Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

Praca dyplomowa inżynierska

Krzysztof Werys

Zastosowanie uczenia się ze wzmocnieniem do wyznaczania strategii w środowisku Robocode

Opiekun pracy:
dr inż. Paweł Cichosz

Ocena .....

.....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalność: Informatyka –
Inżynieria systemów
informacyjnych

Data urodzenia: 13 października 1990 r.

Data rozpoczęcia studiów: luty 2010 r.

Życiorys

Urodziłem się 13 października 1990 roku w Kielcach. W 2009 roku ukończyłem II Liceum Ogólnokształcące im. Jana Śniadeckiego w Kielcach, gdzie uczęszczałem do klasy z rozszerzoną matematyką, fizyką oraz informatyką. W tym samym roku otrzymałem świadectwo dojrzałości. W lutym 2010 roku rozpocząłem studia na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Od trzeciego roku studiów moją specjalnością jest Inżynieria Systemów Informatycznych. W listopadzie 2013 roku rozpocząłem staż dla studentów uczelni technicznych w Europejskiej Organizacji Badań Jądrowych CERN.

.....
podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn. .....

Z wynikiem .....

Ogólny wynik studiów .....

Dodatkowe wnioski i uwagi Komisji .....

.....

Streszczenie

Praca ta prezentuje implementację algorytmu uczenia się ze wzmocnieniem Q-learning oraz jego zastosowanie do wyznaczania strategii w środowisku Robocode. Robocode to gra komputerowa symulująca bitwy czołgów oraz pełniąca rolę środowiska eksperymentalnego do badania algorytmów inteligentnego zachowania. W pracy zostały sprawdzone dwie strategie czołgu uczącego się: strategia agresywna i strategia przetrwania. Zbadana została również możliwość przenoszenia zdobytych umiejętności do innych instancji tego samego zadania.

Słowa kluczowe: *uczenie się ze wzcmocnieniem, Q-learning, Robocode, sztuczna inteligencja*

Abstract

Title: *An application of reinforcement learning in determining the strategy in Robocode.*

This thesis describes implementation of the reinforcement learning algorithm Q-learning and its application in determining the strategy in the game Robocode. Robocode is a computer game simulating tank battles which is an experimental environment for research in artificial intelligence. This thesis tests two robot strategies: aggressive strategy and careful strategy. Also the possible transfer of skills to other instances of the same problem was investigated.

Key words: *reinforcement learning, Q-learning, Robocode, artificial intelligence*

Spis treści

1. Wstęp

1.1. Wprowadzenie

Tworzenie sztucznej inteligencji dla gier komputerowych jest czynnością skomplikowaną. Gracz oczekuje nietrywialnej logiki przeciwników, która byłaby dla niego sporym wyzwaniem oraz która by umożliwiała rywalizowanie na podobnym poziomie zaawansowania. Coraz częściej można spotkać się z "inteligentnymi"botami, które podczas gry uczą się posunięć gracza i w trakcie rozgrywki coraz lepiej potrafią się jemu przeciwstawiać. Gry komputerowe stanowią również dogodny "poligon doświadczalny"dla algorytmów przydatnych także w świecie rzeczywistym (np. autonomiczne roboty, pojazdy bezzałogowe itp.).

1.2. Motywacja

Motywacją tworzenia programów uczących się jest złożoność niektórych zadań jakie są stawiane programom komputerowym. Z tego względu niekiedy sformułowanie poprawnych algorytmów jest utrudnione bądź nawet uniemożliwione.

Algorytmy uczenia maszynowego są wykorzystywane podczas tworzenia sztucznej inteligencji dla gier komputerowych. Są one pomocne przy rozwiązywaniu problemów takich jak znajdowanie ścieżek, współpraca z innymi graczami czy ogólna kontrola bota. Szereg z nich można rozwiązać korzystając z prostej algorytmizacji. Niestety gdy od sztucznej inteligencji oczekiwane jest zachowanie podobne do człowieka lub gdy trzeba podjąć decyzję w warunkach, w których nie posiadamy wszystkich danych o środowisku, takie rozwiązanie może nie być wystarczające. Aby stawić czoło takiemu problemowi można wykorzystać algorytmy uczenia się ze wzmocnieniem.

System opierający się na uczeniu się ze wzmocnieniem jest to taki system uczący, który w trakcie interakcji ze środowiskiem obserwuje jego stan i wykonuje akcje oparte na pewnej strategii. Po wykonaniu akcji otrzymuje nagrodę bądź karę stanowiącą informację zwrotną na temat jakości ostatnio podjętych działań.

1.3. Cel i zakres

Celem niniejszej pracy inżynierskiej jest implementacja oraz zbadanie możliwości zastosowania algorytmu uczenia się ze wzmocnieniem Q-learning w środowisku Robocode.

Robocode [6] jest to gra open-source symulująca bitwy czołgów w czasie rzeczywistym. Roboty muszą odpowiednio nawigować w środowisku tak aby unikać pocisków, kolizji z innymi zawodnikami czy zderzeń ze ścianą. Aby wygrać czołg musi zlokalizować przeciwnika i strzelić w niego odpowiednią ilość razy. Robocode jest symulatorem, czyli sztucznie odtwarza właściwości i zjawiska zachodzące podczas walk robotów. Dlatego też posiada pewne cechy realistyczne:

- Wykonanie pewnych czynności zajmuje czas. Na przykład podczas rotacji robot może zostać postrzelony.
- Dotarcie pocisku do celu wymaga czasu. Jeżeli strzał jest oddawany do poruszającego się obiektu, aby trafić trzeba brać pod uwagę zmianę jego położenia.
- Informacje o przeciwnikach są ukryte. Dane o wrogu można dostać dopiero wtedy gdy zostanie on żauważony"poprzez radar.
- Zderzenia z innymi obiektami ranią.

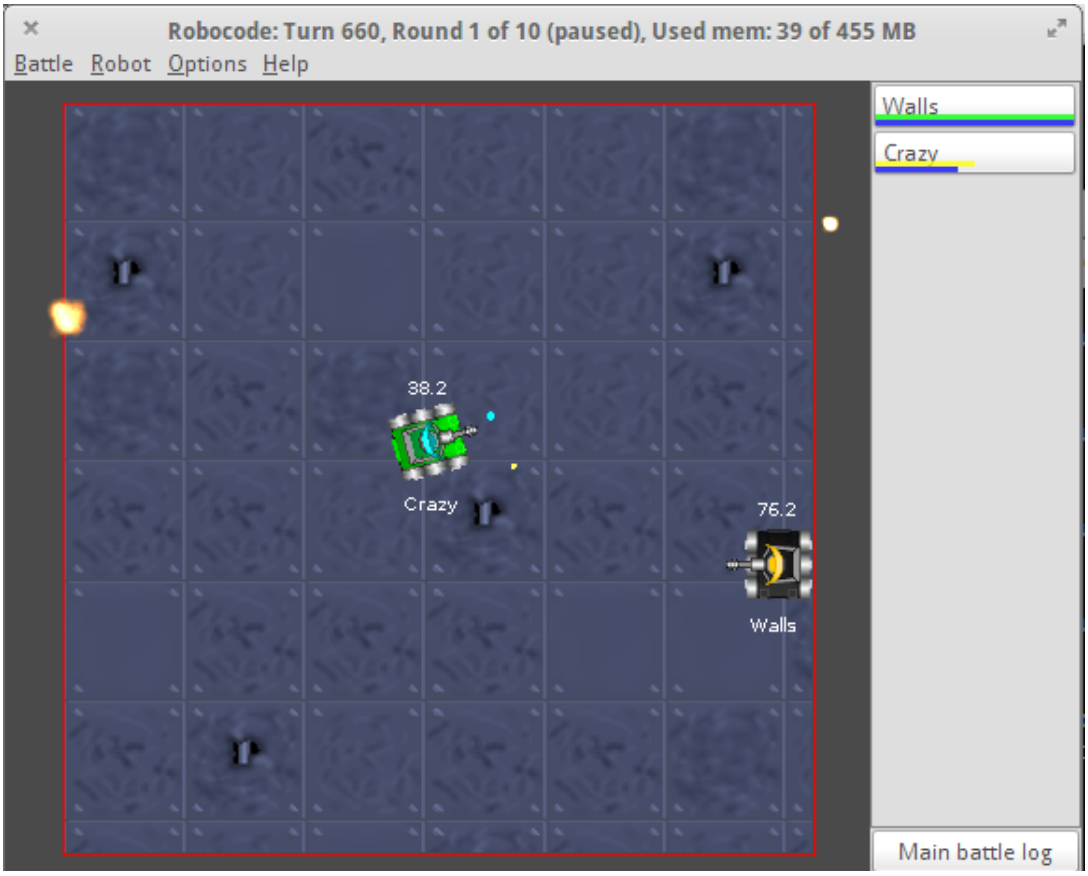
Cechy nierealistyczne środowiska Robocode to:

- Czujniki są idealne i wolne od szumu.
- Czujniki wykrywają energię robota, prędkość oraz kąt pod jakim się znajduje.
- Walka przeprowadzana jest w prostokątnym środowisku 2D bez przeszkód.

