T, liczba węzłów niezablokowanych wynosiła pięć ($k_{init} = 5$). Węzły potomne były stopniowo odblokowywane zgodnie z formułą 4.9:

$$N_p \geq \propto \times \beta^{k-k_{init}} \tag{4.9}$$

gdzie $\boldsymbol{k}$ liczba odblokowanych węzłów, N_p liczba symulacji w węźle P, współczynniki $\propto = 50$ oraz $\beta = 1.3$ zostały dobrane eksperymentalnie

4.4. Wykorzystanie wiedzy w strategii symulacji

Zbudowanie dobrej heurystyki dla gry w Go jest trudne. Drzewo gry jest silnie rozgałęzione i powstaje obawa, że heurystyka może eliminować silne zagrania. Jeszcze większym problemem są *martwe* kamienie, które pozostają na planszy przez wiele ruchów. Komplikuje to statyczną analizę stanu gry.

W podstawowej wersji algorytmu Monte Carlo, w kroku symulacji, ruchy wykonywane są wyłącznie losowo. Jedyną regułą implementowaną w tym kroku (poza sprawdzeniem legalności ruchu) jest pomijanie ruchów granych wewnątrz własnych *oczu*.

Drake i Uurtamo (2007) zaproponowali cztery proste heurystyk, które poprawiają szacowanie wyników w kroku symulacji [10]. Wybór ruchu wykonywany jest albo losowo albo zgodnie z jedną z heurystyk. Wybór heurystyki zależy od przypisanych im miar prawdopodobieństwa. Przykładowo, ruch wykonywany jest losowo z prawdopodobieństwem $1/2$, zgodnie z pierwszą heurystyką z prawdopodobieństwem $1/5$, drugą − $3/20$, trzecią − $1/10$ i czwartą − $1/20$. Suma wszystkich prawdopodobieństw jest równa $1/1$. O wyborze heurystyki decyduje losowanie metodą „koła ruletki”, które gwarantuje, że każdy ruch zostanie wykonany według jednej z heurystyk. Korzystanie z heurystyk spowalnia symulację i zmniejsza liczbę rozgrywek, dlatego w jednym ruchu stosowana jest tylko jedna heurystyka. Zaproponowano następujące heurystyki.

Heurystyka bliskości (ang. proximity). Losowo wybierany jest punkt położony w otoczeniu poprzedniego ruchu w odległości nie większej niż trzy punkty według miary Manhattan. Jeżeli znaleziony ruch nie jest zgodny z regułami gry, to wykonywany jest losowy ruch. Heurystyka działa zgodnie z taktyczną zasadą, że na wykonany ruch przeciwnika należy odpowiedzieć kontrą (najczęściej lokalną).

Heurystyka unikania skrajnych linii planszy (ang. avoid-the-first-two-lines). Losowo wybierany jest punkt, który nie jest położony na dwóch skrajnych liniach planszy. Przykładowo, jeżeli gra prowadzona jest na planszy 5×5 punktów, to jedynym punktem spełniającym ten warunek jest punkt C3. Naturalnie heurystyka może być stosowana tylko na dużych planszach. Jeżeli znaleziony ruch, który nie jest zgodny z regułami gry, to wykonywany jest losowy ruch. Ruchy wykonywane na obrzeżach planszy są słabe, zwłaszcza w początkowej i w środkowej fazie gry. Kamień ułożony na krawędzi kontroluje mniejsze terytorium i jest łatwym celem dla przeciwnika. Wyjątkiem od tej reguły są końcowe fragmenty rozgrywek.

Heurystyka najlepszego ruchu (ang. first-order history). Idea polega na wyborze ruchu, który jest dobrze notowany bez względu na położenie w drzewie gry. Jeżeli ruch został wysoko oceniony w wielu gałęziach drzewa, to najprawdopodobniej jest optymalny w pewnym lokalnym otoczeniu. Implementacja heurystyki polega na aktualizacji tabeli zawierającej licznik ruchów wybieranych przez algorytm w kroku wyboru (np. UCT). Jeżeli koło ruletki wylosuje tę heurystykę, to wybierany jest ruch najczęściej grany przez algorytm.

Heurystyka najczęściej granej obrony (ang. second-order history). W grach niekooperacyjnych ruch grany w fazie inicjatywy może być słabo oceniony, ale ten sam ruch w fazie defensywy, w odpowiedzi na atak przeciwnika, może być silny. W heurystyce wybierany jest ruch, który na podstawie poprzednich symulacji, okazał się najskuteczniejszą obroną przed atakiem przeciwnika. Implementacja heurystyki polega na aktualizacji tabeli zawierającej licznik ruchów odpowiedzi wybieranych przez algorytm w kroku wyboru (np. UCT) dla każdego ruchu każdego z graczy. Jeżeli koło ruletki wylosuje tę heurystykę, to w odpowiedzi na ruch przeciwnika wybierany jest ruch gracza najczęściej wybierany przez algorytm.

Heurystyka otwarcia gry. We wstępnej fazie rozgrywki, w pierwszych 5-10 ruchach można zastosować prostą heurystyką, polegającą na wykonywaniu ruchów wyłącznie na 4. i 5. linii (dla gier na planszy $19/19$).

4.4.1. Algorytm symulacji bazujący na pilności

Algorytm symulacji bazujący na pilności (ang. urgency-based simulation) został zaproponowany przez Bouzy'ego (2005). Wybór ruchu wykonywany jest na podstawie funkcji pilności U_i obliczanej dla każdego potencjalnego ruchu. O wyborze ruchu decyduje losowanie metodą „koła ruletki". Prawdopodobieństwo wylosowania ruchu P_i określone jest formułą 4.10:

$$P_i = \frac{U_i}{\sum_{k \in M} U_k} \tag{4.10}$$

gdzie M jest zbiorem wszystkich dopuszczalnych ruchów w zadanym stanie gry.

Bouzy zaproponował, aby funkcję pilności uzależnić od heurystyk zbijania kamieni i unikania zbicia oraz od heurystyki „dobrych" wzorców. Heurystyki te zostały opisane w punkcie 4.3 „Wykorzystanie wiedzy w strategii przeglądania drzewa".

4.4.2. Algorytm symulacji zbliżonej do sekwencyjnej

W tym punkcie zostanie omówiony algorytm symulacji zbliżonej do sekwencyjnej (ang. sequence-like simulation) (Gelly, 2006). Algorytm łączy heurystykę bliskości z bazą „dobrych" wzorców. Kolejne ruchy wykonywane są w bezpośrednim sąsiedztwie poprzedniego ruchu (w odległości równej 1 w mierze Manhattan). Najlepszy ruch wybierany jest na podstawie bazy wzorców. Jeżeli nie można dopasować wzorca, to ruch wykonywany jest losowo. Poszukiwanie ruchów w tak bliskim sąsiedztwie powoduje, na planszy można zauważyć wyraźną granicę między terytoriami kontrolowanymi przez każdego z graczy.

Wprowadzenie różnorodności

Jedną z metod wprowadzającą różnorodność, w „prawie" sekwencyjnym algorytmie, jest wykonywanie ruchu w pustym fragmencie planszy. Powoduje to zmianę kierunku kontynuacji gry. Stosowana jest następująca procedura. Jeżeli symulacja nie może dopasować wzorca, to wyszukiwany jest węzeł planszy, którego wszystkie sąsiadujące z nim węzły są puste. Jeżeli po $\boldsymbol{n}$ próbach „samotny" węzeł nie może być znaleziony, to wykonywany jest losowy ruch [13].

5. Zrównoleglanie algorytmu MCTS

W tym rozdziale opisano metody zrównoleglania algorytmu MCTS. Zaletą algorytmu MCTS jest możliwość jego efektywnego zrównoleglenia w wielordzeniowych procesorach (CPU), klastrach oraz procesorach graficznym (GPU).

Postęp w budowie komputerów doprowadził do sytuacji, w której procesor w komputerze osobisty może mieć kilka rdzeni. Aby w pełni wykorzystać możliwości oferowane przez sprzęt, należy w miarę możliwości starać się zrównoleglić działanie algorytmu. Przetwarzanie współbieżne jest istotnym zagadnieniem nie tylko w odniesieniu do algorytmu MCTS, ale także każdej innej metody. W algorytmie MCTS, podobnie jak w alfa-beta, im więcej czasu jest na „myślenie" nad ruchem, tym większa jest siła gry programu. Zrównoleglenie MCTS wydaje się być obiecującą techniką zwiększającą siłę gry.

W pracy porównano ze sobą trzy metody metodę zrównoleglania: w liściu (równoległe wykonywanie wielu symulacji z jednego liścia drzewa MCTS), w wierzchołku (równoległe budowanie niezależnych drzew MCTS, a po upływie zadanego czasu wybór ruchu na podstawie gałęzi wychodzących z korzeni wszystkich drzew) i w drzewie (równoległe budowanie tego samego drzewa MCTS). Metody te zostały zaimplementowane i przetestowana na komputerze z wielordzeniowym procesorem.

W pracy rozpatrywany jest przypadek użycia wielordzeniowego procesora, w którym każdy z rdzeni przetwarza tylko jeden wątek. Każdy z wątków ma dostęp do współdzielonego obszaru pamięci. Czas dostępu do pamięci jest jednakowy dla wszystkich wątków. W sytuacji, w której dwa lub więcej wątków wykonuje operację na zasobach współdzielonych, ostateczny wynik tej operacji może być zależny od przeplotu zapisu i w konsekwencji być niejednoznaczny. Dlatego współbieżnie działające wątki, przy dostępie do zasobów współdzielonych, powinny korzystać z mechanizmu wzajemnego wykluczania *mutex* (ang. MUTual EXclusion). Przypadek próby jednoczesnego zapisu przez wiele wątków może wystąpić w krokach 1, 2 i 4 algorytmu MCTS. W kroku 3, podczas symulacji, każda rozgrywka może być wykonywana całkowicie niezależnie. Ta specyfika algorytmu MCTS jest szczególnie intersująca. Symulacje zajmują większość czasu każdej iteracji algorytmu i zrównoleglenie może mieć znaczący wpływ na poprawę wydajności programu.

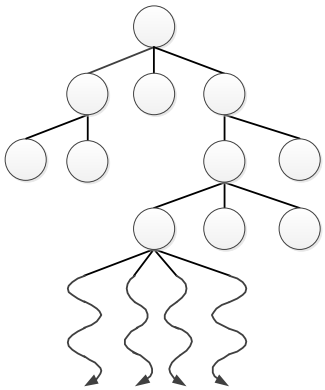
Obliczenia w procesorze graficznym wymagają rozwiązania kolejnych problemów. Przetwarzanie wielowątkowe w GPU z jednej strony zapewnia wielokrotnie większą moc obliczeniową w porównaniu do CPU i z drugiej, wprowadza ograniczenia na implementowane algorytmy oraz techniki programowania (brak rekurencji, uciążliwe zarządzenie pamięcią). Nie każdy algorytm działający współbieżnie w CPU można efektywnie zrównoleglić w GPU. Realizacja idei użycia GPU do obliczeń wcześniej wykonywanych przez CPU wymaga modyfikacji istniejących algorytmów lub opracowanie nowych. W punkcie 5.5 „Metody zrównoleglania w GPU" omówiono techniki dedykowane przetwarzaniu w GPU.

5.1. Zrównoleglanie w liściu

Technika zrównoleglania w liściu po raz pierwszy została zaproponowana przez Cazenave i Jouandeau [8]. W tej technice, drzewo jest przeglądane tylko przez jeden wątek, który po dojściu do liścia, dołącza do drzewa jeden lub więcej węzłów. Odpowiada to krokom 1 i 2 algorytmu. Następnie, rozpoczynając od liścia, każdy z dostępnych wątków wykonuje niezależną symulację (krok 3). Po zakończeniu wszystkich symulacji, rezultat jest wstecznie

propagowany tylko przez jeden wątek. Technika zrównoleglenie w liściu została ideowo przedstawiona na rysunku 5.1.

Technika zrównoleglenia w liściu jest prosta w implementacji i nie wymaga wielokrotnej implementacji mechanizmu wzajemnego wykluczania. Warto zwrócić uwagę na to, że średni czas działania wielu wątków jest dłuższy niż działania jednego, ponieważ wszystkie wątki muszą czekać na zakończenie tego, który działa najdłużej.

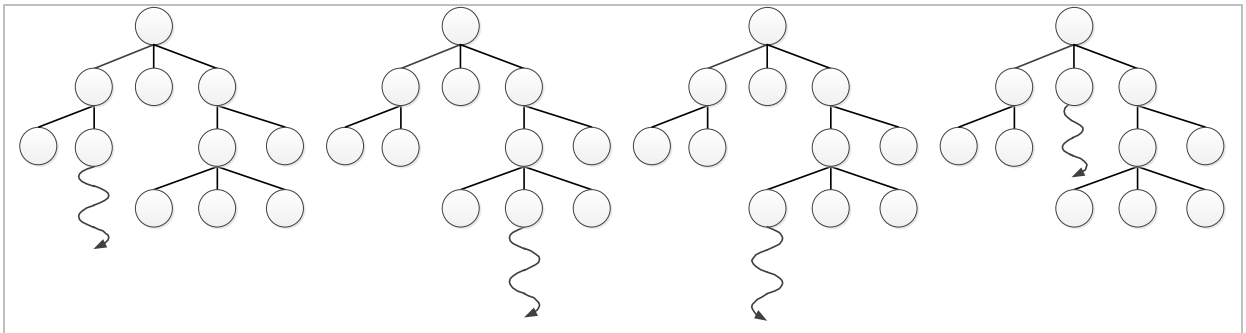


Rysunek 5.1. Technika zrównoleglania w liściu algorytmu MCTS.

Ograniczenie wysiłku obliczeniowego możliwe jest dzięki wykorzystaniu następującej obserwacji. Jeżeli wszystkie rozgrywki, które zakończą się wcześniej, zakończą się przegraną, to jest bardzo prawdopodobne, że podobny wynik zostanie uzyskany w pozostałych jeszcze niezakończonych grach. Można dojść do wniosku, że kontynuowanie pozostałych gier jest stratą czasu. W oparciu o powyższą obserwację warto część gier przerwać. Taka zmiana w algorytmie spowoduje częstsze przeglądanie drzewa oraz to, że nie wszystkie dostępne wątki będą wykorzystane do obliczeń.

5.2. Zrównoleglenie w wierzchołku

Technika zrównoleglania w wierzchołku działa w następujący sposób. Każdy z wątków używany jest do budowania niezależnego drzewa MCTS. Podobnie jak w przypadku zrównoleglania w liściu, wątki działają niezależnie i nie wymieniają ze sobą informacji. Po przekroczeniu limitu czasu przeznaczonego na ruch, węzły potomne wierzchołków ze wszystkich drzew są ze sobą scalane. Scalanie polega na zgrupowaniu wierzchołków odpowiadających temu samu stanowi gry i sumowaniu wyników uzyskanych w każdym z drzew. Po zsumowaniu wartości wyników gier i liczby odwiedzin ze wszystkich drzew, wybierany jest wierzchołek odpowiadający wykonywanemu ruchowi w rzeczywistej grze. Technika zrównoleglenie w wierzchołku została ideowo przedstawiona na rysunku 5.2.

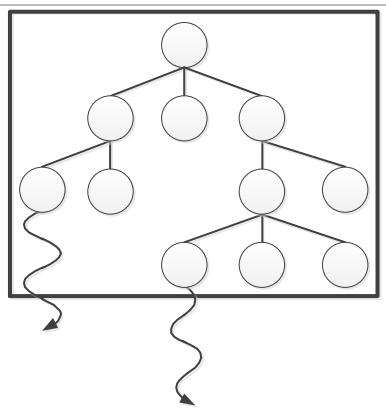


Rysunek 5.2. Technika zrównoleglania w wierzchołku algorytmu MCTS.

5.3. Zrównoleglenie w drzewie

Technika zrównoleglenia w drzewie polega na współdzieleniu drzewa przez kilka równocześnie prowadzonych gier. Ponieważ każdy z wątków może modyfikować informację zapisaną w drzewie, które jest zasobem współdzielonym, dlatego konieczne jest zastosowanie mechanizmu wzajemnego wykluczania. W technice zrównoleglenia w drzewie mogą być użyte dwa niezależne mechanizmy: **sekcji krytycznej** oraz **znacznika odwiedzin**.

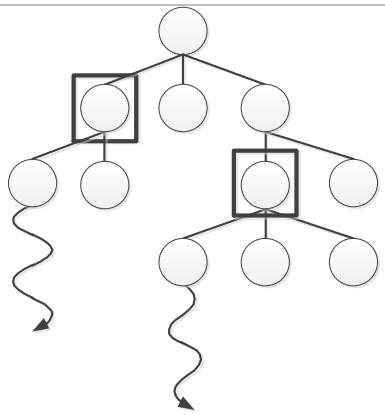
Do zrównoleglania w drzewie można zastosować jedną globalną albo wiele lokalnych sekcji krytycznych. Globalny *mutex* blokuje dostęp do całego drzewa w taki sposób, że w danej chwili tylko jeden wątek ma dostęp do drzewa (kroki 1, 2 i 4 algorytmu). W tym czasie pozostałe wątki mogą wykonywać symulacje (krok 3) rozpoczynając od różnych liści. Jest to podstawowa różnica w stosunku do zrównoleglenia w liściu, gdzie wszystkie symulacje zaczynają się od tego samego węzła. Technika zrównoleglenia w drzewie z użyciem jednej globalnej sekcji krytycznej została ideowo przedstawiona na rysunku 5.3.



Rysunek 5.3. Technika zrównoleglania z użyciem jednej sekcji krytycznej.

