Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Robert Wiatr

Implementacja zaawansowanych strategii gry Reversi w C# dla platformy .NET. Projektowanie modułów AI i gra z użyciem połączenia sieciowego.

Praca inżynierska wykonana w Zakładzie
Mechaniki Kwantowej
Opiekun: **dr Jacek Matulewski**

Toruń 2007

Panu dr Jackowi Matulewskiemu za opiekę,
wskazówki i pomoc w trakcie pisania tej pracy.
Serdecznie dziękuję!

Spis treści

1. Wstęp 4
2. Historia gry 5
3. Reguły gry 6
4. Możliwe strategie gry 7
4.1 Strategia maksymalnej ilości pionów 8
4.2 Strategia stabilnych pionów 9
4.3 Strategia wartości pól 10
4.4 Strategia klinowania 10
4.5 Strategia mobilności 11
4.6 Strategia kontroli centrum 12
4.7 Strategia ruchów oczekujących (tempa) 13
4.8 Strategia pełzania wzdłuż krawędzi 14
4.9 Strategia parzystości 15
4.10 Atak na niezrównoważone krawędzie 16
4.11 Pułapka stonera 17
5. Algorytm Mini – Max 18
5.1 Odcięcie Alfa – Beta 21
6. Implementacja programu 24
6.1 Implementacja strategii mobilności 25
6.2 Implementacja strategii klinowania 28
6.3 Gra sieciowa 32
7. Podsumowanie 38
8. Literatura 39
Dodatek A.: Instrukcja obsługi programu 40
A.1 Wstęp 40
A.2 Interfejs użytkownika (główna forma) 40
A.3 Tryby gry 41
A.4 Konfiguracja strategii komputera 42
A.5 Gra sieciowa 44
A.6 System podpowiedzi 45

1. Wstęp

Niniejsza praca zawiera opis kilku najbardziej rozpowszechnionych strategii wykorzystywanych podczas gry Reversi oraz opis ich implementacji w języku C# przy użyciu narzędzi Microsoft Visual Studio 2005 Express Edition w oparciu o biblioteki.NET Framework 2.0. Wraz z pracą powstała gra, która pozwala nie tylko na grę dwóch osób siedzących przy tym samym komputerze, ale również grę poprzez sieć oraz grę z wirtualnym graczem (botem), a także grę pomiędzy dwoma botami. Praca polegała na rozwinięciu pierwotnego projektu gry (przygotowanego przez dr Jacka Matulewskiego) poprzez dodanie zaawansowanych strategii oraz umożliwienie gry przez sieć.

2. Historia gry

Reversi znana także jako Otello (ang. *Othello*) jest logiczną grą planszową przeznaczoną dla dwóch graczy. Plansza składa się z 64 pól w układzie osiem na osiem. Gra zyskała dużą popularność ze względu na proste reguły oraz krótki czas rozgrywki. Jak twierdzą znawcy, pomimo że jej zasady są łatwiejsze od warcabów, pod względem bogactwa kombinacyjnego, zawiłości strategii i taktyki dorównuje takiej klasycznej grze jak szachy, znacznie przewyższając ją dynamiką rozgrywki [2]. Grę Otello określa się mianem europejskiej wersji Go. Reversi została stworzona pod koniec XVIII wieku. Pierwszy raz opublikowana została przez Lewisa Watermana w Wielkiej Brytanii około roku 1888. Część zasad gry Waterman zaczerpnął z pomysłu Jamesa Molletta, który opublikował w roku 1870 bardzo podobną grę o nazwie "The Game of Annexation" [3].

Do Polski Reversi dotarła już w XVIII wieku. Nigdy jednak nie uzyskała większej popularności. Współcześnie na początku lat 70 XX wieku zauważona i opisana przez Marka Penszko w miesięczniku "Problemy". Od tamtego momentu dość regularnie odbywają się w Polsce turnieje graczy [2].

W 1971 roku japończyk Goro Hasegawa opracował dobrze znaną dziś grę Otello, pod względem reguł bardzo podobną do Reversi. W kilkunastu krajach powstały federacje tej gry, a od roku 1977 odbywają się corocznie mistrzostwa świata. Około roku 1979 powstała pierwsza implementacja Otello na komputery osobiste. Powstawały federacje sympatyków Otello. Wszystko to powodowało, iż implementacje gry stawały się coraz bardziej zaawansowane.

Pod koniec lat 80 ubiegłego wieku firma Microsoft dołączyła grę Reversi do systemu Windows 3.0, co wpłynęło na znaczący wzrost jej popularności. Jednakże do kolejnych wersji systemu Windows gra nie była już dołączana. Wersja sieciowa gry Reversi ponownie pojawiła się dopiero razem z systemem operacyjnym Windows XP. W 1996 roku zainteresowali się nią programiści Javy [3].

3. Reguły gry

Gra rozgrywana jest na kwadratowej jednokolorowej planszy o wymiarach osiem na osiem pól. Uczestniczy w niej dwóch graczy używających brązowych i zielonych pionów. Spotykane są również zestawy z pionami czarnymi i białymi. Celem gry jest zajęcie większej liczby pól niż przeciwnik. Koniec gry następuje w sytuacji, gdy plansza zostaje całkowicie zapełniona pionkami, lub ewentualnie, gdy żaden z graczy nie może wykonać ruchu. Rozgrywkę rozpoczyna się gdy, zajęte są już cztery centralne pola, jest to tzw. pozycja startowa (patrz rysunek 1). Czarne krzyżyki znaczą grupę pól, na które możliwe jest postawienie pionka przez „brązowego" gracza.

Rysunek 1. **Początkowe ułożenie pionków oraz zaznaczenie możliwych ruchów gracza „Brązowego".**

Przyjęło się, iż partię rozpoczyna gracz mający brązowe pionki, lecz nie jest to sztywna zasada. Kolejne ruchy wykonuje się przemiennie, dostawiając kolejne piony swego koloru, oczywiście po jednym w każdym ruchu. W sytuacji, gdy któryś z grających nie może wykonać ruchu jego przeciwnik rusza się aż do momentu, gdy gracz ten odzyskuje możliwość wykonania posunięcia. Ruch jest prawidłowy i może zostać wykonany wówczas, gdy pomiędzy stawianym pionkiem a już stojącymi pionkami danego gracza (w poziomie, w pionie lub po skosie) znajduje się jeden lub więcej pionków przeciwnika oraz nie ma pól pustych.

Rysunek 2. **Wygląd planszy po ruchu gracza „Brązowego" (na pole *F5*), możliwe ruchy „Zielonego".**

W wyniku wykonania ruchu pionki przeciwnika znajdujące się pomiędzy już stojącymi a aktualnie stawianym pionkiem ulegają przejęciu, co oznacza zmianę ich koloru na kolor gracza wykonującego ruch. Rysunek 2 przedstawia sytuację na planszy po zajęciu przez brązowego pola ***D3***, sygnalizuje to niebieska kropka w lewym górnym rogu tego pola. W wyniku tego przejął on pionka przeciwnika z pola ***D4***. Teraz krzyżyki sygnalizują możliwe ruchy gracza „Zielonego” [2].

4. Możliwe strategie gry

Każdą rozgrywkę można podzielić na trzy etapy: otwarcie, środek gry oraz zakończenie. Plansza zawiera 64 pola, przy czym cztery centralne są od początku zajęte. Daje to możliwość wykonania maksymalnie 60 ruchów. Oczywiście nie zawsze każdy z graczy wykona ich dokładnie połowę. Bywają oczywiście sytuacje, gdy gra zakończy się przed zapełnieniem całej planszy. Upraszczając każdy z etapów trwa 20 ruchów, jednak eksperci twierdzą, że etapy otwarcia i zakończenia zwykle wymagają mniej ruchów. Zależnie od etapu rozgrywki można stosować różne strategie, również często opłaca się zmieniać je wielokrotnie podczas tego samego etapu [4].

Dla wygody opisu strategii zastosowano uproszczone oznaczenie poszczególnych pól planszy. Reprezentuje to rysunek 3.

Rysunek 3. **System oznaczeń pól planszy.**

	A	B	C	D	E	F	G	H
1		C	A	B	B	A	C	
2	C	X	D	D	D	D	X	C
3	A	D					D	A
4	B	D					D	B
5	B	D					D	B
6	A	D					D	A
7	C	X	D	D	D	D	X	C
8		C	A	B	B	A	C	

Poza rogami (pola ***A1***, ***H1***, ***A8*** oraz ***H8***) istotną rolę odgrywają pola ***X*** – wyjątkowo niebezpieczne (patrz rozdział 4.3), pola ***C*** – równie niebezpieczne jak ***X***, oraz pola ***A***, ***B*** i ***D***. Często stosuje się nomenklaturę kierunków geograficznych np.:

- wiersz ***1*** określa się jako krawędź północna
- wiersz ***8*** - południowa,
- kolumna ***A*** - zachodnia,

- kolumna ***H*** - wschodnia.

Podobnie rogi:

- ***A1*** – północno zachodni
- ***H1*** – północno wschodni
- ***A8*** – południowo zachodni
- ***H8*** – południowo wschodni [5].

4.1 Strategia maksymalnej ilości pionów

W grze Otello zwycięzcą zostaje ten z uczestników gry, który pod koniec partii zgromadził więcej pól. Początkujący gracze często zajmują takie pola by zawsze zdobyć możliwie największą liczbę pól. Takie zachowanie nazywamy strategią maksymalnej ilości pionów (maksymalizacji, zachłanną). Gdy obaj przeciwnicy są nowicjuszami taka strategia może wydawać się skuteczna, lecz gdy w grze będzie uczestniczył nawet średnio doświadczony gracz łatwo zauważyć, że strategia taka nie jest dobra.

Rysunek 4. **Przykładowe ustawienie pionów ilustrujące wadę strategii maksymalizacji.**

Rysunek 4 przedstawia dosyć mało prawdopodobną sytuację, w której gracz zielony nie może wykonać żadnego ruchu, w związku z tym gracz brązowy wykona najbliższy ruch - zajmie jeden z rogów. Dalsze ruchy również należeć będą do gracza brązowego. W efekcie zajmie więc wszystkie rogi. Ostatecznie partię wygrywa „brązowy” zajmując 40 z 64 pól (patrz rysunek 5), mimo że na cztery ruchy przed zakończeniem gry „zielony” posiadał 59 pól. Wynika z tego prosty wniosek, iż posiadanie dużej ilości pionów, nawet bardzo blisko końca gry, wcale nie gwarantuje ostatecznego zwycięstwa. W tym przykładzie zielony rzeczywiście posiada wiele pionów, ale są one bezbronne i mogą być odwrócone przez przeciwnika.

Rysunek 5. **Końcowy wygląd planszy – zwycięstwo gracza „Brązowego”.**

Wynika stąd wniosek, że aby zagwarantować sobie zwycięstwo należy przede wszystkim dążyć do zdobycia takich pól, które nigdy nie będą przejęte przez przeciwnika. Ten temat zostanie opisany w rozdziale (4.2).

4.2 Strategia stabilnych pionów

Z lektury podrozdziału (4.1) wynika, że istnieją pewne pola, które po zajęciu już nigdy nie zostaną przejęte przez przeciwnika. Na pewno są to rogi. Postawiony tam pion staje się stabilny. Właściwość tą wykorzystuje się w strategii stabilnych pionów [6]. Należy zauważyć, że gdy gracz oprócz rogu posiada przyległe do niego pole ***C*** również i ono staje się stabilne. Uogólniając wszystkie pola osłonięte przez inne nieodwracalne stają się również stabilnymi. Oczywiście często bardzo trudno jest zdobyć stabilne piony nawet na kilka ruchów przed końcem gry, lecz ich zdobycie często przesądza o zwycięstwie. Na rysunku 6 przedstawiony jest diagram ilustrujący pewną rozgrywkę, w której brązowy ma możliwość zajęcia pola ***H8*** uzyskując tym samym 23 stabilne pola i bardzo dobrą perspektywę na zwycięstwo.

Rysunek 5. **Przykładowe ustawienie pionów – gracz „Brązowy” może zdobyć róg *H8*.**

Rysunek 7 prezentuje wygląd planszy po zajęciu rogu. Przy dobrej grze brązowy może zdobyć nawet 33 stabilne piony.

