



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

Analiza i porównanie wybranych algorytmów dla gry karcianej
Analysis and comparison of selected algorithms for the card game

Autor:

Damian Malarczyk

Kierunek studiów:

Informatyka

Opiekun pracy:

dr inż. Edyta Kucharska

Kraków, 2016

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Podziękowania

Spis treści

1. Wprowadzenie	7
1.1. Przedmiot pracy	7
1.2. Cel pracy	7
1.3. Zawartość pracy	7
2. Gra karciana Love Letter	9
2.1. Opis zasad gry	9
2.2. Analiza problemu	15
2.3. Opis formalny problemu	16
3. Przegląd wybranych algorytmów	21
3.1. Algorytm losowy	21
3.1.1. Opis	21
3.1.2. Sposób wykorzystania	21
3.1.3. Zapis pseudokodem	21
3.2. Algorytm zachłanny	22
3.2.1. Opis	22
3.2.2. Sposób wykorzystania	22
3.2.3. Zapis pseudokodem	24
3.3. Algorytm min-max	25
3.3.1. Opis	25
3.3.2. Sposób wykorzystania	26
3.3.3. Zapis pseudokodem	27
3.4. Algorytm Monte Carlo Tree Search	30
3.4.1. Opis	30
3.4.2. Sposób wykorzystania	30
3.4.3. Zapis pseudokodem	31
4. Implementacja	33
4.1. Analiza wymagań	33

4.2.	Koncepcja wykonania i wykorzystane technologie.....	35
4.3.	Diagramy	35
4.3.1.	Diagram klas pakietu model	36
4.3.2.	Diagram klas pakietu player	37
4.3.3.	Diagram klas pakietu engine.....	38
4.4.	Listingi.....	39
4.4.1.	Listing części metod klasy Engine.....	39
4.4.2.	Listing algorytmu MCTS	40
4.5.	Prezentacja systemu.....	44
4.6.	Problemy napotkane w trakcie realizacji.....	49
5.	Rezultaty	51
5.1.	Algorytm losowy versus Algorytm Losowy.....	51
5.2.	Algorytm zachłanny versus Algorytm losowy	54
5.3.	Algorytm minimaksowy versus Algorytm losowy	57
5.4.	Algorytm MCTS versus Algorytm Losowy	59
5.5.	Algorytm Minimaksowy versus Algorytm Zachłanny	61
5.6.	Algorytm Zachłanny versus Algorytm MCTS	65
5.7.	Algorytm Minimaksowy versus Algorytm MCTS	68
5.8.	Wnioski.....	71
6.	Podsumowanie	73

1. Wprowadzenie

1.1. Przedmiot pracy

Od kilku lat coraz większą popularnością cieszą się wszelkiego rodzaju gry planszowe i karciane. Ze względu na ich różnorodną mechanikę, mogą być dobrą podstawą do porównania i analizy działania różnych algorytmów. Gdy grając w jedną z nich, „Love Letter”, mój kolega stwierdził, że „ta gra to tylko los”, postanowiłem sprawdzić to w praktyce, implementując kilka wybranych algorytmów i sprawdzając ich działanie właśnie na podstawie tej gry. Algorytmy zostały przeze mnie wybrane w taki sposób, by mimo deterministycznej natury przypominać intuicyjne zachowanie człowieka w sytuacji dużej niepewności. Niektórzy podejmują decyzje losowo, inni wybierają te zagrania, które najszybciej prowadzą do wygranej, a część stara się minimalizować ewentualne straty. Oprócz algorytmów deterministycznych wybrałem również jeden heurystyczny, który poprzez swoją niejednoznaczność w podejmowaniu wyborów najbardziej przypomina ludzkie zachowanie.

1.2. Cel pracy

Celem tej pracy jest implementacja programistyczna gry oraz kilku różnych algorytmów, których działanie można przeanalizować opierając się na statystykach zdobywanych w symulacji. Następnie, spośród wybranych algorytmów, zostanie wybrany algorytm najlepiej spełniający swoje zadanie - czyli wygrywanie gier.

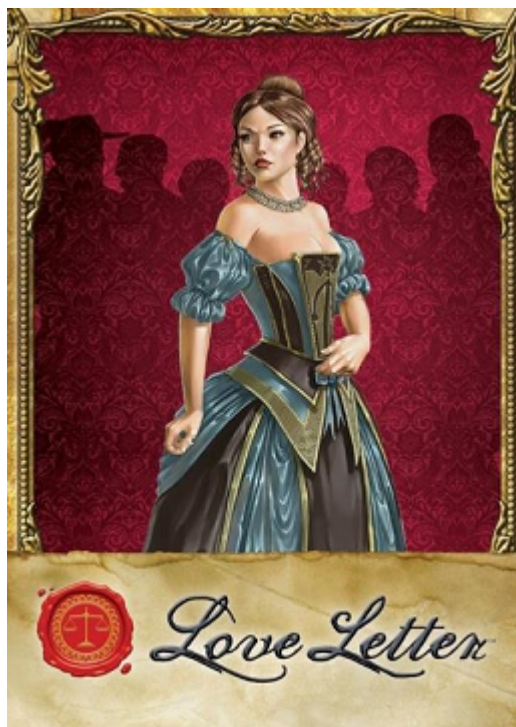
1.3. Zawartość pracy

Rozdział 1 jest wprowadzeniem definiującym przedmiot pracy, jej cel oraz zawartość. W rozdziale 2 opisana została gra karciana „Love Letter” wraz z zasadami. Na ich podstawie oszacowałem ilość możliwych rozwiązań, a następnie dokonałem analizy problemu i zdefiniowałem jego model matematyczny w postaci gry ekstensywnej. Rozdział 3 przedstawia proponowane przeze mnie algorytmy rozwiązujące przedstawiony problem, kolejno: algorytm losowy, algorytm zachłanny, algorytm minimaxowy oraz algorytm Monte Carlo Tree Search. Dla każdego z nich opisany został sposób działania, wykorzystanie w praktyce, oraz mój pomysł na użycie do rozwiązania zadanego problemu. W rozdziale 4 przedstawiłem założenia programu, w którym implementowana jest gra i algorytmy. Następnie zaprezentowane są

poszczególne etapy tworzenia aplikacji, począwszy od analizy wymagań, poprzez koncepcję wykonania, prezentację diagramów UML oraz listingów, a skończywszy na prezentacji programu. W rozdziale 5 przedstawiłem wyniki eksperymentów symulacyjnych, w których brały udział implementowane algorytmy. Statystyki przedstawione zostały na wykresach wraz z komentarzem. Następnie zaprezentowałem wnioski na temat gry oraz poszczególnych algorytmów. W rozdziale 6 przedstawiłem podsumowanie wykonanych czynności, wnioski z eksperymentów oraz osiągnięty cel pracy.

2. Gra karciana Love Letter

W tym rozdziale opisuję kontekst oraz zasady gry Love Letter. Opisuję działanie każdej karty oraz przedstawiam główny cel gry - wygraną określonej ilości rund. Następnie przedstawiam problem i analizuję jego złożoność. Wszystkie załączone zdjęcia oraz instrukcja zaczerpnięte są z [1] oraz [2].



Rysunek 2.1. Love Letter - okładka

2.1. Opis zasad gry

W trakcie gry wcielamy się w rolę jednego z adoratorów księżniczki starającego się o zdobycie jej serca. W tym celu przygotowaliśmy list miłosny, który chcemy jej dostarczyć. Niestety, księżniczka pogrążona jest obecnie w żałobie i nie przyjmuje do siebie nikogo obcego, w związku z czym musimy znaleźć inny sposób na przekazanie jej naszego listu. Oprócz księżniczki, na dworze znajdują się inne postacie, z których każda ma mniejszy lub większy dostęp do komnat naszej wybranki i może oddać jej list. Przekazujemy więc naszą przesyłkę swojemu tajnemu posłańcowi, a na koniec gry księżniczka jako

pierwszy przeczyta ten list, który został przekazany przez najbardziej zaufaną postać. Serce wybranki zdobywa gracz, który jako pierwszy przekaże w ten sposób od 4 do 7 listów, w zależności od liczby graczy.

Cel i ustawienie początkowe rundy

Love Letter rozgrywa się jako serię rund. Grę wygrywa gracz o następującej ilości wygranych rund:

- 7 w grze na 2 graczy,
- 5 w grze na 3 graczy,
- 4 w grze na 4 graczy.

Każda runda dzieli się na tury, w których naprzemiennie jeden z graczy wykonuje ruch. Grę wygrywa ten z nich, który na końcu ostatniej tury posiada kartę o wyższym numerze.

Ustawienie początkowe każdej rundy wygląda następująco:

- przetasuj karty,
- odrzuć 1 wierzchnią kartę nie odkrywając jej (nie bierze udziału w rundzie),
- jeśli gra tylko 2 graczy, odrzuć 3 wierzchnie karty, odkryte,
- rozdaj po 1 karcie wszystkim graczom,
- jeśli jest to pierwsza runda, turę zaczyna gracz, który jako ostatni był na randce, w przeciwnym wypadku zaczyna zwycięzca poprzedniej rundy (jest to zasada wprowadzająca „klimat” do gry i pierwszego gracza można wybrać wg uznania).

Tura gracza i opis kart

Podczas swojej tury gracz dociąga jedną kartę ze stosu. Następnie wybiera jedną z dwóch kart, które posiada już w ręce, kładzie ją przed sobą tak, by była widoczna dla wszystkich i zastosowuje opisany na niej efekt - nawet jeśli jest negatywny. Zagrana karta pozostaje odkryta przez całą rundę, a druga pozostaje w ręce. Następnie tura przechodzi na osobę po lewej stronie aktywnego gracza.

W grze znajduje się 16 kart, w 8 typach. Każda z typów kart posiada wartość od 1 do 8. Są to kolejno: 4 karty Strażniczki, po 2 karty Kapłana, Barona, Pokojówki i Księcia, oraz po jednej karcie Króla, Hrabiny i Księżniczki. Ich szczegółowy opis wraz z wyglądem znajduje się poniżej:

**Rysunek 2.2.** Strażniczka

Na rysunku 2.2 przedstawiona jest karta typu Strażniczka. Zagrywając tę kartę należy wskazać jednego z pozostałych graczy i odgadnąć kartę którą posiada. Jeśli karta została prawidłowo odgadnięta, wskazany gracz odrzuca ją i przegrywa rundę.

**Rysunek 2.3.** Kapłan

Rysunek 2.3 przedstawia kartę typu Kapłan. Zagrywając tę kartę należy podglądać kartę wybranego gracza.



Rysunek 2.4. Baron

Na rysunku 2.4 przedstawiona jest karta typu Baron. Po zagraniu tej karty należy w ukryciu porównać drugą posiadaną kartą z wybranym graczem. Następnie ten gracz, który ma kartę o mniejszej wartości odrzuca swoją kartę i przegrywa rundę. W przypadku remisu nic się nie dzieje.



Rysunek 2.5. Pokojówka

Rysunek 2.5 przedstawia kartę typu Pokojówka. Zagranie tej karty sprawia, że gracz jest niewrażliwy na efekt pozostałych kart do czasu swojej następnej tury.



Rysunek 2.6. Książę

Na rysunku 2.6 przedstawiona jest karta typu Książę. Zagranie pozwala wybrać dowolnego gracza (w tym siebie), zmusić go do odrzucenia posiadanej karty i pociągnięcia następnej.



Rysunek 2.7. Król

Rysunek 2.7 przedstawia kartę typu Król. Po jej zagranie należy wymienić się drugą kartą z innym graczem.



Rysunek 2.8. Hrabina

Rysunek 2.8 przedstawia kartę typu Hrabina. Ta karta ma działanie pasywne. Nie wywiera efektu po zagranie, natomiast zmusza gracza do jej zagrania jeśli równocześnie posiada na ręce kartę typu Książę lub Król.



Rysunek 2.9. Księżniczka

Na rysunku 2.9 przedstawiona jest karta typu Księżniczka. Zagranie tej karty oznacza natychmiastową przegraną w rundzie. Ta zasada działa również, gdy gracz został zmuszony do zagrania tej karty, np. przez efekt karty Książę.

2.2. Analiza problemu

Z wyżej przedstawionych zasad wynika, że gra cechuje się wysokim stopniem losowości i jest niedeterministyczna, więc można ją przedstawić jako problem optymalizacyjny^[4]. W związku z tym powstaje również pytanie, jaka jest najlepsza strategia mieszana bądź czysta^[5] maksymalizująca prawdopodobieństwo wygrania gry. Dla dalszych rozważań zakładam, że gra toczy się pomiędzy dwoma graczami.

Najprostszym sposobem na znalezienie takiej strategii, byłoby stworzenie drzew probabilistycznych dla wszystkich możliwych stanów początkowych gry, a następnie opracowanie algorytmu podejmowania decyzji opartego na danych statystycznych. Jednakże, ilość i rozmiar tych drzew może być zbyt duża by znaleźć rozwiązanie problemu w rozsądnym czasie. Z tego powodu postanowiłem najpierw oszacować ile jest wszystkich możliwych przebiegów gry. Nim przejdę do obliczeń, wprowadzę kilka definicji by ustandaryzować używane pojęcia:

- *Decyzja* - inaczej *Zagranie*, jest to typ karty wraz ze sposobem jej wykorzystania. Przykładowo: Zagranie karty typu Strażniczka z wyborem karty typu Król, lub zagranie karty typu Księż z wyborem na gracza przeciwnego,
- *Podjęcie decyzji* - to wybór zagrania według zastosowanego algorytmu, które zostanie użyte jako ruch w grze,
- *Strategia* - ciąg decyzji podejmowanych według reguł określonych przez algorytm,
- *Scenariusz* - inaczej przebieg gry, chronologiczny ciąg decyzji podjętych przez obu graczy od początku do końca rundy; Jedna ze ścieżek w drzewie probabilistycznym dla danego stanu początkowego rundy, czyli pojedyncze rozwiązanie.

Oszacowanie ilości rozwiązań

Na przebieg każdej rundy wpływ mają następujące czynniki:

- Kolejność kart w talii na początku rundy
- Zagrywanie kart przez graczy

Zacząłem od oszacowania ilości możliwych rozwiązań. W każdej rundzie bierze udział wszystkie 16 kart. Zakładając, że każda z nich jest unikalna, to liczba wszystkich możliwych kolejności kart to permutacja, którą obliczam wzorem podanym w [3]:

$$P_n = n! , \text{ gdzie } n \in N^+$$

Dla $n = 16$, $n! = 20922789888000$. Część kart się powtarza, więc tę liczbę należy jeszcze podzielić przez permutacje powtarzających się kart Strażniczki, Kapłana, Barona, Pokojówki oraz Księcia. Razem jest to $4! * 2! * 2! * 2! * 2! = 384$. Tak więc liczba unikalnych kolejności kart wynosi:

$$20922789888000/32 = 54486432000 - 54\text{mld}, 864\text{mln i } 432 \text{ tys.}$$

Należy zwrócić uwagę, że jest to górne oszacowanie. Może to się wydawać nieintuicyjne, bo przecież jeśli przetasujemy w zadany sposób talię to określimy tylko kolejność w jakiej gracze będą dociągać karty nie biorąc pod uwagę decyzji podjętych przez graczy, ale należy zauważyć, że te podjęte decyzje doprowadzą ewentualnie do innego stanu talii, który już został wzięty pod uwagę w obliczeniach. Co więcej, istnieją warunki zmniejszające zadane oszacowanie, ponieważ część kolejności kart może prowadzić do tego samego rozwiązania. Za przykład niech posłuży sytuacja, w której pierwszy gracz ma w ręce dwie karty typu Książę. O ile zagranie Księcia na przeciwnika nie skończy gry, to doprowadzi do zmiany układu talii na inny, który jest już wzięty pod uwagę w obliczeniach. W tym miejscu należałoby wskazać dolne oszacowanie, niemniej jednak te wyliczenia byłyby zbyt obszerne, a nie są głównym celem pracy.

Z uwagi na rząd wielkości, stworzenie optymalnej strategii na podstawie analizy statystycznej wszystkich dostępnych rozwiązań byłoby czasochłonne. Z tego powodu, zamiast odpowiadać na pytanie „Jaka jest najlepsza strategia podejmowania decyzji?”, dużo łatwiej będzie odpowiedzieć na pytanie „która z podanych strategii jest najlepsza?”. Kierując się tą zasadą, w następnym rozdziale opisałem wybrane strategie, których skuteczność sprawdziłem implementując je w napisanej przeze mnie aplikacji. Pozostaje jeszcze sformalizować przedstawiony problem za pomocą modelu matematycznego.

2.3. Opis formalny problemu

W celu ujednolicenia używanych pojęć i skrótów, przedstawię je następująco:

Tablica 2.1. Pojęcia i skróty

Nazwa (typ) karty	Skrót nazwy	Możliwe zagrania	Skrót zagrania
Strażniczka	Str	Strażniczka z wyborem na [Nazwa karty]	Str_Kap, Str_Bar, ... , Str_K-a
Kapłan	Kap	Zagranie kapłana	Kap_Z
Baron	Bar	Zagranie barona	Bar_Z
Pokojówka	Pok	Zagranie pokojówki	Pok_Z
Książę	K-e	Książę na siebie; Książę na przeciwnika	K-e_S, K-e_P
Król	Krl	Zagranie Króla	Krl_Z
Hrabina	Hra	Zagranie Hrabiny	Hra_Z
Księżniczka	K-a	Zagranie Księżniczki	K-a_Z

Inne:

- „ręka” - karty, które gracz aktualnie posiada.
- Z - zbiór zagrań (decyzji), oznaczanych jako z . Jest to zbiór wszystkich dostępnych decyzji w grze.
- Z_i - zbiór zagrań (decyzji) dostępnych w danym stanie(węźle) i
- DK - druga karta, którą gracz posiada w ręce.
- *Typ Karty* - jeden z ośmiu rodzajów kart.
- $W(Karta)$ - wartość karty od 1 do 8.

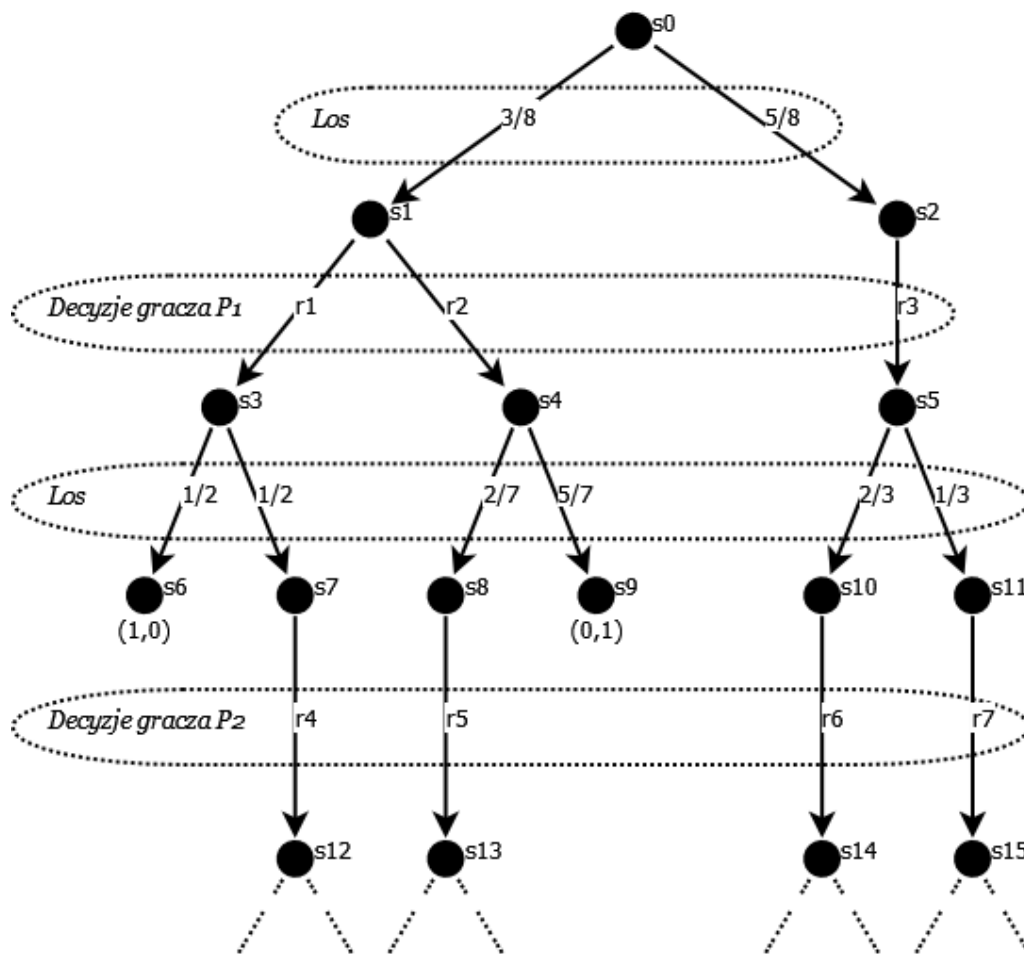
Patrząc z perspektywy jednego z graczy, cała gra składa się z szeregu etapów (tur), gdzie w każdym etapie gracz podejmuje decyzje, a stan początkowy następnego etapu jest wynikiem dwóch akcji: podjętej decyzji (której efekt możemy przewidzieć z pewnym prawdopodobieństwem) i reakcji gracza drugiego, której prawdopodobieństwo jest nieokreślone. Dodatkowo, rzeczywisty stan każdego etapu nie jest w pełni znany graczowi. Z tych cech wynika, że grę „Love Letter” możemy sklasyfikować jako **dynamiczny(wieloetapowy) proces podejmowania decyzji w warunkach niepewności**^[6]. Jej zapis formalny można przedstawić jako grę w postaci ekstensywnej^[7]:

$$T = (S, R), \text{ gdzie}$$

- T - graf skierowany
- S - zbiór wierzchołków (węzłów)
- R - relacja określona na parach wierzchołków (łuki)

Przypisując każdemu wierzchołkowi $s \in S$ jeden ze stanów rundy, a każdemu łukowi $r \in R$ decyzję danego gracza ze zbioru Z lub udział losu, otrzymamy zapis pozwalający jednoznacznie określić możliwe przebiegi konkretnej gry, podjęte przez graczy decyzje, ich stan wiedzy na każdym etapie oraz wypłaty graczy (wynik gry).

Ponieważ w każdej rundzie bierze udział dwóch graczy, oznaczmy ich jako P_1 i P_2 . Dodatkowo, część w której losowana jest karta oznaczmy jako Los . Zauważmy, że skoro korzeń s_0 to stan początkowy rundy przed losowaniem kart z talii, to wszystkie łuki wychodzące z s_0 są efektem losowania, a łuki z następnych poziomów należą kolejno do gracza P_1 , $Losu$, gracza P_2 itd. Poglądowo zilustrowałem to w następujący sposób (Rys. 2.10):



Rysunek 2.10. Przykład drzewa gry w postaci ekstensywnej

Ogólne wnioski:

- wszystkie łuki wychodzące z węzłów należących do $Losu$ oznaczone są prawdopodobieństwem sumującym się do 1.
- węzły s_6 i s_9 są liśćmi drzewa i zawierają wypłaty graczy (wynik gry) - 1 oznacza zwycięstwo, 0 oznacza przegraną. Przykładowo, w prezentowanym drzewie, w węźle s_6 wygrał gracz P_1 .

- należy zauważyć, że łuki r są ponumerowane i nie są wzajemnie jednoznaczne z zagraniami ze zbioru Z . Jak najbardziej wiele różnych węzłów może oznaczać to samo zagranie.

Zdarza się, że gracze nie wiedzą w jakim węźle danego poziomego drzewa się znajdują. Wracając do powyższego przykładu (Rys. 2.10) założmy, że łuki r_4 i r_5 są takim samym zagranem dostępnym dla gracza P_2 . Gracz nie wie czy gra znajduje się w stanie s_7 czy s_8 , ponieważ oba stany udostępniają ten sam zbiór zagrań, co odzwierciedla niepełną wiedzę gracza P_2 . Grupa takich stanów, które udostępniają ten sam zbiór zagrań, tworzy **zbiór informacyjny**. W całym drzewie występuje wiele zbiorów informacyjnych, ich zbiór określimy jako I . Ten zbiór, w którym znajduje się aktualnie gracz będziemy nazywać I_k , gdzie $I_k \subset I$ i $k \leq |I|$, a zbiór udostępnianych zagrań jako X_k (zauważmy, że $X_k \subset Z$). Zbiór węzłów, które następują po węzłach z I_k nazywamy **loterią**.

Przedstawiony model jednoznacznie określa stan gry i graczy. Dla uwzględnienia decyzyjności, do takiego modelu należy dodać jeszcze funkcję wyboru (decyzji). Oznaczmy ją jako d :

$$d(s) \rightarrow r, \text{ gdzie } r \text{ jest dowolnym łukiem wychodzącym ze stanu } s$$

Jest to deklaracja ogólna. Zdecydowanie ważniejsze są funkcje wyboru należące do graczy P_1 i P_2 , które uwzględniają zbiór informacyjny. Oznaczmy je odpowiednio jako d_1 i d_2 . Ich postać będzie następująca:

$$d_n(I_k, X_k) \rightarrow z, \text{ gdzie } z \text{ jest dowolnym zagranem należącym do } X_k \text{ i } n = \{1, 2\}$$

W takiej postaci widać, że gracze podejmą decyzję mając niepełną informację na temat stanu gry. W następnym rozdziale przedstawiam algorytmy, które będą dokładnie definiować funkcję wyboru gracza.

3. Przegląd wybranych algorytmów

W tym rozdziale przedstawiam algorytmy wyboru decyzji dla gry „Love Letter”. Są to kolejno: losowy, zachłanny, minimaxowy oraz Monte Carlo Tree Search. Dla każdego z nich opisuję jego użycie w odniesieniu do gry „Love Letter”. Ponieważ wszystkie algorytmy są implementowane w aplikacji komputerowej, przedstawiłem je również w postaci pseudokodu.