Potencjalna korzyść wynikająca ze zrównoleglania w drzewie jest ograniczona czasem, w którym wątki aktualizują dane we współdzielonym drzewie. Jeżeli *t* jest średnim czasem, w którym pojedynczy wątek zapisuje dane, to maksymalny zysk liczony w jednostkach liczby gier na sekundę wynosi 100/t. Jest to najpoważniejsza wada tej metody.

Drugie rozwiązanie, polegające na użyciu lokalnych sekcji krytycznych pozwala na współbieżny dostęp do drzewa wielu wątkom. Sekcjami krytycznymi obejmowane są pojedyncze węzły. Technika zrównoleglenia w drzewie z użyciem lokalnych sekcji krytycznych została ideowo przedstawiona na rysunku 5.4.



Rysunek 5.4. Technika zrównoleglania z użyciem lokalnych sekcji krytycznych.

5.4. Znacznik odwiedzin

Jeżeli kilka wątków rozpocznie przeglądanie drzewa od wierzchołka, to może się zdarzyć, że będą one przeglądać ten sam fragment drzewa. W rezultacie symulacje gier rozpoczną się od wierzchołów znajdujących się w bliskim sąsiedztwie. Ponieważ drzewo przeszukiwania zwykle składa się z milionów węzłów, wielokrotna eksploracja małego fragmentu jest całkowicie zbędna. Pomysłem na rozwiązanie tego problemu jest ustawienie specjalnego znacznika odwiedzin. Znacznik odwiedzin powoduje zmniejszenie wartości węzła przy każdych jego odwiedzinach. Kolejny wątek wybierze ten węzeł do badań tylko wtedy, jeżeli wartość węzła nadal będzie większa niż wartości w węzłach sąsiednich. Znacznik jest usuwany w czasie wykonywania czwartego kroku algorytmu MCTS – wstecznej propagacji, przez ten sam wątek, który spowodował jego ustawienie. Użycie znacznika odwiedzin powoduje, że węzły wyraźnie lepiej rokujące będą nadal odwiedzane przez pozostałe wątki, podczas gdy węzły o niskiej wartości zostaną odwiedzone tylko przez jeden wątek. Metoda ta realizuje zasadę równowagi pomiędzy eksploracją i eksploatacją w zrównoleglonym algorytmie MCTS.

5.5. Metody zrównoleglania w GPU

Klasyczne algorytmy przeszukiwania nie zawsze udaje się zrównoleglić w GPU z kilku powodów:

1. Problemem są obliczenia w architekturze SIMD[12] (ang. Single Instruction Multiple Data). Jest to związane z architekturą GPU, schematem programowania oraz ograniczeniami komunikacji pomiędzy wątkami.
2. Istotną cechą metod zrównoleglania jest ich skalowalność. Metody zrównoleglenia z użyciem GPU muszą być skuteczne w przypadku współbieżnego przetwarzania przez setki tysięcy wątków.
3. W przypadku algorytmu MCTS problemem jest generowanie w GPU liczb pseudolosowych (dostępne metody są zbyt czasochłonne).

Problem był analizowany przez Rockiego (2011), który zaproponował następujące rozwiązanie [9]. Ponieważ, jak stwierdził, algorytmu MCTS nie można zrównoleglić w wierzchołku (z powodu opisanego w punkcie 1), zbadał zrównoleglenie w liściu. Na podstawie wyników eksperymentów doszedł do wniosku, że możliwe jest zrównoleglenie w liściu, ale jedynie do około 1000 wątków na jeden GPU. Aby w pełni wykorzystać możliwości obliczeniowe oferowane przez GPU, dla przeszukiwania wielkich drzew, opracował nowy schemat zrównoleglenia − blokowy (ang. block-parallel). Zaletą schematu zrównoleglenia blokowego jest możliwość wykonywania obliczeń w GPU, w klastrach i superkomputerach z przesyłaniem komunikatów przez interfejs transmisji wiadomości MSI (ang. Message Passing Interface). W rozwiązaniu liczby pseudolosowe generowane są przez CPU i następnie transferowane do GPU w postaci tabel.

[12] SIMD (ang. Single Instruction, Multiple Data) – architektura komputerowa obejmujący systemy, w których przetwarzanych jest wiele strumieni danych w oparciu o pojedynczy strumień rozkazów. Architektura SIMD jest charakterystyczna dla komputerów wektorowych.

Wykonywanie obliczeń w GPU składa się trzech kroków:

1. Skopiowanie danych do pamięci GPU za pośrednictwem DMA.
2. Uruchomienie wielowątkowego przetwarzania danych w GPU.
3. Skopiowanie za pośrednictwem DMA wyników obliczeń z GPU.

Pojedyncze symulacje wykonywane są szybciej w procesorze CPU niż w procesorach graficznych GPU, ponieważ dane muszą być najpierw skopiowane do pamięci GPU. Siła przetwarzania w GPU polega na efekcie skali i pojawia się w chwili przetwarzania danych przez wiele wątków, co rekompensuje czas tracony na kopiowanie danych do i z pamięci GPU.

5.5.1. Zrównoleglenie w wierzchołku i drzewie

W metodzie zrównoleglenia w wierzchołku każdy z wątków przechowuje kopię drzewa zbudowanego we wcześniejszych iteracjach w krokach wyboru i rozwoju. Problemem jest kopiowanie dużej ilości danych do lokalnej pamięci każdego z wątków i dlatego metoda nie może być stosowana przy rozwiązywaniu złożonych problemów. Metoda zrównoleglenia w drzewie wymaga synchronizacji pomiędzy wątkami i z tego powodu jest trudna do efektywnej implementacji w GPU.

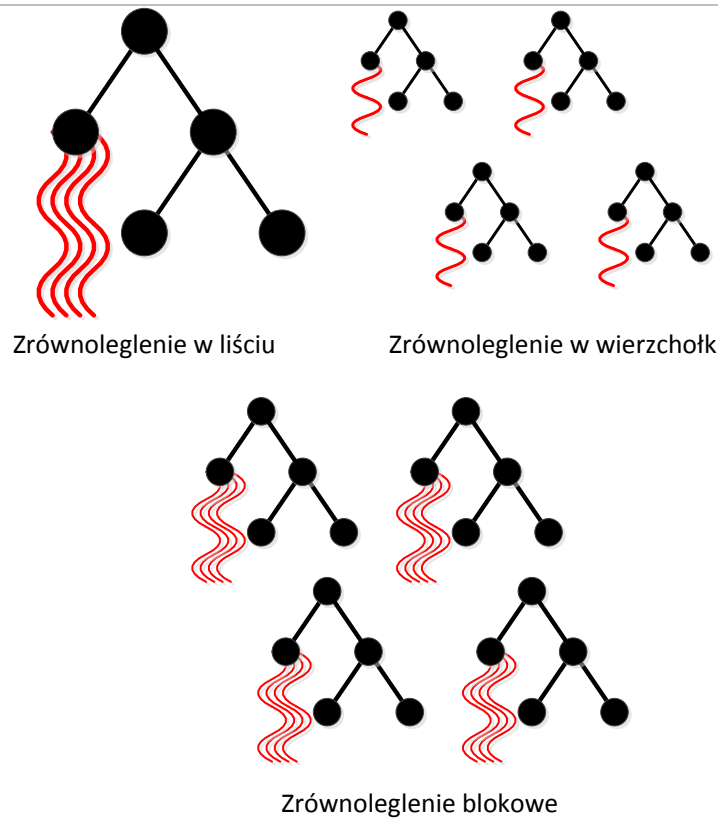
5.5.2. Zrównoleglenie w liściu

W metodzie zrównoleglenia w liściu do wykonywania obliczeń, może być użyty GPU. W kroku wyboru w algorytmie MCTS drzewo jest przeglądane przez jeden wątek działający w CPU. Następnie rozpoczynając od liścia, wykonywane są równoległe symulacje działające w GPU. Każdy z wątków wykonuje pojedynczą symulację. Wszystkie symulacje rozpoczynają się od tego samego węzła. W kroku wstecznej propagacji, po zsumowaniu wyników gier, jeden wątek działający w CPU, aktualizuje wartości w węzłach drzewa. Z czterech kroków algorytmu MCTS trzy wykonywane są w CPU i jeden – symulacja – w GPU. Metoda nie wymaga synchronizacji wątków.

5.5.3. Zrównoleglanie blokowe

Metoda łączy zrównoleglanie w wierzchołku (najbardziej efektywną metodę w CPU) ze zrównoleglaniem w liściu. Metoda jest bardziej efektywna niż samo zrównoleglenie w liściu, ponieważ użycie zrównoleglenia w wierzchołku zmniejsza prawdopodobieństwo znalezienia ekstremum lokalnego (może się to zdarzyć w zrównolegleniu w liściu). Testy wykazały, że metoda jest czterokrotnie bardziej efektywna w odniesieniu do liczby wątków w CPU koniecznych do uzyskania porównywalnych rezultatów [9].

W metodzie zrównoleglenia blokowego kroki wyboru, rozwoju i wstecznej propagacji wykonywane są przez CPU, podobnie jak w metodzie zrównoleglenia w liściu, z tą różnicą że jednocześnie przetwarzanych jest wiele drzew, każde przez osobny wątek CPU. Krok symulacji realizowany jest przez grupy wątków w GPU. Liczba wątków w każdej grupie jest stała. W architekturze GPU, 32 wątki łączone są w tzw. *warpy*, które z kolei formują grupy wątków. Każdy z *warpów* wykonywany jest niezależnie. Liczba symulacji zależy od liczby wątków. Przykładowo, jeżeli zostaną zarezerwowane 4 bloki, każdy składający się z 256 wątków, to w każdej iteracji algorytmu będą wykonywane 1024 symulacje. Techniki zrównoleglenia: w liściu, w wierzchołku i blokowa zostały ideowo przedstawione na rysunku 5.5.



Rysunek 5.5. Techniki zrównoleglenia: w liściu, w wierzchołku i blokowa (GPU).

5.6. Podsumowanie

W tym rozdziale zostały omówione metody zrównoleglania algorytmu MCTS w liściu, wierzchołku i drzewie. Wydajność metody zrównoleglenia na poziomie drzewa można zwiększyć przez zastosowanie dwóch technik: (1) polegającej na zastosowaniu lokalnych sekcji krytycznych zamiast jednej globalnej oraz (2) stosującą wskaźnik odwiedzin zwiększający liczbę gier na sekundę i wzmacniającą siłę gry.

Opisano algorytm zrównoleglania blokowego dedykowany do przetwarzania w GPU. Metoda łączy zrównoleglanie w wierzchołku (najbardziej efektywną metodę w CPU) ze zrównoleglaniem w liściu. Obliczenia w krokach wyboru, rozwoju i wstecznej propagacji zrównoleglone są w wierzchołku i wykonywane w CPU, krok symulacji − w GPU.

6. Program komputerowy do gry w Go

Program składa się z dwóch niezależnych modułów: modułu silnika gry oraz modułu interfejsu. Wydzielenie silnika gry ułatwia przenośność aplikacji i pozwala na uniwersalność zastosowań. W punkcie 6.1 opisano konstrukcję planszy do gry oraz metody weryfikacji poprawności ruchu, zapisywanie informacji o łańcuchach, oddechach, punkcie *ko* itd. W punkcie 6.2 opisano implementację rozgrywek Monte-Carlo. W punkcie 6.3 omówione są techniki oceny wyniku gry.

Mając na uwadze: przenośność („dostępność wszędzie i na każdym sprzęcie"), zwięzłość, szybkość wykonywania programu, wydajność zasobową, łatwość konserwacji − do zakodowania programu wybrano podejście obiektowe i język C++. Program został zbudowany w środowisku programistycznym Visual Studio 2010. W projekcie zostały wykorzystane biblioteki STL, BOOST[13][18] oraz biblioteka graficzna Qt[14][19] do budowy graficznego interfejsu użytkownika.

6.1. Algorytmy i struktury danych w polityce drzewa

Algorytm MCTS można podzielić na dwie polityki: drzewa i symulacji. W polityce drzewa wymagana jest szczegółowa analiza stanu gry, łańcuchów, terytoriów itd. W polityce symulacji najważniejsza jest szybkość działania (praktycznie jedynym wymaganiem jest zgodność ruchu z regułami gry). Obie fazy algorytmu wymagają odmiennych struktur danych i algorytmów.

6.1.1. Struktury danych dla planszy

Przy wyborze struktury danych trzeba z jednej strony wziąć pod uwagę szybkość rejestracji wykonanego ruchu i z drugiej szybkość analizy rozmieszczenia kamieni (łańcuchy, oddechy). Pomimo że naturalnym sposobem implementacji planszy jest tablica dwuwymiarowa zdecydowano się na implementację struktury bazującej na tablicy jednowymiarowej. Do wyznaczenia punku wymagana jest tylko jedna zmienna i liczba parametrów przekazywanych do funkcji będzie mniejsza. Tablica $Board$ ma rozmiar $(s+2) \times (s+1) + 1$, gdzie s jest liczbą punktów w jednej linii planszy. Każdy punkt może mieć jeden z czterech kolorów: czarny, biały, neutralny i poza planszą (BLACK, WHITE, EMPTY, BORDER).

Na rysunku 6.1 pokazano indeksowanie tablicy dla planszy składającej się 5×5 punktów. Tablica jest jednowymiarowa. Narysowanie tablicy w dwóch wymiarach ma jedynie na celu ułatwienie percepcji planszy do gry. Kropkami oznaczono punkty na planszy (kolor EMPTY), znakami # punkty poza planszą (kolor BORDER). Sprawdzenie czy komórka o indeksie idx odpowiada punktowi na planszy wykonywane jest za pomocą formuły 6.1:

$$Board[idx] \; ! = \text{BORDER} \tag{6.1}$$

[13] Biblioteka BOOST jest zbiorem bibliotek poszerzających możliwości języka C++, objętych licencją, która umożliwia użycie ich w dowolnym projekcie.

[14] Biblioteka Qt jest zestawem przenośnych bibliotek i narzędzi programistycznych dedykowanych dla języków C++, QML i Java. Ich podstawowym składnikiem są klasy służące do budowy graficznego interfejsu użytkownika. Aktualnie biblioteka Qt rozwijana jest w dwóch niezależnych projektach: komercyjnym i Open Source. W rozwiązaniu użyto wersji Open Source.

Indeksy sąsiadujących punktów z punktem p mającym indeks idx w tablicy $Board$ obliczane są według formuł 6.2:

$$\begin{aligned} N(idx) &= idx + (s+1) \\ E(idx) &= idx - 1 \\ S(idx) &= idx - (s+1) \\ W(idx) &= idx + 1 \end{aligned} \tag{6.2}$$

gdzie s jest liczbą punktów w jednej linii planszy, idx jest indeksem punktu p, N punktem sąsiadującym „od północy”, E − „wschodu”, S − „południa” i W − „zachodu”. Jeżeli punkt wyznaczony na podstawie formuł 6.2 jest koloru BORDER to znaczy, że punkt p nie ma na planszy punktu sąsiadującego w danym kierunku (na przykład punkt p znajduje się na granicy planszy). Łatwo jest zdefiniować formuły (6.3) dla obliczenia indeksów punktów sąsiadujących w kierunkach: NE, NW, SE, SW:

$$\begin{aligned} \mathrm{NE(idx)} &= \mathrm{idx} + (\mathrm{s}+1) + 1 = \mathrm{idx} + \mathrm{s} + 2 \\ \mathrm{SE(idx)} = \mathrm{E(S(idx))} &= \mathrm{idx} - (\mathrm{s}+1) + 1 = \mathrm{idx} - \mathrm{s} \\ SW(idx) &= W\big(S(idx)\big) = idx - (s+1) - 1 = idx - s - 2 \\ NW(idx) &= idx + (s+1) - 1 = idx + s \end{aligned} \tag{6.3}$$

Zdefiniowanie w strukturze planszy „zagadkowej” komórki z indeksem 42 można „wyjaśnić” wyznaczeniem sąsiadującej komórki z punktem o indeksie 35 w kierunku NE: $NE(35) = 35 + 5 + 2 = 42$. Komórka o indeksie 42 jest poza planszą (kolor BORDER), ale nie wykracza poza indeks tablicy $Board$.

36	37	38	39	40	41	42
30	31	32	33	34	35	
24	25	26	27	28	29	
18	19	20	21	22	23	
12	13	14	15	16	17	
6	7	8	9	10	11	
0	1	2	3	4	5	

#	#	#	#	#	#	#
#	·	·	·	·	·	
#	·	·	·	·	·	
#	·	·	·	·	·	
#	·	·	·	·	·	
#	·	·	·	·	·	
#	#	#	#	#	#	

Rysunek 6.1. Struktura danych dla planszy gry.
Lewa strona. Indeksowanie tablicy jednowymiarowej − $Board$.
Prawa strona. Kropkami oznaczono punkty na planszy, znakami # punkty poza planszą.

Konwersja pomiędzy współrzędnymi punktu i jego indeksem wykonywana za pomocą formuły 6.4:

$$idx(row, col) = row * (s+1) + col \tag{6.4}$$

gdzie s jest liczbą punktów w jednej linii planszy, idx jest indeksem punktu w tablicy $Board$ oraz row, col – odpowiednio wierszem, kolumną we współrzędnych dwuwymiarowych. W celu przyspieszenia konwersji pomiędzy indeksem punktu i współrzędnymi zostały użyte dwie pomocnicze tablice przechowujące wstępnie obliczone wartości współrzędnych dla każdego z punktów.

Indeks pierwszego i ostatniego punktu na planszy jest wyznaczany za pomocą formuł 6.5:

$$\begin{aligned} idxFirst &= s + 2 \\ idxLast &= (s+1) * (s+1) - 1 \end{aligned} \tag{6.5}$$

Pseudokod iteracji przez wszystkie punkty na planszy został pokazany w algorytmie 6.1.