Rysunek 6. **Wygląd planszy po zajęciu rogu H8, gracz zyskuje 23 stabilne pola.**

Zagranie „na róg” jest więc wyjątkowo ważne. Umożliwia ono stosowanie opisanej w rozdziale (4.7) strategii pełzania wzdłuż krawędzi, co prowadzi do łatwego zdobywania kolejnych stabilnych pól. Dlatego nigdy nie należy umożliwiać oponentowi zdobycia rogu, o ile nie ma możliwości wykonania innego ruchu. Metody unikania takich sytuacji opisane zostaną w rozdziale (4.3).

4.3 Strategia wartości pól

Jak już wcześniej wspomniałem szczególnie korzystne jest stawianie pionów na niektórych polach. Szczególnie wartościowe są narożniki oraz inne stabilne pola, które umożliwiają zdobycie nieodwracalnych pól na planszy. Z pewnymi wyjątkami wartościowe są również brzegi planszy. Analiza wielu gier wskazuje, że stawianie pionka na tzw. polach ***C*** oraz ***X*** przy pustym narożniku jest złym wyborem, gdyż umożliwia przeciwnikowi łatwe jego zdobycie. Oczywiście jak w każdej regule, tak i tu istnieją pewne wyjątki. Pomimo, że stawianie pionów w pola ***X*** jest wyjątkowo złym pomysłem, tak wybór pola ***C*** może niekiedy być dobrym wyjściem. Początkującemu graczowi odradza się jednak stawiania pionów w tych polach, jeżeli nie zajął jeszcze pobliskiego narożnika.

4.4 Strategia klinowania

Z uwagi na kluczowe znaczenie pól narożnikowych stawianie pionów na krawędziach jest bardzo istotne. Strategia klinowania [5] opisuje metodę pomagającą wykorzystywać błędy przeciwnika, które popełnił przy zajmowaniu pól na brzegach planszy. Na rysunku 8, rozpatrując południową krawędź widać, że jeśli brązowy zagra na ***C8*** zyskuje możliwość przejęcia rogu ***A8*** niezależnie od następnego ruchu zielonego. Wynika to z tego, iż brązowy pion postawiony na ***C8*** nie może być już nigdy odwrócony,

ponieważ otoczony już jest przez piony przeciwnika. O takim pionie mówi się, że jest zaklinowany. Inna sytuacja jest na krawędzi wschodniej. Tutaj brązowy stawiając piona na ***H6*** bądź ***H5*** nie na możliwości zdobycia rogu ***H8***, gdyż zielony może zagrać odpowiednio na ***H5*** lub ***H6*** eliminując brązowego z tej krawędzi. Stało się tak, ponieważ pola ***H5*** i ***H6*** nie spełniały warunków strategii klinowania.

Rysunek 7. **Ilustracja użycia strategii klinowania, „Brązowy" ma szanse na przejęcie rogu *A1* oraz *A8*.**

	A	B	C	D	E	F	G	H
1		zielony				zielony		
2			zielony	zielony	zielony	brązowy		
3			brązowy	brązowy	brązowy	zielony	brązowy	zielony
4			zielony	zielony	zielony	zielony	brązowy	zielony
5		zielony	zielony	brązowy	zielony	brązowy	zielony	
6			brązowy	zielony	zielony	brązowy	zielony	
7			zielony	brązowy	brązowy	brązowy		zielony
8		zielony		zielony	zielony	zielony		

W przypadku krawędzi północnej jeśli brązowy zagra na ***C1*** lub ***E1***, a zielony wybierze pola ***D1***, to brązowy może zagrać odpowiednio na ***E1*** lub ***C1*** uzyskując tym samym możliwość zdobycia rogu ***A1***.

Dochodzimy do wniosku, że jeżeli pomiędzy pionami tego samego koloru wystąpi nieparzysta liczba wolnych pól, to drugi z graczy może się tam zaklinować. A gdy liczba pól jest parzysta, klina można uniknąć. W tej metodzie rogi są najlepszymi polami do przejęcia, następnie pola ***B***, pola centralne mają wartość neutralną, natomiast pola ***C*** a zwłaszcza pola ***X*** są traktowane jako niebezpieczne. W każdym ruchu gracze zajmują pola o największej wartości (oczywiście, o ile jest to ruch dozwolony).

4.5 Strategia mobilności

Doświadczeni gracze starają się wykonywać takie ruchy by zmniejszyć liczbę możliwych ruchów oponenta. W szczególności dążą do zmuszenie przeciwnika by postawił piona na polach ***C*** lub ***X***, lub starają się pozbawić go możliwości ruchu (strata tury). Taką sytuację ilustruje rysunek 9, w którym brązowy poprzez postawienie piona w polu ***E8*** ogranicza ruchy zielonego do pola ***B2*** oraz ***G2***, które są wyjątkowo niedobrymi polami ***X***.

***Rysunek 9.* Ustawienie pionów ilustrujące zalety stosowania strategii mobilności – brak możliwości wykonania bezpiecznego ruchu przez „Brązowego”.**

W wyniku tego brązowy łatwo przejmuje odpowiednio róg ***A1*** lub ***H1***. Co gorsza zielony starci też północną krawędź, gdy niczym nieograniczony brązowy postawi piona również na drugim z północnych rogów.

Generalna zasadą tej strategii jest ograniczanie liczby możliwych posunięć przeciwnika i w tym samym czasie zwiększenie liczby swoich możliwych ruchów. Kiedy ten cel uda się osiągnąć mówimy, że mamy grę pod kontrolą. Należy jednak pamiętać, że konieczne jest, aby wymusić na przeciwniku zły ruch lub uniemożliwić mu dalszą grę. Nie wystarczy gdy przeciwnik ma tylko jeden ruch, ale taki, który nie prowadzi do jego przegranej.

4.6 Strategia kontroli centrum

Jest to strategia polegająca na ograniczaniu mobilności przeciwnika poprzez tworzenie tzw. granic. Oczywistym jest fakt, że im więcej jest wolnych pól sąsiadujących z pionem przeciwnika, tym więcej potencjalnych ruchów może wykonać gracz (gracz posiada większą mobilność). Pion sąsiadujący z wolnym polem nazywany jest pionem granicznym. Pozostałe piony zwane są wewnętrznymi. Zbiór sąsiadujących ze sobą jednokolorowych pionów granicznych nazywa się granicą. Aby ograniczyć mobilność przeciwnika należy posiadać dużo pionów wewnętrznych i mało granicznych.

Klasyczny przykład wykorzystania tej strategii ukazuje rysunek 10. Brązowy może tutaj zagrać na pole ***A6***, dzięki czemu nie tworzy żadnych pionów granicznych. Zielony, aby uniknąć stawiania pionów na polach ***C*** lub ***X*** wybierze ***F3***. Jednak nie uchroni go to od nieuniknionej przegranej. W następnym ruchu brązowy zagra na ***G3*** ostateczne zmuszając zielonego do zajęcia pola ***X*** lub ***C***. Niezależnie do wysiłków zielonego nieuniknione jest, iż brązowy (jeżeli będzie dobrze grał) zdobędzie wszystkie narożniki.

***Rysunek 10.* Układ pionków ilustrujący zasady stosowania strategii kontroli centrum.**

W tym przykładzie ruch na ***A6*** jest nazywany „doskonale cichym", ponieważ nie odwraca żadnego granicznego piona. Przykład ten uzmysławia nam, że ważne jest nie to gdzie się stawia pion, ale przede wszystkim to jakie piony zostaną odwrócone.

Jedną z metod redukcji mobilności przeciwnika jest unikanie odwracania zbyt wielu pionów granicznych. Lepiej mieć piony wewnętrzne, niż piony graniczne. W skrócie jest to strategia, która polega na uzyskaniu jak największej liczby pionów w centrum, a jak najmniejszej na granicy. Dzięki temu zwiększamy swoją mobilność.

4.7 Strategia ruchów oczekujących (tempa)

Strategia ta polega na zagraniu jednego ruchu więcej w danym obszarze planszy, by tym samym wymusić na przeciwniku przeniesienie gry w inny obszar planszy. Ruch taki prowadzi do wydłużenia granicy przeciwnika w tamtym obszarze. Realizacja tej strategii przedstawiona jest na rysunku 11.

***Rysunek 8.* Wydłużanie granicy przeciwnika poprzez stosowania strategii „tempa”.**

Obaj gracze na północy dzielą między siebie granicę. Oczywistym jest to, że żaden z nich nie ma ochoty właśnie tam stawiać swoich pionów. Wybór pada więc na część południową planszy. Jedynymi rozsądnymi ruchami brązowego wydają się być ***C8*** lub ***D7***. Jeżeli wybierze pierwszą możliwość zielony chcąc również uniknąć zagrania na północy a przez to powiększenia swej granicy wybierze oczywiście ***D7***. Widać wyraźnie, że brązowy

podjął złą decyzje (musi teraz zagrywać na północy), powinien był lepiej zagrać na ***D7***, gdy zielony zagra na ***C8*** wtedy brązowy wybierze ***B8*** zmuszając tym samym zielonego do zagrania na północy. W takiej sytuacji mówi się, że brązowy zyskał „***tempo***” w tej części planszy. Strategię tą można opisać jako zagranie jednego ruchu więcej niż przeciwnik w danym rejonie planszy (często na krawędzi), a przez to wymuszenie na przeciwniku inicjowania gry w innym miejscu (co w konsekwencji wydłuży jego granicę). W podobnym przypadku, lecz gdy na krawędzi południowej są tylko dwa piony, gracz tak zaatakowany może bronić się i o ile to możliwe i bezpieczne postawić piona w polu ***C***.

Są jednak pewne niebezpieczeństwa stosowania takiej strategii. Gracz, który stara się zdobywać tempo za wszelką cenę często może paść ofiarą strategii klinowania lub niezrównoważonych krawędzi.

4.8 Strategia pełzania wzdłuż krawędzi

Jest to strategia maksymalnie wykorzystująca regułę zdobywania tempa. Polega na wstępnym uzyskaniu maksymalnej liczby pól na jednej bądź dwóch przyległych do siebie krawędziach. Stosujący ją gracz stara się szybko pozbawić przeciwnika rozsądnych ruchów zostawiając mu całą granicę oraz centrum. Dobrze rozegrana strategia zmusza przeciwnika do zwiększania swoich granic, lub straty ruchu.

Rysunek 9. **Ustawienie pionków ilustrujące strategię pełzania wzdłuż krawędzi.**

Na rysunku 12, gdy brązowy zagra na ***H7*** zielony ratując się wybierze ***C7***. Lecz gdy brązowy zagra na ***C8*** zielony zostaje zmuszony do oddania rogu poprzez ruch na ***G7***.

Istnieją jednak zagrożenia wynikające ze stosowania takiej strategii. Jeżeli przeciwnik będzie miał więcej możliwości rozsądnych ruchów, może przetrzymać atak gracza, a nawet doprowadzić do wygranej poprzez zmuszenie go do postawienia pionów granicznych.

Pełzanie wzdłuż krawędzi jest strategią krótkoterminową. Aby ją wykorzystać należy ograniczyć przeciwnikowi mobilność i chociaż udane pełzanie wzdłuż krawędzi jest często gwarancją zwycięstwa to jego nieudana próba równie często kończy się nieuniknioną porażką.

4.9 Strategia parzystości

Zakładając, że rozgrywkę rozpoczyna zielony i żaden z graczy nie jest zmuszony do oddania tury wtedy po każdym ruchu brązowego na planszy będzie parzysta ilość pionów, a po każdym ruchu zielonego nieparzysta. Wynika z tego, że ostatni ruch wykona gracz brązowy – on powie ostatnie słowo. Stawia to gracza zielonego w gorszej pozycji.
Z natury rzeczy parzystość daje przewagę graczowi, który grał jako drugi.

Istnieje jednak możliwość zmiany tej sytuacji gdy jeden z graczy musi oddać ruch. Niestety wszystko wraca do normy, gdy jest parzysta liczba oddanych ruchów. Rozwiązaniem problemu jest zdobycie parzystości poprzez wymuszenie na przeciwniku utworzenia nieparzystego obszaru wolnych pól, do którego nie ma on dostępu. Taką sytuacje ilustruje rysunek 13.