3.1. Algorytm losowy

3.1.1. Opis

Jest to najprostszy algorytm podejmowania decyzji. Na wejściu otrzymuje listę dostępnych decyzji, z których w całkowicie losowy sposób wybiera jedną. Każda z dostępnych decyzji ma takie samo prawdopodobieństwo wyboru przez ten algorytm.

3.1.2. Sposób wykorzystania

Z uwagi na prostotę algorytmu jest on wykorzystany głównie do przetestowania poprawności działania implementacji gry. Zastosowałem w nim jedną drobną modyfikację: nigdy nie podejmie decyzji o zagranie Księżniczki - oznacza to natychmiastową przegraną niezależnie od momentu gry, co wypaczałoby wyniki porównań algorytmów.

3.1.3. Zapis pseudokodem

```
1: function  $d_{losowa}(I_n, X_n)$ 
2:   for all  $z_i \in X_n$  do  $\triangleright i = 1..|X_n|$ 
3:     if  $z_i == \text{Księżniczka}$  then
4:        $X_n \leftarrow X_n - z_i$ 
5:     end if
6:   end for
7:   return  $\text{wylosujJeden}(X_n)$ 
8: end function
```

3.2. Algorytm zachłanny

3.2.1. Opis

Algorytm zachłanny polega na wyborze najlepszego możliwego zagrania dostępnego w danej chwili, nie analizując jego konsekwencji w przyszłości. Pomimo, że takie podejmowanie decyzji jest krótkowzroczne, to jest też łatwe w implementacji i daje atrakcyjne wyniki w niektórych problemach, np. przy szukaniu minimalnego drzewa rozpinającego^[10].

3.2.2. Sposób wykorzystania

W kontekście gry 'Love Letter', implementacja algorytmu zachłannego wymaga pewnego doprecyzowania. Najważniejszą częścią jest funkcja kryterialna oceniająca wartość zagrania, która w niektórych przypadkach (na przykład gdy zagranie może zakończyć grę) musi być oparta na prawdopodobieństwie wystąpienia kart u przeciwnika. Prawdopodobieństwo to uzależnione jest od stanu początkowego danej tury.

Rozważmy przypadek, w którym algorytm musi podjąć decyzję o zagraniu karty Strażniczki, lub karty Barona. Jest to pierwszy ruch gracza, a w widocznych kartach odrzuconych na starcie są karty Króla, Księcia i Pokojówki. Oznacza to, że w talii pozostało 9 kart, a jedna z odrzuconych jest niewidoczna, niemniej jednak ją też trzeba brać pod uwagę. Wyliczenie prawdopodobieństwa $P[Karta]$ jaką kartę ma przeciwnik jest tym momencie proste, jednak trzeba jeszcze wziąć pod uwagę drobny szczegół - czy do liczenia $P[Karta]$ wliczać karty posiadane w ręce. Z jednej strony wydaje się to nielogiczne i może prowadzić do wybierania niekorzystnych decyzji (co przeczyłoby idei algorytmu zachłannego) w danym stanie, z drugiej strony można to potraktować jako element blefu, który jest nieodłączną częścią każdej gry towarzyskiej. W swojej implementacji założyłem absolutną zachłanność algorytmu i karty posiadane na ręce są pomijane w obliczeniach. Wobec powyższego, prawdopodobieństwa wystąpienia kart u przeciwnika wynikające z ustalonego stanu początkowego są następujące:

Tablica 3.1. Przykład rozkładu prawdopodobieństwa w przypadku odrzucenia kart Króla, Księcia i Pokojówki

Karta	$P[Karta]$
Strażniczka	30%
Kapłan	20%
Baron	10%
Pokojówka	10%
Książę	10%
Hrabina	10%
Księżniczka	10%

Zagranie karty Baron oznacza porównanie drugiej karty z kartą przeciwnika. Łatwo policzyć, że w 70% przypadków skończyłoby to się porażką, a w 30% nie było by żadnego efektu. Drugim dostępnym ruchem jest zagranie karty Strażniczki, a w jej przypadku najlepszym wyborem jest wytypowanie Kapłana, co daje 20% szans na zwycięstwo i 80% szans, że nie nastąpi żaden efekt. Zauważmy, że gdybyśmy wliczali posiadane karty do obliczenia prawdopodobieństwa, wystąpienie Barona i Kapłana byłoby tak samo możliwe. W takich przypadkach algorytm powinien zawsze celować w kartę z wyższym numerem. By formalnie stwierdzić, jaka decyzja z powinna zostać podjęta, musimy obliczyć funkcję kryterialną dla dostępnych zagrań i wybrać to zagranie, dla której funkcja przyjmuje wyższą wartość. Przyjmijmy, że lokalna funkcja kryterialna wygląda następująco:

$$F(z) = 1 + \text{prawdopodobienstwo_wygranej} - \text{prawdopodobienstwo_przegranej}$$

Po podstawieniu otrzymujemy:

$$\begin{aligned} F(\text{Str_Kap}) &= 1.2 \text{ i } F(\text{Bar}) = 0.3 \\ F(\text{Str_Kap}) &> F(\text{Bar}) \Rightarrow z = \text{Str_Kap} \end{aligned}$$

Najlepszą decyzją w tym wypadku jest zagranie karty Strażniczki z wyborem karty Kapłana. Jak jednak na podstawie powyższego wzoru ocenić zagranie karty Kapłana, Pokojówki lub Króla? Każda z nich wymaga indywidualnej oceny. Mając na uwadze, że w kontekście strategii zachłannej decyzja zawsze powinna być najlepsza w danym stanie, ustaliłem następujące warunki, którymi kierowałem się przy tworzeniu funkcji kryterialnej dla zagrań każdej z kart:

- Strażniczka - ocena zagrania (wartość funkcji kryterialnej) wzrasta gdy pozwala wyeliminować przeciwnika,
- Kapłan - efekt karty jest neutralny i ocena jego zagrania będzie zawsze stała,
- Baron - ocena zagrania wzrasta, gdy mamy drugą kartę silniejszą niż może mieć przeciwnik i maleje, gdy jest odwrotnie,
- Pokojówka - podobnie jak w przypadku kapłana, ocena zagrania karty będzie stała,
- Księżę - ocena zagrania na przeciwnika wzrasta z prawdopodobieństwem Księżniczki u przeciwnika. Ocena zagrania na siebie jest stała, lecz 0, gdy druga karta to Księżniczka,
- Król - jak w przypadku Kapłana i Pokojówki, ocena zagrania jest stała,
- Hrabina - ocena stała, jednak musimy ją zagrać, gdy druga posiadana karta to Król lub Księżę,
- Księżniczka - nie może być nigdy wyrzucona,
- Dodatkowo, jeśli na podstawie powyższych przesłanek oba zagrania mają taką samą wartość, powinna być podjęta decyzja o zagranie karty o niższej wartości W . By to osiągnąć, do oceny dodane są wartości od 0.008 do 0.002.

Opierając się na powyższych wytycznych, zapisałem algorytm w formie pseudokodu.

3.2.3. Zapis pseudokodem

Wykorzystane zmienne pomocnicze:

- $P[]$ - tablica prawdopodobieństwa wystąpień kart u przeciwnika. ($P[Karta]$ oznacza prawdopodobieństwo dla konkretnej karty)
- *decyzja* - zagranie, które ma zostać zwrócone

```

1: function  $d_{zachlanna}(I_n, X_n)$ 
2:    $P[] \leftarrow \text{obliczPrawdopodobienstwo}()$ 
3:    $decyzja \leftarrow NULL$  ▷ Szukanie zagrania o najwyższej wycenie
4:   for all  $z_i \in X_n$  do ▷  $i = 1..|X_n|$ 
5:     if  $F(decyzja, P[]) < F(z_i, P[])$  then
6:        $decyzja \leftarrow z_i$ 
7:     end if
8:   end for
9:   return  $decyzja$ 
10: end function

```

Gdzie funkcja kryterialna wygląda następująco:

```

1: function  $F(z, P[])$ 
2:   switch  $z$  do
3:     case Str_Kap ▷ Zagranie Strażniczki na dany typ karty
4:       return  $1 + P[Kaplan] + 0.008$ 
5:     case Str_Bar
6:       return  $1 + P[Baron] + 0.008$ 
7:     ...
8:     case Str_K-a
9:       return  $1 + P[Ksiezniczka] + 0.008$ 
10:    case Kap_Z
11:      return  $1 + 0.007$ 
12:    case Bar_Z ▷ Kryterium zależy od wartości  $W(DK)$ 
13:       $szansePrzegranej \leftarrow 0$ 
14:       $szanseWygranej \leftarrow 0$ 
15:      for all  $typKarty$  do
16:        if  $W(DK) < W(typKarty)$  then
17:           $szansePrzegranej \leftarrow szansePrzegranej + P[typKarty]$ 
18:        else if  $W(DK) > W(typKarty)$  then
19:           $szanseWygranej \leftarrow szanseWygranej + P[typKarty]$ 
20:        end if

```



```

21:         end for
22:         return  $1 - \text{szansePrzegranej} + \text{szanseWygranej} + 0.006$ 
23:     case Pok_Z
24:         return  $1 + 0.005$ 
25:     case K-e_S
26:         return  $1 + P[K-a] + 0.004$ 
27:     case K-e_P
28:         if  $DK == K-a$  then
29:             return 0
30:         else
31:             return  $1 + 0.004$ 
32:         end if
33:     case Krl_Z
34:         return  $1 + P[K-a] + P[Hra] + 0.003$ 
35:     case Hra_Z
36:         return  $1 + 0.002$ 
37:     case K-a_Z
38:         return 0
39: end function

```

3.3. Algorytm min-max

3.3.1. Opis

Algorytm minimaksowy polega na „minimalizowaniu maksymalnych możliwych strat” bądź alternatywnie na „maksymalizacji minimalnego zysku (wypłaty)”^[11]. Zgodnie z [13] często stosuje się go do gier o następujących zasadach:

- występuje dwóch graczy
- ruchy wykonywane są naprzemiennie
- w każdym stanie istnieje skończona liczba decyzji do podjęcia
- stan i podjęta decyzja jednoznacznie wyznaczają stan następny
- każdy stan może zakwalifikować do jednej z następujących kategorii:
 - wygrana pierwszego gracza
 - wygrana drugiego gracza
 - remis

- sytuacja nierozstrzygnięta

Najczęstsze implementacje polegają na przeszukiwaniu drzewa dalszych przebiegów gry poczynając od zadanego stanu początkowego, tak jak na przykład w warcabach^[12]. Istnieją także implementacje opierające się na liczbowej ocenie ruchu^[13] i na tej idei opieram swoją implementację algorytmu minimaksowego.

3.3.2. Sposób wykorzystania

Jak ustaliliśmy wcześniej, gracz nie wie w jakim dokładnie stanie znajduje się gra, zna natomiast zbiór informacyjny I_n . Z tego względu, z perspektywy gracza P_1 podjęcie decyzji $z_i \in X_n$ nie określa jednoznacznie następnego węzła w drzewie, lecz loterię. Mimo to, jesteśmy w stanie statystycznie ocenić możliwości przeciwnika, a tym samym zminimalizować naszą maksymalną stratę. Ponieważ zysk należy rozumieć jako zwiększenie szansy na wygraną (tak jak w strategii zachłannej), to strata oznacza zwiększenie szansy na przegraną.

Rozważmy scenariusz, w którym gracz pierwszy posiada kartę Księcia oraz Pokojówki. Na stosie pozostało 2 karty, 1 karta jest zakryta zgodnie z zasadami rozpoczęcia rundy i przeciwnik również posiada 1 kartę. Wśród tych 4 nieznanych graczowi kart są karty Strażniczki, Barona, Księcia oraz Księżniczki. Prawdopodobieństwo wystąpienia każdej z nich u przeciwnika wynosi 25% i razem stanowią one tablicę prawdopodobieństwa $P[]$. Przyjmijmy, że wykorzystywana w poprzedniej strategii funkcja kryterialna $F(z, P[])$ to maksymalizacja zysku i nazwijmy ją $F_{max}(z, P[])$. Zgodnie z jej definicją $F_{max}(K-e, P[]) = 1.29$ i $F_{max}(Pokojowka, P[]) = 1.05$, więc zagranie Księcia maksymalizuje szansę na wygraną. Jeśli jednak weźmiemy pod uwagę, że w 75% przypadków nie wygrywamy, wówczas musimy rozważyć odpowiedź przeciwnika. W tym wypadku jest 6 par kart które może posiadać przeciwnik, więc dla każdej z nich, zgodnie ze strategią zachłanną, należy obliczyć funkcję kryterialną oraz szansę na przegraną. Dodatkowo należy zauważyć, że z perspektywy przeciwnika w każdym przypadku gracz pierwszy ma inny rozkład prawdopodobieństwa wystąpienia karty. Pełne obliczenia znajdują się w tabeli poniżej.

Gracz pierwszy po zagranie Księcia posiada tylko kartę Pokojówki. $F(z_1)$ oznacza ocenę zagrania pierwszej karty, $F(z_2)$ oznacza ocenę zagrania drugiej karty.

Tablica 3.2. Scenariusze reakcji przeciwnika na decyzję w zadanym przypadku

Karty 1 i 2	$P[]$ pierwszego gracza	$F(z_1)$	$F(z_2)$	Szanse przegranej
Str ; Bar	$P[Pok] = P[K-e] = P[K-a] = 33.(3)\%$	1.41	0.6	33.(3)%
Str ; K-e	$P[Pok] = P[Bar] = P[K-a] = 33.(3)\%$	1.41	1.37	33.(3)%
Str ; K-a	$P[Str] = P[Bar] = P[K-e] = 33.(3)\%$	1.41	0	33.(3)%
Bar ; K-e	$P[Str] = P[Pok] = P[K-a] = 33.(3)\%$	1.39	1.37	100%
Bar ; K-a	$P[Str] = P[Pok] = P[K-e] = 33.(3)\%$	2.06	0	100%
K-e ; K-a	$P[Str] = P[Pok] = P[Bar] = 33.(3)\%$	1.04	0	0%

Dodatkowo, wprowadźmy funkcję $K(z) = F_{max} - F_{min}$, która wyceni dane zagranie.

Każdy rozważany scenariusz ma taką samą szansę wystąpienia, czyli $\frac{1}{6}$. Statystycznie szansa na przegraną wynosi więc:

$$\frac{1}{6} * (\frac{1}{3} + \frac{1}{3} + \frac{1}{3} + 1 + 1 + 0) = \frac{1}{6} * 3 = 0.50$$

Warunkiem wystąpienia możliwości przegrania, jest brak wygranej po zagranie Księcia na przeciwnika, czyli ostatecznie szanse na przegraną wynoszą:

$$F_{min}(K-e_P, P[]) = 0.75 * 0.50 = 0.375$$

Modyfikując o tę wartość ocenę zagrania Księcia otrzymujemy finalnie:

$$K(K-e_P) = F_{max}(K-e_P, P[]) - F_{min}(K-e_P, P[]) = 1.29 - 0.375 = 0.915$$

W przypadku zagrania Pokojówki szanse przegrania wynoszą 0%, ponieważ zgodnie z zasadami gry jesteśmy odporni na działanie kart przeciwnika:

$$K(Pok_Z) = F_{max}(Pok_Z, P[]) - F_{min}(Pok_Z, P[]) = 1.05 - 0 = 1.05$$

Wynika z tego, że decyzją, która minimalizuje maksymalną stratę jest zagranie $z = Pok_Z$.

Na podstawie powyższych rozważań zapisałem implementację algorytmu zachłannego w postaci pseudokodu.

3.3.3. Zapis pseudokodem

Wykorzystane zmienne pomocnicze:

- $K(z)$ - oznacza wspomnianą wyżej funkcję kryterialną zagrania.
- $L[Y_n]$ - zbiór możliwych zbiorów decyzji przeciwnika na danym poziomie drzewa.

- Y_n - oznacza możliwy zbiór zagrań dostępnych dla przeciwnika na danym poziomie drzewa.
- Funkcja $R(L[Y_n])$ oznacza prawdopodobieństwo przegranej po reakcji przeciwnika.
- RK - zbiór zagrań będących reakcją przeciwnika.
- $P_{porazki}$ - tablica zawierająca prawdopodobieństwo porażki po danym zagranie

```

1: function  $D_{minimaksowa}(I_n, X_n)$ 
2:    $P[] \leftarrow \text{obliczPrawdopodobienstwo}()$ 
3:   for all  $z_i \in X_n$  do ▷  $i = 1..|X_n|$ 
4:      $K(z_i) \leftarrow F_{max}(z_i, P[]) - F_{min}(z_i, P[])$ 
5:   end for
6:    $decyzja \leftarrow z_0$  ▷ Szukanie zagrania o najwyższej wycenie
7:   for all  $z_i \in X_n$  do
8:     if  $K(decyzja) < K(z_i)$  then
9:        $decyzja \leftarrow z_i$ 
10:    end if
11:  end for
12:  return  $decyzja$ 
13: end function

```

Funkcja $F_{max}(z, P[])$ jest identyczna do $F(z, P[])$ występującej w strategii zachłannej, w związku z czym przedstawiam wyłącznie $F_{min}(z, P[])$.

```

1: function  $F_{min}(z, P[])$ 
2:    $L[Y_n] \leftarrow \text{utwórz listę możliwych zbiorów decyzji u przeciwnika}$ 
3:   switch  $z$  do
4:     case Str_Kap ▷ Szanse, że Strażniczka nie przyniesie zwycięstwa
5:       return  $(1 - P[Kap]) * R(L[Y_n])$ 
6:     case Str_Bar
7:       return  $(1 - P[Bar]) * R(L[Y_n])$ 
8:     ...
9:     case Str_K-a
10:      return  $(1 - P[K-a]) * R(L[Y_n])$ 
11:    case Kap_Z
12:      return  $R(L[Y_n])$ 
13:    case Bar_Z ▷ Szanse, że porównanie zakończy się remisem
14:       $szanseRemisu \leftarrow P[DK]$ 
15:      return  $szanseRemisu * R(L[Y_n])$ 
16:    case Pok_Z

```

```

17:         return 0
18:     case K-e_P
19:         return  $(1 - P[K-a]) * R(L[Y_n])$ 
20:     case K-e_S
21:         if  $DK == K-a$  then
22:             return 0
23:         else
24:             return  $R(L[Y_n])$ 
25:         end if
26:     case Krl_Z
27:         return  $R(L[Y_n])$ 
28:     case Hra_Z
29:         return  $R(L[Y_n])$ 
30:     case K-a
31:         return 10
32: end function

```

Funkcja $R(L[Y_n])$ oznacza szansę porażki po reakcji przeciwnika.

```

1: function  $R(L[Y_n])$ 
2:   for all  $Y_n \in L[Y_n]$  do
3:      $P[] \leftarrow \text{obliczPrawdopodobienstwo}()$ 
4:      $RK \leftarrow RK \cup D_{zachlanna}(Y_n, P[])$   $\triangleright RK$  to zbiór reakcji dla każdego ze zbiorów  $Y_n$ 
5:   end for
6:   for all  $z_i \in RK$  do  $\triangleright$  Sprawdzenie szansy porażki,  $i = 1..|RK|$ 
7:     switch  $z_i$  do
8:       case Str_DK
9:          $P_{porazki}[z_i] \leftarrow 1$ 
10:      case Bar_Z
11:        if  $W(DK) < W(DK_{przeciwnika})$  then
12:           $P_{porazki}[z_i] \leftarrow 1$ 
13:        else
14:           $P_{porazki}[z_i] \leftarrow 0$ 
15:        end if
16:      case K-e_Z
17:        if  $DK == K-a$  then
18:           $P_{porazki}[z_i] \leftarrow 1$ 
19:        else
20:           $P_{porazki}[z_i] \leftarrow 0$ 

```

```

21:         end if
22:         case default  $P_{porazki}[z_i] \leftarrow 0$ 
23:     end for
24:     return  $\sum_{i=0}^{|RK|} \frac{1}{|RK|} * P_{porazki}[i]$ 
25: end function

```

3.4. Algorytm Monte Carlo Tree Search

3.4.1. Opis

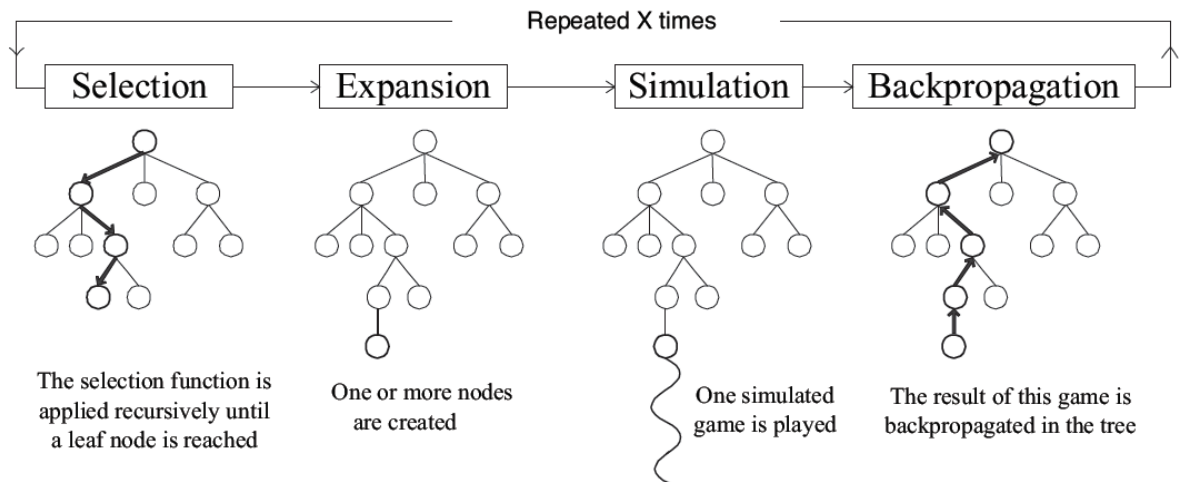
Drzewo Przeszukiwań Monte Carlo jest metodą heurystycznego podejmowania decyzji, często wykorzystywaną w grach typu Hex^[14] czy Go^[15]. W przeciwieństwie do deterministycznych algorytmów przeszukujących drzewo decyzyjne, MCTS buduje drzewo wariantów poprzez losowe próbkowanie najbardziej obiecujących ruchów. Dzięki temu jego dużą zaletą jest możliwość efektywnego wykorzystania w grach o wysokim rozgałęzieniu drzewa decyzyjnego^[16]. Popularność wykorzystania tego algorytmu stale rośnie, czego przykładem jest zastosowanie w niedeterministycznej grze planszowej Osadnicy z Catanu^[19], czy w strategii turowej Total War: Rome II^[18]. Główne cechy algorytmu:

- przerywalność algorytmu w dowolnym momencie
- zależność wydajności od czasu - im dłużej działa algorytm, tym lepsze osiąga i
- uniwersalność - algorytm do działania potrzebuje wyłącznie zbioru dostępnych ruchów oraz możliwości przeprowadzenia losowej rozgrywki od danego stanu
- niezależność od wiedzy eksperckiej występującej w grze^[16]

3.4.2. Sposób wykorzystania

Jak opisali autorzy artykułu *Monte-Carlo Tree Search: A New Framework for Game AI*^[17], zasady działania algorytmu są następujące: otrzymując zbiór informacyjny I_n oraz zbiór dopuszczalnych ruchów (decyzji) X_n , tworzony jest korzeń T , oraz węzły $s_0 \dots s_i$ ($i = 1..|X_n|$) odpowiadające stanom gry po wykonaniu dostępnych ruchów (decyzji). Następnie przez zadany czas t powtarzane są następujące kroki:

1. **Selekcja** - zaczynając od korzenia T , wybieraj kolejne węzły balansując pomiędzy **eksploatacją** i **eksploracją**, aż dotrzesz do węzła-liścia L .
2. **Ekspansja** - jeśli wybór L nie kończy gry, utwórz węzły potomne i wśród nich wylosuj węzeł C
3. **Symulacja** - dla wybranego węzła przeprowadzać losową symulację, aż do osiągnięcia wyniku.
4. **Propagacja wsteczna** - każdy węzeł od C do L jest aktualizowany o wartość wyniku.



Rysunek 3.1. Schemat działania algorytmu MCTS^[17]

Szczególnym elementem jest selekcja, ponieważ ona odpowiada za kierunek rozrastania się drzewa. Eksploatacja oznacza wybieranie ruchów o wysokiej częstości wygranych, a eksploracja to badanie ruchów o niskiej liczbie przeprowadzonych symulacji. Brak równowagi pomiędzy tymi dwoma strategiami próbkowania doprowadzi do zakłamań, na przykład poprzez pułapkę optimum lokalnego. W związku z tym w 2006 roku Kocsis i Szepesvári opracowali wzór równoważący selekcję nazwany Upper Confidence bound applied to Trees (UCT) (górna granica ufności)^[20]:

$$v_i + E * \sqrt{\frac{\ln N}{n_i}}$$

- v_i - estymowana wartość węzła,
- E - empirycznie dobierany parametr eksploracji, zazwyczaj $\sqrt{2}$,
- N - ilość odwiedzin węzła-rodzica,
- n_i - ilość odwiedzin w danym węźle.

Ważne jest, że każdy węzeł posiada informację o szacunkowej wartości opartej na wynikach symulacji, oraz o liczbie przeprowadzonych symulacji.

Ze względu na fakt, że algorytm do działania wymaga wyłącznie listy dostępnych ruchów oraz możliwości przeprowadzenia symulacji (na przykład za pomocą interfejsu w grze), jego implementacja do gry „Love Letter” nie wymaga specjalnych modyfikacji. Powyższa wiedza w pełni pozwala na wykorzystanie algorytmu w grze.