W ramach projektu została przebadana jakość strategii uzyskiwanych za pomocą uczenia się ze wzmocnieniem w porównaniu do algorytmów zdroworozsądkowych. W tym celu zostały sprawdzone dwie strategie robota uczącego się: strategia agresywna i strategia przetrwania. Sprawdzona została również możliwość przenoszenia zdobytych umiejętności do innych instancji tego samego zadania. Jakość działania robotów była sprawdzana względem wyników uzyskiwanych przez roboty dostarczane wraz z implementacją gry.

1.4. Przegląd rozdziałów

Pierwsza częśc pracy zawiera przegląd algorytmów uczenia maszynowego wraz z przykładami. Opisane jest również zastosowanie algorytmów sztucznej inteligencji w grach komputerowych.



Rysunek 1.1. Przykładowa walka w grze Robocode.

Rozdział trzeci opisuje podstawy teoretyczne dotyczące m.in procesów decyzyjnych Markowa oraz algorytmów uczenia się ze wzmocnieniem.

W rozdziale czwartym znajduje się opis środowiska Robocode oraz szczegóły dotyczące zastosowania oraz implementacji algorytmu Q-learning.

Rozdział piąty zawiera opis oraz wyniki przeprowadzonych eksperymentów dotyczących zastosowania uczenia się ze wzmocnieniem w grze Robocode.

W rozdziale szóstym znajduje sie podsumowanie, a w dodatkach informacje o wykorzystanych technologiach oraz o konfiguracji środowiska pracy.

2. Dotychczasowy stan wiedzy

2.1. Przegląd algorytmów uczenia maszynowego

Algorytmy uczenia maszynowego ze względu na postać informacji trenującej można podzielić na trzy poniższe grupy [2], które zostaną opisane w następnych punktach:

- uczenie nadzorowane,
- uczenie bez nadzoru,
- uczenie ze wzmocnieniem.

2.1.1. Uczenie nadzorowane

Uczenie nadzorowane [8, 10, 13] jest to uczenie, które zakłada obecność ludzkiego nadzoru polegającego na dostarczaniu zestawu danych uczących. Zadaniem systemu jest nauczenie się przewidywania prawidłowej odpowiedzi na zadane pobudzenie oraz generalizacja przypadków wyuczonych na przypadki, z którymi system jeszcze się nie zetknął.

Główne zadania rozwiązywane przez uczenie nadzorowane to:

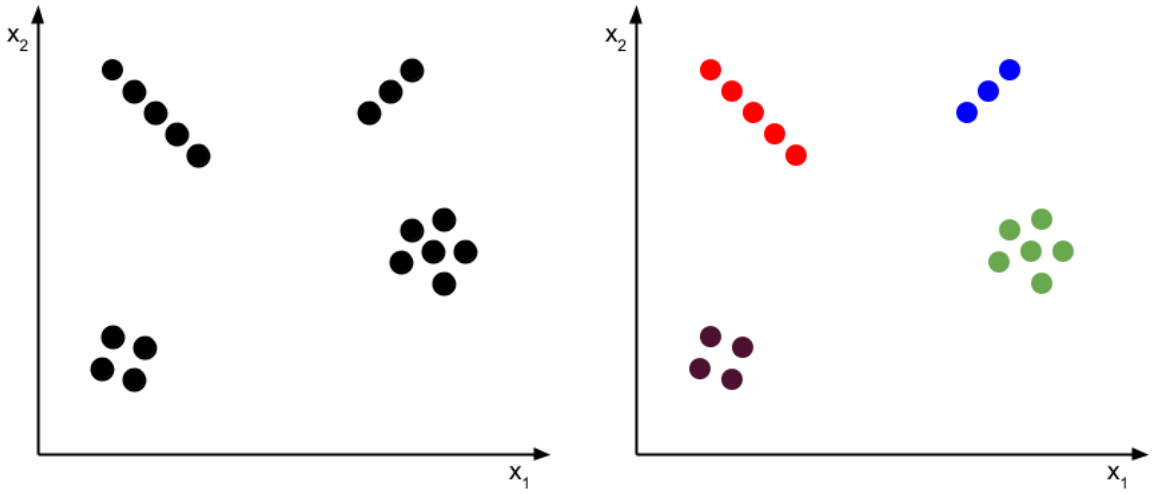
Klasyfikacja Polega na określeniu klasy obiektów na podstawie ich cech. Przykładowym zastosowaniem może być klasyfikacja artykułów z różnych portalów internetowych na artykuły interesujące i nieciekawe dla danego użytkownika [13]. System uczyłby się na podstawie oznaczeń użytkownika czy lubi dany tekst czy nie. Następnie po jakimś czasie byłby w stanie sam klasyfikować nowe artykuły.

Regresja Jest to metoda wykorzystywana w przewidywaniu liczbowych wartości przypisanych obiektom na podstawie ich cech. Przykładem tutaj może być przewidywanie wartości pewnego mieszkania na podstawie ceny mieszkań w okolicy [2].

2.1.2. Uczenie bez nadzoru

Uczenie bez nadzoru [9, 17], w przeciwieństwie do uczenia nadzorowanego, na wejściu nie otrzymuje etykietowanych danych (stwierdzających

przynależność do danej grupy). Uczenie bez nadzoru stara się znaleźć ukryte struktury wśród nieoznaczonych danych. Przykładowym zadaniem uczenia bez nadzoru jest grupowanie, czyli podział danego zbioru elementów na podzbiory (kategorie, grupy). Elementy każdego z podzbiorów powinny być bardziej podobne do innych elementów z tego samego podzbioru, niż do elementów innych podzbiorów.



Rysunek 2.1. Przykładowy rezultat grupowania zbioru danych o dwóch atrybutach: x_1 i x_2 . Przykłady należące do tej samej grupy oznaczone zostały tym samym kolorem.

Przykładem zastosowania uczenia bez nadzoru jest chociażby serwis Google News[1] (rysunek 2.2). Po wcześniejszym przetworzeniu artukułów grupowane są odnośniki prowadzące do wiadomości o tej samej tematyce, a następnie wspólnie wyświetlane. Dzięki temu możliwe jest czytanie tych samych informacji na różnych portalach internetowych [2].

Rysunek 2.2. Przykład zastosowania uczenia bez nadzoru. Odnośniki prowadzące do artykułów o tej samej tematyce są grupowane, a następnie razem wyświetlane.

[1] http://news.google.pl/

2.1.3. Uczenie się ze wzmocnieniem

W przypadku uczenie się ze wzmocnieniem [9, 21] nie są dostarczane żadne przykłady trenujące. System uczący się w trakcie interakcji ze środowiskiem obserwuje jego stan i wykonuje akcje oparte na pewnej strategii. Po wykonaniu akcji otrzymuje nagrodę bądź karę stanowiącą informację zwrotną na temat jakości ostatnio podjętych działań.