Algorytm 6.1. Iteracja przez wszystkie punkty na planszy.

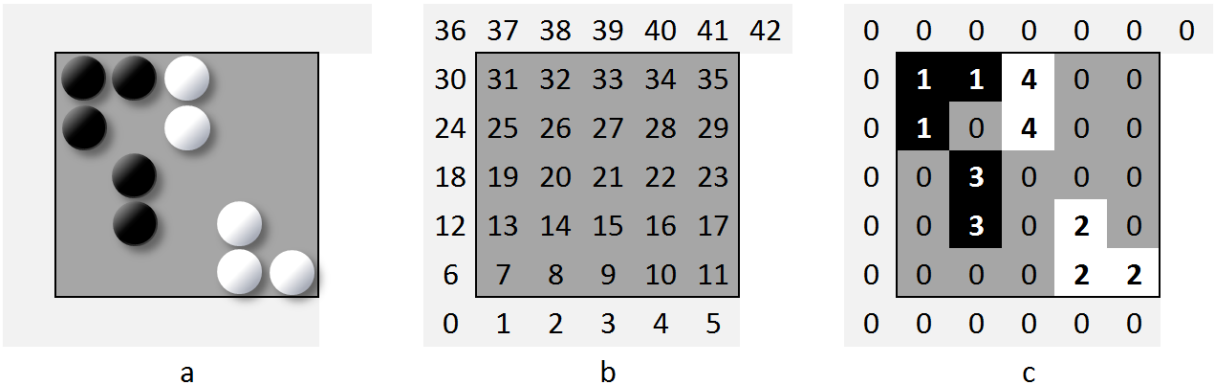
```
idx ← idxFirst
WHILE idx < idxLast DO
   IF Board[idx] != BORDER THEN
      // wykonaj operację na punkcie mającym indeks idx w tablicy Board
   END
   idx ← idx + 1
END
```

W strukturze planszy pamiętana jest też informacja o łańcuchach. Informacja zapisana jest w tablicy jednowymiarowej typu całkowitoliczbowego – $ChainStones$ – mającej rozmiar równy tablicy planszy $(s+2) \times (s+1) + 1$. Tablica zawiera odwzorowania wszystkich zajętych przez kamienie punktów p, na indeks łańcucha w tablicy łańcuchów $ListOfChains$. Puste punkty oraz punkty graniczne mają indeks równy 0.

Każdy łańcuch jest opisany przez strukturę danych $Chain$. Struktury $Chain$ zapisane są w tablicy $ListOfChains$.

Przykładowo, jeżeli punkty p_1, p_2 należą do łańcucha o indeksie c_1 w tablicy $ListOfChains$, to w tablicy $ChainStones$, w komórkach o indeksach p_1, p_2 będzie zapisany indeks łańcucha c_1. Komórka $ListOfChains[c1]$ będzie zawierała strukturę łańcucha, do którego należą punkty p_1, p_2.

Na rysunku 6.2 pokazano przykładowy stan gry, indeksowanie tablicy łańcuchów $ChainStones$ dla planszy składającej się 5×5 punktów oraz wartości komórek tablicy dla czterech łańcuchów.



Rysunek 6.2. Struktura danych dla łańcuchów w polityce drzewa.
a. Stan gry.
b. Indeksowanie jednowymiarowej tablicy łańcuchów – $ChainStones$.
c. Wartości w komórkach tablicy łańcuchów – $ChainStones$.

Dodatkowo w strukturze planszy pamiętana jest liczba zbitych kamieni przez gracza i oponenta, punkt *ko* oraz lista wykonanych ruchów w grze.

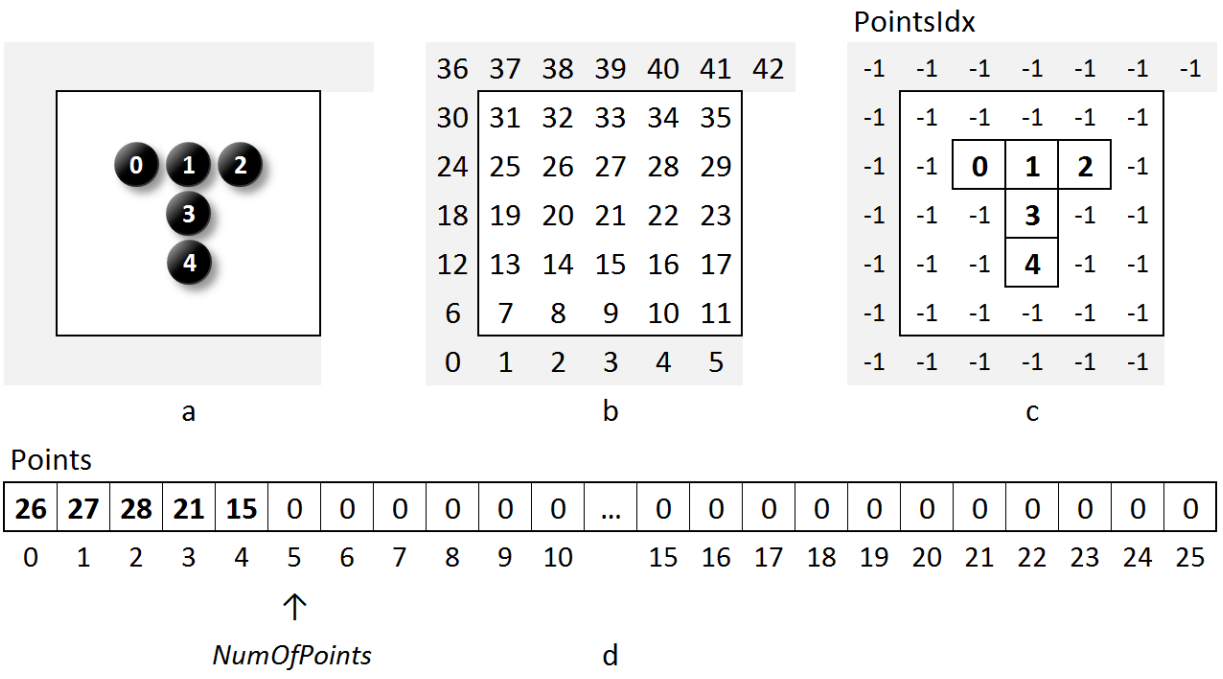
6.1.2. Struktury danych dla łańcuchów

Łańcuch jest grupą sąsiadujących kamieni w tym samym kolorze. Każdy kamień należy do łańcucha (pojedynczy kamień na planszy tworzy jednoelementowy łańcuch). Analiza stanu gry przeprowadzana jest na podstawie analizy łańcuchów. Kamienie w łańcuchu współdzielą *oddechy*. Zbijanie kamieni oraz stan *atari* zależy od liczby *oddechów* łańcucha. Tablica łańcuchów musi być aktualizowana po wykonaniu każdego ruchu.

Łańcuch opisany jest strukturą $Chain$ zawierającą informację o punktach (kamieniach), z których składa się łańcuch oraz o *oddechach*. Zarówno punkty zajęte przez kamienie jak i *oddechy* opisane są za pomocą trzech składowych: (1) tablicy punktów $Points$ o rozmiarze $s \times s$; (2) tablicy indeksów punktów $PointsIdx$ we współrzędnych planszy o rozmiarze $(s+2) \times (s+1) + 1$ oraz (3) licznika punktów $PointsCounter$.

Korzystając z tych struktur danych zostały zaimplementowane operacje na łańcuchach: (1) dodania kamienia do łańcucha; (2) usunięcia kamienia z łańcucha oraz (3) weryfikacji czy dany kamień należy do łańcucha. Analogiczne operacje zostały zdefiniowane dla *oddechów*.

Operacja dodania punktu do łańcucha (kamienia lub *oddechu*) składa się z trzech elementarnych czynności: (1) zapisania punktu na końcu tablicy punktów $Points$; (2) aktualizacji indeksu dla dodanego punktu w tablicy indeksów $PointsIdx$ oraz (3) inkrementacji licznika punktów $PointsCounter$. Na rysunku 6.3 pokazano stan tablic po utworzeniu łańcucha składającego się z pięciu czarnych kamieni.



Rysunek 6.3. Przykład tworzenie łańcucha.
a. Stan gry. Łańcuch pięciu czarnych kamieni został utworzony przez kolejno kładzione kamienie: 0, 1, 2, 3, 4.
b. Indeksowanie jednowymiarowej tablicy planszy do gry.
c. Wartości w komórkach tablicy $PointsIdx$.
d. Wartości w tablicy $Points$ oraz bieżące położenie wskaźnika $NumOfPoints$.

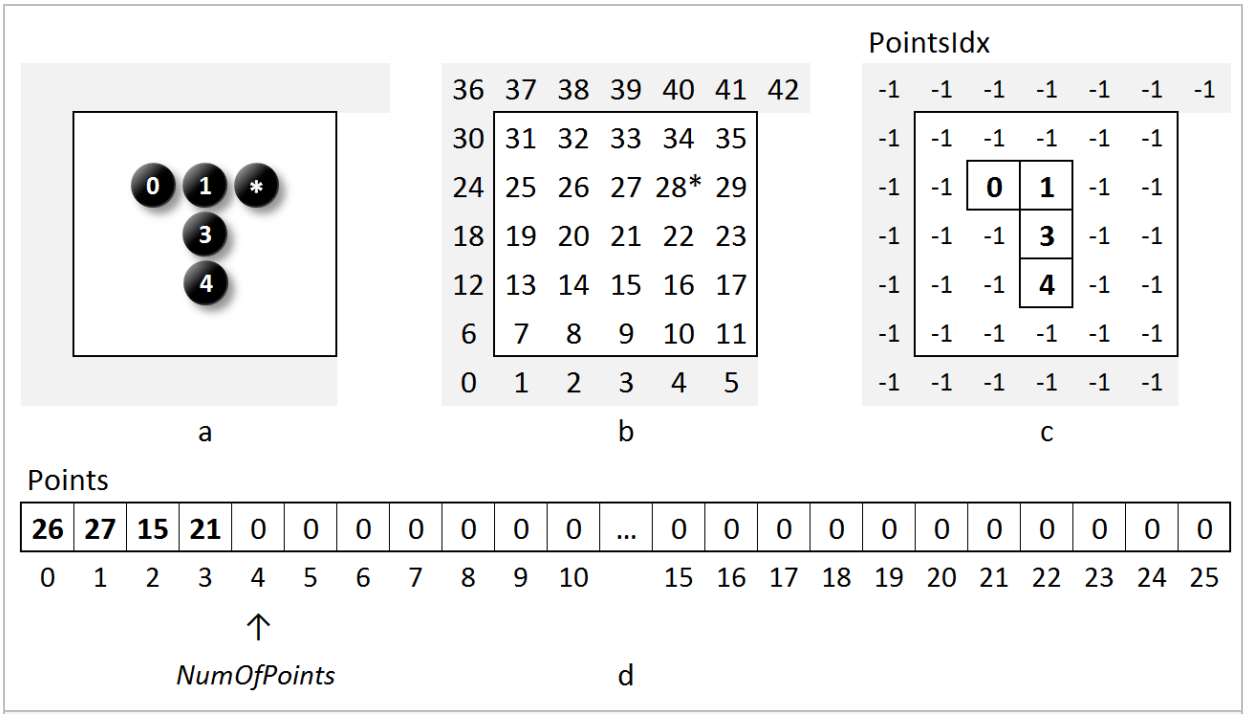
Operacja usunięcia kamienia z łańcucha została zilustrowana na rysunku 6.4. Z łańcucha pięciu czarnych kamieni jest usuwany kamień oznaczony (*).

Weryfikacja czy punkt należy do łańcucha polega na sprawdzaniu czy indeks punktu w tablicy indeksów $PointsIdx$ jest różny od -1.

Pseudokod operacji na łańcuchu, dodania i usunięcia punktu, został przedstawiony w algorytmie 6.3.

Algorytm 6.2. Pseudokod operacji dodawania i usuwania punktów w łańcuchu.

```
DANE WEJŚCIOWE: Punkt Point [dodawany do | usuwany z] łańcucha
DANE WYJŚCIOWE: N/A

PROCEDURE AddPoint(Point)
   // Zapisanie punktu na końcu tablicy punktów Points
   Points[NumOfPoints] ← Point
   // Aktualizacja indeksu dla dodanego punktu w tablicy indeksów PointsIdx
   PointsIdx[Point] ← NumOfPoints
   // Inkrementacja licznika punktów PointsCounter
   NumOfPoints ← NumOfPoints + 1
END PROCEDURE

PROCEDURE RemovePoint(Point)
   // Zamiana miejscami punktu Point z ostatnim punktem w łańcuchu LastPoint
   Idx ← PointsIdx[Point]
   LastPoint ← Points[NumOfPoints-1]
   Points[Idx] ← LastPoint
   PointsIdx[LastPoint] ← Idx

   // Usunięcie punktu Point z tablic Points i PointsIdx
   Points[NumOfPoints-1] ← 0
   PointsIdx[Point] ← -1
   // Dekrementacja licznika punktów PointsCounter
   NumOfPoints ← NumOfPoints - 1
END PROCEDURE

FUNCTION IsInChain(Point)
   // Czy indeks punktu w tablicy indeksów PointsIdx jest różny od −1
   RETURN (PointIdx[Point] != -1)
END FUNCTION
```



Rysunek 6.4. Usunięcie kamienia z łańcucha.

a. Stan gry. Łańcuch z pięciu czarnych kamieni. Kamień (*) jest usuwany.
b. Indeksowanie jednowymiarowej tablicy planszy do gry.
c. Wartości w komórkach tablicy $PointsIdx$ po usunięciu kamienia (*).
d. Wartości w tablicy $Points$ oraz bieżące położenie wskaźnika $NumOfPoints$ po usunięciu kamienia (*).

6.1.3. Kryteria weryfikacji poprawności ruchu

W programie zostały użyte następujące kryteria poprawności ruchu. Ruch jest poprawny, jeżeli spełnione są wszystkie trzy warunki[15]:

1. Kamień został położony w pustym punkcie na planszy.
2. Ruch nie jest rozgrywany w punkcie *ko*.
3. Ruch nie jest *samobójstwem*, innymi słowy ruch nie powoduje zbicia swojego kamienia.

Warunki 1 i 2 wymają jedynie sprawdzenia z planszą i punktem *ko*. Do analizy warunku 3 wymagana jest informacja o łańcuchach.

Ruch nie jest *samobójstwem*, jeśli co najmniej jeden z poniższych warunków jest spełniony:

1. Ruch sąsiaduje z pustym punktem.
2. Ruch jest w sąsiedztwie łańcucha posiadanego przez tego samego gracza i łańcuch ma, co najmniej dwa sąsiadujące punkty swobody (położony kamień zajmie jeden z punktów swobody).
3. Ruch jest w sąsiedztwie łańcucha posiadanego przez innego gracza, który ma dokładnie jeden punkt swobody (położenie kamienia spowoduje zbicie łańcucha przeciwnika).

6.1.4. Wykonywanie ruchów

Wykonanie ruchu jest procedurą składającą się z szeregu elementarnych operacji: łączenie sąsiednich łańcuchów, usuwanie zbitych kamieni, aktualizowanie liczników *oddechów*, aktualizacja punktu *ko* itd.

Pierwszą operacją w procedurze wykonywania ruchu jest utworzenie nowego łańcucha. Następnie należy sprawdzić sąsiednie punkty, ponieważ ich stan determinuje kolejne operacje.

Jeżeli sąsiedni punkt jest **pusty**, to zostanie dodany do tablicy *oddechów* łańcucha.

Jeżeli sąsiedni punkt jest **zajęty przez kamień w tym samym kolorze**, to oba łańcuchy zostaną ze sobą połączone. Możemy połączyć dwa łańcuchy, dodając wszystkie punkty i *oddechy* drugiego łańcucha do pierwszego łańcucha. Po połączniu łańcuchów należy iteracyjnie zweryfikować raz jeszcze wszystkie kamienie i *oddechy*.

Jeżeli sąsiedni punkt jest **zajęty przez kamień przeciwnika**, to należy policzyć liczbę oddechów łańcucha, do którego należy kamień przeciwnika. Jeżeli liczba oddechów łańcucha przeciwnika jest większa niż 1, to należy dekrementować wartość licznika oddechów.

Jeżeli **liczba oddechów jest równa 1**, łańcuch przeciwnika zostaje zbity i jest usuwany z planszy. Usunięcie łańcucha powoduje, że inne sąsiadujące z nim łańcuchy uzyskują nowe *oddechy* i należy zaktualizować liczniki *oddechów* dla tych łańcuchów.

Jeżeli zbity **łańcuch składał się tylko z 1 kamienia** to punkt, w którym został zbity kamień jest punktem *ko*. W przeciwnym przypadku, należy usunąć punkt *ko*, (jeżeli był ustawiony).

Pseudokod procesu wykonywania ruchu został przedstawiony w algorytmie 6.3.

---

[15] Warunek *superko* nie jest sprawdzany.