Rysunek 10. **Utworzenie nieparzystego obszaru pól (pole G8), do którego nie ma dostępu „Brązowy".**

W takiej sytuacji oczywiście brązowy nie może zagrać na ***G8***, a zielony nie powinien tego robić. Może wykorzystać strategię parzystości i wymusić na brązowym obronę rogów. Nie licząc ***G8*** na planszy jest nieparzysta liczba wolnych pól. Jeżeli zielony nie zagra na ***G8*** będzie ono tymczasowo wyłączone z gry, gdyż nic nie zmusza zielonego do zagrania w to pole. Gracz zielony powinien tak to rozegrać, aby po jego ruchu pozostały tylko obszary o parzystej liczbie wolnych pól, czyli zagrać ***G2***. Teraz brązowy jest zmuszony grać jako pierwszy w każdy z dwu dwupolowych obszarów. Sekwencja

ruchów wygląda następująco: ***G2*** zielony – ***H1*** brązowy – ***G1*** zielony – ***A1*** brązowy – ***A2*** zielony, teraz brązowy musi oddać ruch a zielony kończy grę zagraniem na ***G8***.

4.10 Atak na niezrównoważone krawędzie

Do tej pory postawienie piona w polu ***X*** uważane było za zły wybór, jednak nie zawsze musi tak być. Rysunek 14 ilustruje taką sytuację.

Rysunek 11. **Ustawienie pionków ilustrujące zasadę ataku na niezrównoważone krawędzie.**

Gracze zajmują łącznie 50 pól, jednak nikt nie zajął żadnego z rogów. Brązowy wykonuje ruch. Jedynymi możliwymi ruchami są zagrania na ***B2***, ***B7***, ***G2***, ***G7*** oraz ***H2*** czyli wszystkie pola ***X*** oraz jedno pole ***C***. Każde z tych zagrań umożliwia graczowi zielonemu na przejęcie rogów, lecz jeden ruch może stworzyć możliwość wygranej dla brązowego. Po takim wprowadzeniu można wyjaśnić strategię niezrównoważonych krawędzi.

Typową krawędzią niezrównoważoną jest kolumna ***H*** (krawędź wschodnia). Taki układ pionów pozwala graczowi brązowemu łatwo przejąć róg ***H8***. Mimo że brązowy zmuszony jest umożliwić zielonemu przejęcie rogu, jednak w niedługiej perspektywie ma szansę zyskać przewagę. Brązowy stawiając na ***G2*** poświęca róg **H1**, właśnie tam najpewniej zagra zielony, uzyskując 7 stabilnych pionów w wierszu ***1***. Następnie brązowy klinuje na ***H2***, daje mu to możliwość przejęcia rogu ***H8*** a następnie ***A8***, a później nawet ***A1***. W najlepszym wypadku brązowy wygra stosunkiem pól 51 do 13.

Jednak nie każde takie poświecenie rogu zawsze przynosi wygraną. W powyższym przykładzie zielony wcale nie musi przejąć rogu ***H1***, lecz zagrać na jedno z pól w kolumnie ***A***.

4.11 Pułapka stonera

Prowokacja w postaci ataku na niezrównoważoną krawędź nie zawsze prowadzi do przejęcia upatrzonego narożnika, ponieważ atakowany gracz może zdecydować czy chce zająć taki oddany ruch.

Strategia omówiona w tym rozdziale opisuje sytuację w której zagranie na krawędzi gwarantuje zdobycie rogu. Jeżeli pułapka stonera jest dobrze zastawiona, atakowany nie jest w stanie uchronić się przed taką prowokacją. Taką sytuację przedstawia rysunek 15.

Rysunek 12. **Początkowe ustawienia pionków w pułapce Stonera.**

Zagrywający zielony może wykorzystać błąd w ustawieniu brązowego, który w wierszu 8 posiada słabą krawędź. Zielony początkowo zagrywa na ***B7*** i tym samym przejmuje kontrolę nad przekątną ***E4*** – ***B7***. Brązowy nie może przejąć pola ***A8***, więc prawdopodobnie będzie się starał uzyskać piona na przekątnej grając na przykład ***F3***.

Rysunek 13. **Ustawienie końcowe. „Zielony" ma pewność przejęcia któregoś z południowych rogów.**

Dzięki temu zielony może zaatakować na ***D8*** (rysunek 16), a brązowy nie ma możliwości uniknięcia oddania rogu. Mimo że prawdopodobnie postawi piona w polu ***B8***, tym samym przejmując wszystkie pola gracza zielonego w wierszu ***8***, to i tak ostatecznie odda róg ***A8***

oraz ***H8***. Z drugiej strony jeżeli nawet brązowy nie zagra ani ***A8*** ani ***B8***, zielony i tak może zająć róg ***H8***. Jest to zasadnicza różnica pomiędzy pułapką stonera a atakiem na niezrównoważoną krawędź – nieuchronność przejęcia rogu.

Reasumując pułapka stonera przebiega w dwóch etapach. Najpierw atakujący przejmuje kontrolę nad przekątną grając na pole ***X***, a następnie atakuje słabą krawędź przeciwnika, czyli krawędź która zawiera wolne pole ***C***, grożąc zajęciem rogu. Jakikolwiek ruch obronny atakowanego prowadzi nieuchronnie do oddania co najmniej jednego z atakowanych rogów.

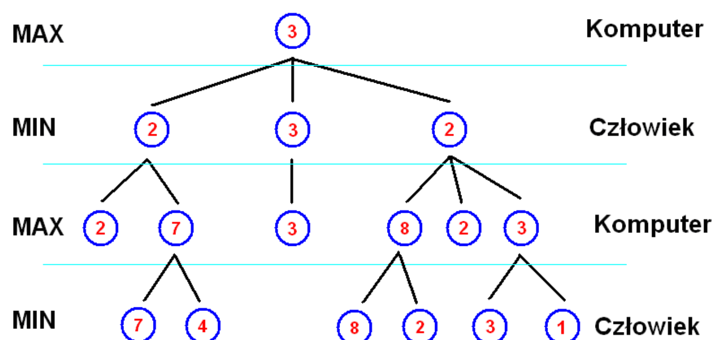
5. Algorytm Mini – Max

Algorytm Mini – Max jest wykorzystywany standardowo w różnorodnych grach turowych (tzn. takich, w których ruchy wykonuje się na przemian przez dwóch graczy), w których jednym z grających jest komputer. Jego typowym zastosowaniem są gry planszowe takie jak warcaby, szachy itp. Zadaniem tego algorytmu jest wybór ruchu zwiększającego prawdopodobieństwo zwycięstwa poprzez analizę wszystkich potencjalnych ścieżek rozwoju sytuacji na planszy.

W przypadku algorytmu Mini – Max komputer symuluje na przemian ruchy własne oraz przeciwnika. Jako ruch własny komputer wybiera najlepszy spośród dostępnych (MAX), zaś dla przeciwnika symuluje się ruch najbardziej dla siebie niekorzystny (MIN). Stąd bierze się nazwa algorytmu. Celem komputera będzie *n*–krotna naprzemienna symulacja najlepszych i najgorszych ruchów. Ilość powtórzeń *n* (nazywana także głębokością symulacji) jest przy tym parametrem zależnym od ustawień użytkownika. Po ukończonej symulacji algorytm zwraca stan gry.

Po utworzeniu listy możliwych posunięć, spośród wszystkich symulacji komputer wybierze najkorzystniejszy dla niego ruch. W odróżnieniu od opisanych wcześniej strategii, w algorytmie Mini – Max najlepszy ruch wybierany jest nie w oparciu o analizę bieżącej sytuacji na planszy, a na podstawie przewidywania jego konsekwencji po kilku kolejnych ruchach. Należy przy tym zwrócić uwagę, że do wyboru ruchów w kolejnych przewidywanych posunięciach wykorzystywane są opisane wcześniej strategie. Strategia Mini – Max jest więc *de facto* meta-strategią.

Rysunek 17. **Drzewo wszystkich możliwych ruchów aż do 3 poziomu włącznie.**



Rysunek 17 schematycznie ilustruje zachowanie algorytmu, którego zadaniem jest przeszukiwanie drzewa, w których mamy dwa typy węzłów Max oraz Min. Każdy z węzłów reprezentuje jeden z możliwych ruchów jakie gracz może wykonać. Oczywiście w grze typu Reversi możliwych posunięć może być dużo więcej. Wartości przypisane każdemu z tych węzłów to np. liczba pól przejętych w danym ruchu przez gracza. Liście drzewa symbolizują koniec gry. Mini – Max to algorytm rekurencyjny, który wykonuje możliwe ruchy określoną liczbę posunięć do przodu, po czym cofa się do wyższego poziomu. Na każdym z poziomów rozpatruje się wszystkie możliwe do wykonania ruchy. Bada się w związku z tym wielką ilość możliwych sytuacji, co dla odpowiednio dużej głębokości przeszukiwań staje się bardzo czasochłonne. Jednak istnieje sposób na ominiecie tego problemu, mianowicie odcięcie Alfa – Beta (podrozdział 5.1). Po utworzeniu listy możliwych ruchów, każdy z nich jest „wirtualnie" wykonywany, po czym zwracana jest liczba określająca sytuację gracza, który wywołał algorytm. W każdym węźle MIN preferuje się najbardziej niekorzystną sytuację, zaś w MAX najkorzystniejszą. Wybrana wartość zwracana jest „rodzicowi". Ostatecznie do najwyższego poziomu zwraca się wynik każdego przeszukania z każdej gałęzi. Wśród tych wyników komputer wybiera najlepszy.

Wartość danego ruchu w badanym (końcowym) węźle drzewa, wyliczona jest na podstawie jednej bądź kilku strategii, a uzależniona jest od globalnej sytuacji gracza na planszy. Sytuacja ta zależy od kilku czynników: liczby zajętych pól przez graczy, liczby rogów w posiadaniu każdego z nich, możliwości zajęcia lub straty któregoś z rogów, możliwości zastosowania strategii klinowania, liczby stabilnych pionów graczy, liczby pól X w posiadaniu każdego z graczy przy sąsiednim rogu wolnym, strategii kontroli centrum korzystającej ze strategii mobilności i parzystości, oraz sytuacji na krawędzi planszy. Czyli

do oceny sytuacji na planszy używanych jest wiele strategii, których dobór i odpowiednie wartościowanie wpływa ostatecznie na wybór właściwego ruchu.

Jeżeli wybór najlepszego ruchu odbywałby się na standardowych zasadach, czyli preferowania najbardziej wartościowego według danej strategii, wtedy mogło by to doprowadzić do błędnych wyników. Ponieważ brane by były pod uwagę tylko ruchy w „najgłębszych" poziomach drzewa, takie ruchy byłyby „wyrwane z kontekstu".

Przykładowo rozpatrując sytuację w dwóch różnych gałęziach A oraz B tego samego drzewa, może się zdarzyć sytuacja, gdy w danym węźle gałęzi A wykonywany jest ruch dzięki któremu istnieje szansa na zdobycie rogu, jak wiadomo takie posunięcia są preferowane, lecz jednak byłby on obarczony ryzykiem utraty innych cennych pól. Zaś w innym badanym węźle w gałęzi B wykonywany jest mniej ryzykowny ale także mniej zyskowny ruch. W związku z tym wybór najlepszego ruchu musi opierać się na ocenie sytuacji gracza, gdyż nie miałoby sensu posiadanie kilku rogów jeżeli sytuacja na planszy i tak nie dawałaby szans na wygraną.

Listingi od 1 do 3 zawierają pseudokod omawianego algorytmu. Przedstawiona w listingu 1 funkcja *minimax* uruchamia algorytm. W linii 4 znajdywane są wszystkie możliwe ruchy jakie w tej chwili może wykonać gracz. Spośród nich wybrany zostanie ten najefektywniejszy. Polecenie z linii 6 tworzy listę ruchów z ich wartościami obliczonymi w gałęziach drzewa. Ostatecznie w linii 8 funkcja zwraca informację o najlepszym możliwym ruchu tj. o tym ruchu w liście, który ma największą wartość.