Uczenie się ze wzmocnieniem jest szeroko wykorzystywane w robotyce, w grach oraz pojazdach/samolotach autonomicznych. Przykładem zastosowań jest:

- Robot uczący się odbijać piłeczkę w tenisie stołowym [18].
- Nauka latania helikopterem do góry nogami [19].
- System uczący się wykorzystujący uczenie się ze wzmocnieniem wraz z siecią neuronową do gry w Tryktraka (ang. backgammon). Program, nie znając wcześniej zasad i tylko na podstawie otrzymywanych nagród osiągnął poziom mistrzowski (należy do kilku najlepszych graczy na świecie) [22].

Ten typ uczenia się jest wykorzystany w pracy i będzie szerzej opisany w następnych rozdziałach.

2.2. Sztuczna inteligencja w grach komputerowych

Sztuczna inteligencja jest szeroko stosowana w grach komputerowych. Dostarczanie konkurencyjnych i nieprzewidywalnych graczy jest bardzo ważną cechą. Gdy logika przeciwników jest mało zaawansowane, gracz jest w stanie szybko odnaleźć optymalną strategię, dzięki której będzie zawsze wygrywał.

Celem sztucznej inteligencji w grach komputerowych jest symulowanie inteligentnego i realistycznego zachowania [3]. Logika przeciwników powinna być tak skonstruowana, aby człowiek był w stanie z nią konkurować. Ważną funkcją jest możliwość operowania i balansowania poziomem zaawansowania. Akcje wykonywane powinny być powtarzalne na podobnym poziomie, ale w taki sposób, aby były trudne do przewidzenia. Warto również zaznaczyć, że gry komputerowe stanowią dogodny "poligon doświadczalny"dla algorytmów przydatnych także w świecie rzeczywistym [5].

Symulatory jazdy

Ciekawym zastosowaniem sztucznej inteligencji jest technologia Drivatar™ [4] wykorzystana w grze Forza Motorsport[2]. Głównym założeniem jest stworzenie własnego awatara, który uczy się stylu jazdy gracza. Następnie gdy osiągnie on pewien poziom perfekcji, gracz może zadecydować, że do pokonywania trasy wykorzystany będzie awatar symulujący jego styl jazdy. Jest to wygodna funkcja, chociażby w przypadku, gdy po raz kolejny trzeba przechodzić tę samą trasę, a gracz nie chce poświęcać już na nią więcej czasu. Jest też możliwe przejście całego trybu gry korzystając z awatara. Dużą zaletą technologii Drivatar jest również możliwość dzielenia się swoimi awatarami ze znajomymi. Daję to opcję konkurowania z przyjaciółmi w czasie, gdy akurat są oni niedostępni. Warto również wspomnieć o tym, że wszyscy przeciwnicy w grze Forza Motorsport są wcześniej przeszkolonymi awatarami.

W komercyjnych symulatorach jazdy trasa jaką powinien pokonać samochód jest ustalana przez ekspertów [16], a następnie testowana w grze przez osoby tworzące dane oprogramowanie. Jest to podejście czasochłonne. Badania przeprowadzone na grze TORCS opisane w [12] pokazują, że jest możliwe wyznaczenie trasy bez udziału człowieka i uzyskanie przy tym bardzo obiecujących wyników. W tym podejściu został wykorzystany algorytm genetyczny.

Gry zręcznościowe

W 2009 roku zostały przeprowadzone zawody "Mario AI Competition" [23] (ze współpracą z IEEE Games Innovation Conference oraz z IEEE Symposium on Computational Intelligence and Games) podczas których konkurowały różne kontrolery grające w grę Infinite Mario Bros[3]. Jest to klon klasycznej gry Super Mario Bros. Celem konkursu było stworzenie najlepszej logiki bota potrafiącego przejść grę. Zwycięzcą był ten, kto zdobył jak największą ilość punktów. Punktacja opierała się na jak najszybszym przejściu poziomu, zdobyciu jak największej ilości monet oraz na ilości pokonanych wrogów. Wyniki zawodów pokazały, że w tej grze najlepsze efekty można uzyskać opierając się na algorytmie A*. Wszystkie inne podejścia, takie jak sieci neuronowe, systemy eksperckie czy programowanie genetyczne uzyskiwały o połowę punktów lub mniej niż podejścia wykorzystujące A*.

Na przykładzie gry Quake 3 Arena, wykorzystywane są między innymi algorytmy genetyczne i algorytmy logiki rozmytej. Przykładem ich użycia jest chociażby podejmowanie decyzji o zbieraniu przedmiotów i broni [24].

[2] http://www.forzamotorsport.net/
[3] http://www.mojang.com/notch/mario

Rysunek 2.3. Wizualizacja możliwych ścieżek rozważanych przez algorytm A*. Każda czerwona linia pokazuję trajektorię Mario (biorąc pod uwagę dynamicznie zmieniające się środowisko). Źródło: [23]

Gry strategiczne

Badania przeprowadzone na podstawie gry Cywilizacja IV [11] pokazują, że uczenie się ze wzmocnieniem może zapewnić potężne narzędzie, które pozwala agentom na szybką adaptację i poprawę działania w grach strategicznych. System uczył się dobierania odpowiedniej wysokopoziomowej strategii (budowa budynków, prowadzenie wojny itp.) przeciwko stałej strategii wroga. Logika bota uzyskana w ten sposób dawała lepsze rezultaty niż ręcznie zaprogramowana sztuczna inteligencja.

Robocode

Praca “Modern AI for games” [20] pokazuje zastosowanie w grze Robocode algorytmu neruroewolucji do poruszania działem i namierzania przeciwników. W tym przykładzie do wyboru siły pocisku został użyty algorytm Q-learning.

Obiecujące wyniki prezentuje również praca “Evolving Robocode Tank Fighters” [14], gdzie został wykorzystany algorytm genetyczny do stworzenia logiki robota.

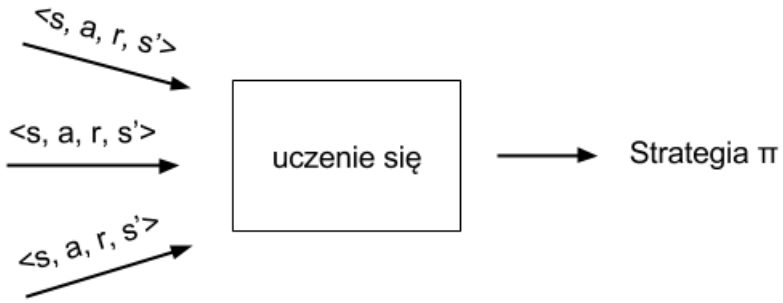
3. Podstawy teoretyczne

3.1. Uczenie się ze wzmocnieniem

Algorytm uczenia się ze wzmocnieniem czerpie swoje inspiracje z biologii. Eksperyment z psem Pawłowa jest na to dobrym przykładem [1]. Iwan Pawłow stwierdził, że podanie psu pokarmu wywołuje u niego wydzielanie śliny. Eksperyment polegał na tym, że przed nakarmieniem psa dzwonił dzwonek. Po kilku próbach pies nauczył się, że zawsze po dzwonku dostaje nagrodę. Późniejsze samo dzwonienie dzwonkiem bez dawania pożywienia wywoływało reakcję ślinienia się u psa.

Uczenie się ze wzmocnieniem polega na dynamicznych interakcjach ucznia ze środowiskiem. W dyskretnych krokach czasu uczeń realizuje swoje zadanie wybierając pewne akcje zgodnie z jego aktualną strategią. Obserwuje w tym samym czasie zmieniający się stan środowiska. Po wykonaniu akcji uczeń otrzymuję informację zwrotną (nagrodę bądź karę) na temat ostatnich zachowań, która wpływa na przyszłe decyzje.

Typowy scenariusz uczenia się ze wzmocnieniem zakłada tryb uczenia się online. Oznacza to, że dane potrzebne do podjęcia decyzji otrzymywane są partiami (w turach). Przykładowy proces jest przedstawiony na grafice 3.1:



Rysunek 3.1. Czarnoskrzynkowy przykład uczenia się.