Algorytm 6.3. Pseudokod procedury wykonywania ruchu.

```
DANE WEJŚCIOWE: Punkt Point, gracz Player.
DANE WYJŚCIOWE: N/A

PROCEDURE MOVE(Point, Player)

   Chain ← CreateChain(Point)      // Utwórz łańcuch Chain z punktem Point
   CapturedPoints ← CreateListOfPoints()   // Inicjalizuj listę zbitych kamieni

   // Dla każdego sąsiada Neighbour punktu Point
   FOR EACH Neighbour IN Point.GetNeighbours()
      // Jeżeli sąsiadujący punkt jest pusty
      IF Neighbour.Color = EMPTY THEN
         // sąsiadujący punkt jest pusty
         Chain.AddLiberty(Neighbour);

      // Jeżeli sąsiadujący kamień jest w kolorze gracza
      ELSE IF Neighbour.Color = Player.Color THEN
         // połącz łańcuchy
         Chain ← Chain.MergeChain(Neighbour.GetChain())
         // aktualizuj oddechy w sąsiadujących łańcuchach
         Chain.UpdateLibertyForChains()

      // Jeżeli sąsiadujący kamień jest w kolorze oponenta
      ELSE IF Neighbour.Color != Player.Color THEN
         // Znajdź łańcuch, do którego należy sąsiadujący kamień
         NeighbourChain ← Neighbour.GetChain()

         // Jeżeli liczba oddechów sąsiadującego łańcucha jest równa 1
         IF NeighbourChain.CountLiberty() = 1 THEN

            // Usuń łańcuch z planszy
            Board.RemoveFromBoard(NeighbourChain)
            // Aktualizuj licznik zbitych kamieni
            Board.Player.UpdatePrisoners(NeighbourChain)
            // Do listy zbitych kamieni dodaj kamieni zbitego łańcucha
            Board.Player.Captures.Add(NeighbourChain.Points())

            // Dla każdego łańcucha sąsiadującego z łańcuchem NeighbourChain
            FOR EACH Chain IN NeighbourChain.GetNeighbours()
               Chain.UpdateLiberty()
            END FOR

         // Jeżeli liczba oddechów sąsiadującego łańcucha jest większa od 1
         ELSE NeighbourChain.CountLiberty() > 1 THEN
            // Aktualizuj oddechy dla sąsiedniego łańcucha
            NeighbourChain.RemoveLiberty(Point)
         END IF

      // Jeżeli sąsiadujący punkt jest granicą planszy
      ELSE Neighbour.Color = BORDER THEN
         // nic nie rób
      END IF
   END FOR

   // Jeżeli liczba zbitych kamieni jest równa 1
   IF CapturedPoints.CountPoint() = 1 THEN
      // Zbity kamień przypisz do punktu ko
      PointKo ← CapturedPoints(0)
   ELSE
      // “Zeruj” punkt ko
      PointKo ← Null
   END IF
END PROCEDURE
```

6.1.5. Historia i anulowanie ruchów

W fazie wyboru (polityka drzewa) często są analizowane ruchy, które w rzeczywistej rozgrywce nigdy nie są wykonywane. Dlatego w programie należy zaimplementować funkcjonalność anulowania ruchów, innymi słowy cofanie wszystkich wprowadzonych zmian w stanie gry. Prostym rozwiązaniem jest zapamiętanie stanu gry (zapisanie migawki) przed wykonaniem ruchu i następnie odtworzenie stanu sprzed ruchu na podstawie zapamiętanej migawki. Rozwiązanie to ma poważną wadę − zajmuje dużo pamięci i czasu procesora.

Innym podejściem jest modyfikacji procedury wykonania ruchu − $Move$ (opisanej w algorytmie 6.3). W tym celu utworzone są cztery struktury danych: (1) gracza, który wykonał ruch – $Player$; (2) punktu, w którym kładziony jest kamień – $Point$; (3) punkt *ko* przed wykonaniem ruchu – $PointKo$ oraz (4) czteroelementowa tablica wartości logicznych (dla czterech sąsiadujących kamieni) – $CapturedNeighbours$ − przyjmująca wartość $true$, jeżeli ruch spowodował zbicie sąsiadującego kamienia i $false$ w przeciwnym przypadku. Przed wykonaniem ruchu, na stosie historii ruchów, odkładana jest struktura zbiorcza zawierająca opisane cztery struktury. Wartości w tablicy $CapturedNeighbours$ wypełniane są po wykonaniu ruchu i analizie zbitych kamieni.

Funkcja $CancelMove$ (odwrotna do funkcji $Move$) zdejmuje ze stosu strukturę odpowiadającą ostatnio wykonanemu ruchowi. Kolor punktu, w którym został położony kamień ustawiany jest na neutralny i chwilowo zerowany jest punkt *ko*. Następnie analizowany jest każdy sąsiedni punkt. Jeżeli sąsiadujący punkt jest częścią łańcucha w kolorze oponenta to punkt, dla którego anulowany jest ruch musi zostać dodany do *oddechów* łańcucha. Jeżeli sąsiadujący punkt jest częścią łańcucha w kolorze gracza, to kolejno przeszukiwane są sąsiednie punkty w celu rekonstrukcji łańcuchów. Rekonstrukcja jest konieczna, ponieważ sąsiednie łańcuchy mogły zostać połączone przez położony kamień i operacja wycofania ruchu musi dokonać ich ponownego podziału. W kolejnym kroku anulowane jest zbicie kamieni. Puste punkty, w każdym z kierunków wyznaczonych na podstawie wartości z tablicy $CapturedNeighbours$, wypełniane są kamieniami w kolorze oponenta. Wypełnianie kolorem pustych punktów wykonywane jest z wykorzystaniem algorytmu rozrostu ziarna[16]. Ostatnią czynnością jest odtworzenie wartości punktu *ko*.

6.2. Algorytmy i struktury danych w polityce symulacji

W fazie symulacji najważniejszym parametrem jest wydajność. Od liczby symulowanych pojedynków, rozegranych w czasie przeznaczonym na „myślenie”, zależy siła gry programu. Struktury danych i algorytmy użyte w polityce drzewa umożliwiały szybką analizę łańcuchów, ale za cenę wolniejszych rozgrywek. W polityce symulacji musi być użyta struktura, która pozwoli na skrócenie czasu aktualizacji danych w strukturach łańcuchów.

[16] Algorytm rozrostu ziarna (ang. flood fill) został przedstawiony w postaci pseudokodu 6.6 na stronie 54.

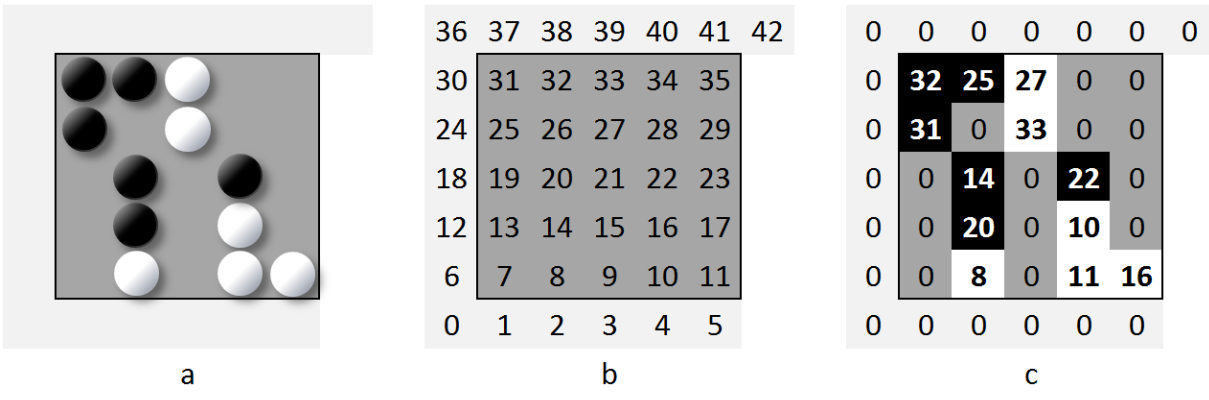
6.2.1. Tablica łańcuchów

Tablice punktów $Points$ oraz indeksów $PointsIdx$ użyte w polityce drzewa zostają zastąpione jedną tablicą kamieni, które tworzą łańcuch – $Stones$. Rozmiar tablicy $Stones$ jest równy jest $(s+2) \times (s+1) + 1$. Jeżeli punkt p_0 jest pusty, innymi słowy jego kolor jest neutralny, to $Stones[p_o] = 0$. Jeżeli w punkcie p_0 jest kamień, to punkt p_0 ma kolor kamienia i $Stones[p_o] = p_1$, gdzie p_1 jest kolejnym punktem w łańcuchu. Jeżeli łańcuch zawiera tylko jeden punkt p_0, to $Stones[p_o] = p_0$. Tablica $Stones$ jest aktualizowana w taki sposób, aby sekwencja

$$p_o,\ Stones[p_o],\ Stones\big[Stones[p_o]\big],\ \ldots$$

iterowała przez wszystkie kamienie w łańcuchu. Innymi słowy tablica $Stones$ jest listą cykliczną kamieni należących do tego samego łańcucha. Na rysunku 6.5 pokazano przykładowy stan gry, indeksowanie tablicy łańcuchów $Stones$ dla planszy składającej się 5×5 punktów oraz wartości komórek tablicy dla sześciu łańcuchów.

W celu szybkiego obliczenia liczbę *oddechów*, w taki sposób aby każdy oddech był liczony tylko jeden raz, została zdefiniowana druga tablica $Liberties$ o rozmiarze$(s+2) \times (s+1) + 1$, w której dla każdego punktu na planszy przechowywany jest znacznik informujący czy *oddechy* były już obliczone. Początkowo wartość znaczników dla wszystkich punktów ustawiana jest na wartość 0 oraz indeks wskazujący na bieżący punkt $LibertiesIdx$ jest ustawiany na wartość 0. Za każdym razem, kiedy obliczana jest liczba oddechów dla łańcucha, indeks $LibertiesIdx$ jest inkrementowany. Jeżeli w tablicy $Liberties$ zostanie znaleziony niepoliczony jeszcze *oddech*, to licznik oddechów jest inkrementowany i znacznik jest ustawiany na bieżącą wartość $LibertiesIdx$.



Rysunek 6.5. Struktura danych dla łańcuchów w polityce symulacji.
a. Stan gry.
b. Indeksowanie jednowymiarowej tablicy łańcuchów – $Stones$.
c. Wartości w komórkach tablicy łańcuchów – $Stones$.

Algorytm wykonania ruchu wymaga jedynie informacji czy liczba oddechów jest równa 0, 1, większa lub równa 2. Dlatego jeżeli licznik oddechów osiąga wartość równą 2, to dalsze liczenie oddechów jest przerywane i procedura zwraca bieżącą wartość licznika oddechów.

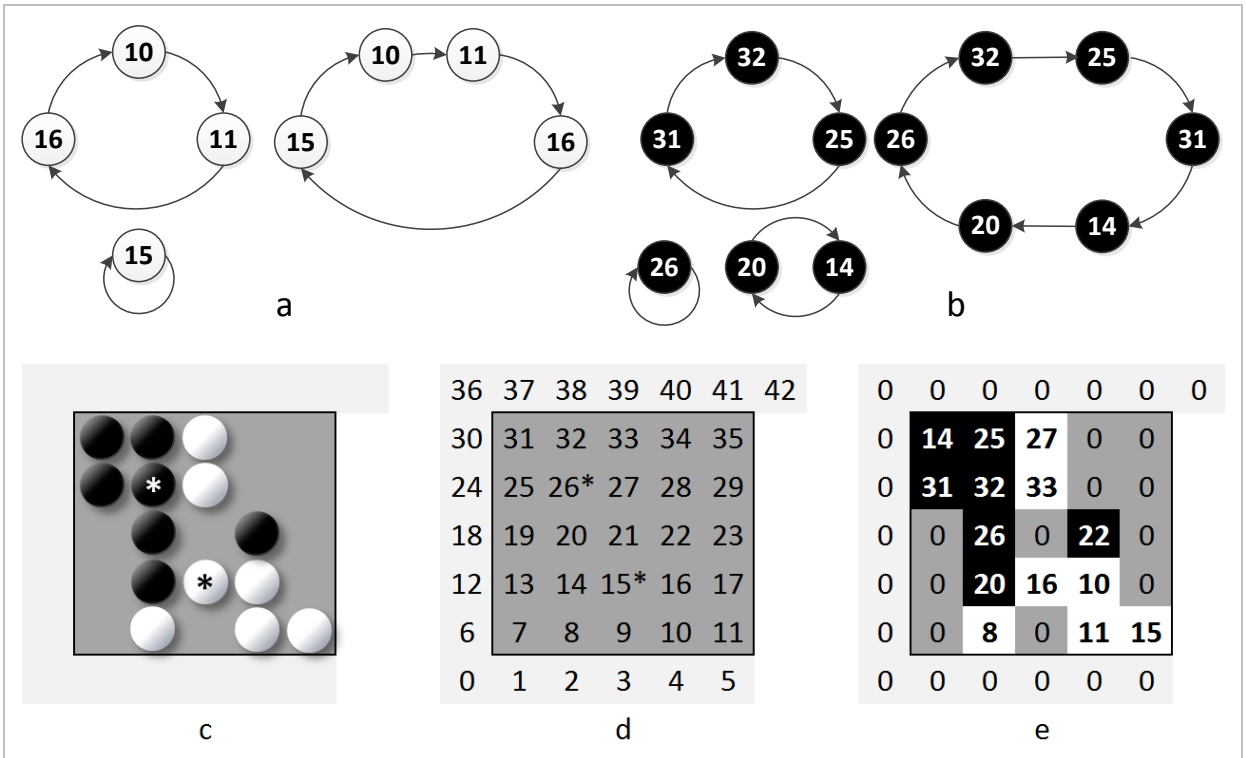
Pseudokod wyznaczania liczby *oddechów* został przedstawiony w algorytmie 6.4.

Algorytm 6.4. Pseudokod wyznaczenia liczby *oddechów* w polityce symulacji.

```
DANE WEJŚCIOWE: Punkt Point, który nie jest pusty i nie jest punktem granicznym planszy
DANE WYJŚCIOWE: Liczba oddechów

FUNCTION CountLiberties(Point)

   // Ustaw wartość początkową licznika oddechów NumOfLiberties na wartość 0
   NumOfLiberties ← 0
   // Zwiększ indeks LibertiesIdx
   LibertiesIdx ← LibertiesIdx + 1
   // CurrentPoint jest bieżącym kamieniem w łańcuchu
   CurrentPoint ← Point

   // Iteruj przez wszystkie kamienie w tablicy Stones
   DO
      // Dla każdego sąsiada Neighbour punktu CurrentPoint
      FOR EACH Neighbour IN CurrentPoint.GetNeighbours()

         // Jeżeli sąsiedni punkt jest pusty
         IF Neighbour.Color = EMPTY AND Liberties[Neighbour] != LibertiesIdx THEN

            NumOfLiberties ← NumOfLiberties + 1
            Liberties[Neighbour] ← LibertiesIdx

         END IF
      END FOR

      // Jeżeli liczba oddechów jest większa równa dwa
      IF NumOfLiberties >=2 THEN
         // Przerwij liczenie i zwróć wartość licznika oddechów
         RETURN NumOfLiberties
      END IF

      // Licz oddechy dla kolejnego kamienia w łańcuchu
      CurrentPoint ← Stones[CurrentPoint]

   // Dopóki nie zostały sprawdzone wszystkie kamienie (lista cykliczna)
   UNTIL CurrentPoint = Point

   // Zwróć wartość licznika oddechów
   RETURN NumOfLiberties

END FUNCTION
```

W fazie symulacji operacja łączenia łańcuchów jest realizowana przez połącznie list w taki sposób, by elementy każdej z osobnych list były elementami nowej jednej listy. Na rysunku 6.8 pokazano przykład połączenia dwóch łańcuchów.

Procedury wykonania ruchu i weryfikacji jego poprawności są analogiczne do wersji opisanych w polityce. Operacje anulowania ruchu nie zostały zaimplementowane, ponieważ wykonanie jednorazowej kopii struktur danych dla planszy, przed rozpoczęciem każdej symulacji, jest szybsze niż wielokrotne anulowanie ruchów (rozgrywka na planszy 19×19 średnio składa się z 200-400 ruchów).



Rysunek 6.6. Łączenia łańcuchów w polityce symulacji.

a. Łączenie dwóch łańcuchów (białe).
Lewa strona przed połączeniem: łańcuch zawierający pojedynczy kamień 15* i drugi łańcuch składający się z trzech kamieni 16, 10, 11.
Prawa strona po połączeniu: jeden łańcuch zawierający kamienie 15*, 10, 11, 16.

b. Łączenie trzech łańcuchów (czarne).
Lewa strona przed połączeniem: łańcuch zawierający pojedynczy kamień 26*, drugi łańcuch składający się z trzech kamieni 31, 32, 25 oraz trzeci łańcuch składający się z dwóch kamieni 20, 14.
Prawa strona po połączeniu: jeden łańcuch zawierający kamienie 26*, 32, 25, 31, 14, 20.

c. Stan gry. Kamień czarny 26* łączy trzy łańcuchy czarnych.
Kamień biały 15* łączy dwa łańcuchy białych.

d. Indeksowanie jednowymiarowej tablicy łańcuchów – $Stones$.

e. Wartości w komórkach tablicy łańcuchów – $Stones$, po złączeniu łańcuchów.

6.2.2. Wykonywanie losowych ruchów

Podczas symulacji program wykonuje losowe ruchy wybierane z jednakowym prawdopodobieństwem ze zbioru dostępnych posunięć. Zbiór utworzony jest z poprawnych ruchów (zgodnych z regułami gry) i dodatkowo nie zawiera pojedynczych *oczu*[17]. Zbiór jest zaimplementowany w postaci listy indeksów pustych punktów na planszy. Struktura listy jest zgodna ze strukturą łańcucha używaną w implementacji polityki drzewa, która zapewnia jednakowy, stały czas wykonania operacji dodawania i usuwania punktu. W pierwszym kroku procedury kopiowana jest tablica dostępnych punktów. Następnie losowana jest liczba

[17] Położenie kamienia w *oku* własnego łańcucha jest dużym błędem, ponieważ taki ruch zabiera *oddech*. W szczególnym przypadku może spowodować, że łańcuch, który wcześniej był bezwarunkowo *żywy*, teraz będzie „tylko” *żywy* i będzie atakowany przez oponenta.

należąca do zakresu indeksów tablicy wolnych punktów. Po wylosowaniu punktu, wykonywane jest sprawdzenia czy ruch jest prawidłowy (zgodny z regułami i nie jest w punkcie pojedynczego *oka*). Jeżeli ruch nie jest prawidłowy, to wylosowany punkt jest usuwany z listy i losowanie jest powtarzane. Jeżeli lista jest pusta i ruch nie został wylosowany gracz nie może wykonać ruchu. Wylosowanie poprawnego ruchu kończy procedurę.