3.4.3. Zapis pseudokodem

Wykorzystane zmienne pomocnicze:

- $Nast(s, z)$ - funkcja wskazująca następnik węzła u po wyborze zagrania (łuku) z ,

- X_L - zbiór możliwych zagrań dostępnych w węźle L ,
- $wynik$ - wypłata gracza.

```

1: function  $d_{mcts}(I_n, X_n)$ 
2:    $T \leftarrow$  utwórz korzeń  $T$  z obecnego zbioru informacyjnego  $I_n$ 
3:   repeat
4:      $L \leftarrow$  selekcja( $T$ )
5:     if  $L$  nie jest końcem gry then
6:       for all  $z_i \in X_L$  do ▷  $i = 1..|X_L|$ ,
7:          $L \leftarrow$  dodajDziecko( $Nast(L, z_i)$ )
8:       end for
9:        $C \leftarrow$  wylosujDziecko( $L$ )
10:       $wynik \leftarrow$  symuluj( $C$ ) ▷ Symulacja
11:      repeat ▷ Propagacja wsteczna
12:         $C \leftarrow$  aktualizuj( $wynik$ )
13:         $C \leftarrow$  rodzic( $C$ )
14:      until  $C == T$ 
15:    end if
16:  until koniec czasu
17:   $decyzja \leftarrow$  najwięcejSymulacji( $R$ ) ▷ ruch z najbardziej obiecującego węzła
18:  return  $decyzja$ 
19: end function

```


4. Implementacja

W tym rozdziale opisuję wymagania wobec projektu, który zawiera implementację gry „Love Letter” i opisanych algorytmów. Następnie przedstawiam koncepcję wykonania wraz z diagramami UML, a na końcu prezentuję system i opisuję problemy napotkane w trakcie realizacji.

4.1. Analiza wymagań

Ponieważ celem pracy jest porównanie działania algorytmów gry karcianej, pierwszym krokiem potrzebnym do jego realizacji jest zaimplementowanie samej gry. Jak zostało to napisane w [24], przed właściwą implementacją programu należy określić jego wymagania funkcjonalne i нефункционалне. Większość wymagań jest już udokumentowana w postaci zasad gry opisanych w rozdziale 2, należy jednak wyszczególnić pewne dodatkowe wymogi:

- program musi udostępniać interfejs do którego można podłączyć różne algorytmy podejmujące decyzje,
- interfejs musi udostępniać aktualny stan gry oraz zbiór decyzji możliwych do podjęcia,
- interfejs musi zawierać funkcjonalność pozwalającą na utworzenie nowej instancji gry z zadanym stanem początkowym i podjętą decyzją. W nowej instancji wszystkie nieznanе elementy gry są losowane na nowo (np. talia karta jest przetasowywana),
- program musi umożliwiać wybranie dwóch dowolnych algorytmów, które będą ze sobą grać, oraz ustawienie liczby gier które rozegrają,
- algorytm MCTS powinien mieć możliwość ustawienia dwóch parametrów: czasu wykonania, oraz algorytmu przeciwnika wykorzystywanego w symulacjach,
- po skończonej serii gier program powinien wyświetlać szczegółowe statystyki rozgrywek w formie wykresu i zapisywać dane do pliku.

Zwracane statystyki powinny zawierać następujące:

- procentowa liczba zwycięstw każdego algorytmu z wyszczególnieniem na tury w których została zakończona runda,

- rozkład procentowy zakończeń gry - ile rund skończyło się zagranieniem karty Barona, Strażniczki, Księcia lub w ostatniej turze po wyczerpaniu talii,
- wyszczególnienie procentowej liczby zwycięstw w zależności od tego który algorytm wykonuje ruch jako pierwszy.

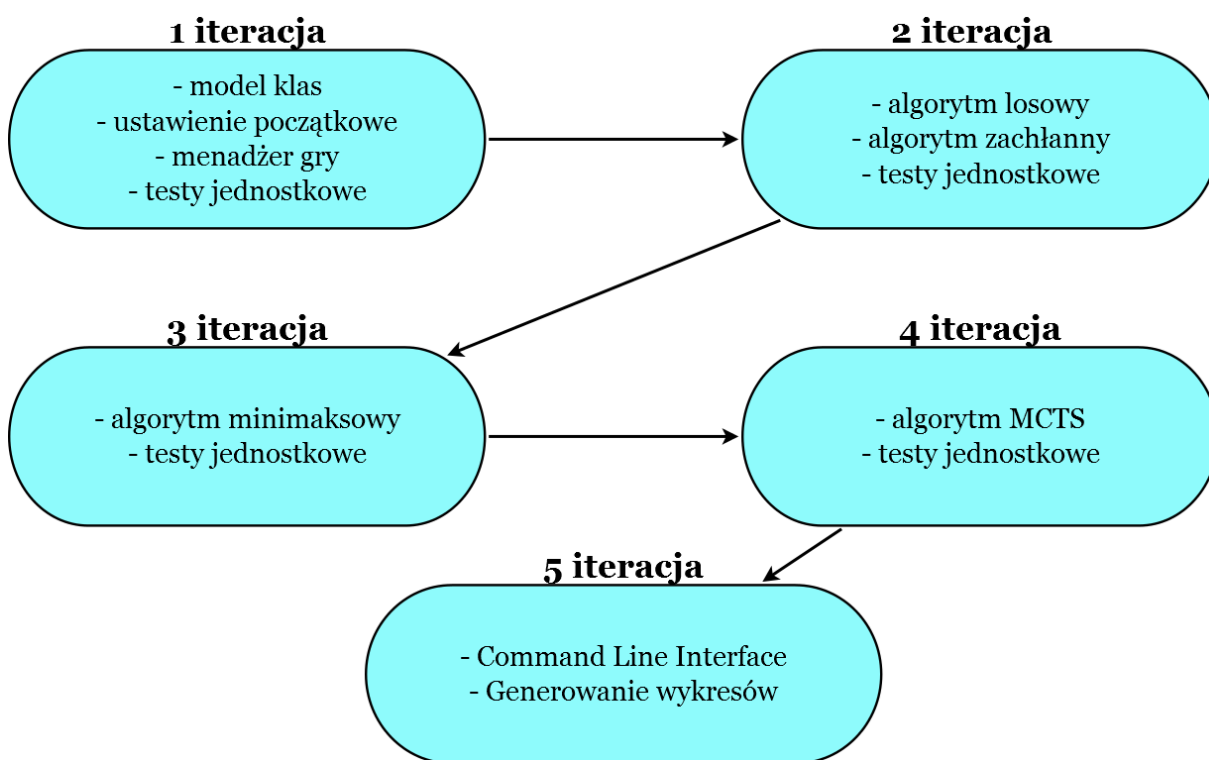
Wymagania niefunkcjonalne:

- program powinien uruchamiać się bez względu na środowisko,
- program nie powinien wymagać od użytkownika znajomości szczegółów implementacji,
- program powinien udostępniać pomoc,
- aplikacja powinna zostać zaprojektowana w taki sposób, żeby można było dodawać kolejne algorytmy do aplikacji bez konieczności modyfikacji kodu.

4.2. Koncepcja wykonania i wykorzystane technologie

By skupić się wyłącznie na merytorycznym działaniu projektu, postanowiłem zaimplementować program w postaci aplikacji konsolowej, gdzie komunikacja odbywa się za pomocą wiersza poleceń^[22]. Wprowadzenie warstwy graficznej jest zbędne wobec postawionych wymagań.

Pomimo prostoty i niezmienności wymagań funkcjonalnych, projekt został wykonany zgodnie z metodyką zwinną (ang. agile methodology)^[25], która wciąż zyskuje na popularności względem metodyk kaskadowych. Jej główną cechą jest porzucenie szczegółowego planowania projektu na rzecz obserwacji i reagowania. Całość oprogramowania jest wytwarzana w kolejnych iteracjach, w których implementowane są kolejne funkcjonalności, oraz testowane i poprawiane wcześniejsze. Poniżej prezentuję zaplanowane przeze mnie iteracje tworzenia projektu (Rys. 4.1.):



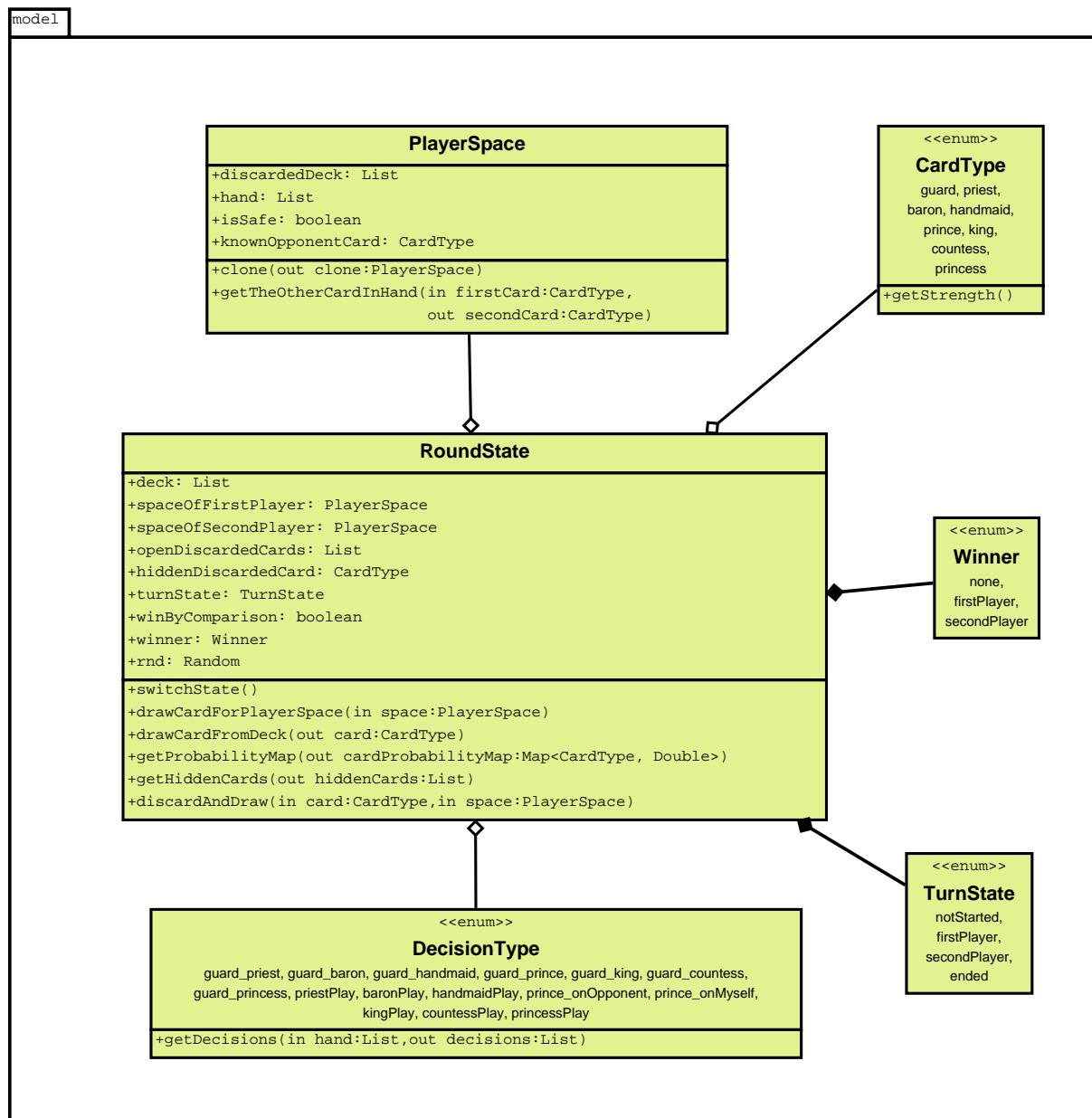
Rysunek 4.1. Zaplanowane iteracje

Ze względu na osobiste umiejętności, do implementacji wybrałem język programowania Java SE w wersji 8^[23]. Jest to język obiektowy, o ścisłym typowaniu, działający na maszynie wirtualnej Javy, co zapewnia kompatybilność ze wszystkimi systemami operacyjnymi. Całość projektu wykonałem w środowisku programistycznym IntelliJ IDEA w darmowej wersji Community^[26].

4.3. Diagramy

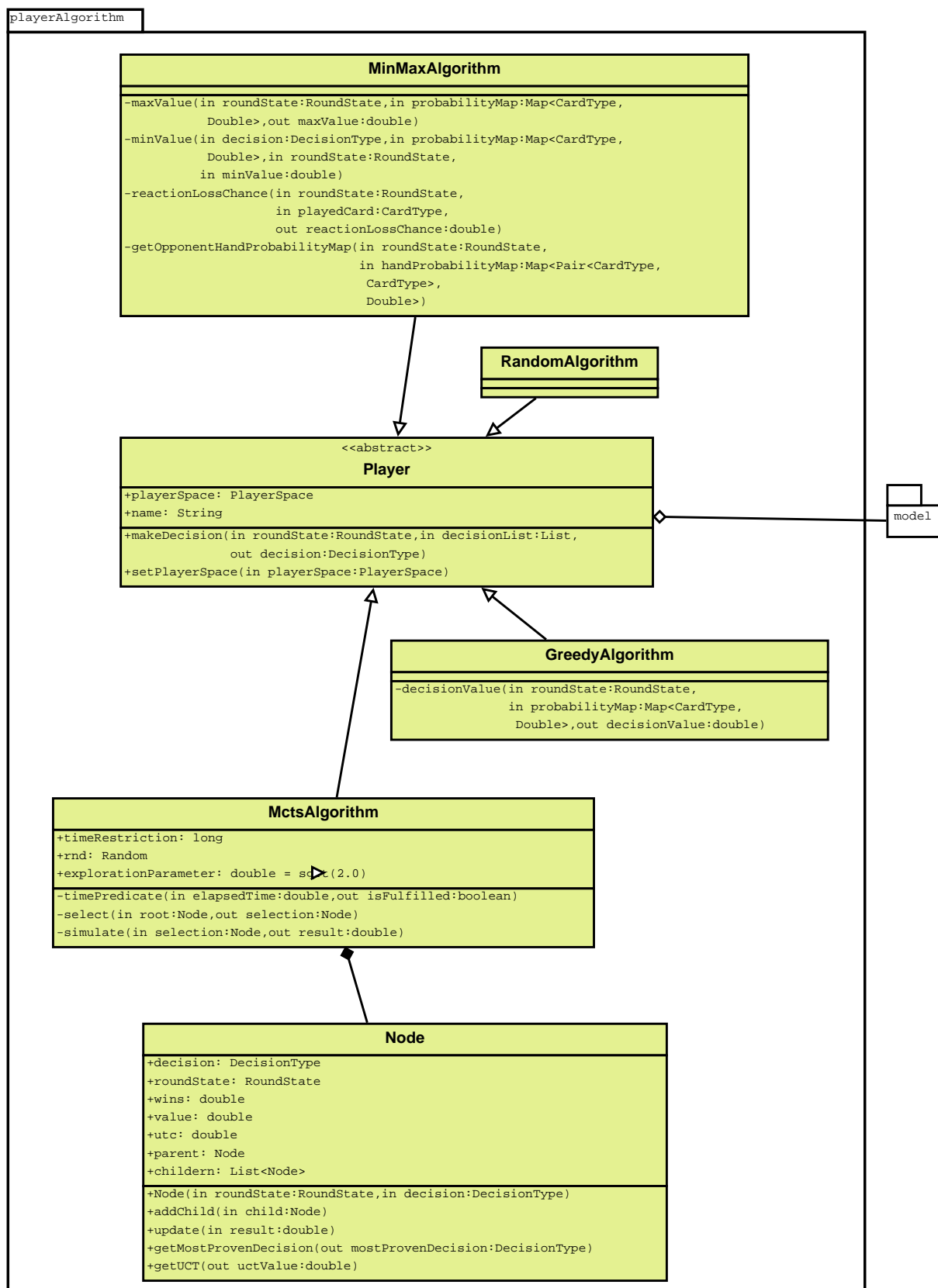
Poniżej prezentuję diagramy klas UML, dokumentujące obecną strukturę kodu.

4.3.1. Diagram klas pakietu model

Rysunek 4.2. Diagram klas pakietu *model*

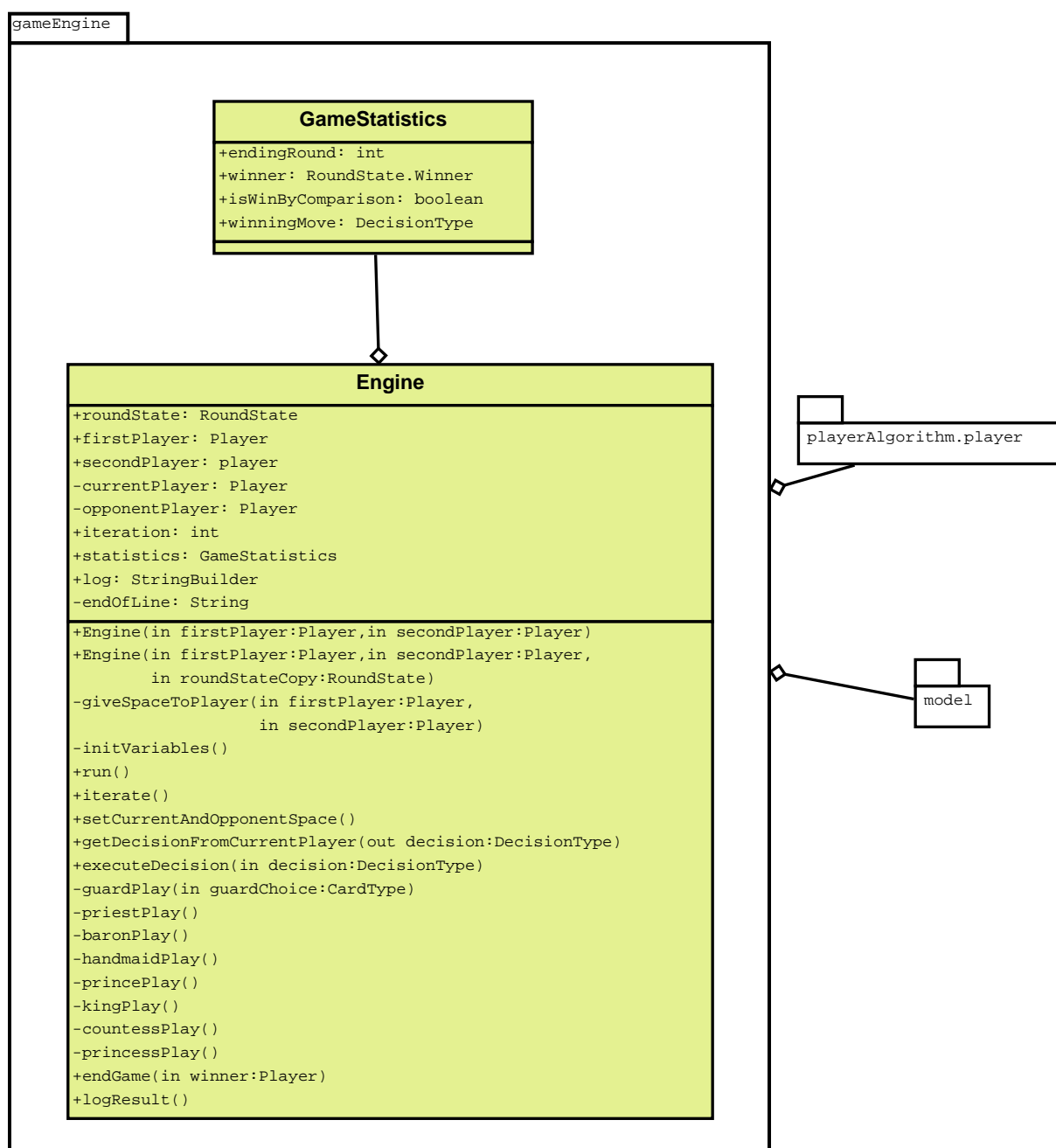
W pakiecie *model* znajdują się wszystkie elementy składające się na samą grę. Głównym elementem jest klasa **RoundState**, odpowiadająca wierzchołkom s_i należącym do zbioru S . Typy kart i zbiór zagrań Z reprezentowane są w postaci klas typu enum.

4.3.2. Diagram klas pakietu player

Rysunek 4.3. Diagram klas pakietu *player*

W pakiecie *player* głównym elementem jest klasa abstrakcyjna *Player*, po której dziedziczą implementowane przeze mnie algorytmy. Warto zauważyć, że wykorzystuje również pakiet *model*. Takie polimorficzne rozwiązanie znacznie ułatwia dodawanie kolejnych algorytmów do aplikacji, bez konieczności modyfikowania jej.

4.3.3. Diagram klas pakietu engine



Rysunek 4.4. Diagram klas pakietu *gameEngine*

Pakiet *gameEngine* realizuje zasady gry w sensie mechaniki działania kart. Klasa *Engine* jest menadżerem i zarządza stanem gry na podstawie decyzji otrzymanej z klasy *Player*. Metoda *giveSpaceToPlayer* przekazuje obiektom klasy *Player* referencję do *PlayerSpace* i kopię obiektu *RoundState*.

4.4. Listingi

Pisząc kod aplikacji, oprócz kierowania się paradygmatami programowania obiektowego, bardzo ważne jest również by dbać o czytelność kodu. W [27] można znaleźć wiele porad i metod zwiększających jakość kodu. Jak taki kod powinien wyglądać bardzo dobrze opisuje Grady Booch:

„Czysty kod jest prosty i bezpośredni. Czysty kod czyta się jak dobrze napisaną prozę. Czysty kod nigdy nie zaciemnia zamiarów projektanta; jest pełen abstrakcji i prostych ścieżek sterowania.”^[28]

Najpopularniejszą metodologią wspomagającą tworzenie czystego kodu jest SOLID. Jest to mnemonik opisujący pięć podstawowych założeń programowania obiektowego^[29]. SOLID oznacza:

- S (Single responsibility principle) - Klasa powinna posiadać jedną odpowiedzialność
- O (Open/closed principle) - Zmiana wymagań wobec aplikacji nie powinna w konsekwencji zmieniać kodu, lecz dodawać nowe funkcjonalności bez usuwania starych.
- L (Liskov substitution principle) - W przypadku dziedziczenia, klasy pochodne muszą mieć możliwość realizacji funkcji klasy bazowej.
- I (Interface segregation principle) - Klient nie może być uzależniony od nieużywanego przez niego interfejsu.
- D (Dependency inversion principle) - Funkcjonalność wysokopoziomowych modułów nie może zależeć od sposobu działania modułów niskopoziomowych.

Poniżej przedstawiam najistotniejsze listingi.