Na wejściu algorytmu podawane są informacje odnośnie przejścia stanów. Są to informacje otrzymane podczas interakcji ze środowiskiem: stan początkowy s w jakim znajdował się uczeń, akcja a jaka została podjęta, otrzymana nagroda r oraz stan końcowy s'.

3.2. Procesy decyzyjne Markowa

Proces decyzyjny Markowa jest to model matematyczny opisujący problem uczenia się ze wzmocnieniem. Jest to formalizacja niedeterministycznego przeszukiwania, czyli przeszukiwania w sytuacji, gdy wyniki akcji są nieznane.

3.2.1. Definicja procesu decyzyjnego Markowa

Każde zadanie uczenia się ze wzmocnieniem, które spełnia własność Markowa nazywamy procesem decyzyjnym Markowa (ang. Markov decision process, MDP) [21]. Jest on opisany poprzez:

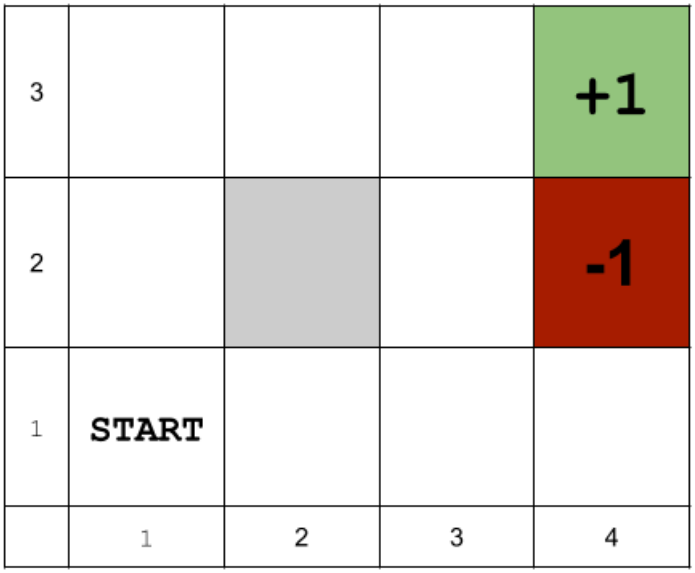
- Skończony zbiór stanów S - jest to zbiór, który reprezentuje wszystkie możliwe stany, w jakim może się znaleźć środowisko,
- Skończony zbiór akcji A - określa wszystkie możliwe czynności, jakie gracz może wykonać w danym stanie,
- Funkcję przejścia stanów $T(s, a, s')$- tę funkcję można najprościej określić jako zasady gry. Zasady, które w trakcie rozgrywki są niezmienne. Funkcję przejść stanów można nazwać "fizyką"danego środowiska. Określa ona prawdopodobieństwo $P(s'|s, a)$ znalezienia się w stanie s', będąc w stanie s i podejmując akcję a,
- Funkcję nagrody $R(s, a)$ - jest to informacja zwrotna na temat przydatności stanu i wykonania w nim akcji a. Warto zaznaczyć, że jest oczekiwana wartość nagrody (nagrody mogą też być stochastyczne).

Przykładowe proste środowisko, które może zostać opisane przez proces decyzyjny Markowa jest przedstawione na rysunku 3.2.

Celem ucznia jest dotarcie na zielone pole. Gdy znajdzie się on na czerwonym polu to przegrywa i musi rozpocząć grę od stanu startowego (1,1). Szare pole (2,2) przedstawia przeszkodę, na którą nie można wejść. Akcje jakimi dysponuje to: GÓRA, DÓŁ, LEWO i PRAWO. Jest to środowisko stochastyczne, to znaczy akcje nie zawsze zostają wykonane zgodnie z planem. W tym przypadku, wykonanie akcji GÓRA kończy się sukcesem w 80% przypadków. W 20% zamiast przejść na planowane pole, uczeń poruszy się w prawo.

Środowisko przedstawione na przykładzie może zostać opisane następująco:

stany (1,1), (1,2), . . . , (4,3)
akcje GÓRA, DÓŁ, LEWO, PRAWO
funkcja przejścia stanów $P(s'|s, a) = 0.8$, gdy a = GÓRA, $P(s'|s, a) = 0.2$ gdy a = GÓRA i s' jest stanem na prawo od s.
funkcja nagrody $R((4, 2), a) = -1$; $R((4, 3), a) = +1$ gdzie a to dowolna akcja.



Rysunek 3.2. Przykładowe środowisko.

3.2.2. Własność Markowa

Własność Markowa [21] określa, że tylko dana chwila ma znaczenie. Kluczowe dla środowiska jest to, że nagrody i przejścia stanów nie zależą od historii wcześniejszych stanów i akcji. Ucznia nie interesuje historia (stany w których był w przeszłości). Do podjęcia decyzji ważny jest stan, w którym aktualnie się znajduje. W każdym kroku nagroda i następny stan zależą tylko od aktualnego stanu i akcji. Drugą ważną rzeczą jest to, że środowisko jest niezmienne.

3.2.3. Strategia

Rozwiązaniem problemu zdefiniowanego jako proces decyzyjny Markowa jest strategia . Strategia, jest to funkcja, która przyjmuje stan, a zwraca akcję: $\pi(s) \rightarrow a$. Czyli innymi słowy, mówi jaką akcję podjąć w danym stanie, tak aby zadanie zostało rozwiązane.

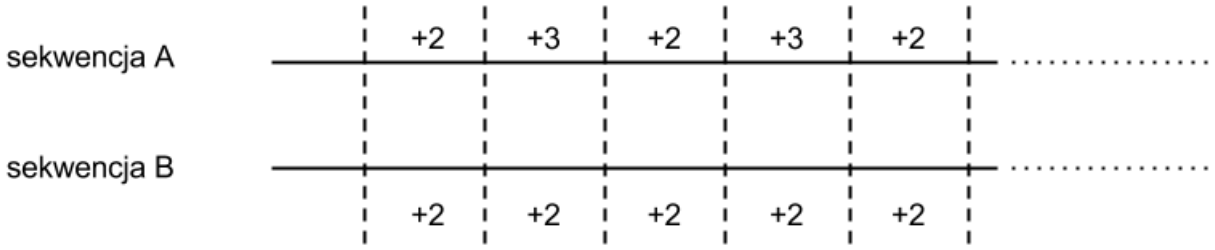
3.2.4. Nagrody

W zadaniu uczenia się ze wzmocnieniem zakłada się, że liczbowa wartość nagrody jest dostępna w każdym kroku. Nie musi ona jednak faktycznie odzwierciedlać efektów bezpośrednio poprzedzającej akcji. Możliwa jest sytuacja, że dopiero po wykonaniu odpowiedniej sekwencji ruchów dostaniemy informację zwrotną od środowiska. Są to tak zwane opóźnione nagrody, co znaczy, że sukces/porażka może zależeć nie tylko od akcji bezpośrednio je poprzedzających, lecz także wcześniejszych. Dobrym przykładem jest gra w

szachy. Całą rozgrywkę, która składa się z wielu tur można przegrać dzięki tylko jednemu złemu posunięciu na początku gry. Mimo to, że cały czas graliśmy optymalnie (poza tym jednym złym ruchem), to dowiemy się o naszym błędzie dopiero na sam koniec, gdy nasz król zostanie zamatowany.

3.2.5. Dyskontowanie nagród

Istotnym problemem jest wybór sekwencji ruchów gdy gra nie ma końca. Załóżmy, że mamy do wyboru jedną z dwóch podanych sekwencji wraz z nagrodami:



Rysunek 3.3. Przykład dwóch sekwencji ruchów.

Czy sekwencja A jest lepsza od B? Do oceny sekwencji zostanie wykorzystana funkcja wartości stanu U(ang. utility function) [7]. Symbol E oznacza wartość oczekiwaną przy założeniu posługiwania się strategią π.

$$U(s) = E_\pi \left[\sum_{t=1}^{\infty} R_t | s_0 = s \right] \quad \text{gdzie s} \in \text{S} \tag{3.1}$$

Dla sekwencji A jak i B, funkcja wartości wynosi nieskończoność. Nieważne jaka akcja zostanie wybrana, to i tak będziemy zbierać nagrody dążąc do nieskończoności. Czyli wybór sekwencji A jest tak samo dobry jak i wybranie sekwencji B.