6.2.3. Generator liczb pseudolosowych

W programie do generowania liczb losowych użyto funkcji biblioteki *boost* GSL[18].

Do generowania liczb pseudolosowych został wybrany złożony generator *Mersenne Twister*, charakteryzujący się wyjątkowo dużym okresem $K = 2^{10037} - 1 > 10^{1000}$ oraz szybkością działania, często znajdujący zastosowanie w metodach Monte-Carlo.

Generatory liczb pseudolosowych wymagają podania liczby inicjującej generowany ciąg wartości losowych (ang. seed). Aby wynikowy ciąg był „nieprzewidywalny" wartość inicjująca także powinna mieć losowy charakter. W przypadku, gdy potrzebna jest tylko jedna wartość, to z reguły używa się do tego celu czasu systemowego. W programie do inicjalizacji generatora MT została użyta formuła 6.6:

$$seed = r + 100 * (M - 1 + 12 * (d - 1 + 31 * (g + 24 * (m + 60 * s)))) \quad (6.6)$$

gdzie r − rok, M − miesiąc, d − dzień, g − godzina, m − minuta, s − sekunda.

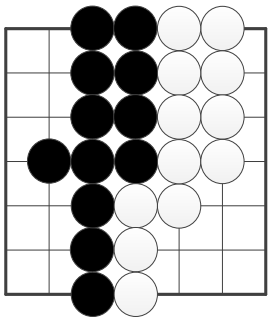
Przykład użycia generatora liczb losowych w bibliotece Boost pokazano w algorytmie 6.5.

Algorytm 6.5. Przykład użycia generatora liczb losowych.

```
namespace Seed
{
   unsigned int generate()
   {
      int r,m,d,g,min,s,seed;

      time_t tt = time(NULL);
      struct tm *date = localtime(&tt);
      r = (*date).tm_year + 1900;
      m = (*date).tm_mon;
      d = (*date).tm_mday;
      g = (*date).tm_hour;
      min = (*date).tm_min;
      s = (*date).tm_sec;
      seed = r + 100*(m - 1 + 12*(d - 1 + 31*(g + 24*(min + 60*s))));
      return seed;
   }
}  //namespace Seed

//ustawienie wartości „zarodka" generatora
BaseRandom::SetSeed(Seed::generate());

//losowanie z listy pustych punktów emptyPts
int length = emptyPts.Length();
int index = random.Int(length);
PointIdx p = emptyPts[index];
```

6.3. Ocena wyniku gry

W grze w Go stosowane są dwie podstawowe metody obliczania wyniku gry[18]. Wynik gry może być obliczony na podstawie: terytorium (ang. territory) albo obszaru (ang. area). Obliczanie wyniku gry na podstawie terytorium stosowane jest w regułach japońskich i polega na zsumowaniu pól kontrolowanego terytorium (punktów otoczonych przez kamienie) oraz wszystkich zbitych kamieni. Obliczanie wyniku gry na podstawie obszaru, stosowane jest w regułach chińskich i polega na zsumowaniu punktów otoczonych przez kamienie i dodaniu wszystkich „żywych" kamieni na planszy. Do liczby punktów uzyskanych przez białe dodawane są punkty *komi* (zwykle 6.5 albo 7.5), które rekompensują przewagę czarnych, które rozpoczęły grę jako pierwsze. Wynik gry obliczony tymi dwiema metodami zwykle różni się, co najwyżej o jeden punkt. W programach komputerowych częściej implementowane są zasady chińskie (wynika to z faktu, że reguły japońskie nie zawsze są jednoznaczne) i w konsekwencji wynik gry obliczany jest metodą według obszaru.



Rysunek 6.7. Obliczenie wyniku gry na podstawie terytorium i obszaru.

Komi = 6.5 punktu
Wynik gry na podstawie terytorium: (C; B) = (13; 12+6,5) = (13; 18,5). Wygrały białe 5.5 pkt.
Wynik gry na podstawie obszaru: (C; B) = (13+12; 12+12+6,5) = (25; 30,5). Wygrały białe 5.5 pkt.

Wynik obliczony metodami według terytorium i obszaru może być różny z następujących powodów:

1. Jeden z graczy zrezygnował z ruchu, podczas gdy drugi z graczy ułożył kamień wewnątrz własnego terytorium.
2. Czarne wykonały większą liczbę ruchów (czarne rozpoczęły i zakończyły grę).
3. Gra toczyła się z wyrównaniem − słabszy gracz (grający czarnymi) ustawił przed rozpoczęciem gry pewną liczbę kamieni wynikającą z różnicy siły gry (handicap).

W uproszczeniu, obliczenie terytorium i obszaru polega na zsumowaniu pól, którym zostały przypisane kolory graczy. Każdy punkt na planszy może być oznaczony kolorem: białym, czarnym albo neutralnym. Kolor punktów zajętych przez kamienie jest jednoznacznie określony przez kolor znajdującego się tam kamienia. Kolor dla niezajętych punktów zależy od tego, który z graczy kontroluje ten punkt.

Kolor może być przypisany według trzech kryteriów: terytorium (ang. territory), *moyo* albo obszaru (ang. area):

[18] Każda odmiana gry w Go (japońska, chińska, amerykańska, nowozelandzka), ma inne zasady obliczania wyniku gry. W pracy odniesiono się wyłącznie do reguł japońskich i chińskich. W programie zaimplementowano tylko chińskie reguły obliczania wyniku gry.

a) **Terytorium** (ang. territory). Punkty planszy, które po zakończeniu gry, podczas obliczania wyniku, mogą się zmaterializować w postaci punktu przyznanego dla jednego z graczy. Innymi słowy są to punkty otoczone przez kamienie gracza.
b) ***Moyo***. „Prawie" terytorium, fragment planszy, który może się łatwo stać terytorium, jeżeli oponent zlekceważy zagrożenie.
c) **Wpływ**[19] (ang. area). Fragment planszy zawierający punkty, w których jeden z graczy ma większy wpływ niż jego oponent.

Ogólnie każde terytorium jest *moyo* oraz każde *moyo* jest wpływem.

Algorytm obliczania wyniku gry na podstawie obszaru może być opisany w następujący sposób:

1. Określ statusy *żywy* albo *martwy* dla wszystkich bloków kamieni.
2. Usuń *martwe* kamienie z planszy.
3. Dla każdego punktu na planszy (zajętego przez kamień lub pustego) przypisz jeden z kolorów: czarny, biały, neutralny.
4. Oblicz wynik gry. Grę wygrywa ten gracz, który kontroluje większą liczbę punktów na planszy.

6.3.1. Metody bezpośrednie

W tym punkcie zostanie opisanych sześć metod bezpośrednich umożliwiających obliczenie terytorium. Każda z tych metod przypisuje wartość liczbową każdemu punktowi na planszy. Przyjęto zasadę, że wartości dodatnie przypisywane są punktom kontrolowanym przez czarne kamienie i ujemne – przez białe.

Metoda jednoznaczna

Metoda jednoznaczna EC (ang. explicit control) oblicza tylko te terytoria, które są jednoznacznie kontrolowane przez jednego z graczy. W metodzie EC, punktom całkowicie otoczonym przez czarne, przypisywana jest wartość $+1$ oraz wartość -1 punktom całkowicie otoczonym przez białe. (Przez całkowicie otoczone punkty należy rozumieć jednopunktowe „oczy".) Wszystkim pozostałym pustym punktom przypisywana jest wartość 0.

Metoda bezpośrednia

Metoda EC jest bardzo niedokładna. W metodzie bezpośredniej DC (ang. direct control), każdemu pustemu punktowi sąsiadującemu z czarnym kamieniem i niesąsiadującemu z białym, przypisywana jest wartość $+1$. Każdemu pustemu punktowi sąsiadującemu z białym kamieniem i niesąsiadującemu z czarnym, przypisywana jest wartość -1. Wszystkim pozostałym pustym miejscom przypisywana jest wartość 0.

Metoda rozrostu ziarna

Metoda rozrostu ziarna[20] (ang. flood fill) przypisuje rekurencyjnie każdemu z pustych punktów jeden z kolorów. Jeżeli wypełnienie obszarów dla czarnych nakłada się na obszar dla białych, to takim punktom na planszy nadawany jest kolor neutralny. (W konsekwencji

[19] Wpływ jest zamiennie nazywany obszarem. Nie należy jednak tego terminu utożsamiać z obliczeniem wyniku gry według obszaru. W pracy, w celu uniknięcia homonimu, jest używany termin „wpływ".
[20] Metoda rozrostu ziarna jest powszechnie używana w programach graficznych do wypełniania zamkniętych obszarów bitmapy zadanym kolorem.

wszystkie punkty mające kolor różny od neutralnego muszą być otoczone przez kamienie w jednym kolorze.) Algorytm rozrostu ziarna może być zaimplementowany z wykorzystaniem stosu (wersja rekursywna) albo kolejki. Wersja z użyciem kolejki został przedstawiona w algorytmie 6.6.

Algorytm 6.6. Algorytm rozrostu ziarna w wersji z kolejką.

```
DANE WEJŚCIOWE: Punkt Point, zastępowany kolor PrevColor, nowy kolor NewColor
DANE WYJŚCIOWE: N/A

PROCEDURE FloodFile (Point, PrevColor, NewColor)
   Queue ← CreateQueue()     // Utwórz nowa pustą kolejkę dla analizowanych punktów
   Queue.Insert(Point)       // Wstaw punkt Point do kolejki

   // Dopóki kolejka nie jest pusta
   DO WHILE (Queue.Length() > 0)

      // pobierz i usuń kolejny punkt z kolejki i przypisz do zmiennej NextPoint
      NextPoint ← Queue.RemovePoint();
      // Jeżeli kolor punktu NextPoint jest zgodny z zastępowanym kolorem
      IF NextPoint.GetColor() = PrevColor THEN

         // Zmień kolor punktu NextPoint
         NextPoint.SetColor(NewColor)

         // Wstaw do kolejki każdy z czterech sąsiadów punktu NextPoint
         Queue.Insert(NextPoint.GetWestNeighbour())  // Punkt sąsiadujący z lewej strony
         Queue.Insert(NextPoint.GetEastNeighbour())  // Punkt sąsiadujący z prawej strony
         Queue.Insert(NextPoint.GetNorthNeighbour()) // Punkt sąsiadujący od góry
         Queue.Insert(NextPoint.GetSouthNeighbour()) // Punkt sąsiadujący od dołu
      END IF
   END DO
END PROCEDURE
```

Metoda bazująca na odległości

Metoda jednoznaczna i metoda bezpośrednia nie rozpoznają regionów otoczonych przez luźno połączone kamienie. Z taki regionami „radzi sobie" metoda DBC bazująca na odległości (ang. distance-based control). W metodzie DBC do pomiaru odległości stosowana jest miara Manhattan. Pustym punktom położonym bliżej czarnego kamienia przypisywana jest wartość $+1$, położonym bliżej białego kamienia, wartość -1. Jeżeli odległość do czarnego oraz białego kamienia jest jednakowa, to punktowi przypisywana jest wartość 0.

Ostateczny wynik gry zależy od różnicy pomiędzy liczbą punktów w kolorach czarnym i białym po uwzględnieniu dodatkowych punktów *komi*. Obliczenia przeprowadzone metodą rozrostu ziarna i odległości powinny dać ten sam wynik. Jeżeli obliczone dwoma metodami wyniki są różne, to oznacza, że na planszy nadal znajdują się duże obszary, które nie są kontrolowane przez graczy. Może być to spowodowane tym, że nie wszystkie *martwe* kamienie zostały usunięte lub, że gra nie powinna zostać jeszcze zakończona, ponieważ są jeszcze punkty do zdobycia przez obu graczy i pojedynek nie jest jeszcze rozstrzygnięty. Porównanie wyników obliczonych tymi dwoma metodami jest dobrym kryterium na sprawdzenie, czy gra jest zakończona oraz czy wynik jest poprawnie obliczony.

Metoda bazująca na wpływie

Słabością metody bazującej na odległości jest pomijanie siły kamieni. W metodzie DBC pojedynczy kamień może mieć porównywalną siłę jak grupa kamieni znajdująca się w tej samej odległości. Z takimi problemami skutecznie radzi sobie metoda IBC bazująca na

wpływie (ang. influence-based control). Po raz pierwszy metoda ta została użyta przez Zobrista (1968). Metoda działa następująco:

1. Wszystkim punktom, w których znajdują się kamienie, przypisz wartość dodatnią dla kamieni czarnych oraz ujemną dla białych. Dla pustych punktów przypisz wartość zero.
2. Czterokrotnie wykonaj następującą operację. Dla każdego punktu, do jego wartości bezwzględnej dodaj liczbę sąsiednich punktów mających ten sam znak i odejmij liczbę sąsiednich punktów mających znak przeciwny. W istocie jest to operacja morfologiczna dylatacji.

Metoda Bouzy'ego

Po raz pierwszy operacje morfologiczne, w komputerowej grze w Go, zostały użyte przez Zobrista (1968). Algorytm Zobrista korzystał jedynie z operacji dylatacji. Algorytm prawidłowo obliczał wpływ, ale nie radził sobie z obliczeniem terytorium. Metoda Bouzy'ego jest modyfikacją algorytmu Zobrista. Bouzy zaproponował operator D_Z będący modyfikacją operatora dylatacji stosowanego w morfologii matematycznej[21]. Operator dylatacji D_Z jest zbliżony (ale nie identyczny) do operatora zaproponowany przez Zobrista. Operator działa w następujący sposób. Dla każdego punktu mającego wartość różną od zera, który nie sąsiaduje z punktem o przeciwnym znaku, dodaj do jego wartości bezwzględnej, liczbą sąsiednich punktów tego samego znaku. Dla każdego punktu mającego wartość zero, którego sąsiednie punkty mają tylko wartość dodatnią, dodaj liczbę dodatnich sąsiednich punktów. Dla każdego punktu mającego wartość zero, którego sąsiednie punkty maja tylko wartość ujemną, odejmij liczbę ujemnych sąsiednich punktów.

Opisany operator dylatacji D_Z nie zapewnia wysokiej jakości rozpoznawania terytorium. Dlatego Bouzy zaproponował użycie kolejnego operatora E_Z – będącego zmodyfikowanym operatorem erozji. Zaproponował również, aby po kilkukrotnym użyciu operatora dylatacji, kilkukrotnie użyć operator erozji. Zmodyfikowany operator erozji działa w następujący sposób. Dla każdego punktu mającego niezerową wartość, odejmij od wartości bezwzględnej liczbę sąsiednich punktów mających wartość zerową lub mających przeciwny znak. Jeżeli wynik operacji powoduje zmianę znaku wartości dla punktu, to przypisz wartość zero.

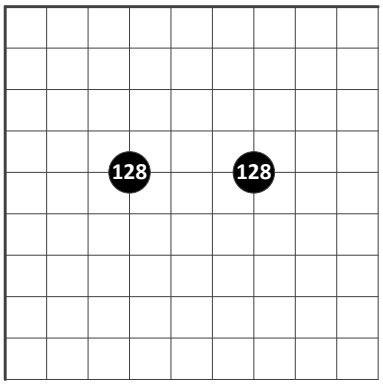
Najpierw kilka razy wykonywany jest operator dylatacji D_Z i następnie kilka razy − operator erozji E_Z. Sekwencję operacji można nazwać operatorem złożonym $Xz(d, e)$, gdzie d jest liczbą operacji dylatacji i e liczbą operacji erozji. Algorytm korzystający z nowego operatora został najpierw zaimplementowany w programie INDIGO i następnie po wykazaniu swojej efektywności także w programie GNUGO. Bouzy zaproponował, aby liczba powtórzeń obu operatorów spełniała formułę $e = 1 + d(d - 1)$ ponieważ odpowiada to operacji jednostkowej dla pojedynczego kamienia znajdującego się w środku dostatecznie dużej planszy. Dodatkowo zalecił, aby liczba powtórzeń dylatacji wynosiła 4 albo 5. Liczba dylatacji jest mniejsza niż liczba erozji, ponieważ operator dylatacji szybciej propaguje niż operator erozji.

[21] Morfologia matematyczna jest działem matematyki bazującym na teorii zbiorów. Metody morfologii matematycznej są często stosowane w przetwarzaniu obrazów. Na przykład, umożliwiają usunięcie szczegółów obrazu, których rozmiar jest mniejszy od zadanego progu.

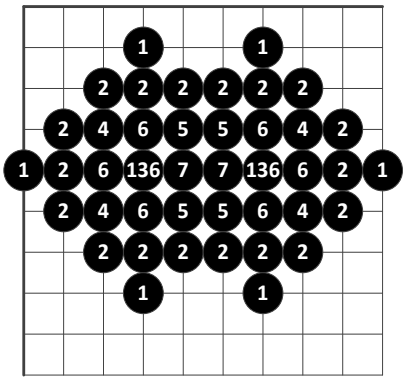
Ogólnie, algorytmy bazujące na użyciu operacji morfologicznych składają się z trzech kroków:

1. Inicjalizacja. Każdemu punktowi na planszy zajmowanemu przez kamień przypisz wartość dodatnią np. $+128$ (Bouzy) lub $+64$ (Zobrist) dla czarnych oraz -128 (Bouzy) lub -64 (Zobrist) dla białych. Pustym punktom przypisz wartość 0.
2. Wykonaj operację dylatacji d razy.
3. Wykonaj operację erozji e razy. (W wariancie Zobrista $e = 0$).