4.4.1. Listing części metod klasy Engine

Listing 4.1. Część metod klasy Engine

```
1 public void run() {  
2     do {  
3         iterate();  
4     } while (roundState.turnState != RoundState.TurnState.ended);  
5     logResult();  
6 }  
7
```

```

8  public void iterate() {
9      setCurrentAndOpponentSpace();
10     roundState.drawCardForPlayerSpace(currentPlayer.playerSpace);
11
12     DecisionType decision;
13     decision = getDecisionFromCurrentPlayer();
14     executeDecision(decision);
15
16
17     statistics.winningMove = decision;
18     if( roundState.deck.size() == 0 && roundState.turnState !=
        RoundState.TurnState.ended) {
19         endGame(null);
20         return;
21     }
22     roundState.switchState();
23     iteration++;
24 }
25
26 public DecisionType getDecisionFromCurrentPlayer() {
27     return currentPlayer.makeDecision(new RoundState(roundState,
        currentPlayer.playerSpace), DecisionType.getDecisions(
        currentPlayer.playerSpace.hand));
28 }

```

4.4.2. Listing algorytmu MCTS

Listing 4.2. Klasa Node

```

1  class Node{
2      DecisionType decision;
3      RoundState roundState;
4      double wins;
5      double value;
6      double visits;
7      double utc;
8      Node parent;
9      List<Node> children;
10
11     Node(RoundState roundState, DecisionType decision){

```



```
12         this.decision = decision;
13         this.roundState = roundState;
14         wins = 0;
15         value = 0.0;
16         visits = 0;
17         parent = null;
18         children = new ArrayList<>();
19     }
20
21     public void addChild(Node child){
22         children.add(child);
23         child.parent=this;
24     }
25
26     public void update(double result){
27         visits++;
28         wins +=result;
29         value = wins/visits;
30         if( visits == 0 || parent == null )
31             utc = -1;
32         else
33             utc = value/visits + explorationParameter * Math.sqrt(
34                 Math.log( parent.visits ) / visits);
35     }
36
37     public DecisionType getMostProvenDecision(){
38         Node mostProvenChild = children.get(0);
39         for( Node child : children ){
40             if( child.visits > mostProvenChild.visits)
41                 mostProvenChild = child;
42         }
43
44         return mostProvenChild.decision;
45     }
46
47     private double getUCT() {
48         if( visits == 0 )
49             return 1000 + rnd.nextInt(20);
```

```

49         return value/visits + explorationParameter * Math.sqrt( Math.
           log( parent.visits ) / visits);
50     }
51 }

```

Listing 4.3. Wybrane metody klasy MctsAlgorithm

```

1  @Override
2  public DecisionType makeDecision(RoundState roundState, List<
    DecisionType> decisionList) {
3      Node root = new Node( new RoundState(roundState, playerSpace),
        null );
4
5      long startTime = System.currentTimeMillis();
6      long elapsedTime;
7      int iteration = 0;
8      boolean shouldContinue = true;
9      do{
10         Node selectedNode = select(root);
11         if( selectedNode.roundState.turnState != RoundState.TurnState
            .ended ) {
12             List<DecisionType> selectionDecisionList = DecisionType.
                getDecisions(selectedNode.roundState.
                    spaceOfFirstPlayer.hand);
13             for (DecisionType decision : selectionDecisionList) {
14                 selectedNode.addChild(new Node(new RoundState(
                    roundState, roundState.spaceOfFirstPlayer),
                    decision));
15             }
16             Node child = selectedNode.children.get(rnd.nextInt(
                selectedNode.children.size()));
17             double result = simulate(child);
18             do {
19                 child.update(result);
20                 child = child.parent;
21             } while (child != null);
22         }
23         elapsedTime = System.currentTimeMillis() - startTime;
24         iteration++;

```

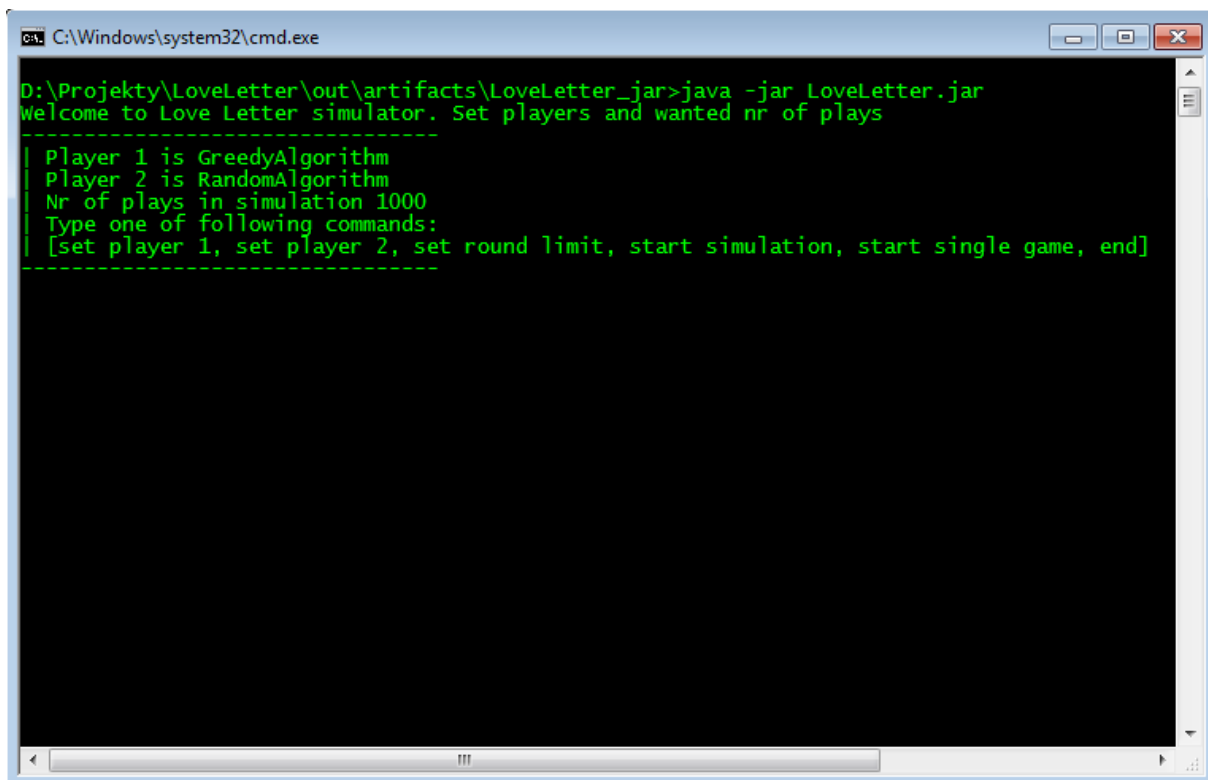
```
25     }while (timePredicate(elapsedTime));
26
27     DecisionType mostProvenDecision = root.getMostProvenDecision();
28     return mostProvenDecision;
29 }
30
31 private Node select(Node node) {
32     if( node.children.size() == 0 ){
33         return node;
34     }
35
36     Node selection = node.children.get(0);
37     Map<DecisionType, Double> utcMap = new HashMap<>();
38     for( Node child : node.children ){
39         if( child.getUCT() > selection.getUCT() || selection.getUCT()
40             == 0 )
41             selection = child;
42         utcMap.put(child.decision, child.getUCT());
43     }
44     return select(selection);
45 }
```

4.5. Prezentacja systemu

Poniższe zrzuty ekranu prezentują trzy widoki aplikacji: widok główny (początkowy), przebieg pojedynczej gry oraz podczas prezentacji wyników symulacji.

Widok główny

Widok główny pojawia się od razu po uruchomieniu aplikacji. Jest to konsola, w której pojawiają się dostępne informacje.



```
C:\Windows\system32\cmd.exe
D:\Projekty\LoveLetter\out\artifacts\LoveLetter_jar>java -jar LoveLetter.jar
Welcome to Love Letter simulator. Set players and wanted nr of plays
-----
| Player 1 is GreedyAlgorithm
| Player 2 is RandomAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
| -----
```

Rysunek 4.5. Widok główny aplikacji

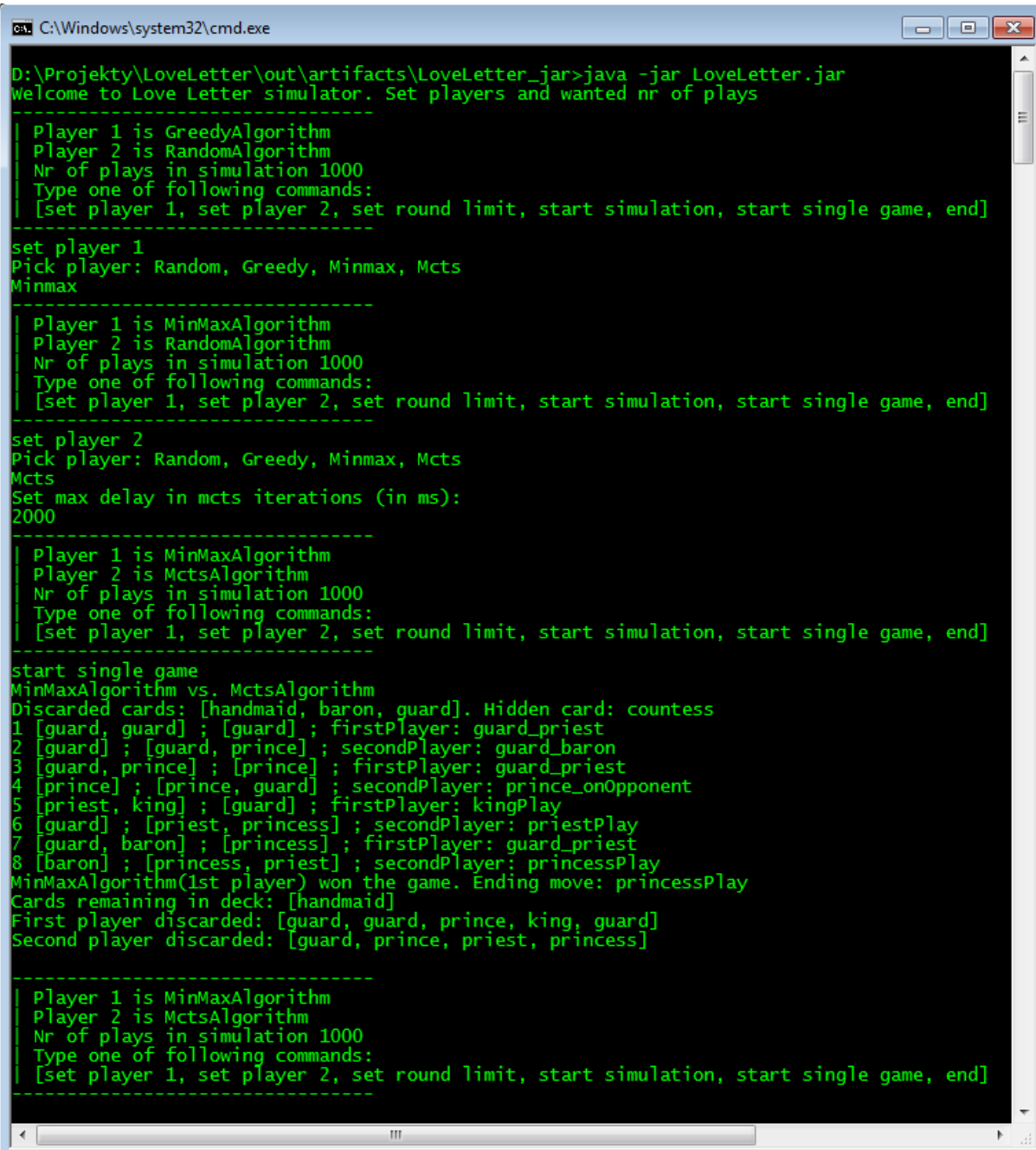
Rys. 4.5 prezentuje menu aplikacji, w którym można wpisywać następujące komendy:

- *set player 1* - ustawia pierwszego gracza,
- *set player 2* - ustawia drugiego gracza,
- *set round limit* - ustawia ilość rund w eksperymencie symulacyjnym,
- *start simulation* - rozpoczyna eksperyment symulacyjny,
- *start single game* - rozpoczyna pojedynczą grę,
- *switch players* - zamienia miejscami gracza pierwszego i drugiego,

- *start mirror simulation* - rozpoczyna eksperyment symulacyjny, w którym gracze rozpoczynają rundy na zmianę.

Widok ustawień i gry

Każdorazowo, po wpisaniu komendy, w konsoli wyświetlają się dodatkowe informacje, szczególnie po rozpoczęciu eksperymentu symulacyjnego.



```
C:\Windows\system32\cmd.exe

D:\Projekty\LoveLetter\out\artifacts\LoveLetter_jar>java -jar LoveLetter.jar
Welcome to Love Letter simulator. Set players and wanted nr of plays
-----
| Player 1 is GreedyAlgorithm
| Player 2 is RandomAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
set player 1
Pick player: Random, Greedy, Minmax, Mcts
Minmax
-----
| Player 1 is MinMaxAlgorithm
| Player 2 is RandomAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
set player 2
Pick player: Random, Greedy, Minmax, Mcts
Mcts
Set max delay in mcts iterations (in ms):
2000
-----
| Player 1 is MinMaxAlgorithm
| Player 2 is MctsAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
start single game
MinMaxAlgorithm vs. MctsAlgorithm
Discarded cards: [handmaid, baron, guard]. Hidden card: countess
1 [guard, guard] ; [guard] ; firstPlayer: guard_priest
2 [guard] ; [guard, prince] ; secondPlayer: guard_baron
3 [guard, prince] ; [prince] ; firstPlayer: guard_priest
4 [prince] ; [prince, guard] ; secondPlayer: prince_onOpponent
5 [priest, king] ; [guard] ; firstPlayer: kingPlay
6 [guard] ; [priest, princess] ; secondPlayer: priestPlay
7 [guard, baron] ; [princess] ; firstPlayer: guard_priest
8 [baron] ; [princess, priest] ; secondPlayer: princessPlay
MinMaxAlgorithm(1st player) won the game. Ending move: princessPlay
Cards remaining in deck: [handmaid]
First player discarded: [guard, guard, prince, king, guard]
Second player discarded: [guard, prince, priest, princess]
-----
| Player 1 is MinMaxAlgorithm
| Player 2 is MctsAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
```

Rysunek 4.6. Widok ustawień i gry

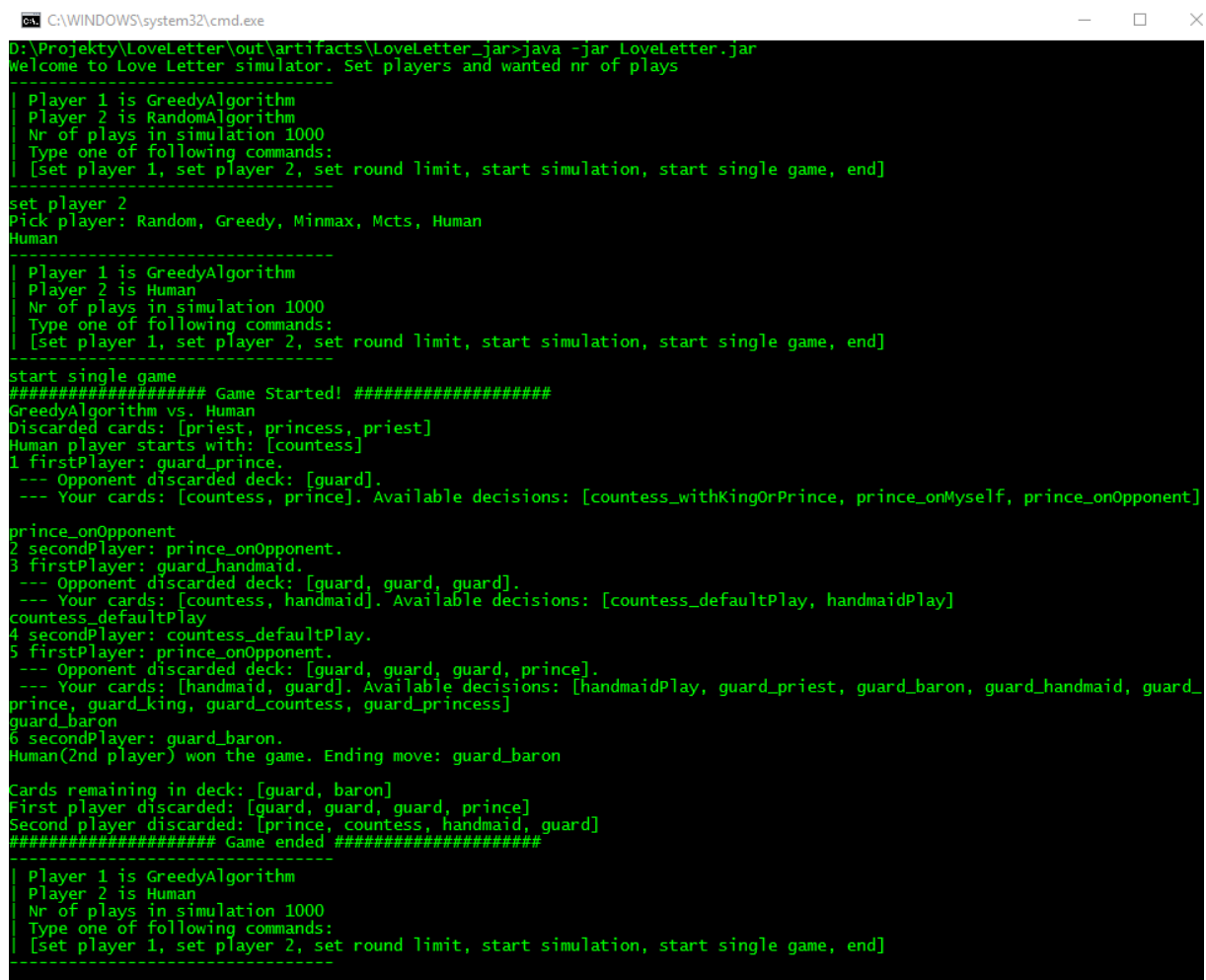
Na rys. 4.6 można zauważyć ustawianie algorytmu dla gracza pierwszego i drugiego oraz przebieg pojedynczej gry. Na początku znajduje się informacja które algorytmy ze sobą grają, następnie jakie karty zostały odrzucone w układzie początkowym, a potem przebieg w następującej postaci:

[numer rundy] [karty pierwszego gracza] ; [karty drugiego gracza] ; [czyj ruch]: [podjęta decyzja]

Na końcu wskazany jest zwycięski algorytm, ruch kończący, oraz stosy kart odrzuconych.

Widok gry człowieka z zaimplementowanym algorytmem

Jeśli za pomocą komend jako jednego z graczy ustawimy człowieka, a jako drugi algorytm, informacje o przebiegu gry będą się ograniczały wyłącznie do tych znanych człowiekowi.



```

C:\WINDOWS\system32\cmd.exe
D:\Projekty\LoveLetter\out\artifacts\LoveLetter_jar>java -jar LoveLetter.jar
Welcome to Love Letter simulator. Set players and wanted nr of plays
-----
| Player 1 is GreedyAlgorithm
| Player 2 is RandomAlgorithm
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
set player 2
Pick player: Random, Greedy, Minmax, Mcts, Human
Human
-----
| Player 1 is GreedyAlgorithm
| Player 2 is Human
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----
start single game
##### Game Started! #####
GreedyAlgorithm vs. Human
Discarded cards: [priest, princess, priest]
Human player starts with: [countess]
1 firstPlayer: guard_prince.
--- Opponent discarded deck: [guard].
--- Your cards: [countess, prince]. Available decisions: [countess_withKingOrPrince, prince_onMyself, prince_onOpponent]
prince_onOpponent
2 secondPlayer: prince_onOpponent.
3 firstPlayer: guard_handmaid.
--- Opponent discarded deck: [guard, guard, guard].
--- Your cards: [countess, handmaid]. Available decisions: [countess_defaultPlay, handmaidPlay]
countess_defaultPlay
4 secondPlayer: countess_defaultPlay.
5 firstPlayer: prince_onOpponent.
--- Opponent discarded deck: [guard, guard, guard, prince].
--- Your cards: [handmaid, guard]. Available decisions: [handmaidPlay, guard_priest, guard_baron, guard_handmaid, guard_prince, guard_king, guard_countess, guard_princess]
guard_baron
6 secondPlayer: guard_baron.
Human(2nd player) won the game. Ending move: guard_baron

Cards remaining in deck: [guard, baron]
First player discarded: [guard, guard, guard, prince]
Second player discarded: [prince, countess, handmaid, guard]
##### Game ended #####
-----
| Player 1 is GreedyAlgorithm
| Player 2 is Human
| Nr of plays in simulation 1000
| Type one of following commands:
| [set player 1, set player 2, set round limit, start simulation, start single game, end]
-----

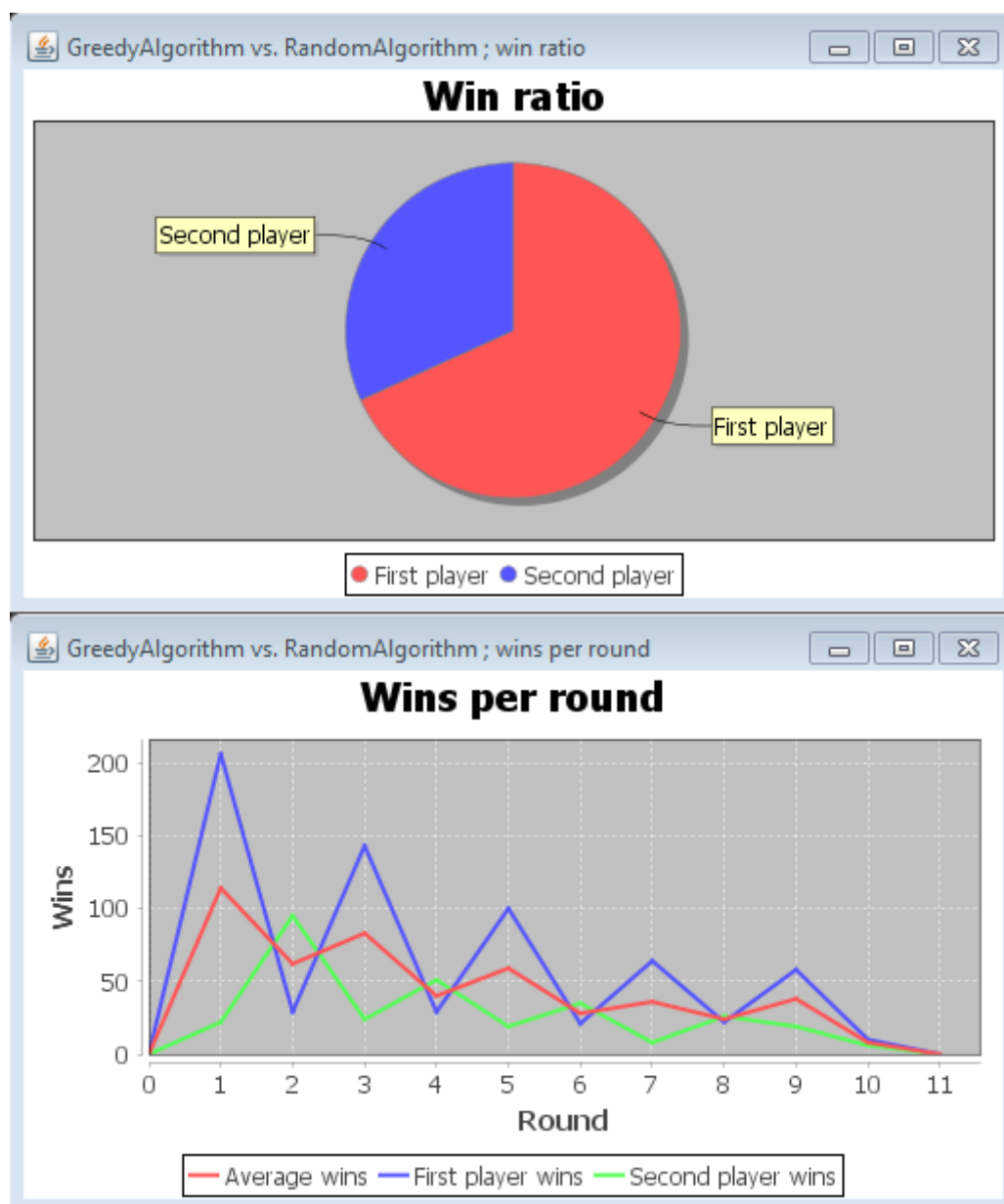
```

Rysunek 4.7. Widok gry człowieka z przeciwnikiem

Na rys. 4.7 przedstawiony został przebieg gry człowieka z przeciwnikiem komputerowym (algorytmem). Na początku należy ustawić jedno z graczy jako *Human*, korzystając z komend *set player 1* lub *set player 2*. Następnie należy włączyć pojedynczą grę komendą *start single game*.

Wykresy eksperymentu symulacyjnego

Na wykresach przedstawione są informacje o przebiegu eksperymentu symulacyjnego, między innymi średnia ilość zwycięstw obu algorytmów w danej turze, co pozwala zauważyć pewne trendy.



Rysunek 4.8. Wykresy zwycięstw

Na powyższym wykresie (Rys. 4.8) wskazany jest stosunek zwycięstw na wykresie kołowym, oraz ilość zwycięstw każdego z algorytmów w danej rundzie, z uwzględnieniem średniej.



Rysunek 4.9. Wykresy zwycięskich decyzji

Powyższe wykresy słupkowe na rys. 4.9 prezentują rozkład wygrywających decyzji danego gracza w zależności od rundy.

4.6. Problemy napotkane w trakcie realizacji

Jednym z problemów w kodzie aplikacji jest bardzo duża ilość instrukcji *switch*. Wynika to z błędnie przyjętego modelu odwzorowania dostępnych decyzji w postaci klasy *Enum*. Zdecydowanie lepszym rozwiązaniem byłoby utworzenie hierarchii klas odpowiadających każdej decyzji, które dziedziczyłyby po jednej wspólnej klasie, np. *Play*. Takie podejście znacznie zredukowałoby ilość niepotrzebnego kodu i zwiększyło jego czytelność.

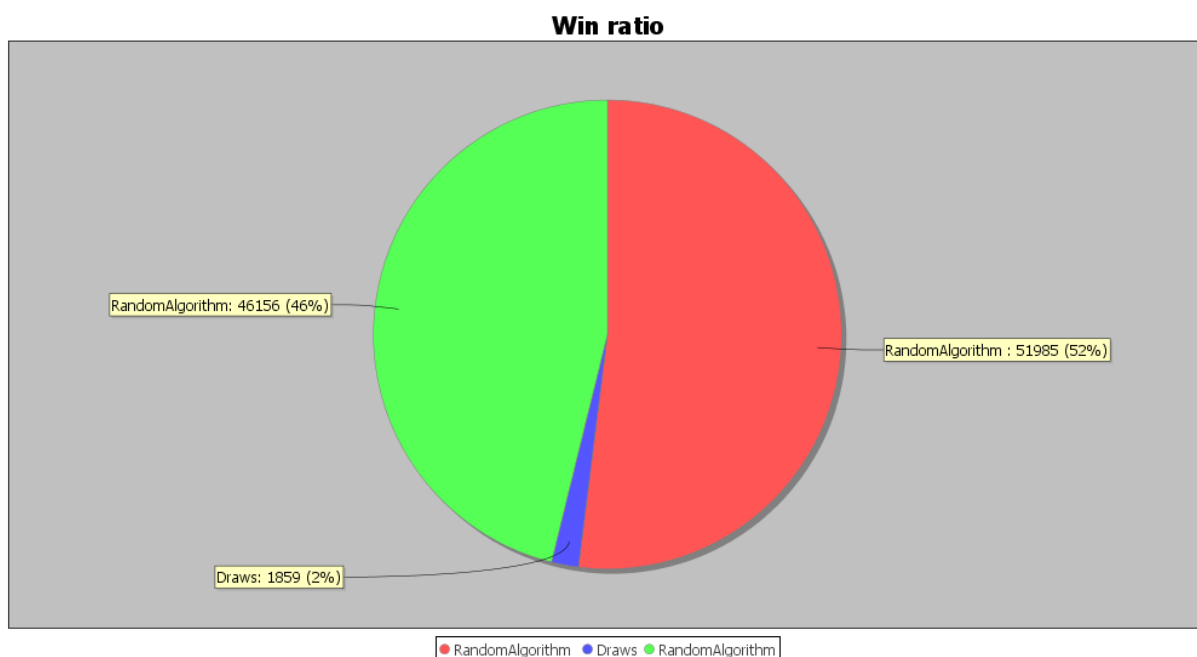
Kolejną rzeczą sprawiającą wiele problemów jest kwestia interfejsu *Copy* w Javie. Podczas testów przesyłania kopii stanu rundy do algorytmu podejmującego decyzje powstawało wiele problemów z kopiowaniem referencji, wobec czego postanowiłem ręcznie napisać metodę kopiującą.