Aby wybór sekwencji stanów był łatwiejszy w przypadku niekończącej się gry, można wprowadzić współczynnik dyskontowania γ [13]. Przy takim podejściu funkcja wartości będzie wyglądała następująco:

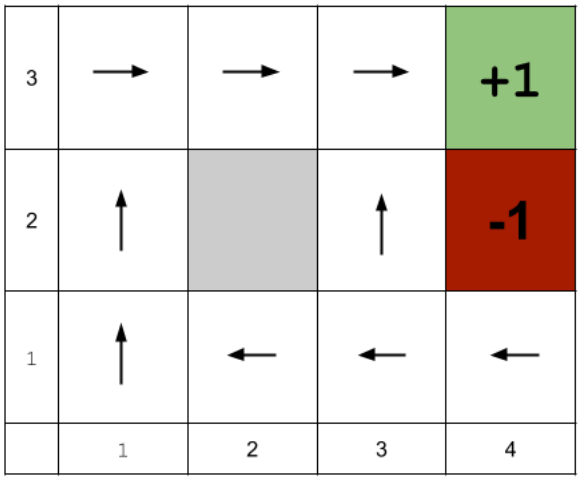
$$U(s) = E_\pi \left[\sum_{t=1}^{\infty} \gamma^t R_t | s_0 = s \right] \quad \text{gdzie } 0 \leq \gamma < 1; \text{ s} \in \text{S} \tag{3.2}$$

Współczynnik dyskontowania określa miarę naszego zainteresowania nagrodami w przyszłości. Gdy współczynnik jest bliski 0 oznacza to, że

zwracamy bardziej uwagę na natychmiastowe nagrody. Jeśli współczynnik jest bliski 1 to długoterminowe nagrody mają większe znaczenie.

3.2.6. Strategia optymalna

Strategia optymalna oznaczona przez π^* jest to taka strategia, która maksymalizuje wartość każdego stanu [21]. Bardzo ważnym elementem MDP jest odpowiedni dobór wartości nagród i kar. W zależności jak zostaną one dobrane, takiej strategii uczeń się nauczy.

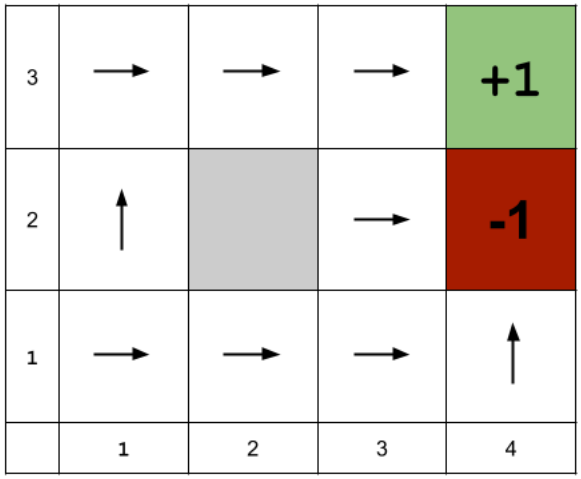


Rysunek 3.4. Przykład 1: $R(s) = -0.04$

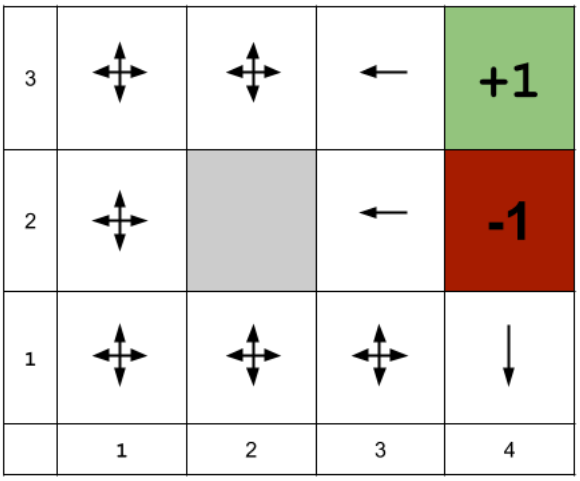
Na rysunku 3.4 został umieszczony przykład strategii optymalnej, w przypadku gdy uczeń otrzymuje karę za każdą rundę o wartości -0.04. Zmusza to do szybszego dotarcia do celu, czyli stanu (4,3). Warto zauważyć, że w stanie (3,1) mimo to, że uczeń ma o wiele krótszą drogę do zielonego pola i tak wybiera drogę naokoło. Uzasadnieniem takiej decyzji jest chęć uniknięcia ryzyka (wejście na czerwone pole) będąc w stanie (3,2). Jest to niebezpieczne miejsce ze względu na to, że środowisko jest stochastyczne (istnieje 20% szansa, że zostanie wykonana akcja PRAWO zamiast akcji GÓRA). Idąc dłuższą ścieżką uczeń otrzyma karę w wysokości 7 * (-0.04) = -0.28. Jest to lepsze rozwiązanie niż pójście w górę ryzykując otrzymaniem -1.

W przypadku przedstawionym na rysunku 3.5 uczeń w każdej turze otrzymuje karę o wartości -2. Na przykładzie widać wyraźnie, że najlepszym rozwiązaniem jest dotarcie do jakiegokolwiek punktu końcowego. Nie ważne jest, czy to jest zielone czy czerwone pole. Bardziej opłacalne jest zakończenie gry w stanie (4,2) niż dostanie kolejnych kar i zakończenie w stanie (4,3).

W ostatnim przykładzie 3.6 uczeń otrzymuje nagrodę za każdą turę w wysokości +2. W tej sytuacji unika on zakończenia gry. Nieopłacalne jest dojście do punktów terminalnych. Nawet znalezienie się na zielonym polu



Rysunek 3.5. Przykład 2: $R(s) = -2$



Rysunek 3.6. Przykład 3: $R(s) = 2$

jest niechciane, bo poruszając się w przeciwnym kierunku, uczeń otrzyma o wiele większą nagrodę.

3.2.7. Rozwiązywanie procesów decyzyjnych Markowa

Rozwiązaniem procesu decyzyjnego Markowa jest strategia optymalna, czyli strategia maksymalizująca wartość każdego stanu [21].

$$\pi^*(s) = argmax \sum_{s'} T(s, a, s')U^*(s') \tag{3.3}$$

Do otrzymania wyniku pomocne będzie rozwiązanie następującego równania opisującego właściwość optymalnej funkcji wartości ze względu na

strategię optymalną, nazywane również równaniem Bellmana [21]:

$$U^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') U^*(s') \right] \tag{3.4}$$

Jest to wzór określający wartość stanu na podstawie nagrody $R(s)$ jaką otrzymamy za bycie w danym stanie oraz zdyskontowanej wartości nagrody jaką otrzymamy w przyszłości.

Aby otrzymać optymalną strategię, można wykorzystać jeden z możliwych algorytmów programowania dynamicznego - iterację wartości. Polega na podążaniu za kolejnymi krokami:

1. Początkowe określenie przydatności dla każdego stanu.
2. Uaktualnienie wartości przydatności na podstawie sąsiednich stanów:

$$U^*_{t+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') U^*_t(s') \right] \tag{3.5}$$

   gdzie $\sum_{s'} T(s,a,s') U^*_t(s')$ oznacza maksymalną przypuszczalną wartość funkcji przydatności.
3. Powtarzanie czynności nr 2 aż do zbieżności.

Powyższy algorytm nie wyznacza wprost strategii optymalnej, lecz optymalną funkcję wartości. Strategię optymalną można zidentyfikować na jej podstawie. Będzie to wtedy strategia zachłanna.