Listing 1. **Metoda uruchamiająca algorytm Mini-Max.**

```
01    function minimax(stanGry)
02    {
03        // znalezienie wszystkich możliwych potomków
04        foreach dziecko in potomkowie(stanGry) do
05            // utworzenie listy wszystkich możliwych ruchów i ich wartości
06            Lista.Add(dziecko, węzełMax(dziecko));
07        // wyszukanie ruchu o maksymalnej wartości w drugim polu listy
08        return najlepszyRuch(Lista);
09    }
```

Wywoływana z funkcji *minimax* funkcja *wezelMax* dotyczy węzła w którym wybiera się gałąź o maksymalnej wartości tj. ruch najbardziej korzystny dla gracza wywołującego algorytm Mini – Max, a jednocześnie najmniej korzystny dla jego oponenta. W linii 14 listingu 2 znajduje się warunek sprawdzający czy warto kontynuować badanie drzewa. Jeżeli nie, to zwracana jest wartość informującą o stanie gry (np. kto wygrał). W linii 17 wyszukiwane są wszystkie możliwe ruchy następnego gracza, a linia 18 to wybór

najlepszego ruchu przeciwnika. Ostatecznie w linii 19 zwracana jest do funkcji *minimax* obliczona wartość gałęzi.

Listing 2. **Wybór gałęzi o maksymalnej wartości.**

```
10	function węzełMax(stanGry)
11	{
12		// Jeżeli to liść lub ograniczenie penetracji
13		// zwrot stanu gry
14		if ostatniBadanyWęzeł(stanGry)
15			return wartość(stanGry);
16		v = - ∞;
17		foreach dziecko in potomkowie(stanGry) do
18			v = Max(v,węzełMin(dziecko));
19		return v;
20	}
```

Funkcja *wezelMin* dotyczy węzła w którym wybierana jest gałąź o minimalnej wartości. Jej działanie jest zatem podobne do funkcji *wezelMax*, z tą różnicą że w linii 29 (listing 3) dokonuje się wyboru „najgorszego" ruchu.

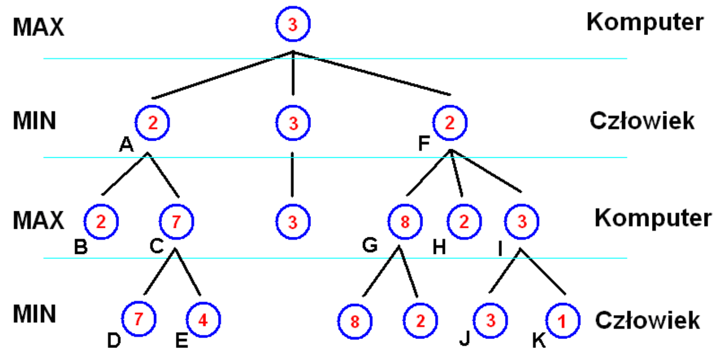
Listing 3. **Wybór gałęzi o minimalnej wartości.**

```
21	function węzełMin(stanGry)
22	{
23		// Jeżeli to liść lub ograniczenie penetracji
24		// zwrot stanu gry
25		if ostatniBadanyWęzeł(stanGry)
26			return wartość(stanGry);
27		v = + ∞;
28		foreach dziecko in potomkowie(stanGry) do
29			v = Min(v,węzełMax(dziecko));
30		return v;
31	}
```

5.1 Odcięcie Alfa – Beta

Przeszukiwanie całego drzewa jest procesem długotrwałym, tym dłuższym im głębiej się je bada. Oczywistą sprawą jest to, że niezależnie od rodzaju gry i doboru funkcji stanu, im głębiej przeszukujemy drzewo gry tym lepszym graczem jest nasz program, a ograniczeniem jest tylko czas i moc obliczeniowa komputera. Jest jednak sposób by ograniczyć zachłanność algorytmu. Rozwiązaniem jest odcięcie Alfa – Beta.

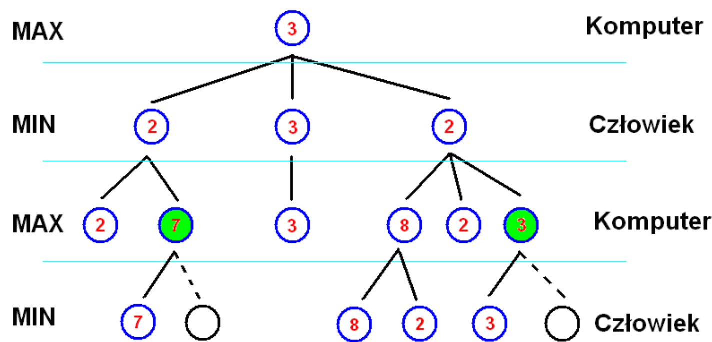
Rysunek 14. **Oznaczenia węzłów.**



Przyglądając się rysunkowi 18 można zauważyć ciekawą sytuację. Mianowicie w lewej gałęzi na poziomie MAX istnieje węzeł ***C*** o wartości 7. Jego rodzicem jest węzeł ***A*** zaś rodzeństwem węzeł ***B*** o wartości 2. Skoro rodzic to MIN więc algorytm na pewno wybierze 2 (***B***). Jednocześnie dla węzła algorytm ***C*** stara się wybrać maksimum spośród ***D*** i ***E***. W związku z tym można zauważyć że nie jest istotna wartość węzła ***E***. Gdyby ***E*** było większe niż 7, ***C*** przyjęłoby jego wartość, jednak węzeł ***A*** oczywiście pozostałby przy 2. Gdyby nawet wartość węzła ***E*** była mniejsza niż 2, wtedy węzeł ***C*** stanowił by barierę. Reasumując badanie węzła ***E*** nie miałoby żadnego znaczenia. W związku z tym na węźle C można dokonać odcięcia BETA (odcięcie dokonane na węźle MAX) – odcięcia wszystkich kolejnych gałęzi. Podobna sytuacja jest w gałęzi prawej w węźle I.

Rysunek 19 ukazuje sprawdzone węzły. Jak widać odcięcia nie wpłynęły na wynik algorytmu – w korzeniu drzewa nadal jest liczba 3.

Rysunek 15. **Ilustracja „odcinania" gałęzi, których sprawdzanie jest zbyteczne.**



Zajmowanie się analizę przypadków, które są w oczywisty sposób bezsensowne niepotrzebnie wydłuża działanie algorytmu. Koniecznym jest więc wprowadzenie zmian do pseudokodu. Tabela 1 przedstawia ilości przebadanych węzłów przy uwzględnieniu odcięcia Alfa – Beta oraz przy zwykłym Mini – Max'ie.

Tabela 1. **Liczba przebadanych węzłów w zależności od głębokości penetracji drzewa przy zastosowaniu odcięcia Alfa – Beta oraz w przypadku jego braku.**

Głębokość	Mini – Max + odcięcie Alfa – Beta	Mini – Max
1	4	4
2	18	18
3	63	79
4	200	428
5	926	2 478
6	2 897	16 251
7	12 238	113 805
8	43 163	865 135
9	163 649	7 007 933

Otrzymane wyniki uzyskane zostały po pierwszym ruchu gracza na pole ***C5***.

Listingi od 4 do 6 zawierają pseudokod algorytmu Mini – Max z uwzględnieniem odcięcia Alfa - Beta. Funkcja *minimax* (listing 4) zawiadująca całym algorytmem nie uległa istotnym zmianom:

Listing 4. **Metoda uruchamiająca algorytm Mini-Max.**

```
01    function minimax(stanGry)
02    {
03        // znalezienie wszystkich możliwych potomków
04        foreach dziecko in potomkowie(stanGry) do
05            // utworzenie listy wszystkich możliwych ruchów i ich wartości
06            Lista.Add(dziecko, węzełMax(dziecko, -∞, +∞));
07        // wyszukanie ruchu o maksymalnej wartości w drugim polu listy
08        return najlepszyRuch(Lista);
09    }
```

Różnice w stosunku do poprzedniej wersji funkcji *wezelMax* (listing 2) polegają na zmianie jej argumentów oraz dodaniu w listingu 5 linii od 20 do 22. W liniach tych następuje odcięcie BETA, o ile wartość zwrócona przez potomka jest większa lub równa argumentowi *beta* obliczonemu w węzłach nadrzędnych. W linii 22 następuje wyliczenie nowej wartości zmiennej *alfa*.

Listing 5. **Wybór gałęzi o maksymalnej wartości.**

```
10    function węzełMax(stanGry, alfa, beta)
11    {
12        // Jeżeli to liść lub ograniczenie penetracji
13        // zwrot stanu gry
14        if ostatniBadanyWęzeł(stanGry)
```

```
15                      return wartość(stanGry);
16              v = - ∞;
17              foreach dziecko in potomkowie(stanGry) do
18              {
19                      v = Max(v, węzełMin(dziecko, alfa, beta));
20                      if v >= beta then
21                              return v;
22                      alfa = Max(alfa, v);
23              }
24              return v;
25      }
```

Podobnie rzecz ma się z funkcją *wezelMin* (listing 6).

Listing 6. **Wybór gałęzi o minimalnej wartości.**

```
26      function węzełMin(stanGry, alfa, beta)
27      {
28              // Jeżeli to liść lub ograniczenie penetracji
29              // zwrot stanu gry
30              if ostatniBadanyWęzeł(stanGry)
31                      return wartość(stanGry);
32              v = + ∞;
33              foreach dziecko in potomkowie(stanGry) do
34              {
35                      v = Min(v, węzełMax(dziecko, alfa, beta));
36                      if v <= alfa then
37                              return v;
38                      beta = Min(beta, v);
39              }
40              return v;
41      }
```

6. Implementacja programu

W tym rozdziale omówione są implementacje dwóch wybranych strategii (mobilności oraz klinowania) oraz modułu odpowiedzialnego za realizację gry dwóch osób przez sieć. Kod źródłowy całego programu został umieszczony w kilku plikach źródłowych. W każdym z nich znajduje się osobny funkcjonalnie moduł gry. Plik *FormaGłowna.cs* zawiera implementację interfejsu gry, w tym moduł odpowiedzialny za inicjację planszy oraz obsługę kliknięć jej pól. Zasada działania tego modułu została opisana w książce Jacka Matulewskiego, *Język C#. Programowanie dla platformy .NET w środowisku Borland C# Builder* [10]. Kod zawarty w *FormaGłowna.cs* oraz *ReversiSilnik.cs* jest znacznym rozszerzeniem opisanego w tej książce kodu. Plik *Menu.cs* obejmuje wszystkie metody związane z obsługą menu aplikacji. Pliki *Siec.cs*, *Plik.cs* oraz *Symulacja.cs* zawierają moduły odpowiedzialne za komunikację sieciową między grającymi, zapisem oraz odczytem stanu gry z plików, a także grę między dwoma botami. Dzięki funkcji zapisu i odczytu stanu gry użytkownik ma możliwość odtworzenia każdej

partii na jej dowolnym etapie. Kolejne dwa pliki, *Polaczenie.cs* oraz *Ustawienia.cs*, zawierają klasy i metody wykorzystywane do konfigurowania adresów IP przeciwnika, a także wyboru strategii użytej przez komputer. Poza samą grą przez sieć program umożliwia także prowadzenie dyskusji wykorzystując wbudowany prosty komunikator internetowy. Nas jednak w szczególny sposób interesować będą pliki *ReversiSilnik.cs* oraz *Minimax.cs*, w których zaimplementowane zostały wszystkie strategie używane przez komputer.

6.1 Implementacja strategii mobilności

Jak wspomniano wyżej strategia mobilności polega na ograniczaniu liczby możliwych posunięć przeciwnika, a tym samym na zwiększeniu liczby własnych możliwych ruchów. Niezbędna jest zatem lista wszystkich dostępnych ruchów. Komputer tworzy ją przeglądając całą planszę, a dokładniej jej puste pola i sprawdzając czy dozwolone jest postawienie na nich własnych pionów.