Ze względu na dużą ilość losowanych parametrów, trudność sprawiło napisanie odpowiednich testów pokrywających wszystkie przypadki. W związku z tym testy sprawdzające fragmenty związane z losowaniem uruchamiane są wielokrotnie. Co prawda takie podejście sprawia, że nigdy nie mamy 100% pewności przejścia wszystkich testów bez zmiany kodu, natomiast pozwala wykryć znacznie większą ilość przypadków brzegowych, które normalnie nie były by pokryte testem.

5. Rezultaty

W tym rozdziale zamieszczam rezultaty eksperymentów symulacyjnych z użyciem opisanych wcześniej algorytmów. Dla każdego z nich przedstawiam średni procent zwycięstw oraz zagrania, które najczęściej kończyły grę. Początkowo prezentuję statystyki z eksperymentów w których każdy z algorytmów gra z algorytmem losowym, będącym zawsze jest drugim graczem. Następne prezentuję wyniki z eksperymentów, w których biorą udział algorytm zachłanny, algorytm minimaksowy i algorytm MCTS. W tych przypadkach symulacje ustawione są tak, by każdy z algorytmów był naprzemiennie pierwszym i drugim graczem. Rozdział kończę opisaniem wniosków ogólnych oraz z działania poszczególnych algorytmów. W każdym porównaniu prezentuję statystyki oparte na próbie 100 tysięcy symulacji, z wyjątkiem tych, gdzie porównywany jest algorytm MCTS. Dla tych przypadków, ze względu na czas trwania jednej rozgrywki, ograniczyłem próbę do 1 tysiąca.

5.1. Algorytm losowy versus Algorytm Losowy



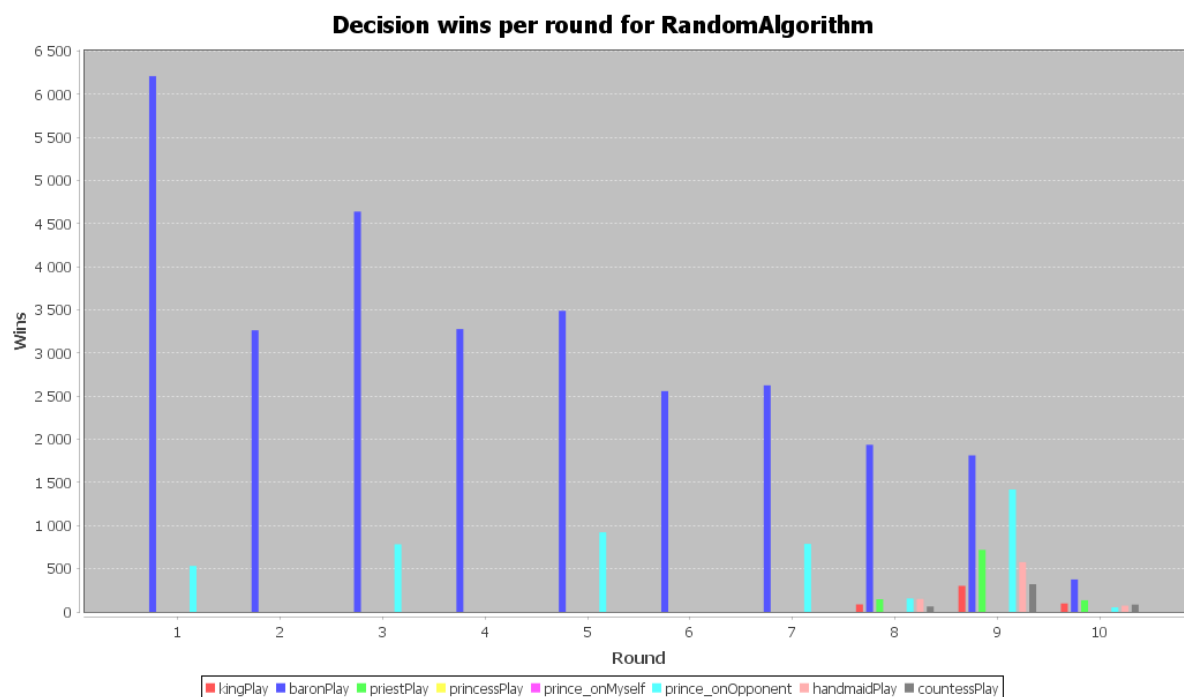
Rysunek 5.1. Wykres kołowy zwycięstw i remisów

Powyższy wykres na rys. 5.1 wskazuje na pewną przewagę gracza rozpoczynającego.

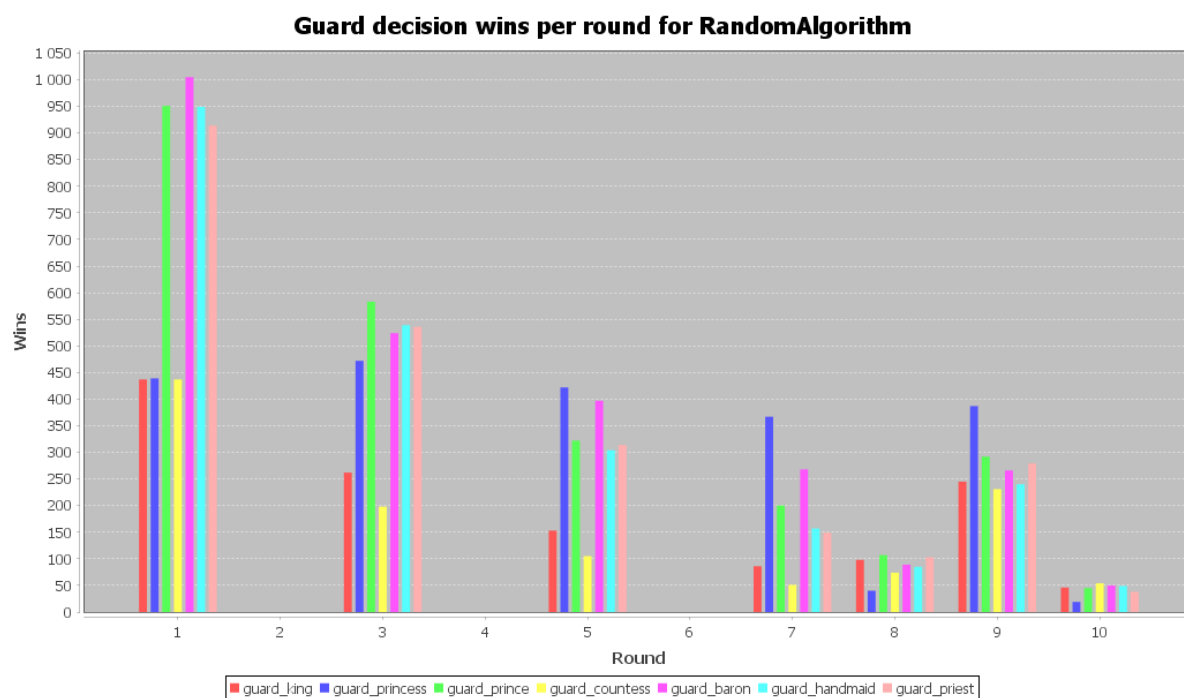


Rysunek 5.2. Wykres wygranych w danej rundzie

Na rys. 5.2 widać wyraźną tendencję do spadku zakończeń gry wraz z kolejnymi rundami, za wyjątkiem przedostatniej, gdzie następuje wzrost.



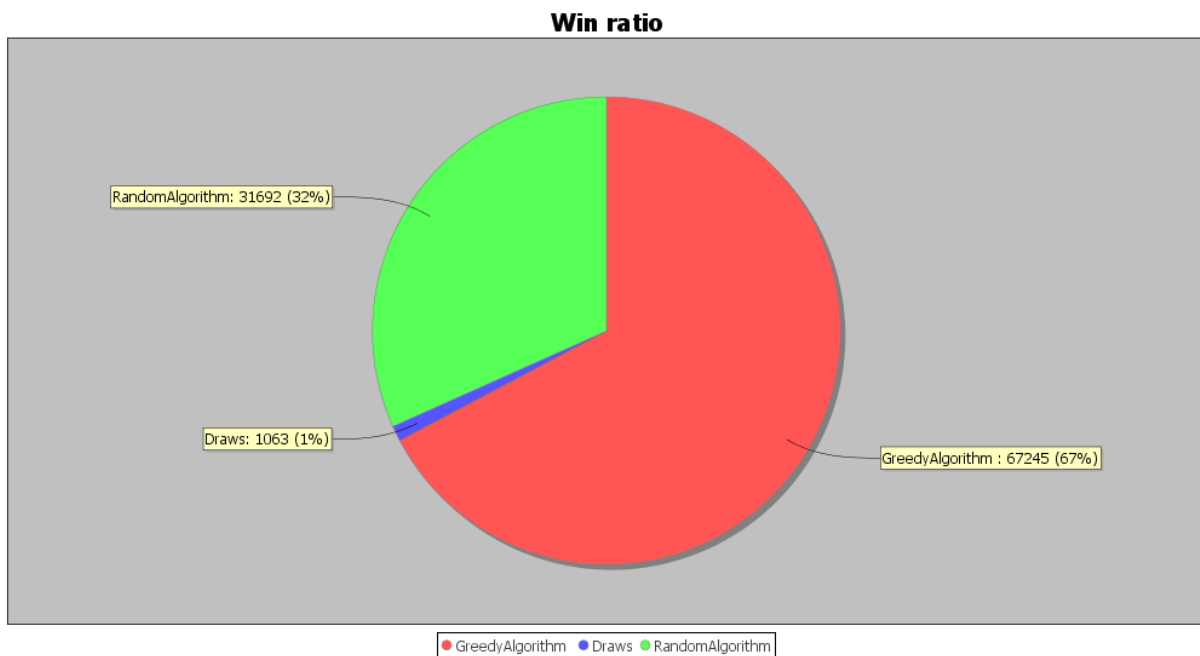
Rysunek 5.3. Wykres zwycięskich zagrań



Rysunek 5.4. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

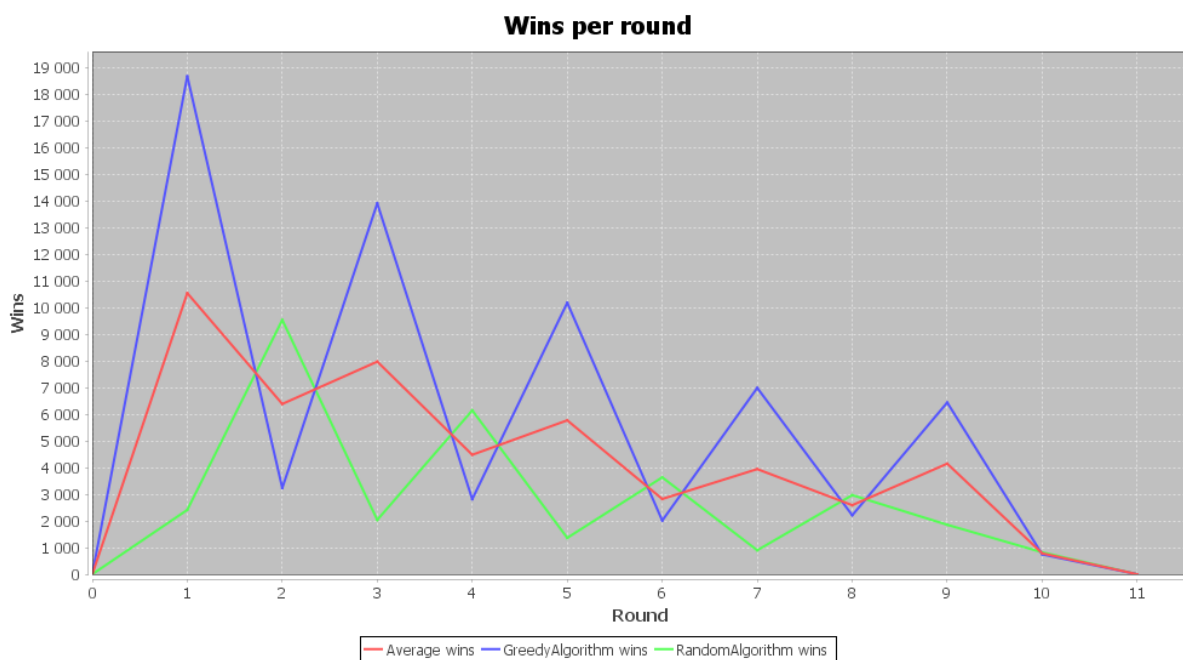
Z dwóch powyższych wykresów (rys. 5.1 i rys. 5.4) wynika, że zdecydowanie najczęstszym wygrywającym ruchem jest zagranie karty Baron. Około połowę mniej zwycięstw jest w wyniku zagrania karty Strażniczki. Warty uwagi jest fakt, że im bliżej końca gry tym częściej zwycięskim zagraniem jest zagranie karty Księcia na przeciwnika.

5.2. Algorytm zachłanny versus Algorytm losowy



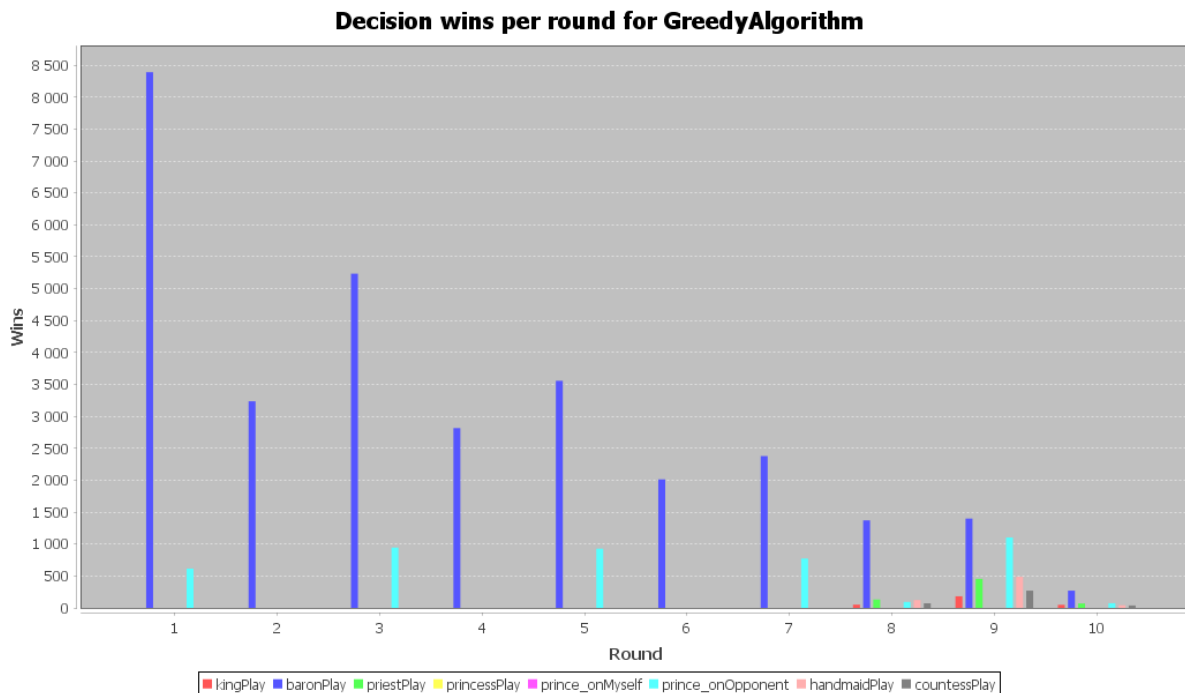
Rysunek 5.5. Wykres kołowy zwycięstw i remisów

Powyżej (rys. 5.5) widać zauważalną przewagę algorytmu zachłannego nad losowym.

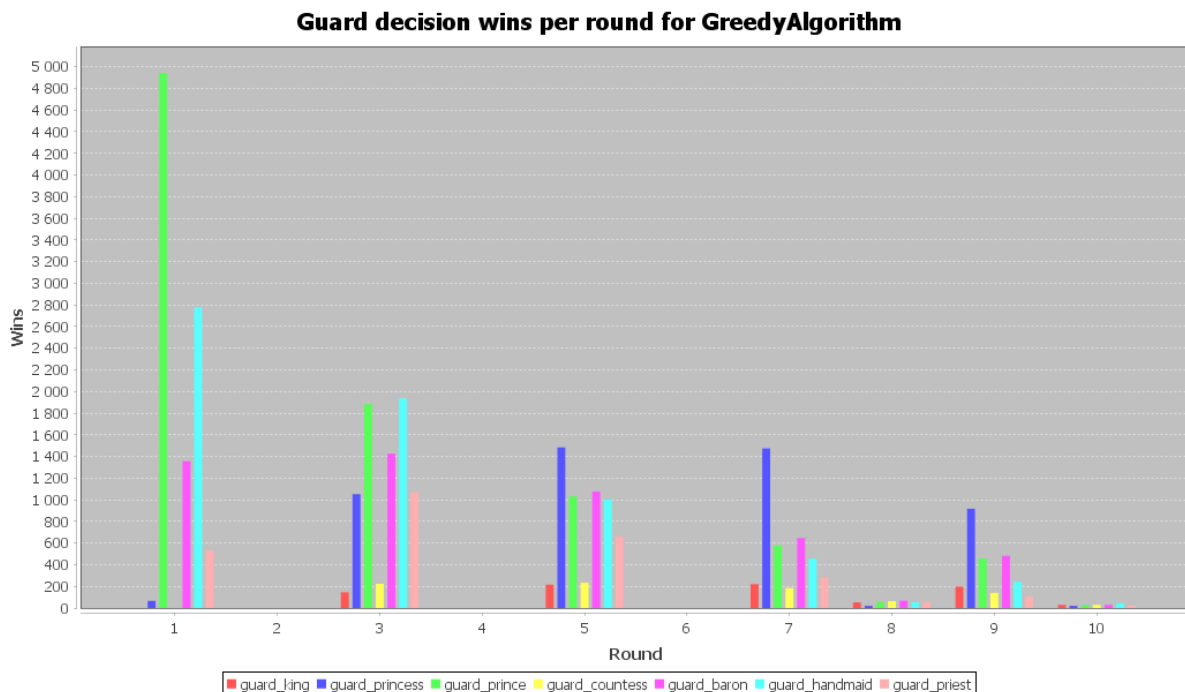


Rysunek 5.6. Wykres wygranych w danej rundzie

Na rys. 5.6 widać różnicę w porównaniu do eksperymentu poprzedniego, gdzie obaj gracze grają losowo. Nie ma tak wyraźnego wzrostu liczby gier skończonych w przedostatniej rundzie. Wynika to ze znacznie mniejszej ilości gier zakończonych remisem.



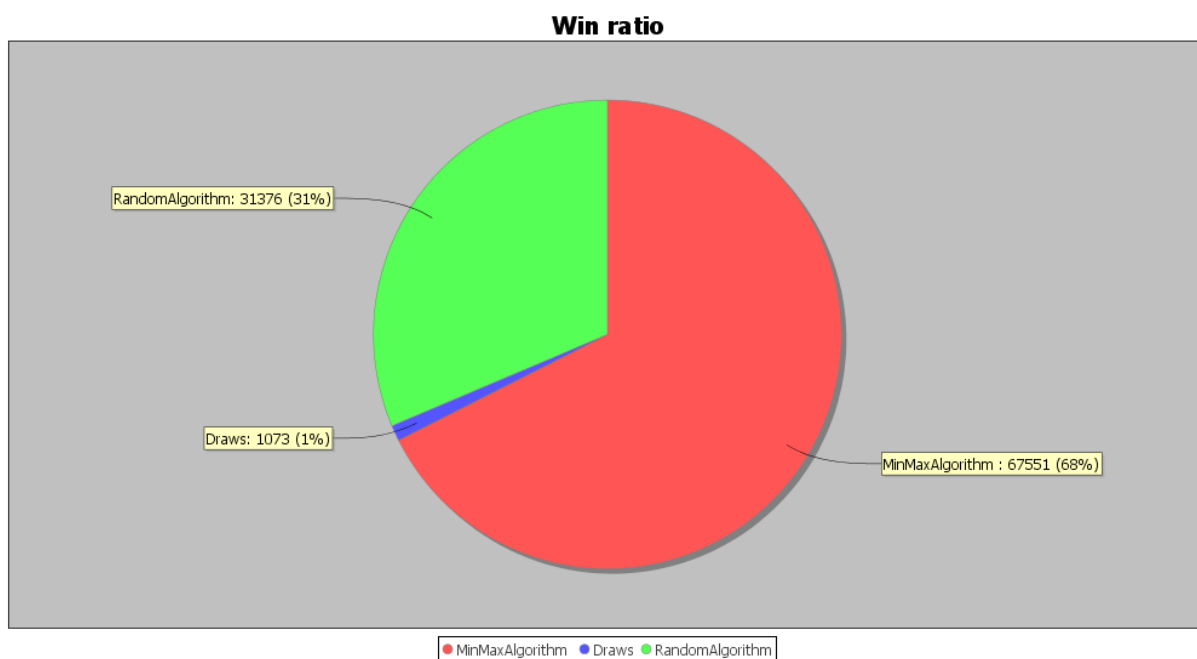
Rysunek 5.7. Wykres zwycięskich zagrań



Rysunek 5.8. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

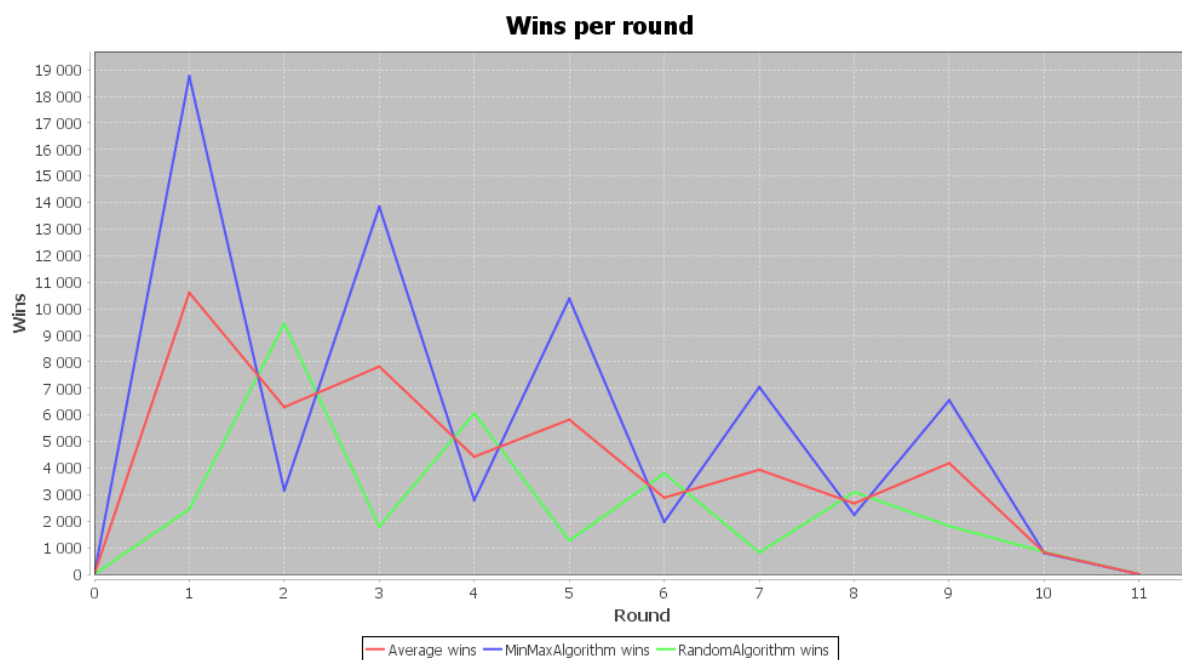
Porównanie wykresów na rysunkach 5.4 oraz 5.8 pokazują ogromny wzrost znaczenia zagrania karty Strażniczki z wyborem karty Księżę. Algorytm zachłanny osiąga zwycięstwo zagrywając Strażniczkę niemal tak samo często co przy zagraniu Barona.

5.3. Algorytm minimaxowy versus Algorytm losowy



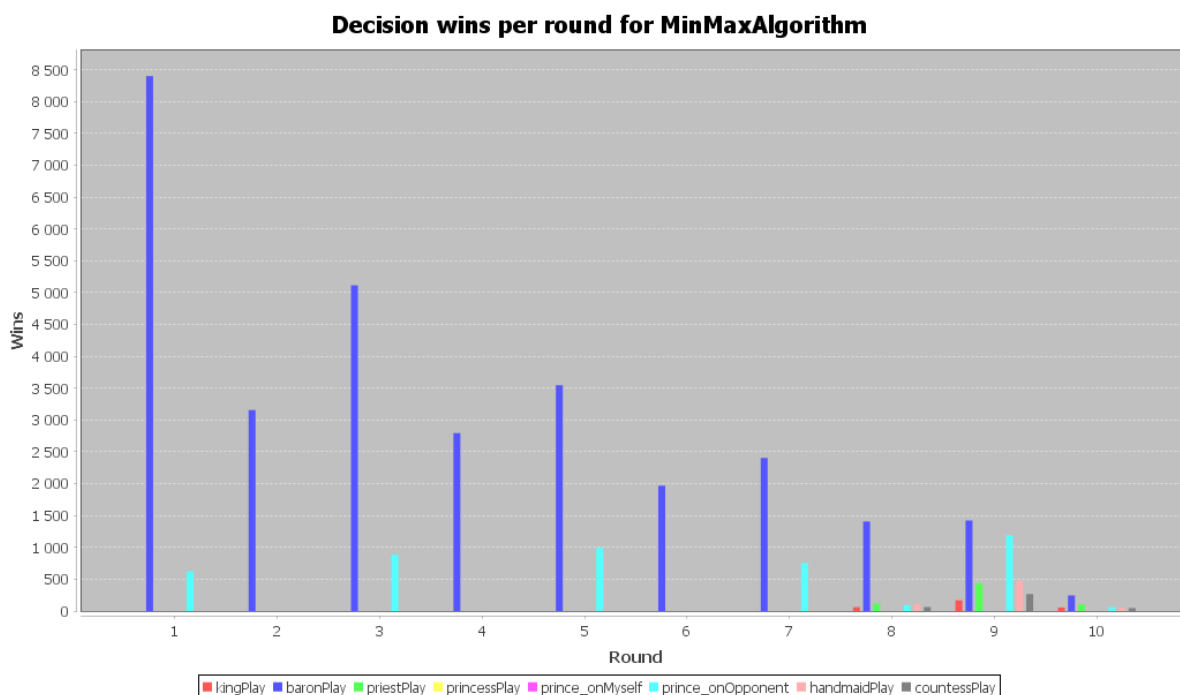
Rysunek 5.9. Wykres kołowy zwycięstw i remisów

Powyższy wykres (rys. 5.9) pokazuje bardzo podobne wyniki jak w przypadku algorytmu zachłanego.