Warto chwilę się zastanowić, dlaczego ten algorytm działa, mimo to, że początkowe wartości funkcji przydatności przypisaliśmy arbitralnie. Duże znaczenie ma wartość nagrody $R(s,a)$, która jest prawdziwą wartością otrzymaną od środowiska. Jest poprawną oceną ostatnich ruchów. Ta wartość w każdym kroku algorytmy będzie propagowana, aż do otrzymania rozwiązania.

3.3. Algorytmy uczenia się ze wzmocnieniem

3.3.1. Q-learning

W poprzednim podrozdziale odnośnie procesów decyzyjnych Markowa zostało przetoczone równanie Bellmana dotyczącego optymalnej funkcji wartości U^*:

$$U^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') U^*(s') \right] \tag{3.6}$$

oraz równanie o strategii zachłannej, która polega na wyborze jednej z akcji, które są dostępne w danym stanie s, na podstawie szacowanych wartości kolejnych stanów s':

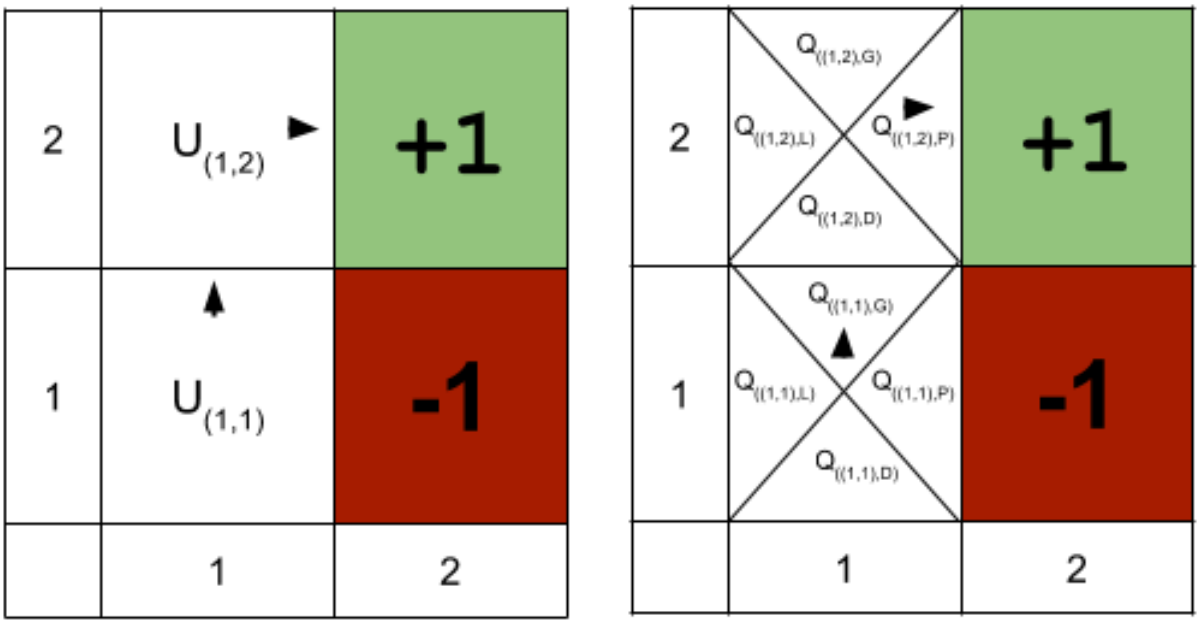
$$\pi^*(s) = argmax \sum_{s'} T(s,a,s')U^*(s') \tag{3.7}$$

W algorytmie Q-learning wprowadzona została nowa funkcja wartości par stan-akcja, funkcja $Q(s,a)$ [21]:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_a Q(s',a) \tag{3.8}$$

Określa ona jakość dla stanu s, w którym się aktualnie znajduje się uczeń oraz akcji a, która zostanie wykonana. Wartość Q jest wyliczona na podstawie nagrody $R(s,a)$ oraz zdyskontowanej szacowanej maksymalnej wartości Q w następnym stanie.

Przykład 3.7 prezentuje sposób wyznaczania strategii optymalnej przy pomocy funkcji $U(s)$ oraz funkcji $Q(s,a)$. Warto zauważyć, że funkcja Q jest wyliczana dla każdej akcji jaka może być wykonana w danym stanie. Funkcja U natomiast obliczana jest tylko dla samego stanu.



Rysunek 3.7. Po lewej przedstawiona jest strategia optymalna wyznaczona przy pomocy funkcji $U(s)$. Po prawej widnieje strategia wyznaczona przy użyciu funkcji $Q(s,a)$.

Przy pomocy wartości Q można również zdefiniować funkcję wartości U. Jest ona równa wartości Q dla takiej akcji a, która ją maksymalizuje [7]:

$$U(s) = \max_a Q(s, a) \tag{3.9}$$

Wyznaczenie strategii przy użyciu Q polega na wybraniu takiej akcji a ,która maksymalizuje wartość Q w danym stanie [7]:

$$\pi(s) = argmax_a Q(s, a) \tag{3.10}$$

Aby modyfikować w trakcie uczenia się wartość Q, wykorzystany jest poniższy wzór [13]:

$$Q(s, a) \leftarrow^{\alpha} r + \gamma \max_a Q(s', a') \tag{3.11}$$

gdzie $V \leftarrow^{\alpha} X$ można rozpisać jako $V \leftarrow (1 - \alpha)V + \alpha X$. Do $Q(s, a)$ jest przypisywana pewna część wartości stanu, która jest obliczana korzystając z nagrody r oraz zdyskontowanej maksymalnej wartości następnego stanu.

Parametr α jest nazywany współczynnikiem uczenia. Określa on tempo nauki ucznia. Gdy wartość wynosi 0, nauka nie ma miejsca. Gdy natomiast wartość jest równe 1, uczeń zapomina wszystkiego czego się nauczył i starą informację zastępuje całkowicie nową. Gdy $\alpha = 1/2$, jako wartość Q przypisywana jest średnia ze starej i nowej wartości Q.

3.3.2. Uczenie się ze wzmocnieniem a programowanie dynamiczne

Rozwiązanie problemu uczenia się ze wzmocnieniem różni się od rozwiązania procesu decyzyjnego Markowa (przedstawionego w podpunkcie 3.2.7) tym, że nie znamy naszego środowiska. Nie mamy informacji odnośnie funkcji przejścia stanów $T(s, a, s')$ oraz nagrody $R(s)$. Otrzymanie strategii polega na obserwacji środowiska i wykorzystaniu informacji jakie nam przekazuje. To znaczy wykorzystana zostaje informacja na temat każdej tranzycji $< s, a, r, s' >$. Są to jedyne dane jakimi uczeń dysponuje. Wie on, że był w stanie s, wykonał akcję a, dzięki której przeszedł do stanu s' otrzymując przy tam nagrodę w wysokości r.

3.3.3. Wybór akcji

Algorytm Q-learning nie definiuje w jaki sposób powinny być wybierane akcje. Określając strategię wyboru akcji, trzeba stawić czoła problemowi jakim jest wybór pomiędzy eksploracją a eksploatacją. Uczeń może pozyskiwać wiedzę bądź dążyć do zebrania większej ilości nagród. Od ucznia oczekujemy efektów nauki, poprawy działania czyli coraz większych nagród. Ale jeśli jego aktualna strategia nie jest optymalna, to dobrym pomysłem byłaby eksploracja nowych stanów, które spowodowałyby zwiększenie nagród.

Warto zauważyć, że Q-learning do funkcjonowania nie wymaga eksploracji. Jest ona tylko potrzebna do poprawy jakości działania. Wymagana natomiast jest eksploatacja, która gwarantuje znalezienie optymalnej strategii.

Wybór losowy

Gdy akcje są wybierane w sposób losowy, uczeń stawia głównie na eksplorację. Zbiera informacje odnośnie wielu stanów dowiadując się czy są one wartościowe. Niestety przy takiej taktyce cała nabyta wiedza nie zostaje wykorzystana w trakcie wyboru akcji. Będąc blisko dużego źródła nagród uczeń jego nie zauważy i zamiast eksploatować, będzie błądził w jego pobliżu.