We wierszu 1 listingu 7 tworzona jest lista do której wpisane będą wszystkie możliwe ruchy gracza wraz z ich wartościami. Jest to kolekcja typu *ArrayList*, która jest w stanie przechowywać referencję do klasy *Object*, na którą można zrzutować referencję do dowolnego obiektu. W tym przypadku lista przechowywać będzie instancje struktury (typu wartościowego) struktury *MozliwyRuch*. W liniach od 2 do 4 znajdują się polecenia organizujące pętlę przeszukującą wszystkie wolne pola na planszy. Wewnątrz niej (linia 6) następuje sprawdzenie, czy postawienie piona w wolnym polu jest zgodne z zasadami gry. Jeżeli tak, zmienna *priorytet* ustawiana jest na wartość większą od 0. Z kolei w linii 9 sprawdzana jest ilość możliwych ruchów, jakie będzie mógł zrealizować przeciwnik w kolejnym posunięciu – zgodnie z implementowaną strategią należy wybrać taki ruch, w którym liczba ruchów przeciwnika będzie maksymalnie ograniczona. W linii 10 tworzona jest struktura zawierająca informację o współrzędnych badanego pola oraz liczbie możliwych ruchów przeciwnika. Ponieważ zmienna *liczba*, która przechowuje ilość możliwych ruchów przeciwnika, będzie miała wartość 0 lub większą należy ją przemnożyć przez -1, celem późniejszego wybrania najmniej korzystnego dla przeciwnika ruchu. W wierszu 12 uzupełnia się listę możliwych ruchów gracza o kolejny wpis.

Listing 7. **Przeglądanie planszy i wybór najlepszego ruchu.**

```
01	ArrayList mozliweRuchy = new ArrayList();
02	for (int i = 0; i < planszaRozmiar; i++)
03		for (int j = 0; j < planszaRozmiar; j++)
04			if (plansza[i, j] == 0)
05			{
06				int priorytet = UstawPionek(i, j, true);
07				if (priorytet > 0)
08				{
09					int liczba = MobilnoscTest(i, j);
10					MozliwyRuch mr = new MozliwyRuch(i, j, liczba);
11					mr.priorytet *= -1;
12					mozliweRuchy.Add(mr);
13				}
14			}
15	mozliweRuchy.Sort(new PorownywaczRuchow());
16	MozliwyRuch najlepszyMozliwyRuch = (MozliwyRuch)mozliweRuchy[0];
```

Ostatecznie w wierszu 15 dokonywane jest sortowanie otrzymanej listy ze względu na liczbę możliwych ruchów przeciwnika, a w końcu w linii 16 wybiera się najbardziej optymalny ruch. Dalej wynik działania tego kodu zwracany jest do procedury odpowiedzialnej za wykonanie danego ruchu (wywołującą omawianą metodę).

Obliczanie ilości pól, jakie może zająć przeciwnik odbywa się w metodzie *MobilnoscTest* (listing 8), której argumentami są dwie współrzędne pola pobrane z listy możliwych ruchów gracza. W linii 19 stan planszy (tablica *plansza*) kopiowany jest do tablicy tymczasowej. Następnie przeprowadzane są niezbędne w tej strategii obliczenia, w których modyfikowana jest zawartość tablicy *plansza*, po czym w linii 51 przywracany jest jej pierwotny stan. W ramach tych obliczeń w linii 20 następuje „wirtualne" wykonanie ruchu przez gracza. Zmienne *złeRuchyX* oraz *złeRuchC* będą informować ile ruchów przeciwnik może wykonać odpowiednio na pola ***C*** lub ***X***. W liniach od 25 do 27 przeszukiwane są wszystkie wolne pola na planszy. W linii 29 sprawdzane jest czy w danym wolnym polu przeciwnik może postawić piona. Jeżeli tak, zmienna *priorytet* będzie większa od 0. W linii 33 sprawdzane jest, czy przeciwnik może postawić piona na którymś z czterech pól ***X***, zaś w linii 35 analogicznie dla pól ***C***. W linii 39 następuje sprawdzenie czy sytuacja na planszy umożliwia przeciwnikowi wykonać jedynie ruchów na pola ***X*** lub ***C***. Gdy przeciwnik może zająć tylko pola ***X***, metoda zwróci wartość 1 (najbardziej korzystną dla gracza), jeżeli pola ***X*** oraz ***C*** wtedy zwróci 2, jeżeli tylko pola ***C*** wtedy 3. Polecenie w linii 49 weryfikuje czy przeciwnik może zająć także inne pola metoda i wówczas zwraca ich liczbę powiększoną o 3 (zabezpieczenie w przypadku gdyby przeciwnik mógł zająć tylko 1, 2 lub 3 pola różne od ***X*** oraz ***C***). Linie 52 i 53 przekazują wirtualny ruchu graczowi oraz zwracają obliczoną wartość.

Listing 8. **Wyznaczanie liczby pól, jakie przeciwnik może przejąć w następnym ruchu.**

```
17      private int MobilnoscTest(int poz, int pio)
18      {
19        plansza_test = (byte[,])plansza.Clone();
20        int liczba = UstawPionek(poz, pio, false);
21        int zleRuchyX = 0;
22        int zleRuchyC = 0;
23        liczba = 0;
24        int priorytet = 0;
25        for (int i = 0; i < planszaRozmiar; i++)
26          for (int j = 0; j < planszaRozmiar; j++)
27            if (plansza[i, j] == 0)
28            {
29              priorytet = UstawPionek(i, j, true);
30              if (priorytet > 0)
31              {
32                liczba++;
33                if ((i == 1 && j == 1) ||
                      (i == 1 && j == planszaRozmiar - 1) ||
                      (i == planszaRozmiar - 1 && j == 1) ||
                      (i == planszaRozmiar - 1 && j == planszaRozmiar - 1))
34                  zleRuchyX++;
35                if (((i == 0 || i == planszaRozmiar - 1)  &&
                       (j == 1 || j == planszaRozmiar - 2)) ||
                      ((i == 1 || i == planszaRozmiar - 2)  &&
                       (j == 0 || j == planszaRozmiar - 1)))
36                  zleRuchyC++;
37              }
38            }
39        if ((liczba != 0) && (liczba == zleRuchyX + zleRuchyC))
40        {
41          if (zleRuchyX != 0)
42            if (zleRuchyC == 0)
43              priorytet = 1;
44            else
45              priorytet = 2;
46          else
47            priorytet = 3;
48        }
49        else if ((liczba != 0) && (liczba != zleRuchyX + zleRuchyC))
50          priorytet += 3;
51        plansza = (byte[,])plansza_test.Clone();
52        Pass();
53        return priorytet;
54      }
```

Arbitralnie dobrane wartości priorytetów dla każdych z ruchów dość dobrze spełniają swoje zadanie. Jest to proste rozwiązanie problemu implementacji strategii mobilności. Zastosowanie zbyt skomplikowanego systemu doboru współczynników zaciemniłoby tylko istotę rzeczy. Jak widać dzięki takiemu doborowi wartości bardzo łatwo można zróżnicować nawet dość skomplikowane ustawienia pionów.

6.2 Implementacja strategii klinowania

Wśród wielu strategii służących do oceny sytuacji gracza, a wykorzystanych w ramach algorytmu Mini - Max (patrz rozdział 5) skuteczne zastosowanie strategii klinowania opisanej w podrozdziale 4.4 daje duże prawdopodobieństwo zdobycia rogu. Jak już wspomniano wykorzystanie błędu przeciwnika poprzez „zaklinowanie się" między jego dwoma pionami stwarza szansę zdobycia pobliskiego rogu. W ramach strategii klinowania sprawdzana jest każda krawędź planszy. Oceniana jest wówczas możliwość wykorzystania istniejącego już klina bądź szansa na utworzenie nowego. Kryterium oceny będzie potencjalna możliwość zdobycia rogu w najbliższym ruchu lub szansa utworzenia przyczółka (klina), pozwalającego zdobyć róg w kolejnym posunięciu. Jest to oczywiście pewne ograniczenie tej strategii, ponieważ nie uwzględnia sytuacji w której gracz ma szansę zdobyć róg w drugim lub trzecim ruchu. Jednak postanowiłem, że zastosuje takie uproszczenie aby uniknąć rozpatrywania ogromnej liczby dodatkowych kombinacji. Poza tym przy stosowaniu algorytmu Mini – Max prowadziło by to do dublowania weryfikowanych możliwości.

Omawiany poniżej fragment kodu (listing 9) dotyczy sprawdzenia możliwości zdobycia północno zachodniego rogu czyli pola ***A1***. W liniach od 6 do 9 pobierane są wartości pól leżących na brzegach planszy, których wspólnym polem jest róg ***A1***, przy czym w nowo stworzonej tablicy pole o indeksie 0 zawsze odpowiada rozpatrywanemu narożnikowi. Jeżeli pole jest wolne odpowiednia komórka tablicy ma wartość 0. Jeżeli jest zajęte przez gracza nr 1 – wartość 1, dla gracza nr 2 – wartość 2.

Listing 9. **Sprawdzanie możliwości zdobycia rogu *A1*.**

```
01    int iloscPol_C_Gracza = 0;
02    int iloscPol_C_Przeciwnika = 0;
03    int[] krawedzPionowa = new int[planszaRozmiar];
04    int[] krawedzPozioma = new int[planszaRozmiar];
05    int wynik = 0;
06    for (int i = 0; i < planszaRozmiar; i++)
07       krawedzPozioma[i] = plansza[i, 0];
08    for (int i = 0; i < planszaRozmiar; i++)
09       krawedzPionowa[i] = plansza[0, i];
10    if (krawedzPozioma[0] == 0)
11    {
12       if (krawedzPozioma[1] != 0)
13       {
14          wynik = klinowanie(krawedzPozioma);
15          if (wynik > 0)
16             iloscPol_C_Gracza++;
17          else if (wynik < 0)
18             iloscPol_C_Przeciwnika++;
19       }
```

```
20          if (krawedzPionowa[1] != 0)
21          {
22            wynik = klinowanie(krawedzPionowa);
23            if (wynik > 0)
24              iloscPol_C_Gracza++;
25            else if (wynik < 0)
26              iloscPol_C_Przeciwnika++;
27          }
28        }
```

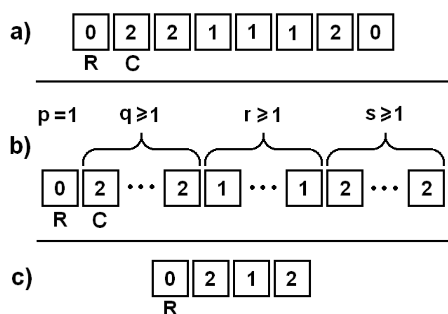
Aby istniała możliwość szybkiego przejęcia rogu pole ***C*** (położone na krawędzi a zarazem bezpośrednio sąsiadujące z rogiem) musi być zajęte przez przeciwnika. Warunek ten jest sprawdzany w liniach 12 i 20. Szanse na zdobycie rogu zależą od tego, który z graczy zajmuje pole ***C***. Jeżeli pole to zajmuje gracz nr 1, wtedy sprawdzane jest czy jego rywal ma możliwość zdobycia przyległego rogu. Aby nie powielać linii kodu dla poszczególnych pól ***C*** zastosowałem metodę polegającą na zamianie wartości w tablicy *krawedz* (listing 10, linia 34). W metodzie *negacja* następuje zmiana wartości każdej komórki z 1 na 2 i na odwrót. Komórki z wartością 0 pozostają bez zmian.

Listing 10. **Konwersja tablicy *krawedz*.**

```
29        private int klinowanie(int[] krawedz)
30        {
31          int c = 1;
32          if (krawedz[1] == 1)
33          {
34            negacja(ref krawedz);
35            c = -1;
36          }
37          return c * metodaKlinowania(parsacja(krawedz));
38        }
```

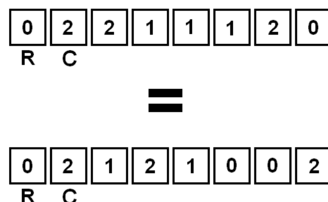
W metodzie *parsacja* konwertuje się tablice do typu *string*. Właściwe zastosowanie strategii klinowania odbywa się w metodzie *metodaKlinowania*. Wstępnie usuwa się w niej ciągi powtórzeń tych samych znaków (o ile są różne od 0), pozostawiając tylko jedno wystąpienie (linie 42 – 48). W linii 49 dodaje się znaki sygnalizujące koniec właściwych danych. Robi się to by uniknąć przekroczenia zakresu podczas procesu porównywania krawędzi ze wzorcami. Schemat działania tej metody przedstawia rysunek 20.