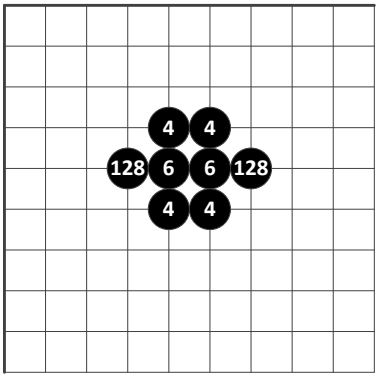
Przykład działania algorytmu z wykorzystaniem operacji morfologicznych pokazano na rysunku 6.8.



Inicjalizacja. Wartości pól po inicjalizacji.



Dylatacja. Wartości pól po wykonaniu trzech kolejnych operacji dylatacji



Erozja. Wartości pól po wykonaniu siedmiu kolejnych operacji erozji

Rysunek 6.8. Działanie algorytmu Bouzy'ego z użyciem operacji morfologicznych.

Operacje morfologiczne, w komputerowej grze w Go, można wykorzystać nie tylko do klasyfikacji terytoriów, ale także *Moyo*, wpływu, a nawet *oddechów*. Istotne jest, aby przed zastosowaniem algorytmu wcześniej zidentyfikować i usunąć z planszy *martwe* grupy kamieni.

Terytorium można obliczyć używając operatora $Xz\,(5, 21)$ i stąd nazwa algorytmu 5/21. Alternatywnie można zastosować mniej dokładny, ale szybszy algorytm 4/13. Algorytm 4/13 jest stosowany do szacowania terytorium podczas gry, a bardziej dokładny i wolniejszy 5/12 do obliczenia końcowego wyniku. Wpływ może być wyznaczony przy użyciu algorytmu 4/0 (inaczej algorytm Zobrista), zaś *Moyo* przy użyciu algorytmu 5/10. Algorytm może również pomóc w obliczeniu *oddechów* dla grupy kamieni – 1/0 czy „silnych" terytoriów (regionów całkowicie otoczonych przez kamienie w jednym kolorze) − 2/2.

6.3.2. Ograniczenia metod bezpośrednich

Wysoką jakość klasyfikacji terytorium zapewniają metody z zastosowaniem operacji morfologicznych, które jako jedyne znajdują terytoria z dużym poziomem ufności. Regiony z małym poziomem usuwane są przez operację erozji. Z tego powodu na planszy mogą pozostawać regiony z nierozstrzygniętym terytorium. Regiony te można podzielić za pomocą metody DBC bazującej na odległości. Połączenie tych dwóch metod klasyfikuje terytoria z jakością porównywalną do uczenia maszynowego. Pozostałe metody bezpośrednie mogą być jedynie użyte do szacowania terytoriów w trakcie gry (np. do budowy funkcji oceny).

Metody bezpośrednie nie są wrażliwe na status kamieni: *żywy* / *martwy*, co jest ich poważną wadą. Dlatego przed ich użyciem, w celu obliczenia wyniku gry, należy usunąć wszystkie *martwe* kamienie z planszy. Jeżeli metody bezpośrednie są używane do szacowania terytorium w trakcie gry, to rozpoznawanie statusu kamieni jest jeszcze trudniejszym zadaniem. Najczęściej problem ten „atakowany jest" za pomocą funkcji heurystycznej.

Alternatywą dla metod bezpośrednich jest maszynowe uczenie. Metody te zapewniają największą dokładność wyznaczania terytoriów [5][12].

6.3.3. Identyfikacja *martwych* kamieni

Bez względu na zastosowaną metodę bezpośrednią do obliczania wyniku gry, należy wcześniej zidentyfikować i usunąć z planszy *martwe* grupy kamieni. Dlatego rozwiązanie problemu obliczenia wyniku gry wymaga znalezienia metody prawidłowego rozpoznaniu *martwych* kamieni, które muszą być usunięte z planszy przed rozpoczęciem „kolorowania" punktów. Kamienie, które nie mogą być zbite (nawet jeżeli gracz zrezygnuje z ruchu) klasyfikowane są jako bezpieczne albo bezwarunkowo *żywe*. Takie kamienie nie muszą być bronione i nie warto ich atakować. Benson (1976) wykazał, że grupa kamieni mająca parę *oczu* jest bezpieczna [14]. Pozostałe kamienie można sklasyfikować jako *żywe* albo *martwe*. Jeżeli oponent może zbić kamienie gracza, bez względu na obronę i nawet, jeżeli wykona ruch jako drugi, to kamienie są *martwe*. Kamienie *żywe* są celem ataku i muszą być bronione. Najprostszym rozwiązaniem problemu jest przeszukanie wszystkich możliwych (zgodnych z regułami) ruchów, aż do zakończenia gry i znając wynik gry przewidzieć ruchy graczy[22]. Prowadzi to jednak do kombinatorycznej „eksplozji". Takie rozwiązanie zaproponował Wolf (2001). Metoda jest dokładna, ale ze względu na wysiłek obliczeniowy nie może być stosowana w trakcie gry. Rozstrzygnięcie czy kamienie są *żywe* czy *martwe* należy do klasy problemów PSPACE trudnych. Dla problemów NP-trudnych, które muszą być rozwiązane w czasie rzeczywistym najczęściej stosowane są funkcje heurystyczne. Jeszcze inną propozycją rozwiązania jest użycie bazy wzorców.

W pracy przyjęto założenie, że kryterium oceny statusu kamieni będzie bazowało na liczbie *oddechów*. Każdy blok mający jeden *oddech* to znaczy taki, który może być zbity w wyniku jednego ruchu, uznawany jest za *martwy* i jest usuwany z planszy.

[22] Algorytm alfa-beta

6.4. Implementacja rozwiązania

W punkcie opisano szczegóły rozwiązania, konfigurowanie programu, metodę testowania oraz wyniki testów wydajności symulacji Monte-Carlo oraz siły gry programu dla różnych ustawień.

6.4.1. Opis rozwiązania

Rozwiązanie zostało zakodowane w C++ i zbudowane w środowisku programistycznym Visual Studio 2010 pod kontrolą systemu operacyjnego Windows 7 – 32 bity.

Rozwiązania składa się z czterech projektów: **BASE**, **GAME**, **MCTS** i **GOUI**. Rozwiązanie ma budowę czterowarstwową. Warstwy wyższe korzystają z funkcji warstw niższych. Każdy z projektów realizuje inne funkcje w programie. Na najniższym poziomie znajduje się moduł **BASE**, na najwyższym – **GOUI**.

W rozwiązaniu użyto biblioteki BOOST[23] oraz biblioteki QT[24] (biblioteki CORE i GUI).

Projekt **BASE** zawiera implementację struktur ogólnego przeznaczenia:
a) Zbiór (wektor, tablica, iterator, stos);
b) Plansza (plansza, punkt, kolor, zbiór punktów);
c) Drzewo przeszukiwania (węzeł);
d) Kontrola czasu;
e) Generator liczb losowych.

Projekt **GAME** zawiera implementację struktur dedykowanych dla gry w Go:
a) Zbiór kamieni (region, łańcuch, blok, „oczy”);
b) Plansz (plansza, historia, synchronizacja);
c) Gracz;
d) Reguły gry;
e) Ocena wyniku gry.

Projekt **MCTS** zawiera implementację algorytmu UCT:
a) Plansza optymalizowana dla symulacji Monte-Carlo;
b) Implementacja fazy przeglądania drzewa;
c) Implementacja fazy rozgrywek.

Projekt **GOUI** zawiera implementację graficznego interfejsu użytkownika:
a) Główne okno programu;
b) Okienka parametrów;
c) Obsługa plików WE/WY (w tym komunikacja z programem GNU Go).

Moduły BASE, GAME, MCTS kompilowane są do postaci bibliotek statycznych LIB. Moduł GOUI do pliku wykonywalnego EXE.

[23] Boost w wersji 1.47.0. Boost jest zbiorem bibliotek poszerzających możliwości języka C++, objętych licencją, która umożliwia użycie ich w dowolnym projekcie.
[24] Qt w wersji 4.8.5. Biblioteka Qt jest zestawem przenośnych bibliotek i narzędzi programistycznych dedykowanych dla języków C++, QML i Java. Do budowy rozwiązania użyto wersji Open Source.

6.4.2. Konfigurowanie programu

Ustawienia programu, które można zmieniać w interfejsie użytkownika (bez konieczności zmian w kodzie i kompilacji nowego rozwiązania) zostały podzielone na trzy grupy: parametry gry w Go, ustawienia algorytmu przeszukiwania MCTS oraz ustawienia turniejów (pojedynków komputer − komputer).

Parametry gry w Go, które można zmieniać w programie:

a) Rozmiar planszy;
b) Punkty *komi*;
c) Gra z wyrównaniem.

W programie dostępne są plansze od 5×5 do 19×19 punktów. Domyślnym rozmiarem planszy jest 9×9 punktów. Rozmiar planszy można zmienić tylko przed rozpoczęciem gry lub turnieju.

W programie dostępne są punkty *komi* od 4.5 do 7.5 punktu. Domyślnie *komi* ma wartość 6.5. Liczbę punktów *komi* można zmienić w dowolnej chwili, także w trakcie gry.

Liczbę kamieni handicap można określić tylko przed rozpoczęciem gry. Liczby kamieni nie można zmieniać w trakcie gry. Kamienie umieszczane są w predefiniowanych punktach. W programie nie ma funkcji umieszczania kamieni w dowolnych punktach. Każdy z rozmiarów plansz ma maksymalną dopuszczalną liczbę kamieni handicap oraz listę predefiniowanych, optymalnych punktów.

Parametry algorytmu przeszukiwania MCTS, które można zmieniać w programie:

a) Liczba wątków od 1 do 16 – domyślna liczba wyznaczana jest przez sprzęt;
b) Czas „myślenia" nad ruchem – domyślna wartość 5s;
c) Kryterium wyboru ruchu (Maksymalny, Silny, Bezpieczny, Zrównoważony) – wartość domyślna Silny;
d) Strategia przeszukiwania (UCT, Monte-Carlo, heurystyki) – wartość domyślna UCT;
e) Heurystyki.

Program może wykonywać symulacje rozgrywek używają od 1 do 16 wątków. Można ustawić większą liczbę wątków niż jest liczba fizycznie dostępnych w procesorze. Najwyższą wydajność uzyskuje się przy ustawieniu „Hardware", dla którego program korzysta ze wszystkich dostępnych sprzętowo wątków. Domyślną wartością liczby wątków jest „Hardware". Liczbę wątków można zmienić tylko przed rozpoczęciem gry lub turnieju.

Limit czasu „myślenia" nad ruchem można ustawiać w zakresie od 1 do 999 sekund. Czas można zmieniać w trakcie gry. Empirycznie stwierdzono, że nie należy ustawiać dłuższych czasów niż 10 sekund (zwłaszcza na szybkim sprzęcie). Pamięć zaalokowana w programie dla drzewa przeszukiwania szybko ulega wyczerpaniu. W praktyce czas powinien wynosić od 3 do 5 sekund.

W programie dostępne są heurystyki stosowane w podanej kolejności (nie można zmienić kolejności heurystyk, ale można je wyłączyć):

a) Atari – atak;
b) Atari – obrona;
c) Minimalizacja liczby oddechów w łańcuchach kamieni oponenta;
d) Zbicie;
e) Losowy ruch.

6.4.3. Metoda testowania

W celu przetestowania poprawności działania programu, wydajności oraz siły gry zaimplementowano w programie funkcję rozgrywania turniejów. Turnieje składają się pojedynków rozgrywanych pomiędzy dwoma programami komputerowymi. Oponentem jest program GNU GO, który w badaniach najczęściej jest wybierany, jako program referencyjny. Kod programu GNU GO został pobrany z sieci Internet [17], następnie zostało utworzone rozwiązanie w środowisku programistycznym Visual Studio 2010 za pomocą, którego został wygenerowany plik wykonywalny. Program GNU GO uruchamiany jest z poziomem dziesiątym odpowiadającym największej sile gry programu. Komunikacja pomiędzy programami komputerowymi wykonywana jest przez pliki w formacie SGF[25]. Format SGF jest uniwersalny i jest powszechnie używany do wymiany danych pomiędzy programami dla różnych gier turowych (między innymi w programach do gry w Go).

Dla każdej rozgranej partii podczas turnieju tworzony jest rekord gry. Rekord gry zawiera podstawowe informacje, które umożliwiają dalszą analizę statystyczną gier. W rekordzie gry rejestrowane są następujące dane:

a) Data rozpoczęcia gry;
b) Godzina rozpoczęcia gry;
c) Dostępna pamięć w MB;
d) Pamięć zarezerwowana dla drzewa przeszukiwania w MB;
e) Maksymalna liczba węzłów w drzewie w tysiącach;
f) Liczba wątków;
g) Rozmiar planszy;
h) Limit czasu na „myślenie” nad ruchem;
i) Strategia przeszukiwania drzewa;
j) Kryterium wyboru najlepszego ruchu;
k) Liczba punktów *komi*;
l) Liczba zbitych czarnych kamieni;
m) Liczba zbitych białych kamieni;
n) Liczba czarnych kamieni na planszy;
o) Liczba białych kamieni na planszy;
p) Liczba wykonanych ruchów w grze;
q) Przyczyna zakończenia gry (B[]W[] − Czarne PASS, Białe PASS; W[]B[] − Białe PASS, Czarne PASS; B[r] – rezygnacja Czarnych, W[r] – rezygnacja Białych);
r) Zwycięzca (B – Czarne, W – Białe);
s) Liczba punktów przewagi;
t) Liczba symulacji na sekundę;
u) Data zakończenia gry;
v) Godzina zakończenia gry.

Rekordy gier zapisywane są w zbiorczym pliku w formacie CSV (wspólnym dla wszystkich gier w turnieju), który następnie „ręcznie” wczytywany jest do arkusza kalkulacyjnego i poddawany dalszemu przetwarzaniu. Dodatkowo dla każdej gry tworzony jest plik tekstowy

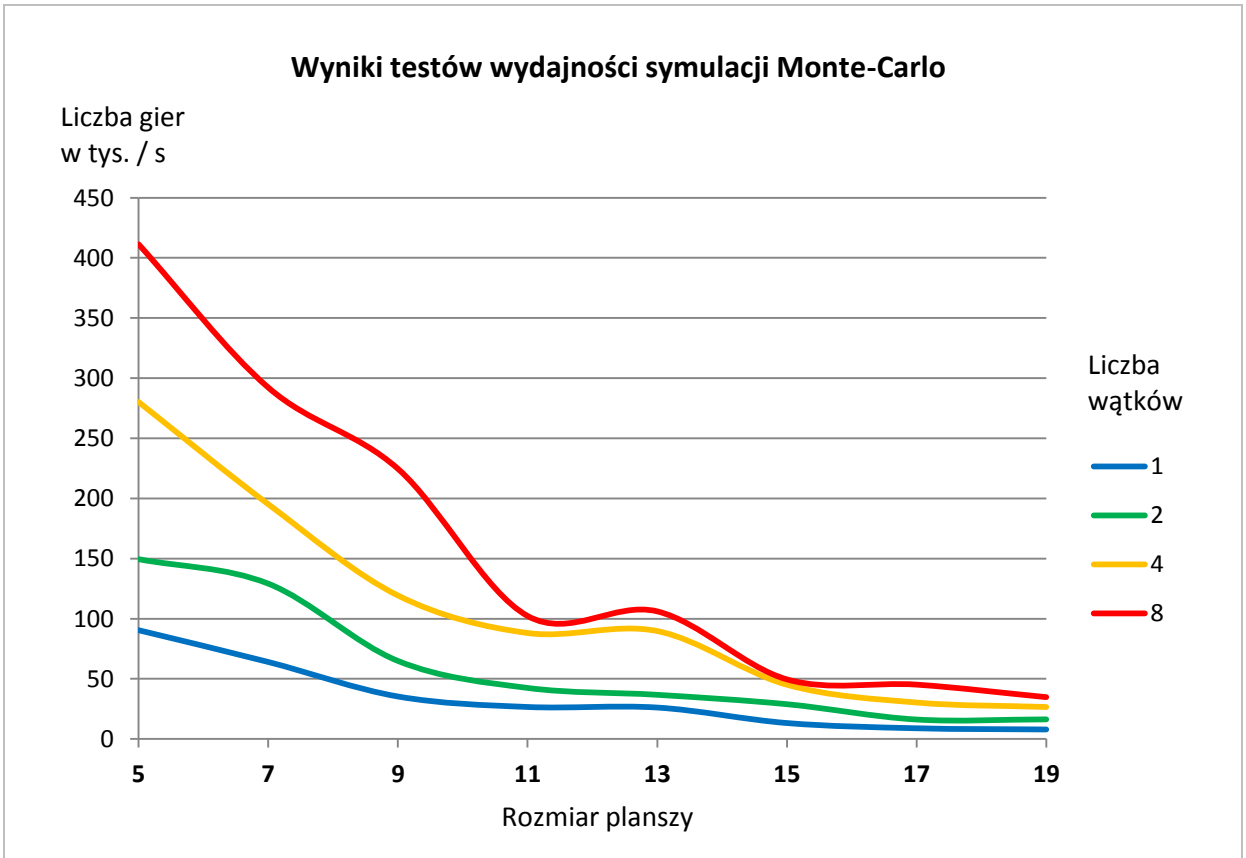
[25] Szczegóły formatu SGF oraz treść przykładowego pliku można znaleźć w Załączniku 3. „Instrukcja użytkownika programu GoGame”.

zawierający informacje o każdym ruchu wykonywanym przez komputer (dostępna pamięć, pamięć zarezerwowana dla drzewa przeszukiwania, rozmiar drzewa, potencjalne ruchy uporządkowane w kolejności od najlepszego, wartość najlepszego ruchu itd.).