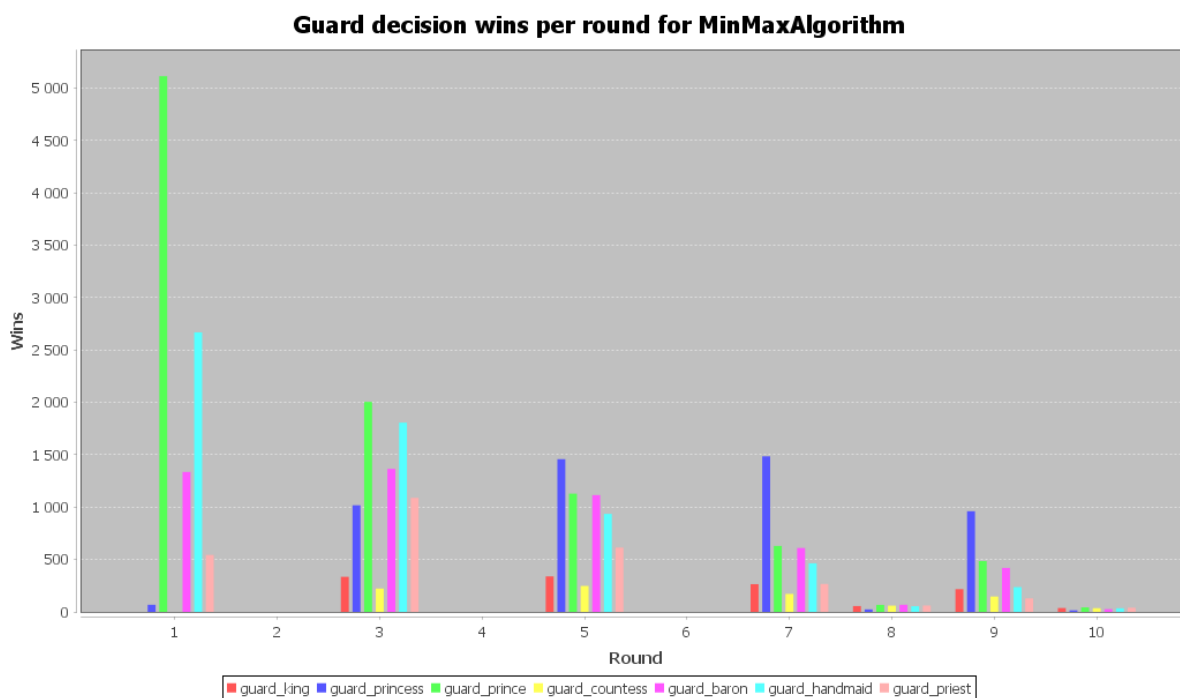
Rysunek 5.10. Wykres wygranych w danej rundzie

Wykres na rys. 5.10 pokazuje wyraźną tendencję do spadku średniej liczby wygranych gier wraz z kolejnymi turami, co występuje również u algorytmu zachłannego.



Rysunek 5.11. Wykres zwycięskich zagrań

Wykres na rysunku 5.11 również obrazuje zbliżone wyniki do algorytmu zachłannego - algorytm minimaxowy ma zdecydowaną przewagę nad algorytmem losowym.

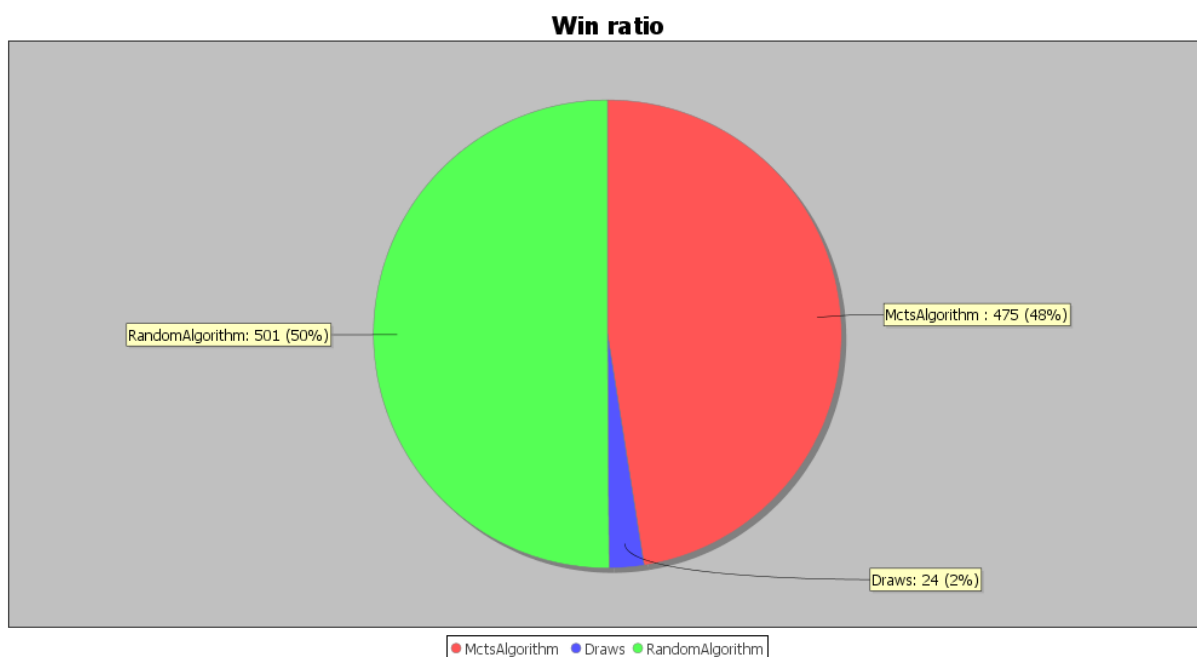


Rysunek 5.12. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

Na rys. 5.12 widać, że najczęściej zwycięstw daje zagranie karty Strażniczki z wyborem na kartę Księcia. Zdecydowanie mniej korzystny jest wybór karty Pokojówki lub Barona.

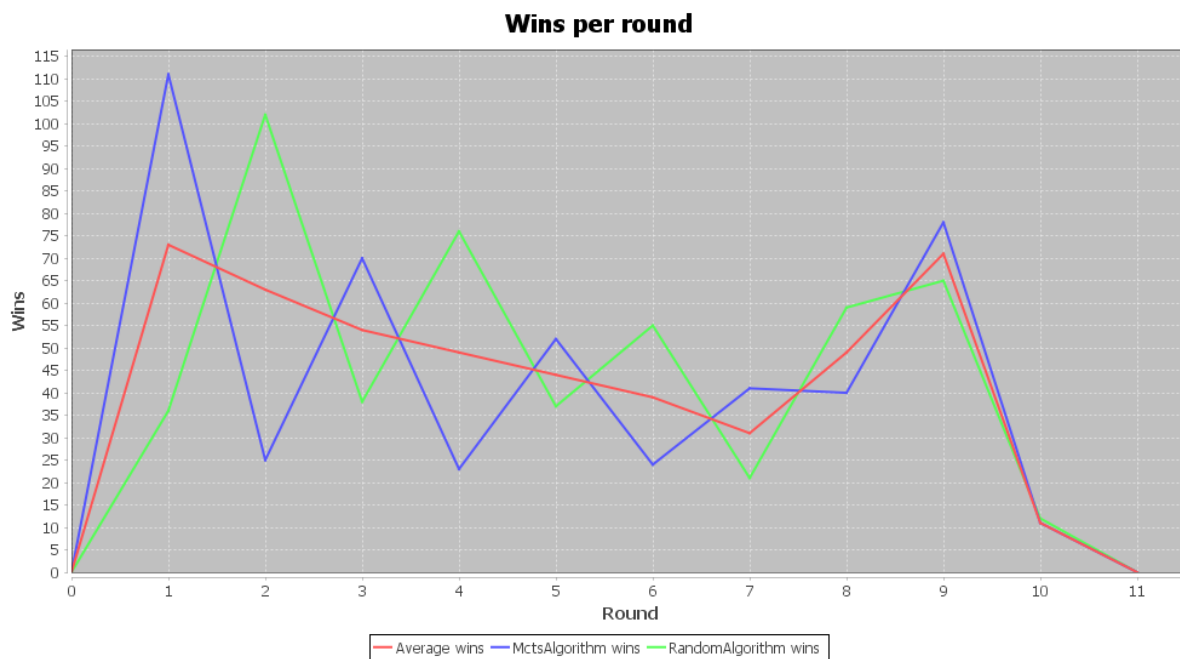
5.4. Algorytm MCTS versus Algorytm Losowy

Dane z eksperymentów symulacyjnych, w których bierze udział algorytm MCTS, opierają się na próbie 1000 gier, ponieważ większa próba znacząco zwiększa czas oczekiwania na wyniki. Wynika to z charakteru algorytmu, który działa przez zadany czas. W przeprowadzonych symulacjach czas działania algorytmu ustawiony był na 100ms.



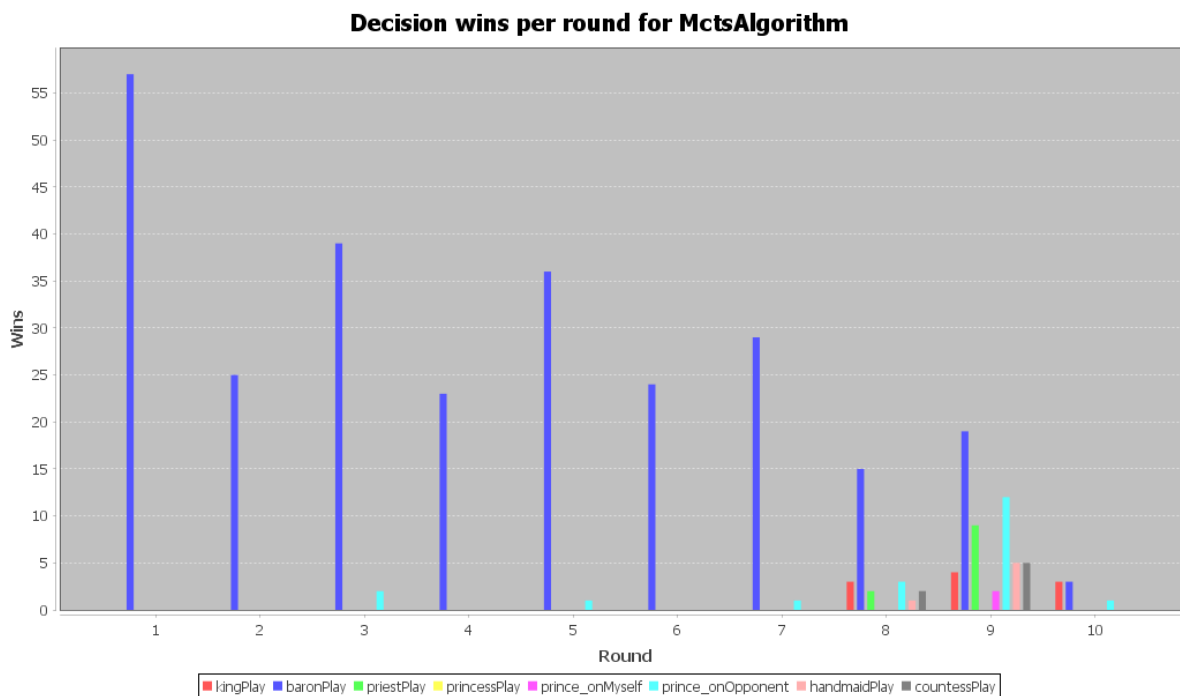
Rysunek 5.13. Wykres kołowy zwycięstw i remisów

Powyższy wykres (rys. 5.13) pokazuje w tym wypadku zauważalną przewagę algorytmu losowego nad algorytmem Monte Carlo Tree Search, mimo, że był drugim graczem. Spowodowane jest to między innymi błędnie przyjętym założeniem, że podstawowa forma algorytmu MCTS osiągnie dobre wyniki mimo czynnika losowego, o ile ilość symulacji na turę będzie odpowiednio duża.

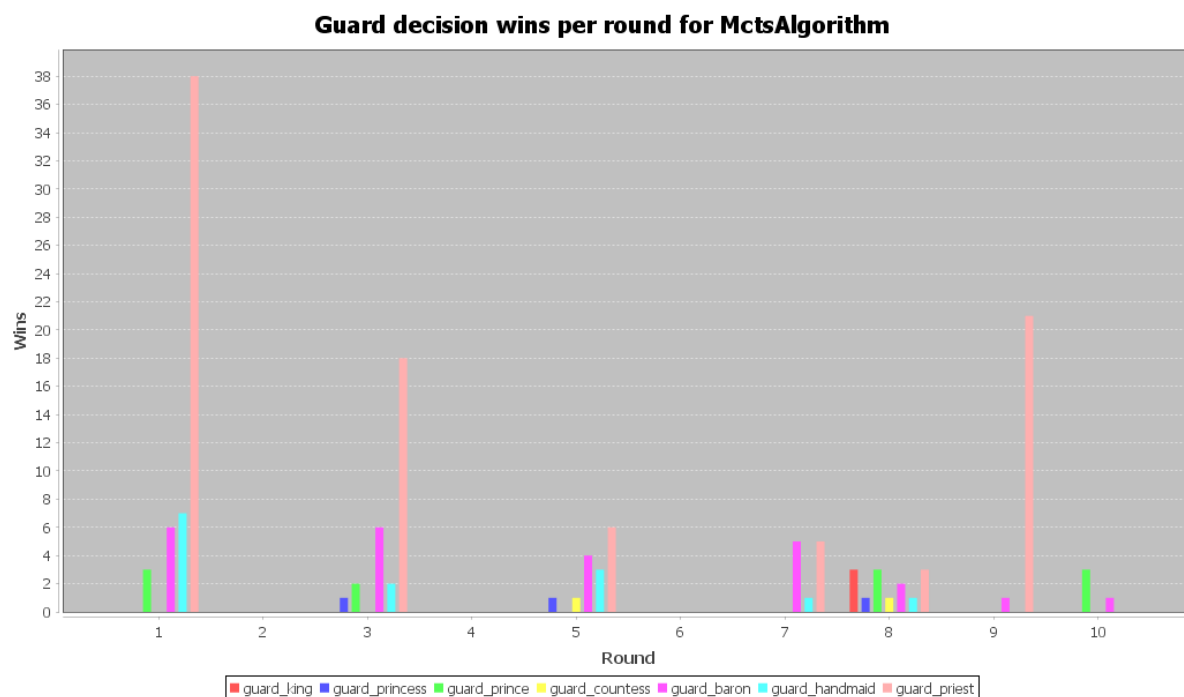


Rysunek 5.14. Wykres wygranych w danej rundzie

Wykres na rys. 5.14 pokazuje, że średnia liczba zwycięstw wzrasta w ostatnich turach, co jest charakterystyczne również dla eksperymentu z dwoma algorytmami losowymi.



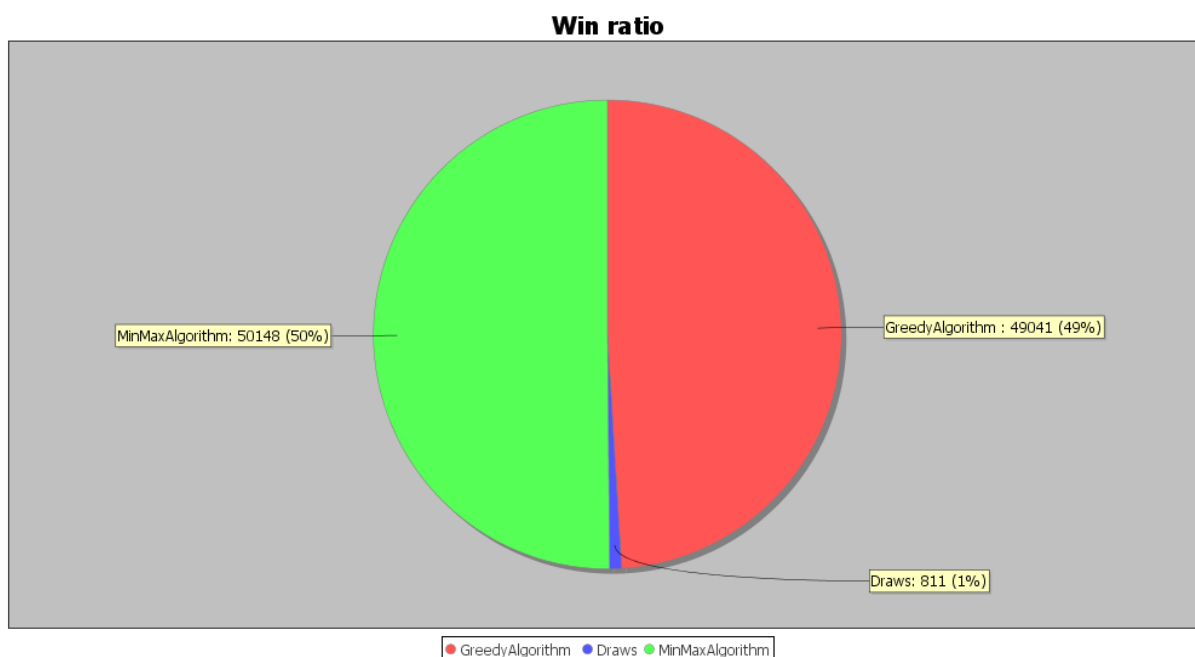
Rysunek 5.15. Wykres zwycięskich zagrań



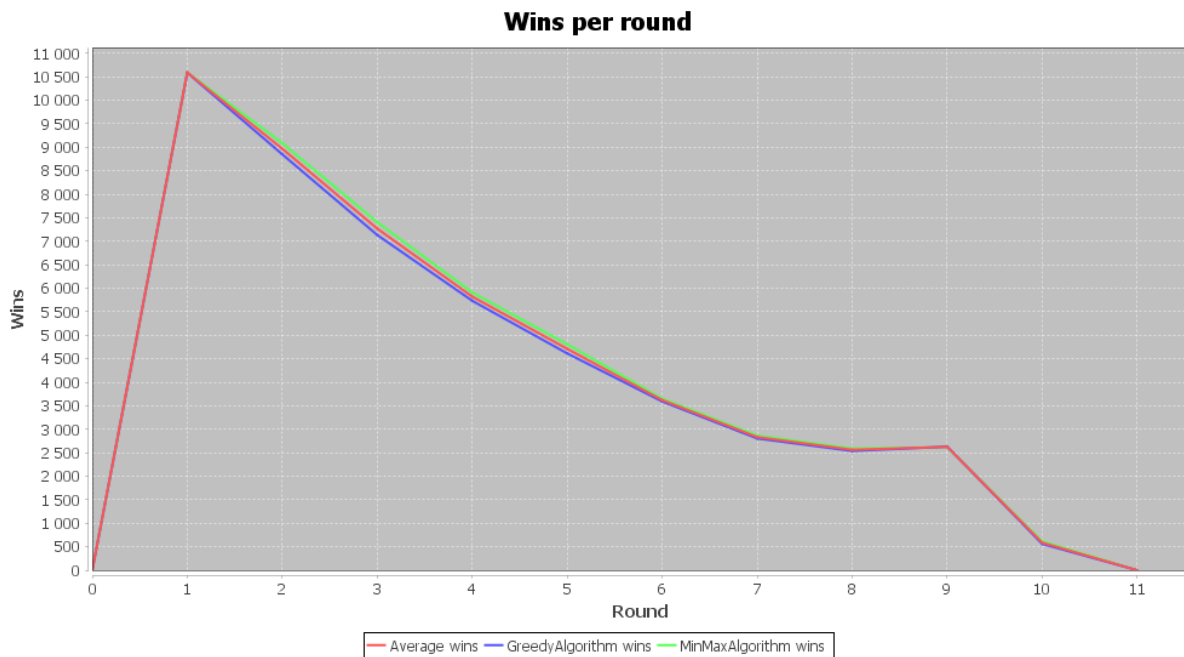
Rysunek 5.16. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

Wykresy na rysunkach 5.15 i 5.16 pokazują wyraźną tendencję algorytmu MCTS do zagrywania karty Strażniczki z wyborem na Kapłana.