Rysunek 20. **Schemat konwersji zapisu stanu krawędzi wykonywany przez metodę *metodaKlinowania*.**



Rysunek 20.a przedstawia przykładowe ułożenie pionów wzdłuż badanej krawędzi. Literą ***R*** oznaczony jest róg, którego możliwość zajęcia jest weryfikowana, natomiast litera ***C*** odnosi się do pola ***C*** (patrz rozdział 4). Wyraźnie widać, że piony z numerem ***1*** są zaklinowane pomiędzy tymi z numerami ***2***. Dzięki temu gracz ***1*** ma nie tylko szansę, ale nawet pewność zdobycia rogu ***R***. Ma on również możliwość przejęcia przeciwległego rogu, lecz ta możliwość nie będzie teraz badana (jest to uzależnione od sposobu pobierana wartości z krawędzi planszy - linie kodu od 6 do 9). Rysunek 20.b ilustruje sytuację reprezentującą wszystkie możliwe wariacje „pewnego klina” (dającego pewność zdobycia rogu). Wartości nad polami oznaczają liczbę sąsiadujących ze sobą identycznych komórek (seria jednakowych wartości). Warunkiem istnienia takiego klina jest otoczenie piona lub grupy pionów gracza pionami przeciwnika. Nie ma znaczenia, jak dużo jest tych pionów, musi ich być jednak więcej niż 1. Stąd biorą się warunki na wartości *q*, *r* oraz *s* (por. linie 42 – 48 w listingu 11). Rysunek 20.c przedstawia uproszczenie tych wszystkich przypadków. Z każdej serii identycznych pól pozostawia się tylko jeden element. Operacji tej nie podlegają tylko serie pól pustych (oznaczonych numerem 0), ponieważ takie upraszczanie prowadziło by do błędnych wyników. Powyższe konwersje odnoszą się do kodu programu w liniach od 42 do 48.

Rysunek 16. **Ilustracja prezentująca różne wersje tego samego „klina”.**



Na rysunku 21 pokazane są dwie różne kombinacje ułożenia pól na planszy. Widać iż są to dwa przypadki tego samego klina, przedstawionego na rysunku 20.c. Wynika stąd,

że można przedstawić wiele różnych kombinacji ustawień pionów na krawędzi za pomocą kliku/kilkunastu wzorcowych zestawów (patrz tabela 2).

***Tabela. 1.* Lista możliwych ustawień pionów na krawędziach.** [1]

Lp.	Wzór
1	0 2 0 2 0 *
2	0 2 0 2 0 E
3	0 2 0 2 1 0 0 2 E *
4	0 2 0 2 1 0 1 E
5	0 2 0 2 1 0 2 E *
6	0 2 0 2 1 0 2 1 E
7	0 2 0 2 1 2
8	0 2 1 E
9	0 2 1 0 1 E
10	0 2 1 0 1 0
11	0 2 1 0 2 1
12	0 2 1 0 1 2 1
13	0 2 1 2

W tabeli 2 wyszczególnione są wzorce ustawień z którymi porównywana jest badana krawędź. Wzorce z numerami 1, 3 oraz 5 są dodatkowo włączone do listy, mimo że nie dają pewności zdobycia rogu. W ich przypadku można jednak zaryzykować stwierdzenie, iż zajmując pole i wykorzystując piony z głębi planszy stworzenia klina jest na tyle prawdopodobne, że warte jest ryzyka. Dowodzenie słuszności tego stwierdzenia wymagałoby dodatkowej pracy procesora, co spowodowałoby dalsze spowolnienie działania programu.

***Listing 11.* Sprawdzanie możliwości wykorzystania klina.**

```
39        private int metodaKlinowania(string krawedz)
40        {
41            string temp = "0";
42            for (int i = 1; i < planszaRozmiar; i++)
43            {
44                if (krawedz[i] != krawedz[i - 1])
45                    temp += krawedz[i];
46                if (krawedz[i] == '0')
```

[1] * – dodatkowo włączone do listy, E – pionek oznaczony cyfrą po lewej stronie leży w przeciwległym rogu badanej krawędzi (np. 0 2 1 E - pionek z 1 leży w drugim rogu badanej krawędzi narożniku).

```
47              temp += '0';
48          }
49          temp += "--------";
50          if (temp.Substring(0, 5) == wzor01)
51            return -1;

          ...

52          else
53            return 0;
54          }
55        }
```

Jeżeli wartość zmiennej *wynik* (linie 14 – 18 oraz 22 – 26 listingu 9) jest większa od 0, zwiększane jest saldo możliwych przejęć rogów przez gracza. Natomiast jeżeli jest ujemna, to wzrasta saldo przeciwnika. Takie obliczenia wykonywane są dla wszystkich rogów. Na końcu odejmuje się saldo przeciwnika od salda gracza. Ostatecznie program korzystający ze strategii klinowania wybierze taki ruch, w którym ta różnica będzie jak największa, czyli sytuacja na planszy będzie gwarantowała pewne przejęcie dużej liczby rogów.

6.3 Gra sieciowa

Ze względu na znaczną ilość linii kodu implementujących funkcje gry sieciowej (jest ich ponad 800) omówione zostaną tylko najistotniejsze fragmenty tego modułu. Grę w wersji sieciowej można uruchomić w trybie serwer lub klient. Jako przykład posłużyła mi implementacja komunikatora internetowego z książki Sławomira Orłowskiego (projekty od 51 do 54) [8].

Metoda *startujPolaczenie* (listing 12) służy do pobierania domyślnego adresu IP oraz numeru portu hosta, z którym chcemy się połączyć, a także do przekazania ich do obiektu związanego z osobną formą odpowiedzialną za konfigurację połączenia. Domyślnie zostanie wybrany adres pętli zwrotnej 127.0.0.1 oraz port 25000 (linie 5 i 6). Metoda ta przekazuje również wszystkie numery IP przyporządkowane do danego komputera (linie od 7 do 8). Najpierw blokowane są przyciski menu (linia 3), dopiero później otwierane jest nowa forma (okienko), w którym wybierany jest docelowy adres i port (linia 4).

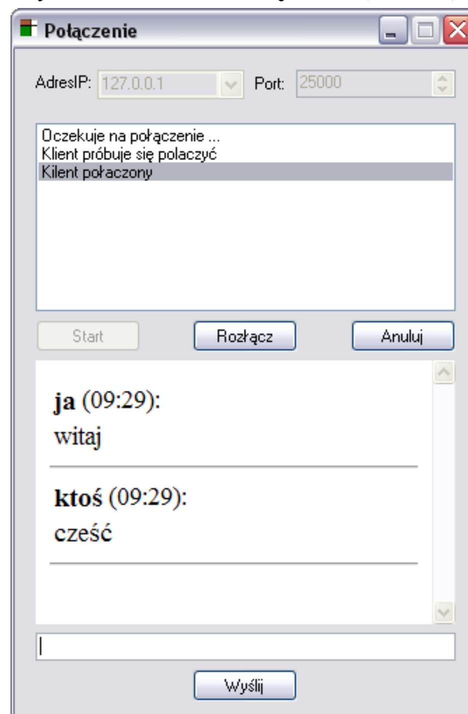
***Listing 12.* Otwieranie okna *Połączenie*, wybór portu oraz adresu IP.**

```
001     private void startujPolaczenie()
002     {
003       zablokujPrzyciskiMenu();
```

```
004        form3 = new ConectionForm(this);
005        form3.pobierzAdresIP(adresIP);
006        form3.pobierzPort(port);
007        IPHostEntry adresyIP = Dns.GetHostEntry(Dns.GetHostName());
008        foreach (IPAddress pozycja in adresyIP.AddressList)
009          form3.pobierzAdresIP(pozycja.ToString());
010      }
```

Po ustaleniu adresu hosta z którym chcemy się połączyć i kliknięciu przycisku START w oknie *Połączenie* (forma3 – rysunek 22) następuje pobranie żądanego adresu IP oraz numeru portu.

***Rysunek 17.* Okno *Połączenie* (forma3).**



Wykonuje to metoda *pobierzIP* (linie 13 oraz 14 listingu 13). Jeżeli gracz uruchomił serwer, to w osobnym wątku zostanie zainicjowany proces oczekujący na otwartym porcie na żądanie połączenia przysłane przez klienta (linia 18). Jeżeli gracz zainicjował połączenie sieciowe przyjmując rolę klienta, zostanie uruchomiony wątek łączenia ze wskazanym w ustawieniach serwerem. Ponieważ kody źródłowe dotyczące serwera i klienta są dość podobne, tu zaprezentuję jedynie modułu odpowiedzialne za działanie serwera.

***Listing 13.* Pobranie numeru portu oraz adresu IP, uruchomienie wątków połączenia sieciowego.**

```
011      public void pobierzIP()
012      {
013        adresIP = form3.przekazAdresIP();
```

```
014       port = form3.przekazPort();
015       if (typ == "Serwer")
016       {
017         graczSieciowy = 2;
018         backgroundWorkerNET_serwer_1.RunWorkerAsync();
019       }
020       if (typ == "Klient")
021       {
022         graczSieciowy = 1;
023         backgroundWorkerNET_klient_1.RunWorkerAsync();
024       }
025     }
```

Metoda *backgroundWorkerNET_serwer_1_DoWork* (listing 14) służy do zainicjowania działania serwera w osobnym wątku (korzysta przy tym z klasy *BackgroundWorker* dostępnej w bibliotekach platformy .NET od wersji 2.0). W liniach od 29 do 37 sprawdzane jest, czy użyto poprawnego zapisu adresu IP. Jeżeli wszystko jest w porządku, w linii 41 adres ten zostaje użyty do zainicjowania połączenia. Następnie rozpoczyna się oczekiwanie na połączenie klienta. Jeżeli pierwszą wiadomością wysłaną przez klienta jest komunikat powitalny „###HI###", następuje jego uwierzytelnienie. W linii 51 inicjalizuje się interfejs gry. Polega to na przywróceniu początkowych ustawień planszy. Następnie uruchamiany jest jeszcze jeden wątek odpowiedzialny za komunikację serwera z klientem. Jeżeli klient nie przeszedł wymaganej autoryzacji połączenie zostaje przerwane, a serwer zrestartowany (linie 54 do 58). Serwer podobnie zachowa się jeżeli w trakcie próby połączenia klient się rozłączy (linie 60 do 64).

Listing 14. **Inicjowanie działania serwera.**

```
026     private void backgroundWorkerNET_serwer_1_DoWork(object sender, DoWorkEventArgs e)
027     {
028       IPAddress serwerIP;
029       try
030       {
031         serwerIP = IPAddress.Parse(adresIP);
032       }
033       catch
034       {
035         MessageBox.Show("Błędny adres IP");
036         return;
037       }
038       serwer = new TcpListener(serwerIP, port);
039       try
040       {
041         serwer.Start();
042         form3.SetText("Oczekuje na połączenie ...");
043         klient = serwer.AcceptTcpClient();
044         NetworkStream ns = klient.GetStream();
045         form3.SetText("Klient próbuje się polaczyć");
046         czytanie = new BinaryReader(ns);
047         pisanie = new BinaryWriter(ns);
048         if (czytanie.ReadString() == "###HI###")
049         {
050           form3.SetText("Kilent połaczony");
```

```
051                Inicjalizacja();
052                backgroundWorkerNET_serwer_2.RunWorkerAsync();
053              }
054              else
055              {
056                form3.SetText("Klient nie wykonał wymaganej autoryzacji. połączenie przerwane");
057                rozłączToolStripMenuItem_Click(sender, EventArgs.Empty);
058              }
059            }
060            catch
061            {
062            form3.SetText("Połączenie zostało przerwane");
063            rozłączToolStripMenuItem_Click(sender, EventArgs.Empty);
064            }
065          }
```

Metoda *backgroundWorkerNET_serwer_2_DoWork* (listing 15) odpowiedzialna jest za odczytywanie wiadomości wysyłanych przez klienta. W pętli typu *while* sprawdzane są wszystkie odebrane wiadomości. Pętla skończy działanie jeżeli klient wyśle komunikat pożegnalny „###BYE###", wtedy powyższy moduł zakończy działanie, a ponownie uruchomiony zostanie moduł oczekujący na podłączenie się klienta (linie 102 – 104). Każde polecenie, czy to zwykły tekst przesyłany w celu wypisania go w komunikatorze przeciwnika, czy polecenie rozpoczęcia nowej gry lub postawienia piona, wysyłane jest pomiędzy grającymi w formie komunikatów tekstowych. Odpowiednio je formatując, tzn. dodając pewne znaki specjalne na końcu oraz początku każdego z nich, umożliwia się łatwe ich rozróżnienie, co pozwala na właściwą reakcję programu.