Wybór zachłanny

W tym przypadku uczeń może bardzo łatwo znaleźć lokalne maksimum. Niestety wybierając akcje zachłannie i poprzez to, że nie eksplorował, nie dowie się o maksimum globalnym.

Wybór ε-zachłanny

Jest to strategia, która polega na tym, że z pewnym prawdopodobieństwem ε wybierana jest losowa akcja, a w pozostałych przypadkach wybierana jest akcja najlepsza [21].

$$\pi(s) = \begin{cases} argmax_a Q(s,a) & \text{gdy } P = q - \varepsilon \\ \text{losowa akcja} & \text{gdy } P = \varepsilon \end{cases} \tag{3.12}$$

3.3.4. Reprezentacja funkcji Q

Podstawowa reprezentacja funkcji Q jest to reprezentacja tablicowa. Zakłada ona, że dla każdego stanu s i akcji a wartość $Q(s,a)$ jest przechowywana w oddzielnej komórce tablicy. W tym wypadku aktualizacja wartości Q jest wykonywana następująco [21]:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_a Q(s',a)\right) \tag{3.13}$$

Warto zauważyć, że gdy przestrzeń stanów ma duży rozmiar, tablicowa reprezentacja stanów może być niemożliwa, ze względu na duże zapotrzebowanie pamięci. Jeżeli nawet dysponujemy dużym zapasem pamięci, uczenie się może być powolne, ponieważ aktualizacja każdego ze stanów może wymagać wielu kroków. Aby rozwiązać ten problem stosowane jest połączenie

algorytmów uczenia się ze wzmocnieniem z metodami regresji [21]. Przy takim podejściu możliwe jest przewidywanie wartości stanów, które nie zostały jeszcze odwiedzone.

4. Uczenie się strategii w środowisku Robocode

4.1. Środowisko Robocode

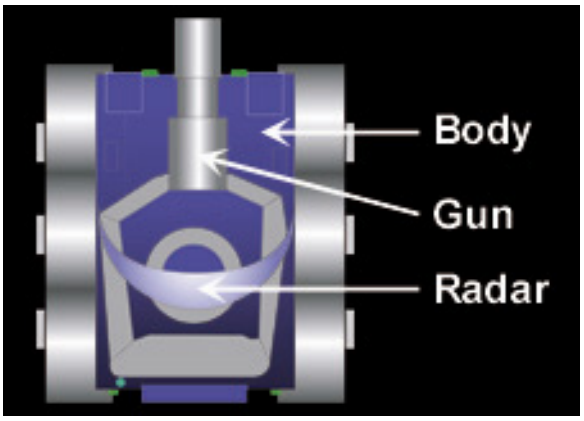
4.1.1. Budowa robota

Czołgi w grze Robocode są skonstruowane z trzech głównych części:

Nadwozie - umożliwia poruszanie się robota (ruch do przodu, do tyłu, skręcanie w lewo i w prawo). Utrzymuje również działo i radar.

Działo - po wcześniejszym wycelowaniu (skręt w lewo/prawo) używane do wystrzeliwania pocisków w kierunku wroga.

Radar - jest to jedna z niezbędnych części robota. Bez niego strzelanie do przeciwnika byłoby niemożliwe oraz ruch czołgu byłby losowy. Są to tak zwane óczyżobota. Radar umożliwia namierzanie i zbieranie informacji na temat przeciwników.



Rysunek 4.1. Budowa czołgu.

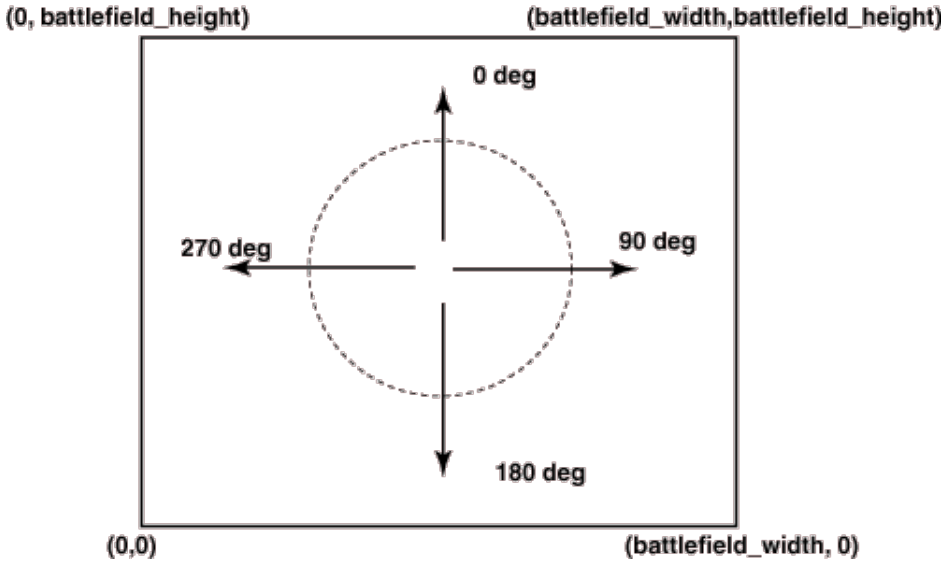
4.1.2. Fizyka gry

Układ współrzędnych

Robocode wykorzystuje kartezjański układ współrzędnych. Punkt (0,0) jest umieszczony w lewym dolnym rogu pola bitwy.

Kąty

Wielkość kąta rośnie zgodnie ze wskazówkami zegara. Północ jest wyznaczona przez kąt 0° (lub 360°), wschód to 90°, południe to 180° a zachód to 270°.



Rysunek 4.2. Układ współrzędnych i rozmieszczenie kątów w środowisku Robocode.

Czas

Czas w grze Robocode jest mierzony przy użyciu tyknięć (ang. tick). Jedna tura oznacza jedno tyknięcie.

Odległości

Jednostką odległości jest piksel. Warto zauważyć, że odległości są reprezentowane przez liczby zmiennoprzecinkowe (double). Dzięki temu jest mozliwy ruch robota o pewien ułamek piksela.

Przyspieszenie robota

Wartość przyspieszenia a robota wynosi 1 piksel/tura^2. Robocode we własnym zakresie dobiera przyspieszenie na podstawie dystansu jaki czołg ma pokonać.

Prędkość robota

Prędkość robota $v = at$ nie może przekroczyć 8 pikseli/turę.

4.1.3. Rotacja nadwoziem, działem i radarem

Maksymalna rotacja nadwozia	$(10-0.75*abs(\text{prędkość}))$ stopni/turę. Im szybciej robot się porusza, tym wolniej się obraca.
Maksymalna rotacja działa	20 stopni/turę
Maksymalna rotacja radaru	45 stopni/turę

4.1.4. Pociski

Obrażenia	4 * siła pocisku. Jeżeli siła ognia jest większa niż 1, to generowane są dodatkowe obrażenia = 2 * (siła pocisku - 1).
Prędkość	20 - 3 * siła ognia
Rozgrzanie działa	1 + siła pocisku/5 Jeśli siła pocisku jest większa niż 0 to robot nie może strzelać. Wszystkie działa na początku każdej rundy są rozgrzane. Z biegiem czasu działo jest chłodzone.
Zwracana energia podczas trafienia	3 * siła pocisku

Kolizje

Kolizja z innym robotem	Każdy robot otrzymuje 0.6 obrażeń
Kolizja ze ścianą	abs(prędkość) * 0.5 - 1

4.1.5. Główna pętla gry

Kolejność głównych zdarzeń w Robocode jest następująca:

1. Widok gry jest odświeżony.
2. Wszystkie roboty wykonują swój kod aż do podjęcia działań i następnie pauzują.
3. Czas jest uaktualniony (czas = czas + 1).
4. Wykonany jest ruch wszystkich pocisków i sprawdzenie kolizji.
5. Wszystkie roboty wykonują swój ruch (poruszanie działem, radarem itp)
6. Roboty wykonują namierzanie radarem.
7. Wszystkie roboty są wznowione do podjęcia nowych akcji.
8. Każdy robot przetwarza swoją kolejkę zdarzeń.