5.5. Algorytm Minimaksowy versus Algorytm Zachłanny

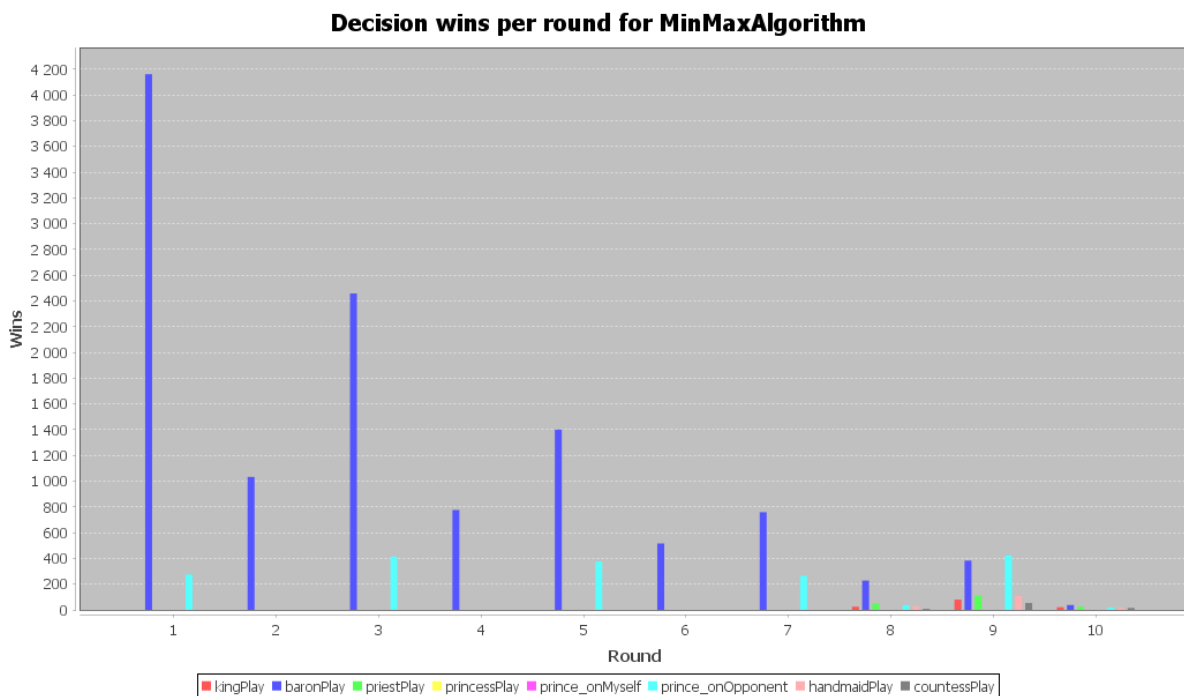


Rysunek 5.17. Wykres kołowy zwycięstw i remisów

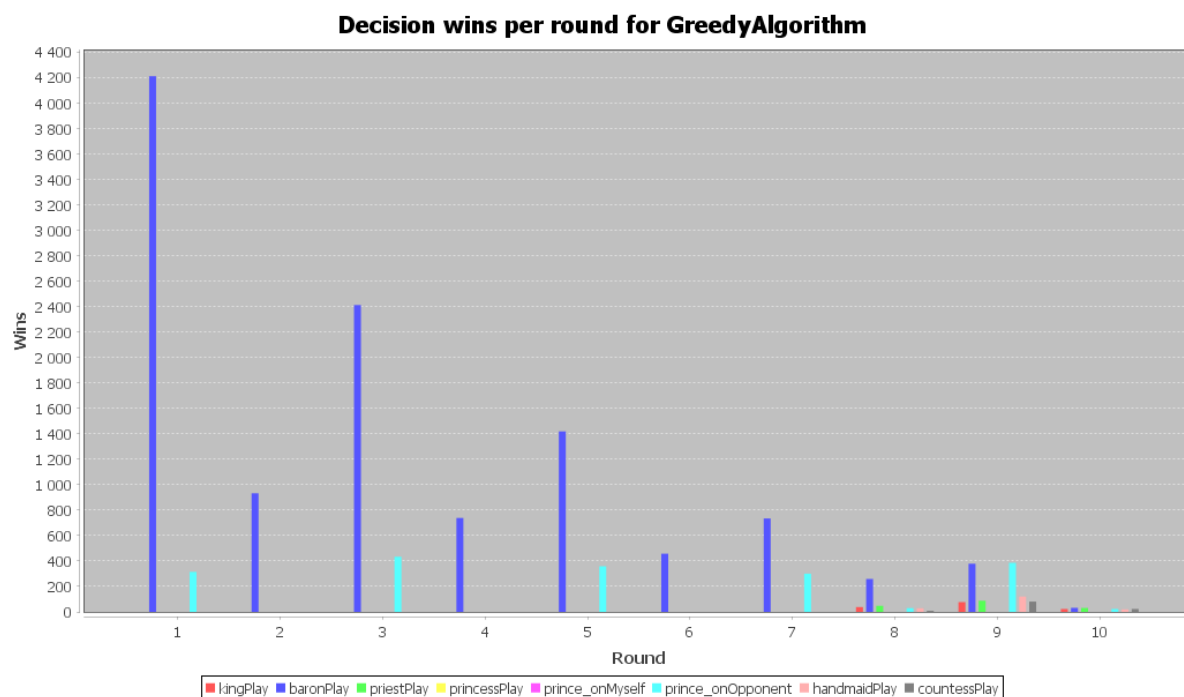


Rysunek 5.18. Wykres wygranych w danej rundzie

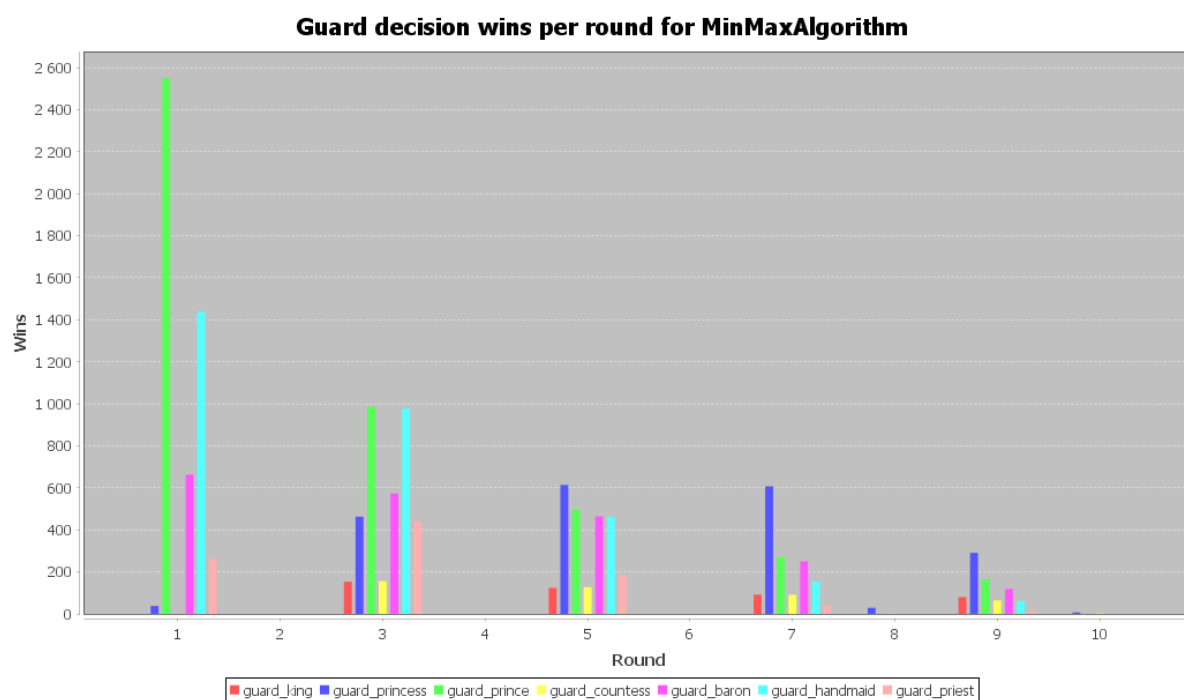
Powyższe wykresy (rys. 5.17 i 5.18) pokazują niewielką przewagę algorytmu minimaksowego nad zachłannym, który w rundach od 1 do 8 zachowuje niewielką przewagę. W rundach 9 i 10 przeważa algorytm zachłanny, z czego wynika, że częściej wygrywa gry kończące się porównaniem siły kart.



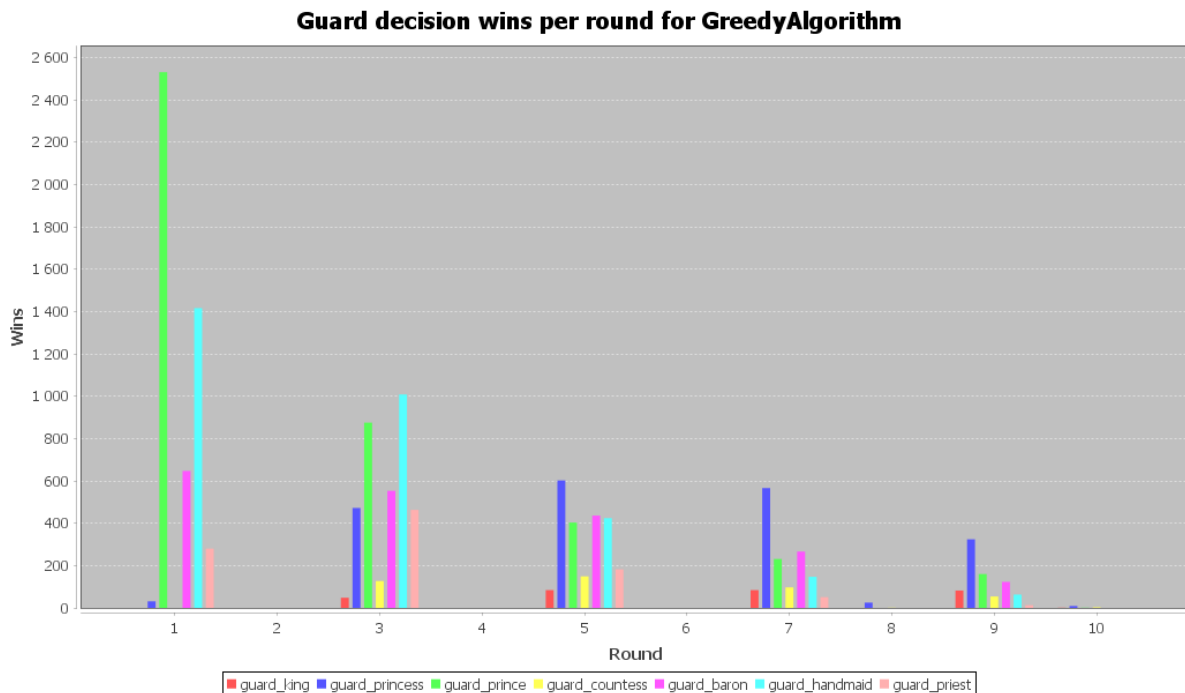
Rysunek 5.19. Wykres zwycięskich zagrań



Rysunek 5.20. Wykres zwycięskich zagrań



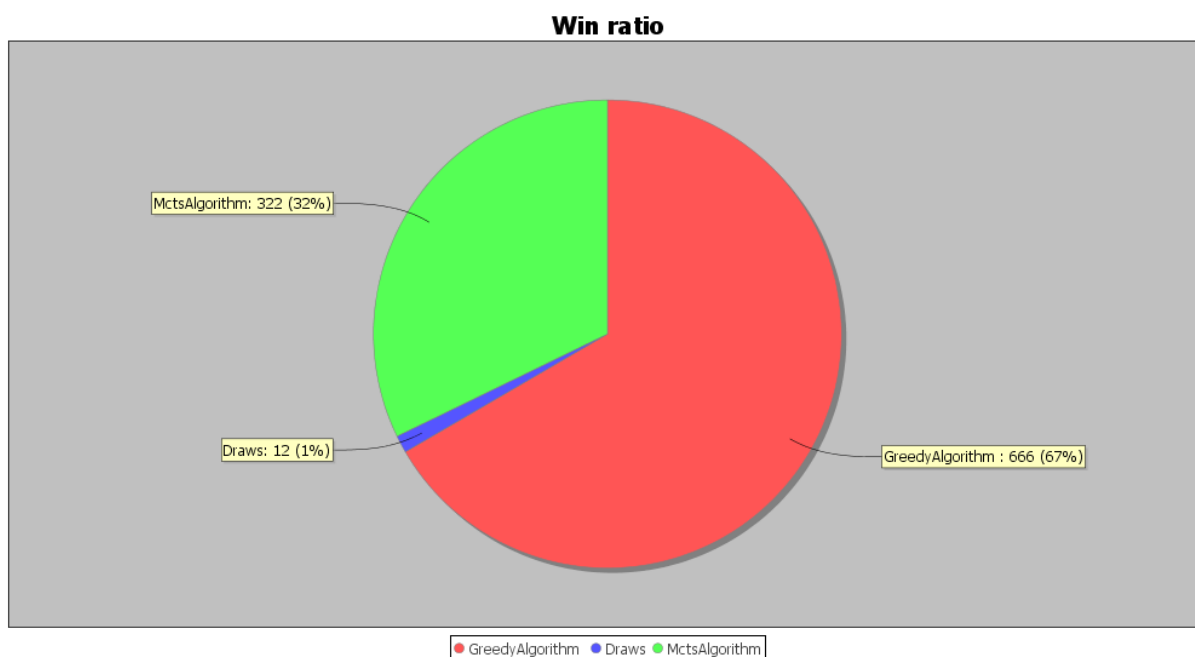
Rysunek 5.21. Wykres szczegółowy zwycięskich zagrań karty Strażniczki



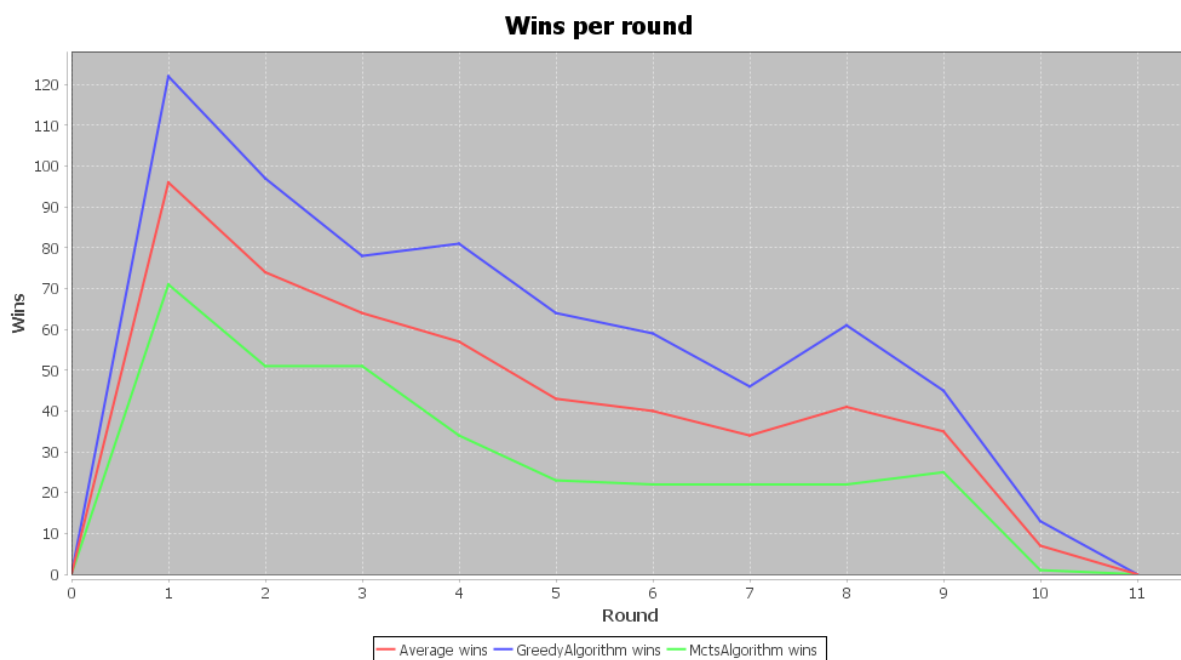
Rysunek 5.22. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

Z wykresów na rysunkach 5.19, 5.20, 5.21 i 5.22 wynika, że oba algorytmy wykazują duże podobieństwo - najczęściej wygrywają poprzez zagranie karty Barona, a zagrywając kartę Strażniczki najczęściej wybierają kartę Księcia.

5.6. Algorytm Zachłanny versus Algorytm MCTS

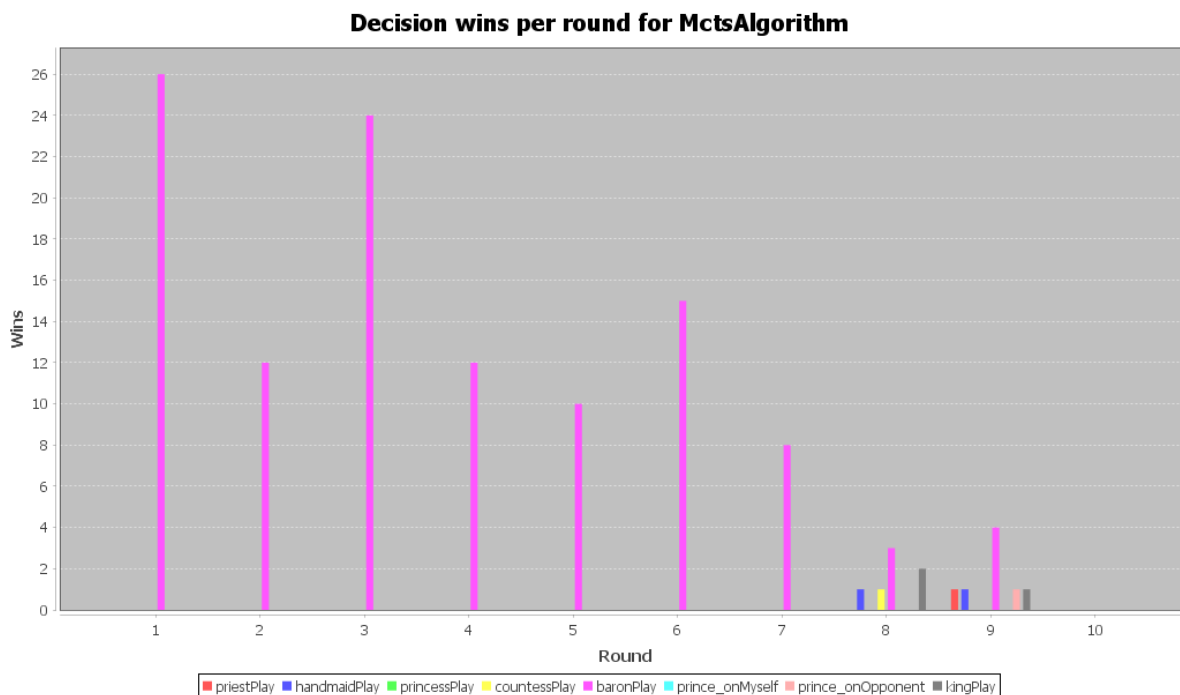


Rysunek 5.23. Wykres kołowy zwycięstw i remisów

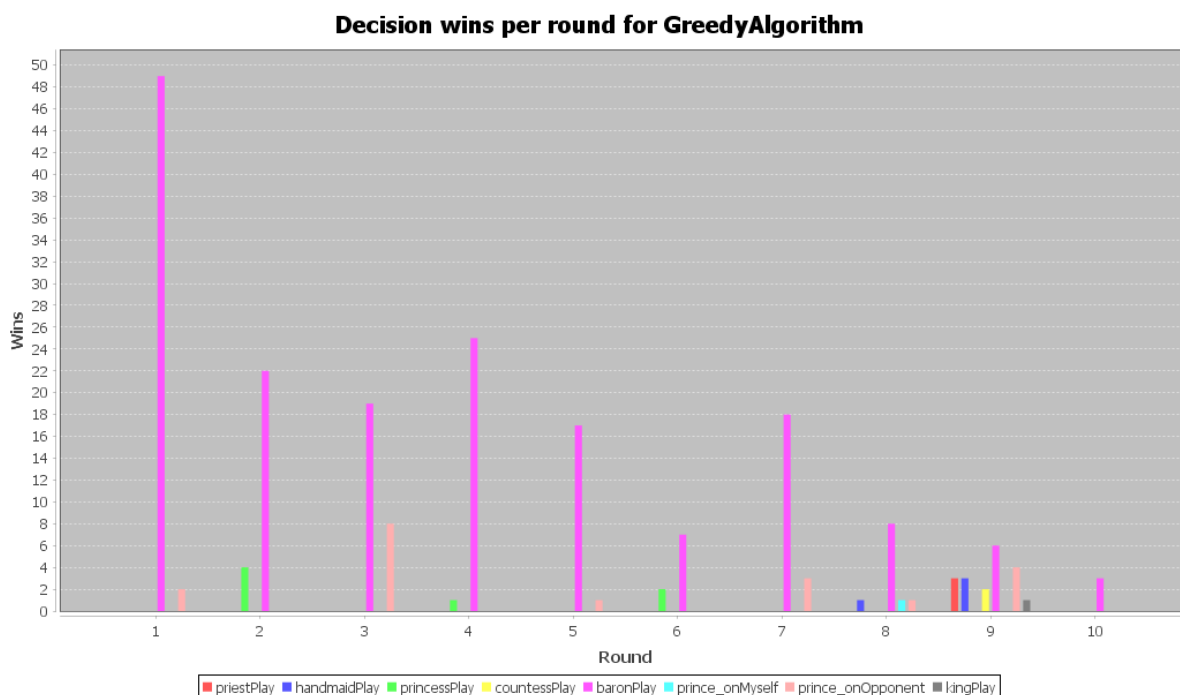


Rysunek 5.24. Wykres wygranych w danej rundzie

Powyższe wykresy (rys. 5.23 i 5.24) pokazują bardzo wyraźną przewagę algorytmu zachłannego nad MCTS.

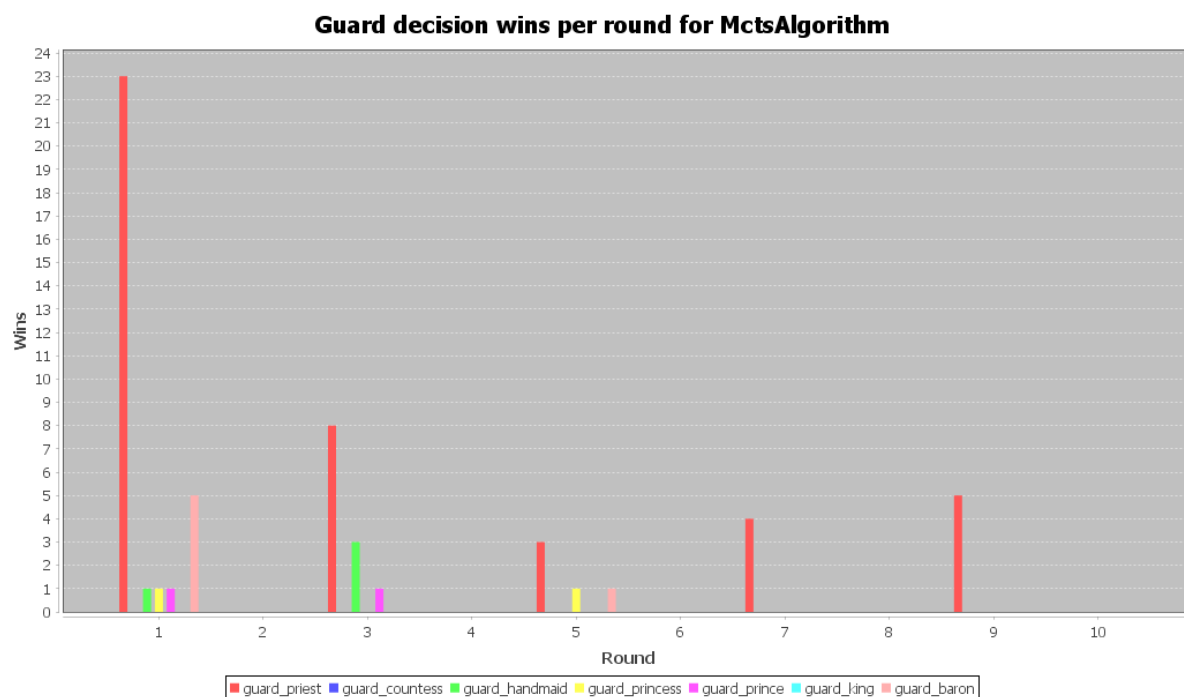


Rysunek 5.25. Wykres zwycięskich zagrań algorytmu MCTS

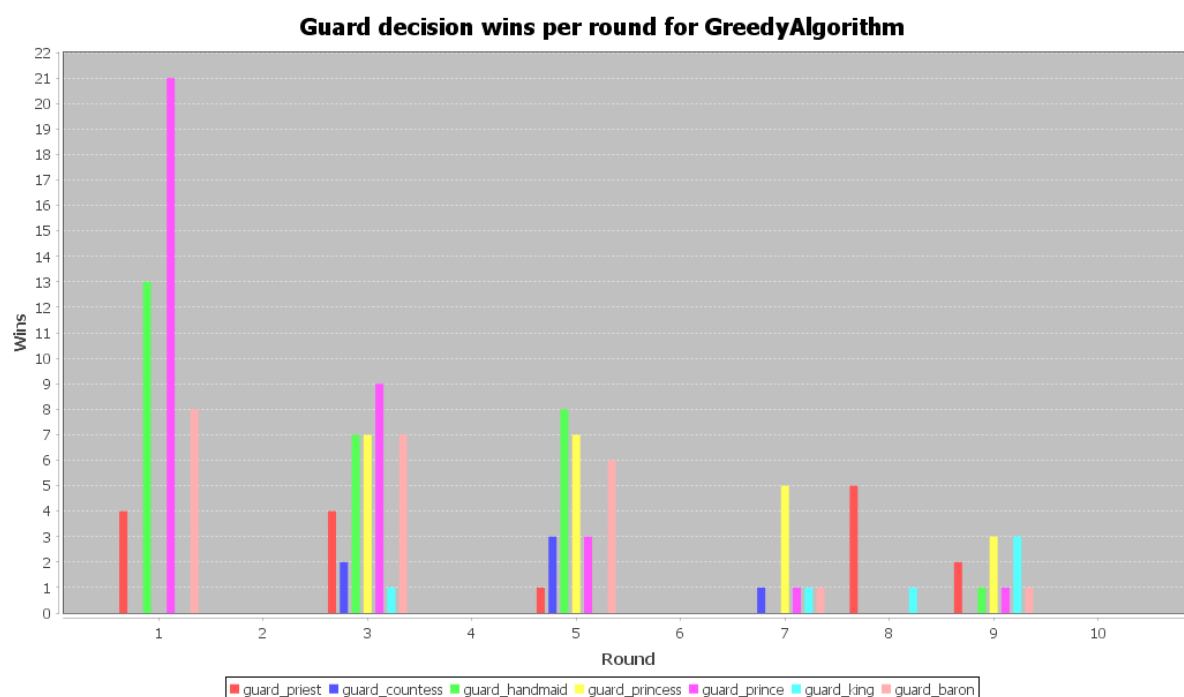


Rysunek 5.26. Wykres zwycięskich zagrań algorytmu zachłannego

Powyższe wykresy (rys. 5.25 i 5.26) zwracają uwagę na nieregularność algorytmu MCTS do zwyciężania zagraniami karty Baron. Może to wynikać z tego, że MCTS częściej zatrzymuje w ręce kartę o wyższym numerze, i kiedy przeciwnik zagrywa kartę Barona, przegrywa.