Listing 15. **Przechwytywanie i interpretowanie wiadomości wysyłanych przez klienta.**

```
066          private void backgroundWorkerNET_serwer_2_DoWork(object sender, DoWorkEventArgs e)
067          {
068            string wiadomosc;
069            try
070            {
071              while ((wiadomosc = czytanie.ReadString()) != massageNetwork1)
072              {
073                if (wiadomosc == massageNetwork5)
074                {
075                  noweRozdanie2 = true;
076                  if (noweRozdanie1)
077                    sekcjaKrytyczna();
078                }
079                if (wiadomosc == massageNetwork2)
080                {
081                  Inicjalizacja();
082                }
083                else if (wiadomosc == massageNetwork3)
084                {
085                  if (MessageBox.Show("Przeciwnik poddał grę\nCzy chcesz zagrać ponownie", "",
                     MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)
086                  {
087                    Inicjalizacja();
088                    nowaGra();
089                  }
090                  else
```

```
091                {
092                   rozłączToolStripMenuItem_Click(sender, EventArgs.Empty);
093                }
094             }
095             else if (wiadomosc.StartsWith(massageNetwork4))
096             {
097                kliknijBezpiecznie(wiadomosc.Substring(19, 2));
098             }
099             else
100                form3.WpiszTekst("ktoś", wiadomosc);
101          }
102          pisanie.Write(massageNetwork1);
103          form3.SetText("Połączenie zostało przerwane przez klienta");
104          rozłączToolStripMenuItem_Click(sender, EventArgs.Empty);
105       }
106       catch
107       {
108          form3.SetText("Klient rozłączony");
109          rozłączToolStripMenuItem_Click(sender, EventArgs.Empty);
110       }
111    }
```

W liniach od 73 do 78 sprawdzane jest czy klient wysłał propozycję rozpoczęcia nowej gry. Ponieważ po ukończeniu gry pytanie o możliwość rozpoczęcia nowej może wysłać każdy z graczy, więc aby uniknąć zapętlenia się poprzez ciągłe wysyłanie zapytań i potwierdzeń należy zastosować w tym miejscu tak zwany semafor. W platformie .NET jest on implementowany przez klasę *Mutex*. Jeżeli klient pierwszy zdążył nadesłać wiadomość z pytaniem o rozpoczęcie nowej gry wtedy po wyrażeniu na nią zgody nie trzeba wysłać kolejnego pytania, a już tylko tę grę rozpocząć po czym jedynie wysłać potwierdzenie jej rozpoczęcia. Pierwotnie zmienne *noweRozdanie1* oraz *noweRozdanie2* mają wartość *false*. Jeżeli gracz ***A*** naciśnie przycisk potwierdzający chęć rozpoczęcia nowej gry, wtedy w metodzie *odpowiedzSieciowa* (listing 16) następuje zmiana wartości zmiennej *noweRozdanie1* na *true* (linia 114). W przypadku gdy gracz ***B*** nie proponuje rozpoczęcia nowej gry, (np. nie zdążył jeszcze podjąć decyzji), gracz ***A*** wysyła do niego pytanie (linia 118). Komunikat docierając do odbiorcy (w tym przypadku do gracza ***B***) jest rozpoznany w linii 73 (listing 15), gdzie następuje zmiana wartości *noweRozdanie2* na *true* (oznacza to że przeciwnik, czyli gracz ***A*** zamierza rozpocząć nową grę). Ostatecznie, jeżeli gracz ***B*** również chcę kontynuować grę, wykonana zostanie metoda *sekcjaKrytyczna* (listing 16), jeżeli nie, gra jest „zawieszona" do czasu podjęcia decyzji przez gracza ***B***. Jeżeli jednak gracz ***B*** zdąży nadesłać pytanie o rozpoczęcie nowej gry zanim sprawdzany będzie warunek z linii 115, może zdarzyć się sytuacja, w której kod w sekcji krytycznej będzie wykonywany przez obie metody „jednocześnie". I właśnie aby uniknąć takiej sytuacji zastosowano klasę *Mutex*. W zależności od tego, która z metod pierwsza zostanie dopuszczona do sekcji krytycznej, dostęp drugiej zostaje zablokowany (linia 122). Zdjęcie

blokady następuje w linii 130. Wówczas jeżeli *noweRozdanie1* oraz *noweRozdanie2* mają wartość *true* (obaj gracze wyrażają chęć na rozpoczęcie nowej gry), nastąpi zmiana ich wartości na *false*. Dzięki temu sekcja krytyczna nie będzie wykonywana dwukrotnie. Następnie nastąpi wyczyszczenie planszy (powrót do ustawień początkowych) oraz wysłane potwierdzenie o zamiarze rozpoczęcia nowej gry.

Listing 16. **Metoda *odpowiedzSieciowa* odpowiada za wysyłanie propozycji rozpoczęcia nowej gry, metoda *sekcjaKrytyczna* służy do jej bezpiecznego zainicjowania.**

```
112    private void odpowiedzSieciowa()
113    {
114      noweRozdanie1 = true;
115      if (noweRozdanie2)
116        sekcjaKrytyczna();
117      else
118        nastepnaGra();
119    }

120    private void sekcjaKrytyczna()
121    {
122      mut.WaitOne();
123      if (noweRozdanie1 && noweRozdanie2)
124      {
125        noweRozdanie1 = false;
126        noweRozdanie2 = false;
127        Inicjalizacja();
128        nowaGra();
129      }
130      mut.ReleaseMutex();
131    }
```

W omawianej już metodzie *backgroundWorkerNET_serwer_1_DoWork*, w warunku z linii 79 (listing 15) sprawdzane jest czy klient wysłał komunikat o potwierdzeniu zamiaru rozpoczęcia nowej gry. Jeżeli tak, następuje inicjalizacja planszy. W linii 83 sprawdza się, czy przeciwnik chce poddać grę oraz czy gracz lokalny ewentualnie wyraża zgodę na rozpoczęcie ponownej rozgrywki. Jeżeli wszystkie warunki zostały spełnione następuje inicjalizacja planszy i wysłanie potwierdzenia do zdalnego gracza (linia 88). W przeciwnym wypadku połączenie zostanie zerwane, a serwer wprowadzony w stan oczekiwania na połączenie kolejnego klienta. Kluczowe znaczenie ma linia 95. Tutaj właśnie sprawdza się czy klient przesłał komunikat o kliknięciu na pole planszy. Jeżeli warunek jest spełniony, wskazane pole zostaje przejęte. Ostatecznie jeżeli żaden z dotychczasowych warunków nie był spełniony odebrana wiadomość zostanie wyświetlona w oknie komunikatora. W razie wystąpienia jakiegoś wyjątku połączenie zostanie zerwane a serwer restartowany (linie 102 – 104).

7. Podsumowanie

W niniejszej pracy opisana została implementacja gry planszowej Reversi (alt. *Otello*) napisana w języku C# dla platformy .NET. Aplikacja ta pozwala na grę między dwiema osobami, znajdującymi się przy jednym komputerze, zdalnie przez sieć lub grę z komputerem. Istnieje również możliwość gry, w której ruchy obu graczy planuje oraz realizuje komputer. W grze wykorzystane zostało wiele zaawansowanych strategii opracowanych dla tej gry.

8. Literatura

[1] http://gamescrafters.berkeley.edu/inside/games/othello.html
(Oficjalna strona UC Berkeley GamesCrafters)

[2] http://www.othello.pl/
(Oficjalna strona Polskiej Federacji Othello)

[3] http://www.pressibus.org/reversi/gen/gborigines.html
(Strona o historii Othello)

[4] http://home.nc.rr.com/othello/strategy/
(Oficjalna strona Othello University)

[5] http://matotek.webpark.pl/
(Wszystko o Othello)

[6] http://vieka.com/reversi/othello.htm
(Othello: Zasady gry)

[7] Jacek Matulewski
Visual C# 2005 Express Edition. Od podstaw
Wydawnictwo Helion, Gliwice 2006

[8] Sławomir Orłowski,
C#. Tworzenie aplikacji sieciowych. 101 gotowych projektów
Wydawnictwo Helion, Gliwice 2007

[9] Evan A. Sultanik
„MiniMax Search by Example”
http://gicl.cs.drexel.edu/people/evan/classes/ai_fall06/minimax.pdf

[10] Jacek Matulewski
Język C#. Programowanie dla platformy .NET w środowisku Borland C# Builder
Wydawnictwo Help, 2004

Dodatek A.: Instrukcja obsługi programu

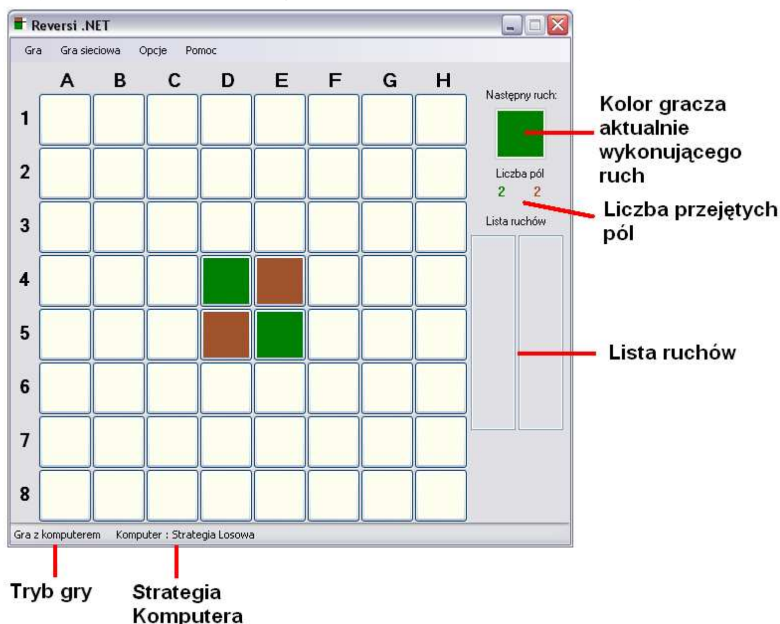
A.1 Wstęp

Korzystając z aplikacji Reversi.NET możesz grać w grę Reversi (alt. *Othello*). Program pozwala na grę pomiędzy dwiema osobami używającymi tego samego komputera, poprzez sieć oraz grę z komputerem. Możliwa jest także symulacja, w której za ruchy obu graczy odpowiada komputer. Istnieje możliwość zapisu przebiegu rozgrywki, a także jej odczytu z pliku. Natomiast w trybie gry sieciowej możliwe jest prowadzenie rozmowy pomiędzy grającymi przy użyciu komunikatora internetowego.

A.2 Interfejs użytkownika (główna forma)

Aplikacja zaopatrzona jest w szereg kontrolek informujących o stanie gry. Ich omówienie widoczne jest na rysunku A - 1.

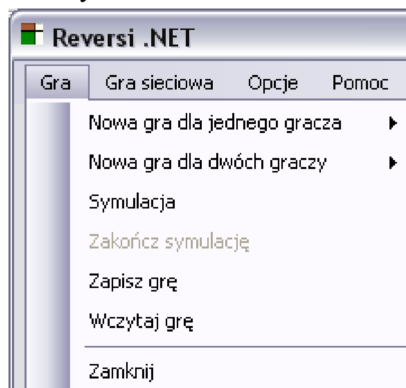
***Rysunek A – 1.* Forma główna. Kontrolki interfejsu użytkownika.**



A.3 Tryby gry

W menu ***Gra*** zebrane zostały opcje odpowiedzialne za uruchomienie gry w trybie lokalnym. Dodatkowo znajdują się tu przyciski służące do zapisu i odczytu przebiegu gry do i z pliku. Dodatkową funkcją jest możliwość gry symulacyjnej (przycisk ***Symulacja***), w której w roli obu graczy występuje komputer. W menu ***Opcje*** można wybrać strategie wykorzystywane przez oba boty. Przycisk ***Zakończ symulację*** służy do przerywania gry pomiędzy botami.