Na zakończenie wszystkie pliki tworzone podczas turnieju (pliki komunikacji, statystyk i informacyjne) przenoszone są do jednego katalogu.

6.4.4. Testy wydajność symulacji Monte-Carlo

Przeprowadzone testy wykazały, że program bez zrównoleglania, korzystający tylko z jednego wątku, działający na komputerze[26] może przeprowadzić 100 tys. symulacji Monte-Carlo w ciągu jednej sekundy dla rozgrywek na planszy 5×5 oraz 10 tys. symulacji na planszy 19×19. Program korzystający ze zrównoleglenia operacji dla ośmiu wątków może przeprowadzić 500 tys. symulacji Monte-Carlo w ciągu jednej sekundy dla rozgrywek na planszy 5×5 oraz 40 tys. symulacji na planszy 19×19. W tabeli 6.1 zebrano wyniki pomiarów w dla różnych rozmiarów plansz[27]. Wyniki w formie graficznej zilustrowano na rysunku 6.9. Limit czasu „myślenia" na ruchem w wszystkich testach był równy 3 sekundy. Dla każdej pary: plansza, wątek zostało rozegranych 10 partii. Liczba symulacji jest średnią arytmetyczną z 10 partii. Liczba symulacji jest najmniejsza na początku partii i stopniowo rośnie osiągając maksimum w końcowej fazie gry.



Rysunek 6.9. Wyniki testów wydajności symulacji Monte-Carlo.

[26] DELL, Intel® Core™ i7-2600K CPU @3.40GHz, RAM 12.0 GB, Windows 7 64-bity

[27] Dane statystyczne dla każdej gry w formie plików CSV oraz dane zbiorcze w arkuszu MS Excel znajdują się w Załączniku 4.

Tabela 6.1. Wyniki testów wydajności symulacji Monte-Carlo.

Liczba gier w tys. / s	Wątki							
	1		2		4		8	
Plansza	Min.	Maks.	Min.	Maks.	Min.	Maks.	Min.	Maks.
5 × 5	75,5	105,4	139,8	159,2	239,4	321,0	299,6	523,0
7 × 7	40,1	87,9	79,7	178,6	138,1	252,3	201,8	382,6
9 × 9	19,2	51,5	36,6	93,2	66,0	172,5	106,7	342,5
11 × 11	14,7	38,5	25,5	59,4	60,9	115,3	71,8	132,8
13 × 13	10,7	41,5	20,5	52,9	34,8	144,6	59,5	152,7
15 × 15	10.2	16.3	14.8	43.0	26.4	63.8	31.6	67.5
17 × 17	6.3	11.4	11.9	20.4	21.0	39.7	38.1	52.2
19 × 19	5.8	10.0	13.0	19.6	16.8	36.3	26.3	43.2

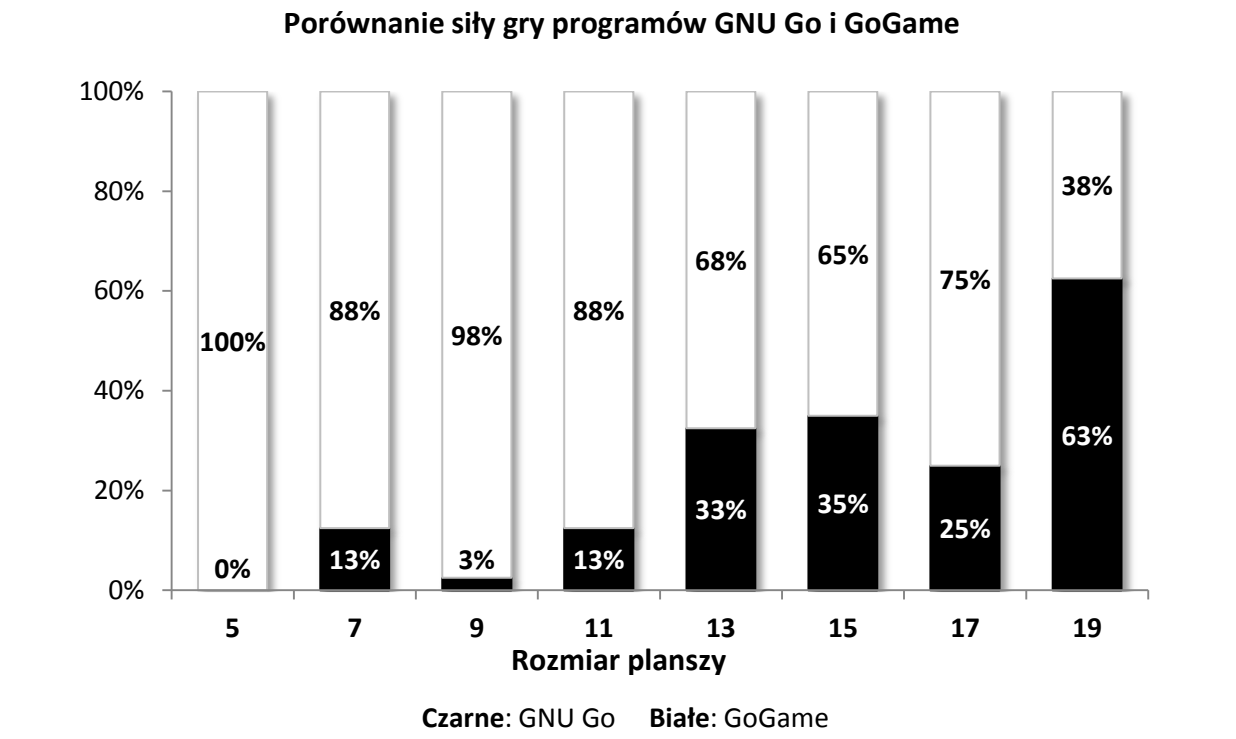
6.4.5. Testy siły gry programu

W tabeli 6.2 przedstawiono wyniki pojedynków z programem referencyjnym GNU GO grającym na poziomie dziesiątym (najsilniejszym). Pojedynki rozgrywane były zgodnie z regułami Chińskimi, na planszach od 5 × 5 do 19 × 19 punktów, bez handicapu, *komi* − 6.5 punktu. W programie ustawiono parametry: limit czasu „myślenia" nad ruchem − 3 sekundy, kryterium wyboru najlepszego ruchu – Silny, strategia przeglądania drzewa – UCT. W symulacjach były używane wszystkie dostępne heurystyki.

Tabela 6.2. Wyniki pojedynków z programem referencyjnym GNU Go.

			Plansza							
			5	7	9	11	13	15	17	19
Zwycięstwa		Czarne	0%	13%	3%	13%	33%	35%	25%	63%
		Białe	100%	88%	98%	88%	68%	65%	75%	38%
Przyczyna zakończenia gry		Rezygnacja Czarnych	90%	30%	3%	10%	25%	38%	25%	62%
		Rezygnacja Białych	0%	0%	0%	0%	0%	0%	0%	0%
		PASS: Czarne, Białe	10%	70%	98%	90%	75%	63%	75%	38%
		PASS: Białe, Czarne	0%	0%	0%	0%	0%	0%	0%	0%
Liczba punktów przewagi		Min	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
		Max	8.5	17.5	31.5	33.5	56.5	47.5	59.5	96.5
		Średnia	4.7	7.9	12.2	12.0	13.4	16.2	15.6	23.6
Liczba ruchów		Min	7	24	28	42	68	82	138	178
		Max	30	62	74	115	182	239	258	341
		Średnia	14.0	38.0	49.2	84.7	124.6	160.4	199.0	277.7
Liczba zbitych kamieni	Czarnych	Min	0	0	0	0	0	0	0	2
		Max	2	13	11	14	13	10	18	20
		Średnia	0.28	1.93	1.13	1.75	3.78	3.60	5.03	7.25
	Białych	Min	0	0	0	0	0	0	1	3
		Max	6	11	10	22	31	31	30	62
		Średnia	1.2	3.1	2.1	4.2	9.7	12.8	12.7	26.3

Na rysunku 6.10 porównano siłę gry programu GO GAME z programem referencyjnym GNU GO. Można zauważyć, że dla małych i średnich rozmiarów plansz program GO GAME ma większą siłę gry, ale dla plansz o maksymalnym rozmiarze 19×19 przewagę uzyskuje program GNU GO.



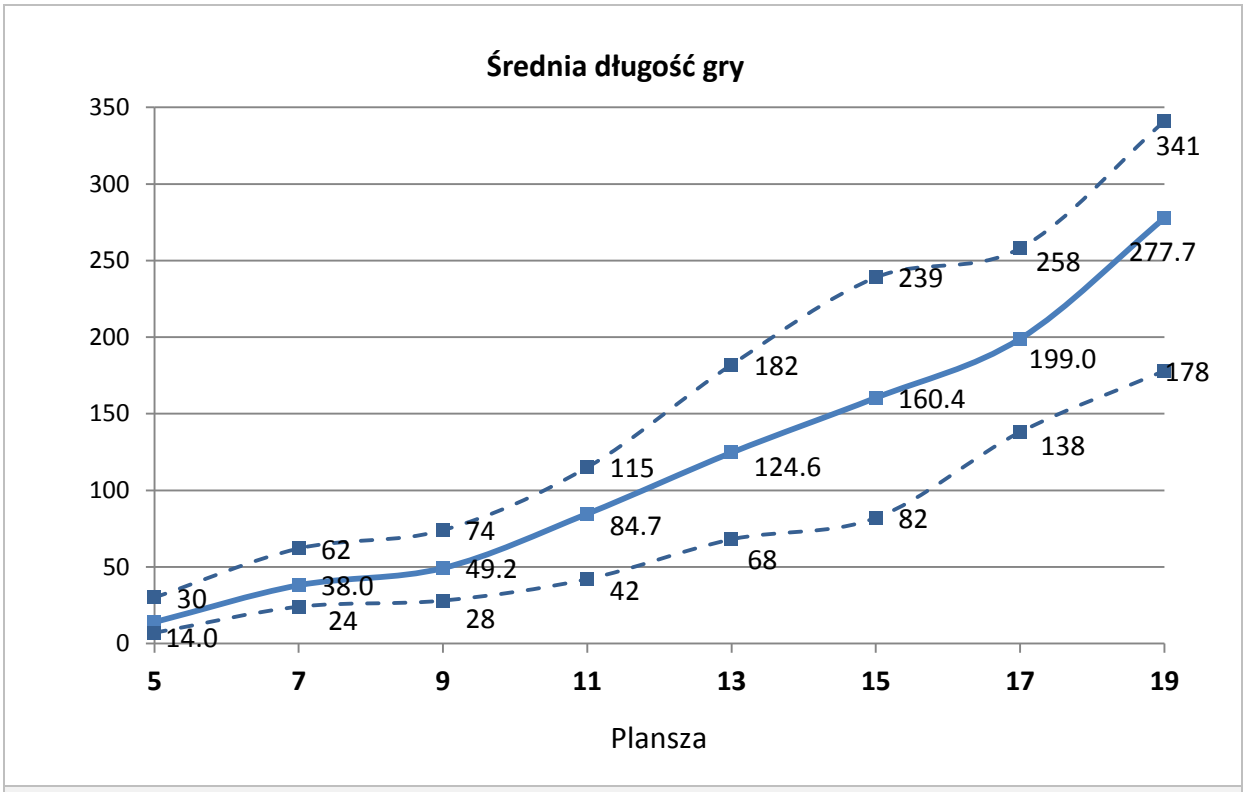
Rysunek 6.10. Siła gry programu w odniesieniu do programu referencyjnego GNU Go.

Na rysunku 6.11 przedstawiono długość gier rozgrywanych na różnych planszach w turnieju z programem GNU GO. Gry na planszy 5×5 miały średnią długość 14 ruchów. Liczba ruchów rosła razem z rozmiarem planszy i dla planszy 19×19 gry miały średnio 277 ruchów. Najkrótsza gra składała się 7 ruchów (5×5), najdłuższa z 341 (19×19).

Gry kończyły się zawsze z inicjatywy Czarnych (GNU GO). Czarne rezygnowały z gry (62% gier) albo jako pierwsze proponowały zakończenie gry (ruch PASS), na co Białe „wyrażały zgodę" również rezygnując z ruchu (38% gier). Białe wielokrotnie rezygnowały z ruchu (PASS), jednak Czarne nigdy nie opowiedziały ruchem PASS (najprawdopodobniej zakładając, że mają jeszcze szansę wygrać pojedynek).

Średnia liczba punktów przewagi rosła razem z rozmiarem planszy. Dla planszy 5×5 miała wartość 4.7 punktu i dla planszy 19×19 – 23.6 punktu. Minimalna przewaga miała wartość 0.5 punktu, maksymalna − 96.5.

Na uwagę zasługuje liczba zbitych kamieni. W tej kategorii program GNU GO okazał się dużo bardzie skuteczny niż program GO GAME. Dla każdego rozmiaru planszy liczba zbitych kamieni białych była około trzykrotnie większa niż liczba zbitych kamieni czarnych. Można postawić hipotezę, że heurystyki w programie GNU GO w pierwszej kolejności starają się bronić (Atari – obrona) a dopiero później atakować (Atari – atak), odwrotnie niż jest to w programie GO GAME.



Rysunek 6.11. Średnia długość gry w turnieju z programem GNU Go.
(Liniami przerywanymi pokazano najdłuższe i najkrótsze rozgrywki.)

6.4.6. Testy wpływu strategii UCT na siłę gry programu

W tej grupie testów porównano siłę gry programu z użyciem strategii przeszukiwania UCT i programu używającego do oceny pozycji prostej symulacji Monte-Carlo. Testy były przeprowadzone na planszy 9×9. Rozegrano dwa turnieje z programem GNU GO, każdy składający się z 50 pojedynków. W pierwszym turnieju program GO GAME korzystał ze strategii UCT, w drugim z oceny pozycji Monte-Carlo. W obu turniejach limit czasu był ustawiony na 5 sekund. Wyniki przedstawiono w tabeli 6.3. Wyniki potwierdzają wcześniejsze oczekiwania, że strategia UCT radykalnie wzmacnia siłę gry programu.

Tabela 6.3. Porównanie siły gry programu dla strategii UCT i funkcji oceny Monte-Carlo.

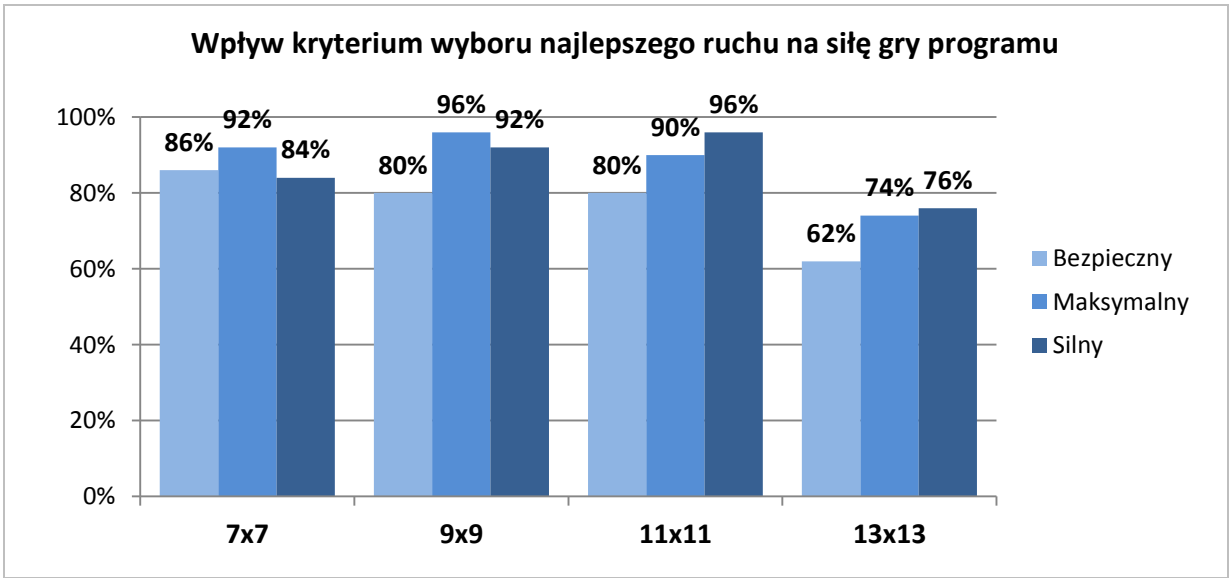
	Monte-Carlo		UCT	
Liczba gier wygranych przez Czarne	31	62%	4	8%
Liczba gier wygranych przez Białe	19	38%	46	92%

6.4.7. Testy kryterium wyboru najlepszego ruchu

W tej grupie testów sprawdzono, jaki jest wpływ kryterium wyboru najlepszego ruchu na siłę gry programu. W tym celu rozgrano mecz z programem GNU GO składający się z 12 turniejów. Turnieje były rozgrywane na planszach o rozmiarach od 7×7 do 13×13 punktów dla każdego z trzech kryteriów wyboru najlepszego ruchu (Maksymalnego, Silnego, Bezpiecznego). W każdym turnieju rozgrano 50 gier z limitem czasu na „myślenie” nad ruchem równym 5 sekund. Wyniki przedstawiono w tabeli 6.4 oraz zilustrowano na rysunku 6.12.