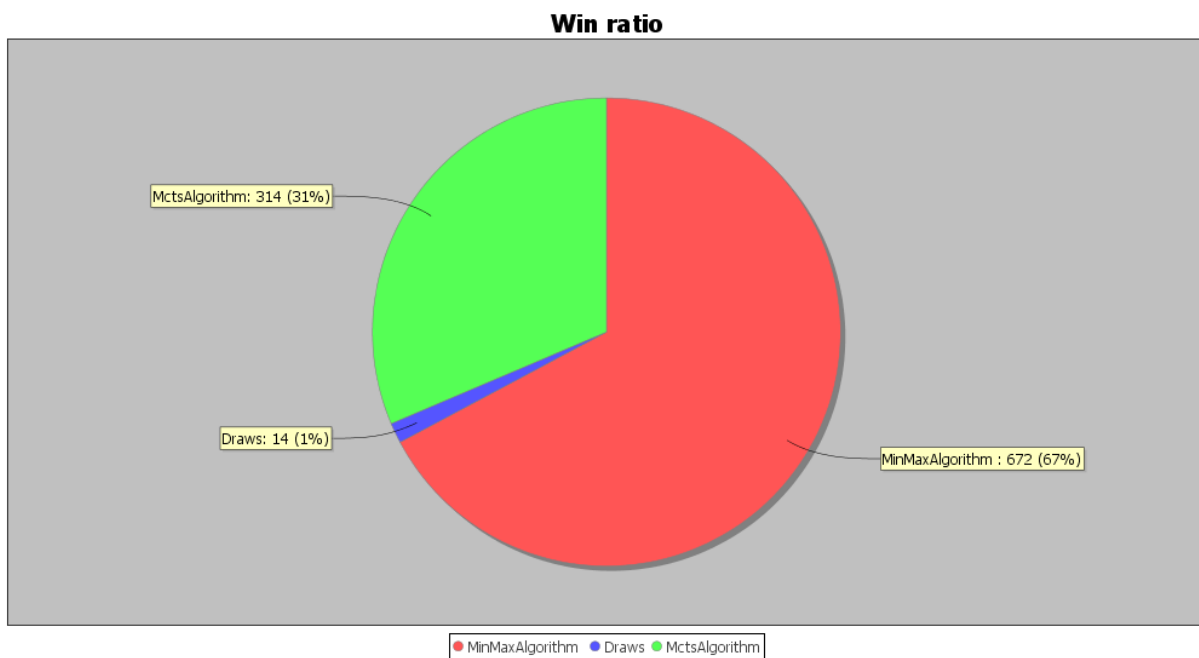
Rysunek 5.27. Wykres szczegółowy zwycięskich zagrań karty Strażniczki



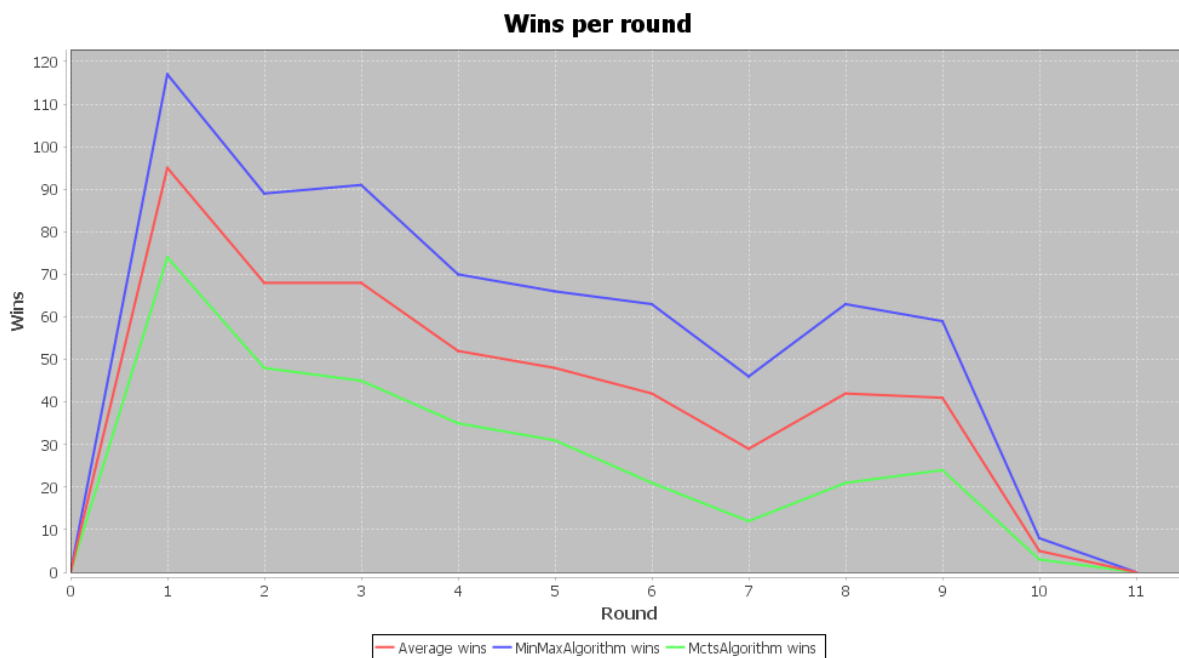
Rysunek 5.28. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

Na powyższych wykresach (rys. 5.27 i 5.28) można zauważyć zupełnie inne tendencje do zagrywania karty Strażniczki. Algorytm MCTS niemal zawsze wybiera kartę Kapłana, natomiast algorytm zachłanny mimo że preferuje wybór Księcia, to niemal równie często wybiera inne typy kart.

5.7. Algorytm Minimaksowy versus Algorytm MCTS

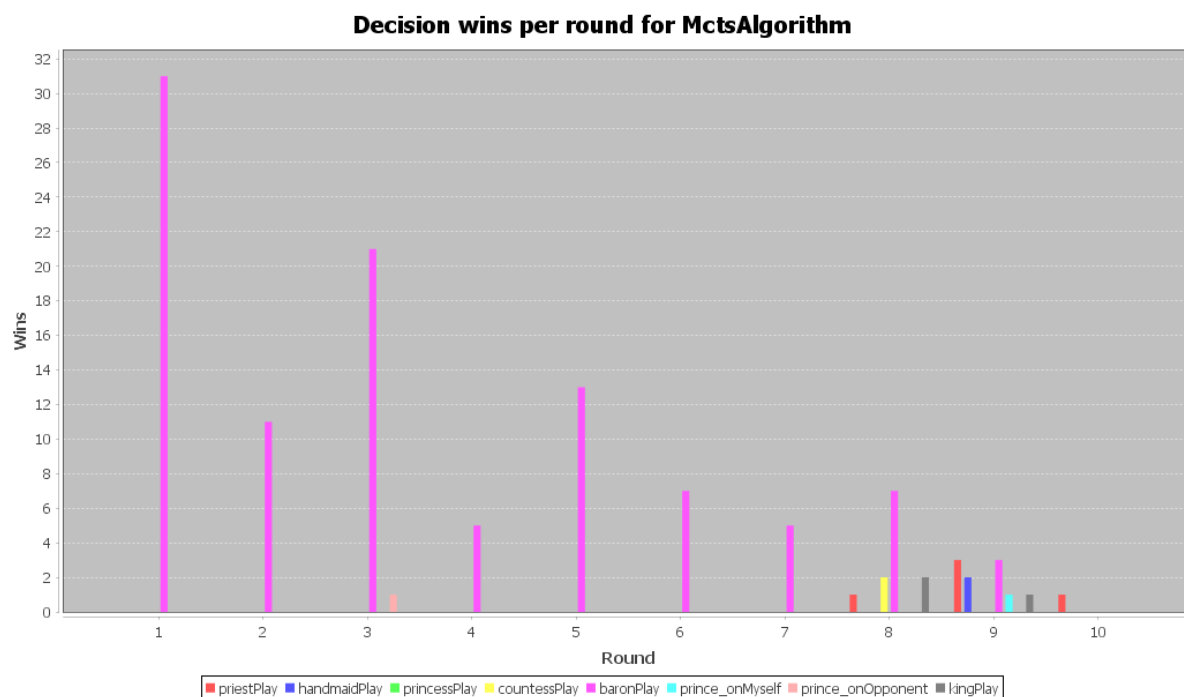


Rysunek 5.29. Wykres kołowy zwycięstw i remisów

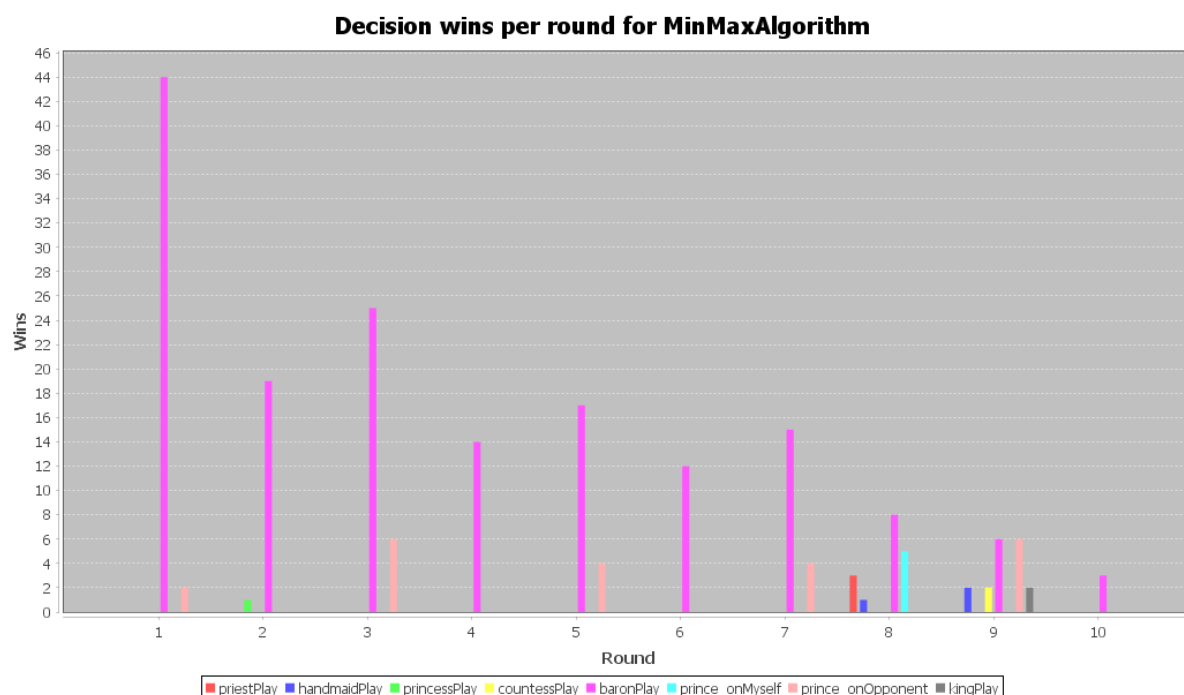


Rysunek 5.30. Wykres wygranych w danej rundzie

Powyższe wykresy (rys. 5.29 i 5.30) pokazują przewagę algorytmu minimaksowego nad algorytmem MCTS.

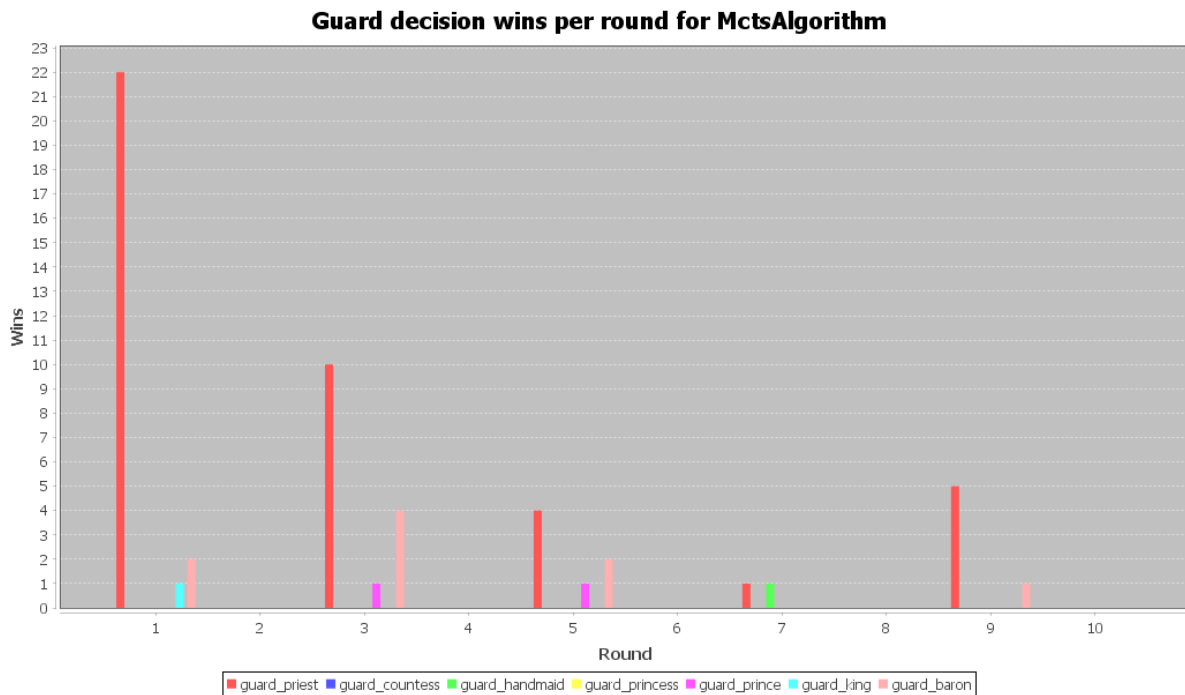


Rysunek 5.31. Wykres zwycięskich zagrań algorytmu MCTS

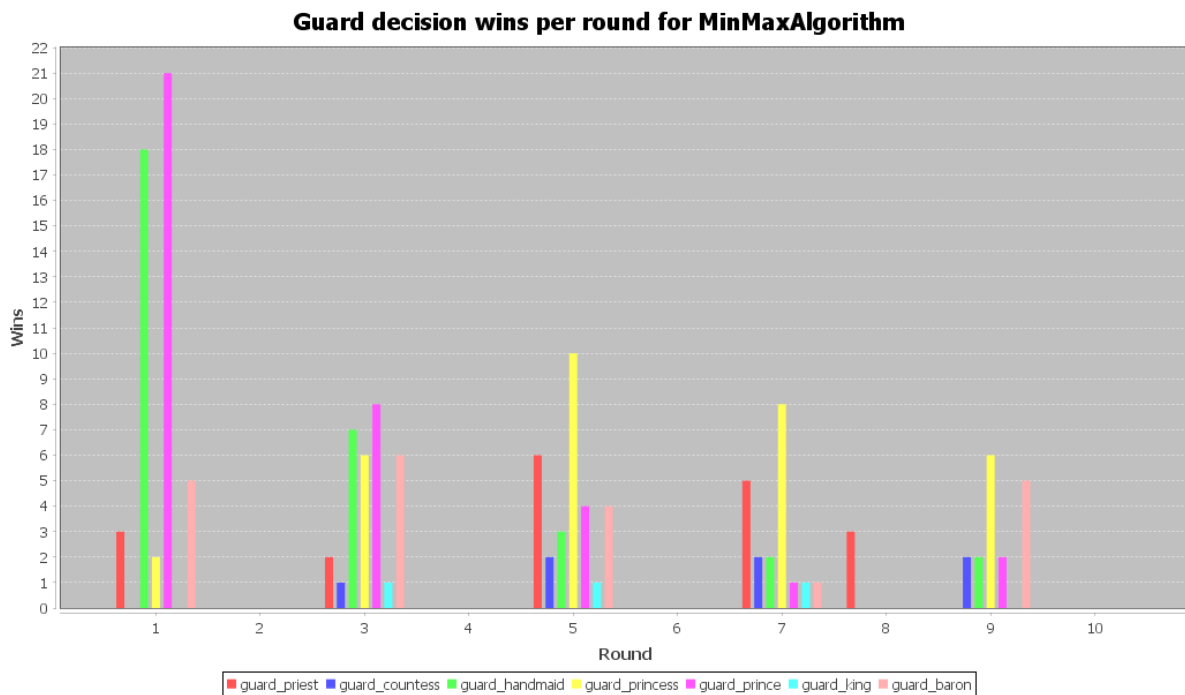


Rysunek 5.32. Wykres zwycięskich zagrań algorytmu zachłannego

Na powyższych wykresach (rys. 5.31 warto zwrócić uwagę na statystykę zwycięskich zagrań w ostatnich rundach. Algorytm minimaksowy do końca zagrywa karty Barona i często wykorzystuje Księcia. Zwycięskie zagrania algorytmu MCTS w ostatnich rundach są bardziej równomiernie rozłożone, a w ostatniej rundzie nigdy nie zagrywa karty Barona.



Rysunek 5.33. Wykres szczegółowy zwycięskich zagrań karty Strażniczki



Rysunek 5.34. Wykres szczegółowy zwycięskich zagrań karty Strażniczki

Na powyższych wykresach (rys. 5.33 i 5.34) widać, że algorytm MCTS zachowuje się podobnie jak wcześniej. Natomiast algorytm minimaksowy zdecydowanie częściej zagrywa kartę Strażniczki z wyborem Księżniczki niż w było to w przypadku algorytmu zachłannego.

5.8. Wnioski

Na podstawie przeprowadzonych badań można sformułować następujące wnioski:

1. Zdecydowana większość gier kończy się w pierwszych 3 rundach, głównie poprzez zagranie kart Strażniczki i Barona. Wynika z tego również, że gracz rozpoczynający ma znaczną przewagę nad drugim graczem.
2. W kolejnych rundach średnia ilość zwycięstw maleje, i następnie rośnie w dwóch ostatnich. Wynika to z dwóch zjawisk: po pierwsze, przy końcu gry gracze wyzbyli się już kart ofensywnych, czyli Strażniczki, Barona i Księcia, wobec czego dochodzi do porównania sił kart. Po drugie, jeśli karty ofensywne pozostały grze, znacznie wzrasta skuteczność ich użycia, co widać po częstoci zagrań Strażniczki ze wskazaniem na Księżniczkę, bądź Księcia ze wskazaniem na przeciwnika (spodziewając się, że ma Księżniczkę).
3. Ze statystyk można wywnioskować, że gra zdecydowanie sprzyja ofensywnym zagraniom.
4. Algorytm losowy nie wymaga głębokiej analizy - niemniej jednak warty uwagi jest fakt, że jest bardziej skuteczny niż algorytm MCTS.
5. Algorytm zachłanny osiąga wysokie wyniki w porównaniu z innymi algorytmami i niewiele niższe niż algorytm minimaksowy. Wynika to ze wspomnianego faworyzowania przez grę zagrań, które jak najszybciej zakończą grę.
6. Algorytm minimaksowy osiąga wyniki tylko niewiele lepsze niż algorytm zachłanny, pomimo znacznie dłuższego czasu działania. Warto pamiętać, że w przedstawionym wariantcie dokonuje on tylko przeszukania drzewa rozwiązań do 1 ruchu w przód. Algorytm ten osiąga przewagę nad zachłannym w rundach środkowych, jednak różnica zwycięstw jest niewielka.
7. Efektywność algorytmu Monte Carlo Tree Search jest zdecydowanie poniżej oczekiwań, co widać szczególnie po wynikach eksperymentu z algorytmem losowym. Przyczyną tego stanu rzeczy jest moje błędne założenie, że negatywny wpływ elementu losowego na algorytm może zostać zniwelowany przez ilość symulacji przeprowadzanych przez MCTS. Niestety, implementacja algorytmu w formie podstawowej sprawia, że wpada on w pewną pułapkę już podczas pierwszego kroku. Jest to spowodowane tym, że gracz w danym momencie gry nie zna całego jej stanu, lecz zbiór informacyjny I , w związku z tym w korzeniu drzewa zawarty jest jeden wylosowany stan z tego zbioru informacyjnego. Biorąc pod uwagę ilość stanów w tym zbiorze, szansa, że zostanie wylosowany ten faktyczny, jest niewielka. Każda symulacja wykonana na błędnie założonym stanie początkowym w korzeniu drzewa algorytmu MCTS będzie prowadzić do błędnych wniosków. W efekcie MCTS osiąga gorsze wyniki niż algorytm losowy. Można by to podsumować potocznym stwierdzeniem, że „lepszy jest brak wiedzy niż wiedza nieprawdziwa”.

6. Podsumowanie

Celem mojej pracy była analiza i porównanie efektywności wybranych algorytmów w podejmowaniu decyzji w grze „Love Letter”. Podstawowym założeniem było nie przeszukiwanie całego drzewa rozwiązań, lecz jedynie najbliższy poziom, bądź posłużenie się heurystyką.

Opisałem zasady gry „Love Letter”, a następnie oszacowałem ilość możliwych rozwiązań. Na tej podstawie przedstawiłem ją jako problem optymalizacyjny, który stał się podstawą do porównania efektywności algorytmów. Model gry zapisałem w postaci gry ekstensywnej opierając się na Teorii Gier.

Do analizy i porównania wybrałem następujące algorytmy: losowy, zachłanny, minimaksowy oraz Monte Carlo Tree Search, który jest algorytmem heurystycznym. Sposób działania każdego z nich został opisany i podałem przykłady ich wykorzystania w praktyce. Następnie zaprezentowałem pomysł ich użycia do podejmowania decyzji w grze „Love Letter”.

Kolejno zaprezentowałem założenia programu, w którym zaimplementowane zostały zasady gry oraz wspomniane algorytmy. Przedstawiłem analizę wymagań oraz diagramy objaśniające strukturę programu. W analizie post implementacyjnej wskazałem główny problem związany z napisaniem programu, jakim okazał się niedostateczny poziom abstrakcji, znacznie zwiększający objętość kodu i tym samym ryzyko popełnienia błędu.

Następnie zaprezentowałem wyniki przeprowadzonych przeze mnie symulacji gier. W analizie skupiłem się na ilości zwycięstw danego algorytmu oraz najczęściej wygrywających zagraniach. Ważnym wnioskiem było wskazanie najskuteczniejszego z analizowanych algorytmów, jakim okazała się moja implementacja algorytmu minimaksowego. Z pewnością jego skuteczność mogłaby być wyższa, jednak z racji charakteru gry, który preferuje szybkie kończenie rund ponad zagrywki przedłużające grę, czas działania algorytmu rósł by zdecydowanie szybciej niż jego efektywność.

Największym zaskoczeniem są wyniki algorytmu MCTS, gorsze nawet od wyników algorytmu losowego. Taki stan rzeczy wynika z błędnie przyjętych na początku założeń, że podstawowa wersja tego algorytmu będzie w stanie osiągać dobre wyniki w tej grze. Mimo to uważam, że po odpowiedniej modyfikacji polegającej na zamianie stanów znajdujących się w węzłach na zbiory informacyjne, byłby on w stanie osiągać wyniki równe, a nawet lepsze, od algorytmu minimaksowego.

Bibliografia

- [1] Alderac Entertainment Group, *Love Letter*, 2012.
- [2] Alderac Entertainment Group, *Love Letter*, <http://www.alderac.com/tempest/love-letter>, 2016-02-08.
- [3] Alicja Cewe, Halina Nahorska, Irena Pancer, *Tablice matematyczne*, Wydawnictwo Podkowa, Gdańsk 2002, rozdział *Kombinatoryka*.
- [4] *Wikipedia*, hasło *Problem optymalizacyjny*. https://pl.wikipedia.org/wiki/Problem_optymalizacyjny, 2016-02-21.
- [5] *Wikipedia*, hasło *Strategia w teorii gier*. https://pl.wikipedia.org/wiki/Strategia_mieszana, 2016-04-10.
- [6] Andrzej Z. Grzybowski *Matematyczne modele konfliktu. Wykłady z Teorii Gier i Decyzji*, Wydawnictwo Politechniki Częstochowskiej, Częstochowa 2012, rozdział *1.1 Klasyfikacja problemów decyzyjnych*.
- [7] Andrzej Z. Grzybowski *Matematyczne modele konfliktu. Wykłady z Teorii Gier i Decyzji*, Wydawnictwo Politechniki Częstochowskiej, Częstochowa 2012, rozdział *2.1 Gry w postaci ekstensywnej*.
- [8] *Wikipedia*, hasło *Drzewo (matematyka)*. [https://pl.wikipedia.org/wiki/Drzewo_\(matematyka\)](https://pl.wikipedia.org/wiki/Drzewo_(matematyka)), 2016-06-01.
- [9] *Wikipedia*, hasło *Teoria decyzji*. https://pl.wikipedia.org/wiki/Teoria_decyzji, 2016-04-10.
- [10] Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, *Algorytmy*, Wydawnictwo Naukowe PWN, Warszawa 2012, rozdział *Algorytmy Zachłanne*, s. 133.
- [11] *Wikipedia*, hasło *Algorytm min-max*. https://pl.wikipedia.org/wiki/Algorytm_min-max, 2016-04-24.
- [12] *Funkcja oceniająca do algorytmu minimaksu w grze warcaby*, http://sequoia.ict.pwr.wroc.pl/witold/aiarr/2009_projekty/warcaby/, 2016-05-04.

- [13] *Studia Informatyczne*, Sztuczna inteligencja/SI Moduł 8 - Gry dwuosobowe. http://wazniak.mimuw.edu.pl/index.php?title=Sztuczna_inteligencjaSI_Modul_8_-_Gry_dwuosobowe, 2016-04-24.
- [14] Broderick Arneson, Ryan Hayward, Philip Henderson, *MoHex Wins Hex Tournament*, ICGA Journal Vol. 32 No. 2 s. 114–116, Czerwiec 2009
- [15] *Wikipedia*, hasło *Monte-Carlo Tree Search*. https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search, 2016-05-04.
- [16] *Introduction to Monte Carlo Tree Search*, <https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/>, 2016-05-04.
- [17] Guillaume Chaslot, Sander Bakkes, Istvan Szita, Pieter Spronck, *Monte-Carlo Tree Search: A New Framework for Game AI*, Universiteit Maastricht / MICC. P.O. Box 616, NL-6200 MD Maastricht, The Netherlands
- [18] *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI*, <http://aigamedev.com/open/coverage/mcts-rome-ii/>, 2016-05-04.
- [19] Istvan Szita, Guillaume Chaslot, Pieter Spronck, *Monte-Carlo Tree Search in Settlers of Catan* Volume 6048 of the series Lecture Notes in Computer Science, s. 21-32, 2010.
- [20] *Monte Carlo Tree Search*, sekcja *About*, <http://www.cameronius.com/research/mcts/about/index.html>, 2016-05-05.
- [21] Krzysztof Sacha *Inżynieria oprogramowania* Wydawnictwo Naukowe PWN, Warszawa 2014, rozdział *Inżynieria wymagań*, s. 50.
- [22] *Wikipedia*, hasło *Wiersz poleceń*. https://pl.wikipedia.org/wiki/Wiersz_poleceń, 2016-05-08.
- [23] Oracle Corporation *Java* <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, 2016-05-08.
- [24] Krzysztof Sacha *Inżynieria oprogramowania* Wydawnictwo Naukowe PWN, Warszawa 2014, rozdział *Metodyka zwinna*, s. 334.
- [25] *Wikipedia*, hasło *Model kaskadowy*. https://pl.wikipedia.org/wiki/Model_kaskadowy, 2016-05-08.
- [26] JetBrains *IntelliJ IDEA* <https://www.jetbrains.com/idea/>, 2016-05-08.
- [27] Robert C. Martin *Czysty Kod* Wydawnictwo Helion, Gliwice 2014
- [28] Robert C. Martin *Czysty Kod* Wydawnictwo Helion, Gliwice 2014, rozdział 1. *Czysty Kod*, s. 30.
- [29] *Wikipedia*, hasło *SOLID (programowanie obiektowe)*. [https://pl.wikipedia.org/wiki/SOLID_\(programowanie_obiektowe\)](https://pl.wikipedia.org/wiki/SOLID_(programowanie_obiektowe)), 2016-06-28.