***Rysunek A – 2.* Menu *Gra*.**

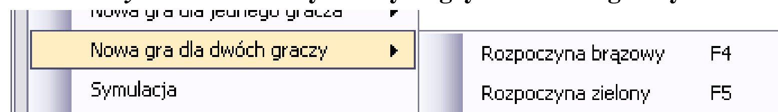


Tryb gry stacjonarnej pomiędzy komputerem a człowiekiem może rozpocząć komputer (skrót klawiaturowy F2) lub człowiek (skrót klawiaturowy F3).

***Rysunek A – 3.* Wybór trybu gry z komputerem.**

Tryb gry stacjonarnej pomiędzy dwiema osobami może rozpocząć gracz brązowy (skrót klawiaturowy F4) lub zielony (skrót klawiaturowy F5).

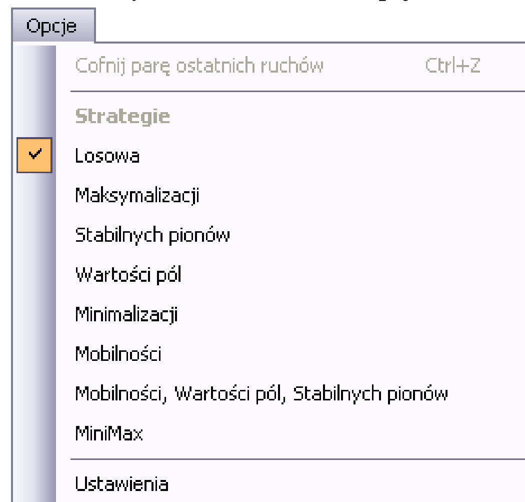
***Rysunek A – 4.* Wybór trybu gry dla dwóch graczy.**



A.4 Konfiguracja strategii komputera

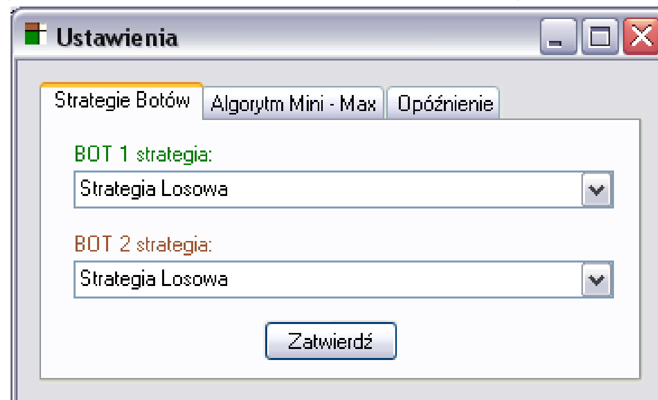
Gracz ma do wyboru wiele strategii ułożonych hierarchicznie z góry do dołu od najprostszych do najbardziej zaawansowanych (i tym samym najbardziej skutecznych). Strategie te wykorzystywane są przez komputer w trakcie gry jednego gracza.

***Rysunek A – 5.* Menu *Opcje*.**



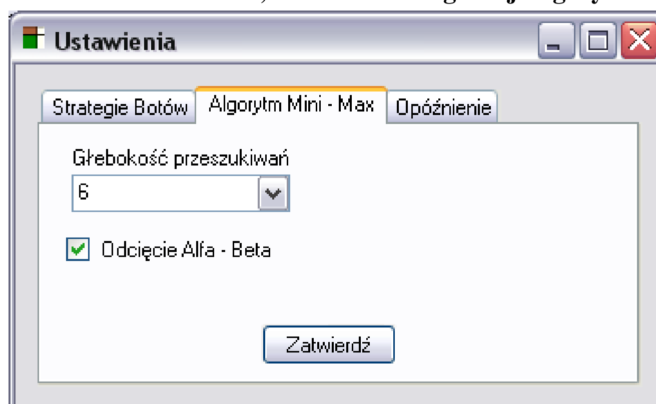
Dodatkowo poprzez kliknięcie przycisku ***Ustawienia*** można skonfigurować strategie przypisane dla obu botów podczas gry symulacyjnej (patrz rysunek A – 6).

***Rysunek A – 6.* Forma *Ustawienia*, zakładka wyboru strategii.**



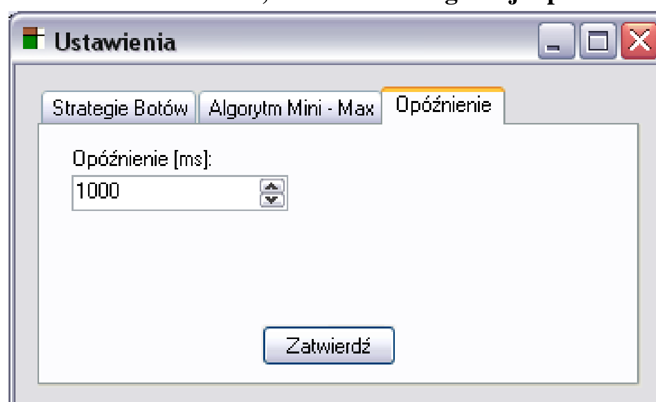
Kiedy komputer wykorzystuje algorytm Mini – Max, w prezentowanej na rysunku A - 7 zakładce istnieje możliwość ustawienia głębokości badania drzewa ruchów. Zakładka ta zaopatrzona jest także w kontrolkę, dzięki której w algorytmie uwzględniane będą odcięcia Alfa – Beta. Dokładny opis wymienionych ustawień znajduje się w rozdziale 5.

***Rysunek A – 7.* Forma *Ustawienia*, zakładka konfiguracji algorytmu Mini – Max.**



Grający ma możliwość ustawienia czasu opóźnienia reakcji komputera, co pozwala na przyjrzenie się zmianom, jakie wprowadził na planszy nasz ruch. Proponowany interwał to 1 sekunda.

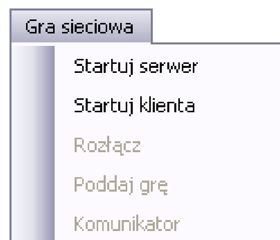
***Rysunek A – 8.* Forma *Ustawienia*, zakładka konfiguracji opóźnienia komputera.**



A.5 Gra sieciowa

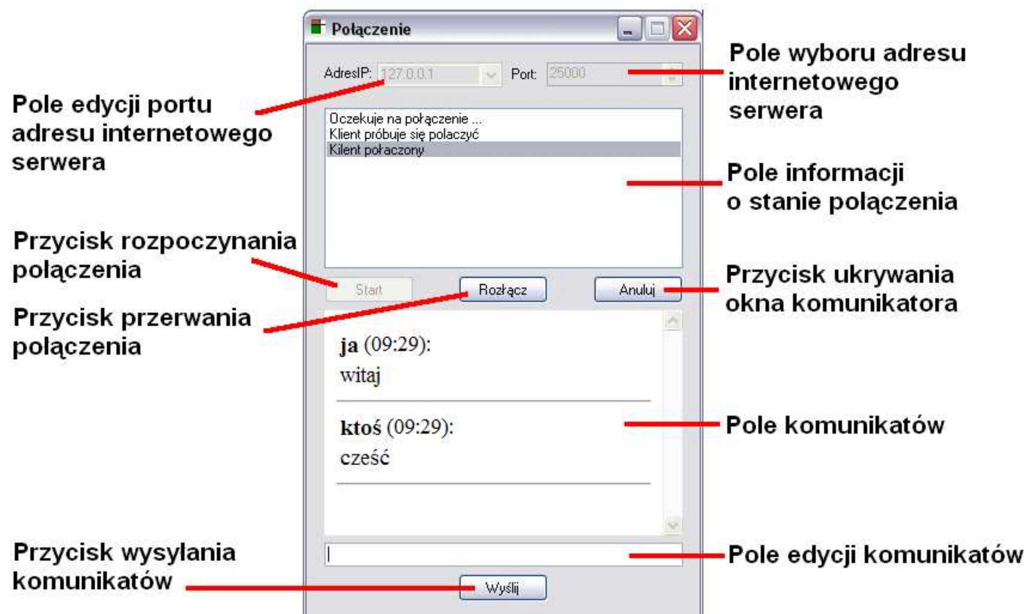
Ten tryb gry rozpoczyna się klikając na przycisk ***Startuj serwer*** lub ***Startuj klienta*** w menu ***Gra sieciowa*** (patrz rysunek A – 9). Po ich naciśnięciu pojawia się okno komunikatora o nazwie ***Połączenie***. Grę można zakończyć przyciskając klawisz ***Rozłącz***. Jest on również umieszczony w oknie ***Połączenie***. Po zamknięciu okna komunikatora, jeżeli połączenie jest nadal aktywne, można je ponownie otworzyć klikając na przycisk ***Komunikator***. Istnieje także możliwość poddania gry – służy do tego przycisk ***Poddaj grę***.

Rysunek A – 9. **Menu *Gra sieciowa*.**



Rysunek A – 10 prezentuje okno komunikatora ***Ustawienia***.

Rysunek A – 10. **Forma *Połączenie*. Kontrolki konfiguracji adresu IP oraz komunikator internetowy.**



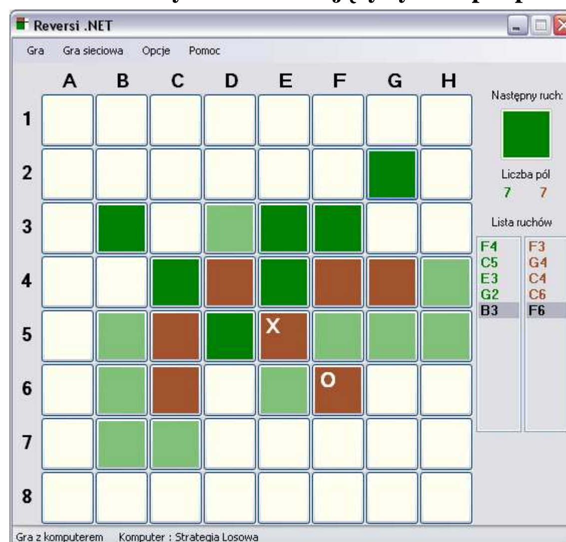
A.6 System podpowiedzi

Menu *Pomoc* posiada zestaw pięciu istotnych przycisków pomagających graczowi w wyborze najlepszego ruchu.

Rysunek A – 11. **Menu *Pomoc*.**

Pierwszym z nich jest ***Podpowiedź ruchu*** (skrót klawiaturowy F6). Dzięki tej funkcji grający ma możliwość zapytania komputera o najlepszy możliwy ruch. Proponowany ruch będzie oznaczony kolorem gracza stosując jaśniejsze zabarwienie. Przykładem takiej podpowiedzi jest ruch na pole ***D3*** widoczny na rysunku A - 12. Wybór oparty jest o strategie, którą aktualnie posługuje się komputer. Przycisk ***Ruch wykonany przez komputer*** (F7) pozwala na tymczasowe (na czas jednego ruchu) przekazanie komputerowi decyzji w sprawie aktualnego ruchu.

Rysunek A – 12. **Rysunek ilustrujący system podpowiedzi.**



Kolejnym przyciskiem jest ***Pokaż wszystkie możliwe ruchy*** (skrót klawiaturowy F8). Podświetlone zostaną wszystkie możliwe ruchy gracza, co widać na rysunku numer A - 12. Ciekawą funkcję spełnia przycisk ***Dodawaj oznaczenia*** (F9). Jego działanie polega na dodawaniu oznaczeń kółek i krzyżyków na polach które ostatnio uległy zmianie. Znak ”o” stawiany jest w miejscu w którym ostatni z grających postawił swojego piona, „x” stawiane jest na polach które ostatni z grających odebrał przeciwnikowi.

Przycisk *Usuń oznaczenia i podpowiedzi (F10)*, jak sama nazwa wskazuje, służy do czyszczenie planszy z wszelkiego typu dodatkowych informacji.