Tabela 6.4. Wpływ kryterium wyboru najlepszego ruchu na siłę gry programu.

		Plansza							
		7 × 7		9 × 9		11 × 11		13 × 13	
Bezpieczny	Liczba gier wygranych przez Czarne	7	14%	10	20%	10	20%	19	38%
	Liczba gier wygranych przez Białe	43	86%	40	80%	40	80%	31	62%
	Liczba punktów zdobytych przez Czarne	21.5	7%	73	15%	73	14%	218.5	36%
	Liczba punktów zdobytych przez Białe	283.5	93%	428	94%	455	92%	387.5	84%
Maksymalny	Liczba gier wygranych przez Czarne	4	8%	2	4%	5	10%	13	26%
	Liczba gier wygranych przez Białe	**46**	**92%**	**48**	**96%**	45	90%	37	74%
	Liczba punktów zdobytych przez Czarne	20	7%	27	6%	37.5	8%	76.5	16%
	Liczba punktów zdobytych przez Białe	283.5	93%	428	94%	455	92%	387.5	84%
Silny	Liczba gier wygranych przez Czarne	8	16%	4	8%	2	4%	12	24%
	Liczba gier wygranych przez Białe	42	84%	46	92%	**48**	**96%**	**38**	**76%**
	Liczba punktów zdobytych przez Czarne	52	13%	29	6%	33	5%	231	34%
	Liczba punktów zdobytych przez Białe	335	87%	459	94%	579	95%	451	66%
Najbardziej efektywne kryterium		Maksymalny		Maksymalny		Silny		Silny	
Najmniej efektywne kryterium		Silny		Bezpieczny		Bezpieczny		Bezpieczny	



Rysunek 6.12. Wpływ kryterium wyboru najlepszego ruchu na siłę gry programu.

Dla małych plansz 7×7 oraz 9×9 najbardziej efektywnym kryterium jest wybór węzła, który ma największą wartość stosunku liczby zwycięstw do liczby odwiedzin − Maksymalny. Dla średnich plansz 11×11 oraz 13×13 najbardziej efektywnym kryterium jest wybór węzła, który ma największą liczbę odwiedzin − Silny. Dla plansz 9×9, 11×11 oraz 13×13 zdecydowanie najmniej efektywnym kryterium okazał się wybór węzła Bezpiecznego mającego maksymalną dolną granicę ufności (maksymalną wartość formuły $v + A/\sqrt{n}$, gdzie A jest stałą dobieraną doświadczalnie, v jest wartością węzła, n liczbą odwiedzin).

7. Podsumowanie pracy

Starożytna orientalna gra w Go od dawna uważana jest za wielkie wyzwanie dla metod Sztucznej Inteligencji. Gra w Go, ze względu na ogromną przestrzeń przeszukiwania, duży współczynnik rozgałęzienia, opóźnione skutki działania oraz trudną do skonstruowania funkcję oceny, uosabia wyzwania stojące przed rzeczywistymi sekwencyjnymi problemami decyzyjnymi. W ostatnich latach komputerowa gra w Go korzysta z nowego paradygmatu bazującego na drzewie przeszukiwania z wykorzystaniem metod Monte-Carlo. Programy komputerowe, w których zaimplementowano drzewo przeszukiwania Monte-Carlo mogą grać na poziomie mistrzowskim i zaczynają kwestionować prymat profesjonalnych graczy.

Podstawowy cel pracy – prezentacja i weryfikacja algorytmów przeszukiwania drzewa Monte-Carlo przy wykorzystaniu zbudowanego programu komputerowego do gry w Go − został zrealizowany. W pracy przedstawiono i przebadano podstawowe elementy algorytmu MCTS oraz jego rozszerzenia.

Program komputerowy GO GAME umożliwia grę człowiek − komputer na wszystkich rozmiarach plansz (od 5×5 do19×19 punktów), daje możliwość gry z wyrównaniem oraz pozwala na wybór liczby punktów *komi*. Siłę gry programu można regulować przez zmianę limitu czasu na „myślenie" nad ruchem i zmianę liczby wątków używanych przez program podczas symulacji. Można eksperymentować, jaki wpływ na siłę gry programu ma kryterium wyboru najsilniejszego ruchu. Można też porównać jak strategia przeszukiwania UCT wzmacnia siłę gry programu w stosunku do prostej oceny pozycji na podstawie symulacji Monte-Carlo. Moduł turniejów pozwala na automatyczne rozgrywanie wielu gier z programem referencyjnym GNU GO (pojedynki komputer − komputer) i akwizycję danych o rozgrywkach. Zebrane dane mogą być użyte do analiz statystycznych gier.

Zbudowany program jest dobrym środowiskiem testowym, które zachęca do przeprowadzania kolejnych eksperymentów. W pracy wykonano tylko podstawowe testy, których rezultaty potwierdziły jedynie wcześniejsze oczekiwanie. W dalszych pracach warto rozbudować moduł parametryzacji programu w taki sposób, aby za pomocą przeprowadzanych eksperymentów okrywać nowe, niezbadane jeszcze możliwości algorytmów bazujących na drzewie przeszukiwania MCTS. Niestety w tej pracy tego celu nie udało się osiągnąć.

Paradygmat drzewa gry Monte-Carlo jest oryginalnym podejściem pozwalającym na efektywne przeszukiwanie wielkich przestrzeni i który jest stosowany nie tylko do rozwiązywania problemów w grach kombinatorycznych, ale także wspomagania decyzji, sterowania, problemów optymalizacyjnych i innych, które mogą być zapisane w postaci par: stan i decyzja i w których sekwencja decyzji prowadzi do stanu terminalnego, w którym graczowi wypłacana jest nagroda. Algorytmy przeszukiwania bazujące na algorytmie UCT (i jego licznych wariantach) są bardzo obszerną dziedziną wiedzy, nie do końca jeszcze zbadaną, która daje możliwość realizowania i testowania wielu nowatorskich pomysłów.

Bibliografia

[1] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, Olivier Teytaud, *The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions*, Communications of the ACM, Vol. 55, No. 3, 2012, pp. 106-113

[2] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton**,** *A Survey of Monte Carlo Tree Search Methods*, IEEE Transactions On Computational Intelligence and AI in Games, Vol. 4, No. 1, 2012, pp. 1-43

[3] Tristan Cazenave i Nicolas Jouandeau, *Parallel Nested Monte-Carlo Search*, 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009, pp. 1-6

[4] Erik van der Werf, *AI techniques for the game of Go*, Universitaire Pers Maastricht, 2004

[5] David Silver, *Reinforcement Learning and Simulation-Based Search in Computer Go*, University of Alberta, 2009

[6] Guillaume Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk and Bruno Bouzy, *Monte-Carlo Strategies for Computer Go*, In Proceedings of the 18th BeNeLux Conference on Artifcial Intelligence, 2006, pp. 83-90

[7] Tristan Cazenave i Nicolas Jouandeau, *On the Parallelization of UCT*, UniversiteitMaastrich, Proceedings of the Computer Games Workshop, 2007, pp. 93-101

[8] Guillaume Chaslot, Mark H. M. Winands, H. Jaap van den Herik, *Parallel Monte-Carlo Tree Search*, Computers and Games, 6th International Conference, Beijing, China, September 29 - October 1, 2008, pp. 60-71

[9] Kamil Marek Rocki, *Large Scale Monte Carlo Tree Search on GPU*, The University of Tokyo, 2011

[10] Peter Drake, Steve Uurtamo, *Heuristics in Monte Carlo Go*, International Conference on Artificial Intelligence, Las Vegas, Nevada, 2007, pp. 171-175

[11] David P. Helmbold, Aleatha Parker-Wood, *All-Moves-As-First Heuristics in Monte-Carlo Go*, International Conference on Artificial Intelligence, Las Vegas, Nevada, 2009, pp. 605-610

[12] Erik C. D. van der Werf, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, *Learning to score final positions in the game of Go*, Theor. Comput. Sci. 349(2), 2005, pp. 168-183

[13] Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, *Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search*, ACG, 2009, pp. 1-13

[14] David B. Benson, *Life in the game of Go*, Information Sciences, Vol.10 (1), 1976, pp.17-29

[15] Claude E. Shannon, *Programming a Computer for Playing Chess*, Philosophical Magazine Ser.7, Vol. 41, No. 314, 1950

[16] Sylvain Gelly, David Silver, *Monte-Carlo tree search and rapid action value estimation in computer Go*, Artificial Intelligence, Volume 175(11), 2011, pp. 1856-1875

[17] GNU Go − http://www.gnu.org/software/gnugo

[18] BOOST C++ Libraries − http://www.boost.org

[19] Qt Project − http://qt-project.org

[20] Kiseido Go Server − http://www.gokgs.com

[21] Computer Go Server − http://cgos.boardspace.net

Dodatek A. Słownik terminów

Podstawowe terminy z dziedziny gry w Go zostały zebrane w tabeli 7.1.

Tabela 7.1. Słownik dziedziny

Termin	Opis	J. Ang.
Atari	Kamienie, które mogą być zbite w wyniku jednego ruchu oponenta. Kamienie, które maja tylko jeden oddech.	
Baduk	Koreańska nazwa gry w Go.	
Dame	Punkty, które nie są kontrolowane przez graczy. Punkty neutralne.	
Dan	Jednostka miary używana do określenia siły gry mistrza.	
Drabina	Zbicie kamieni w wyniku sekwencji wielu ruchów.	Ladder
Gote	Ruch, który oddaje inicjatywę oponentowi, np. wewnątrz kontrolowanego przez gracza terytorium.	
Grupa	Luźno połączona grupa kamieni kontrolująca ten sam obszar.	Group
Handicap	Gra z wyrównaniem. Słabszy gracz, grający czarnymi, ustawia na planszy kilka kamieni przed rozpoczęciem gry. W pojedynkach amatorów, jeden stopień różnicy w sile gry (kyu/dan) odpowiada jednemu kamieniowi.	
Jigo	Remis	
Kamienie *martwe*	Kamienie, które nie mogą uniknąć zbicia są martwe. Po zakończeniu gry, przed obliczeniem wyniku, kamienie martwe są usuwane z planszy.	Dead stones
Kamienie bezwarunkowo *żywe*	Kamienie, które nie mogą być zbite są bezwarunkowo żywe. Kamienie bezwarunkowo żywe mają parę *oczu* albo są w *seki*.	Live stones
Kamienie *żywe*	Kamienie, które nie są *martwe* oraz nie są bezwarunkowo *żywe*.	Alive stones
Ko	Reguła *ko* zabrania graczom natychmiastowego „odbicia” kamienia. Przeciwnik musi zagrać najpierw w innym punkcie planszy. Ta reguła obowiązuje jedynie, gdy zbijamy i „odbijamy” tylko jeden kamień.	
Komi	Określona liczba punktów dodawana podczas oceny wyniku gry do liczby punktów zdobytych przez białe jako rekompensata za rozpoczęcie gry przez czarne. Zwykle *komi* jest równe 6.5 punktu. Ułamek punktu ma na celu unikanie remisu.	
Kyu	Jednostka miary używana do określenia siły gry ucznia.	

Moyo	Terytorium oraz "prawie" terytorium, tzn. fragment planszy, który może się łatwo stać terytorium, jeżeli oponent zlekceważy zagrożenie.	
Łańcuch	Zbiór sąsiadujących kamieni w jednym kolorze.	Chain, String
Obszar	Zbiór punktów na planszy składający się z terytorium i otaczających go kamieni w jednym kolorze.	Area
Oddech	Pusty punkt, który sąsiaduje z kamieniem.	Liberty
Oko	Terytorium otoczone przez kamienie w jednym kolorze zawierające co najmniej jedną oddech.	Eye
Punkt	Przecięcie linii planszy. Węzeł siatki. Miejsce, w którym kładziony jest kamień.	Point
Punkty sąsiednie	Dwa punkty na planszy połączone linią, pomiędzy którymi nie ma innego punktu.	
Ruch samobójczy	Ruch poświęcenia w celu uzyskania lepszej pozycji.	Suicide
Seki	Dwa lub więcej bloków żywych kamieni, które współdzielą jeden lub więcej oddechów i które nie mają pary *oczu*.	
Sente	Ruch, na który oponent musi odpowiedzieć, aby zachować inicjatywę.	
Super Ko	Reguła zabraniająca powtórzenia tej samej pozycji wszystkich kamieni na całej planszy. Została ona wprowadzona, aby uniemożliwić nieskończone powtarzanie sytuacji w pozycjach, których nie obejmuje zwykła reguła *Ko*.	
Terytorium	Punkty otoczone przez kamienie w jednym kolorze i kontrolowane przez jednego z graczy.	Territory
Weiqi* / *Weichi	Chińska nazwa gry w Go.	
Wpływ	Gracz ma wpływ w regionie, jeśli jest to bardzo prawdopodobne, że jest on zdolny do utworzeniu w tym regionie swojego terytorium.	Influence

Dodatek B. Wyniki najlepszych programów komputerowych

W dodatku B przedstawiono historyczne wyniki najlepszych programów komputerowych do gry w Go w pojedynkach przeciwko profesjonalnym graczom.

Pierwszym programem, w którym zastosowano algorytm MCTS był MoGo. Program MoGo był też pierwszym, w którym zastosowano algorytm RAVE oraz bazę wzorców w połączeniu z techniką zajmowania jak największego terytorium. W 2007 roku program MoGo na planszy 9 × 9 jako pierwszy program komputerowy pokonał profesjonalnego gracza (5 *dan*), w grze błyskawicznej z limitem czasu, po 10 minut dla każdego z graczy. Osiągnięcie to zostało powtórzone w 2008 roku, z wydłużonym czasem gry. W 2008 roku, na planszy 19 × 19 program MoGo wygrał mecz z profesjonalnym zawodnikiem (8 dan). Jednak wynik ten został osiągnięty z handicapem 9 kamieni. W tym samym roku, program Crazy Stone wygrał dwa mecze przeciwko profesjonalnemu graczowi (4 dan), z 8 i 7 handicapowymi kamieniami. W 2009 roku program MoGo wygrał z jednym z najlepszych graczy zawodowych (9 dan) w pojedynku z 7 kamieniami handicapowych. Program Fuego w wersji z przetwarzaniem wielowątkowym pokonał na planszy 9 × 9 profesjonalnego mistrza (9 *dan*) i na planszy 19 × 19 w grze z 4 kamieniami handicapowymi − mistrza amatorskiego (6 *dan*). Program Erica, który zdobył złoty medal na olimpiadzie w 2010 roku na planszy 19 × 19 jest aktualnie oceniany, na serwerze KGS (Kiseido Go Server), na poziomie 3 *dan*. W tabeli 7.1 pokazano ranking *Elo* najlepszych programów do gry w Go na planszy 9 × 9 na serwerze CGO (Computer Go Server) [16].

Tabela 7.2. Ranking najlepszych programów na planszy 9 × 9 na serwerze CGO.

Rok	Program	Zastosowane techniki	*Elo*
2006	Indigo	Baza wzorców, symulacja Monte-Carlo	1400
2006	GNU Go, Many Faces	Baza wzorców, algorytm alfa-beta	1800
2007	MoGo, Crazy Stone	Algorytmy MCTS i RAVE	2500
2008	Fuego	Warianty heurystyki MC-RAVE	2700
2010	Many Faces, Zen	Warianty heurystyki MC-RAVE	2700
2011	Erica	Warianty heurystyki MC-RAVE	2800
2012	Zen	Warianty heurystyki MC-RAVE	2900

Warto na dwóch przykładach pokazać jaki postęp dokonał się w algorytmach programów grających w Go. Przed pojawieniem się programów korzystających z technik Monte-Carlo, najsilniejsze „klasyczne programy" np. Many Faces of Go zostały ocenione na serwerze KGS na 6 *kyu* (stopień średnio zawansowanego gracza amatorskiego). Na początku 2012 roku najwyżej notowanym na serwerze KGS był program Zen, który osiągnął poziom 4 dan amatorów. Jak można się domyślać program korzystał z metod MCTS.

Najsilniejsze programy Many Faces of Go i Aya stosujące wcześniej „tradycyjne" podejście polegające na korzystaniu w szerokim zakresie z eksperckiej wiedzy dziedzinowej i rozbudowanej bazy wzorców, aktualnie także korzystają z algorytmu MC-RAVE (co można zauważyć po sile ich gry). Program Many Faces of Go jest aktualnie sklasyfikowany na poziomie 2 *dan* na serwerze KGS, program Aya na poziomie 1 *dan*.

W chwili edycji tej pracy (lipiec 2013), na serwerze CGO najwyżej sklasyfikowanymi programami były: na planszy 19 × 19 − Crazy Stone 3048 *Elo* i Zen 2935 *Elo* oraz na planszy 9 × 9 – Fuego 3147 *Elo* i Crazy Stone 3095 *Elo*.

Dodatek C. Opis dołączonej płyty CD

Dołączona płyta CD zawiera:

Tekst pracy dyplomowej.

Załącznik 1. Dokumentacja projektu: m.in. opis struktura rozwiązania, ustawienia parametrów, biblioteki, konwencje nazewnictwa zmiennych.

Załącznik 2 Kod źródłowy projektu (wersja Visual Studio 2010) i biblioteki.

Załącznik 3. Instrukcja użytkownika programu.

Załącznik 4. Dokumentacja przeprowadzonych testów.

Tekst pracy dyplomowej, dokumentacja projektu i instrukcje zostały zapisane w formatach Microsoft Word 2010 i PDF. Wyniki testów zostały zapisane w formacie MS Excel 2010 oraz w CSV.