4.1.6. Zdobywanie punktów

Po każdej zakończonej bitwie pokazywane są wyniki prezentujące ogólną punktację (rysunek 4.3)

Results for 10 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	sample.SpinBot	2259 (46%)	800	160	1011	139	110	38	8	0	2
2nd	sample.RamFi...	1410 (29%)	250	0	843	0	235	82	0	5	5
3rd	sample.Fire	1254 (25%)	450	40	705	58	1	0	2	5	3

Save OK

Rysunek 4.3. Wyniki po ukończonej bitwie.

- **Total Score** - Ogólny wynik bitwy. Jest to suma wszystkich punktów w każdej kategorii.
- **Survival Score** - Punkty za przetrwanie. Podczas gdy pewien robot ginie na polu bitwy, każdy czołg który utrzymuje się przy życiu otrzymuje 50 punktów.
- **Last Survivor Bonus** - Bonus za przetrwanie. Ostatni robot, który utrzymał się w grze dostaje 10 punktów za każdego innego robota, który wcześniej zginął.
- **Bullet Damage** - Obrażenia od pocisków. Robot otrzymuje 1 punk za każdy punkt obrażeń, jaki zada przeciwnikowi.
- **Bullet Damage Bonus** - Bonus za obrażenia od pocisków. Gdy przeciwnik zostanie zniszczony, robot otrzymuje dodatkowe 20% wszystkich zadanych obrażeń danemu przeciwnikowi.
- **Ram Damage** - Punkty za taranowanie. Za każdy punkt obrażeń zadany przeciwnikowi poprzez taranowanie, robot otrzymuje 2 punkty.
- **Ram Damage Bonus** - Bonus za taranowanie. Gdy taranując przeciwnika zostanie on zniszczony, robot otrzymuje dodatkowe 30% wszystkich zadanych obrażeń danemu przeciwnikowi.

- **1sts, 2nds, 3rds** - Ranking miejsc. Przedstawia kto ile razy był na którym miejscu.

4.2. Zastosowanie algorytmu Q-learning

4.2.1. Uzasadnienie wyboru algorytmu

Zastosowanie algorytmu Q-learning jest sensowne gdy spełnione są poniższe postulaty:

1. Nietrywialne zadanie - zdroworozsądkowe proste strategie nie dają zadowalającej jakości działania.
2. Opóźnione nagrody - sukces bądź porażka nie zależą tylko od akcji bezpośrednio je poprzedzających, lecz także wcześniejszych.
3. Stany zawierają informacje wystarczające do podejmowania optymalnych decyzji.

Wszystkie z powyższych postulatów są spełnione. Implementacja konkurencyjnej logiki robota jest zadaniem niełatwym. Zdroworozsądkowe strategie nie dają zadowalających rezultatów. Łatwo jest napisać taką logikę robota, która jest w stanie pokonać jeden konkretny czołg. Trudniejsze jest natomiast stworzenie jednej ogólnej sztucznej inteligencji, która miałaby możliwość konkurowania z robotami o różnych strategiach. Opóźnione nagrody występują w środowisku. Robot otrzymuje nagrodę bądź karę, gdy uderzy w przeciwnika. Wcześniej, żeby do niego dojechać, musi wykonać odpowiedni ciąg akcji obrotów i podejść. Do podjęcia decyzji jest dostępna duża ilość informacji, m.in gdy radar namierzy przeciwnika, można poznać jego pozycję, ile ma energii etc.

4.2.2. Przestrzeń stanów

Odpowiednie zdefiniowanie przestrzeni stanów jest kluczowe dla algorytmów uczenia się ze wzmocnieniem. W zależności od liczby stanów zależy szybkość z jaką zostanie osiągnięty planowany rezultat. Rozważając różne warianty reprezentacji stanów, chciałem znaleźć takie rozwiązanie, które by umożliwiało swobodne przenoszenie zdobytych umiejętności do innych instancji tego samego zadania. Pozwoliłoby to chociażby na oddzielną naukę z różnymi robotami.

W projekcie została wykorzystana następująca reprezentacja stanów:

- Energia robota uczącego się = {[0, 10], (10, 100]}
  Maksymalna energia robota wynosi 100. Energia została podzielona na

dwa przedziały, ze względu na to, że czołg powinien inaczej się zachowywać mając dużo punktów życia, np. bardziej agresywnie. Gdy robot jest bliski przegranej, powinien zmienić strategię na ostrożniejszą.

- Energia ostatnio widzianego przeciwnika = {[0, 10], (10, 100]}
  Energia przeciwnika też została podzielona na dwa przedziały. W tym przypadku argumentacją jest następująca możliwa strategia. Gdy przeciwnik ma dużo energii to nasz robot stara się go omijać. Czeka, aż energia przeciwnika odpowiednio zmaleje np. poprzez strzelanie lub zderzenia ze ścianą (czołg gdy strzela, traci pewną część swojej energii). W momencie kiedy wróg jest łatwym celem, uczący się robot może go ostatecznie staranować.
- Odległość od przeciwnika = {[0, 100], (100, 500], (500, $+\infty$)}
  Jest to pomocna informacja, chociażby w strategii omijania pocisków przeciwnika. Im bliżej robot jest wroga, tym łatwiejszym jest celem.
- Strona, po której znajduje się przeciwnik = {przód, tył, lewo, prawo}
  W celu dojechania do przeciwnika, np aby go staranować, robot musi wiedzieć, w którą stronę warto się poruszać.
- Odległość od ściany = {[0, 50], (50, ∞]}
  Zderzając się ze ścianą, robot traci pewną ilość punktów energii. Uwzględniając te dwa atrybuty, robot może nauczyć się unikania zderzeń ze ścianą.

Przestrzeń stanów została skonstruowana tak, aby jej rozpiętość była możliwie mała. Im jest ona mniejsza, tym robot szybciej się uczy, a co za tym idzie, obliczenia trwają o wiele krócej. Aby obliczyć rozdzielczość przestrzeni stanów, wszystkie możliwe stany należy przemnożyć ze sobą.

$$2 * 2 * 3 * 4 * 2 = 96$$

4.2.3. Akcje

Akcje, jakimi dysponuje uczący się robot zostały ograniczone do minimum. Są to:

- Ruch do przodu o 25 pikseli
- Ruch do tyłu o 25 pikseli
- Skręt w lewo o 30°
- Skręt w prawo o 30°

Sytuacje, gdy robot chce skręcić o większy kąt niż 30° lub poruszyć się dalej niż 25 pikseli większej liczby kroków.

Liczba par stan-akcja wynosi:

$$96 * 4 = 384 \text{ par}$$

Oznacza to, że tabela Q będzie zawierała 384 elementy, gdzie każdy z nich powinien być równie często uaktualniany zgodnie ze wzorem 3.13.

4.2.4. Nagrody

Celem ucznia jest nauczenie się takiej strategii, która będzie maksymalizować nagrody w każdym stanie. Jeden ze sposób, aby manipulować strategiami to odpowiednie zdefiniowanie nagród. W mojej pracy inżynierskiej skupiłem się na nauce dwóch różnych strategii.

Strategia agresywna

W grze Robocode punkty zdobywa się m.in uderzając przeciwnika. Poprzez odpowiednie wybranie nagród, chciałem zbadać czy jest możliwa nauka taranowania. W tym celu uczący się czołg dostawał nagrodę za każde uderzenie przeciwnika.

Strategia unikania przeciwników

Na tym przykładzie sprawdziłem, czy jest możliwe nauczenie się unikania przeciwników. Pozwoliłoby to na dłuższe utrzymywanie się na polu bitwy. W przypadku gdy w grze byłoby paru przeciwników, umożliwiałoby to na trzymanie się z dala od niebezpiecznych miejsc na planszy i być może, nawet na przeczekanie, aż przeciwnicy wyeliminują się nawzajem. Aby została osiągnięta taka strategia, uczeń otrzymuje nagrodę za każdą przetrwaną rundę. Robot jest karany gdy zbliży się na pewną odległość do przeciwników bądź gdy zostanie trafiony pociskiem.

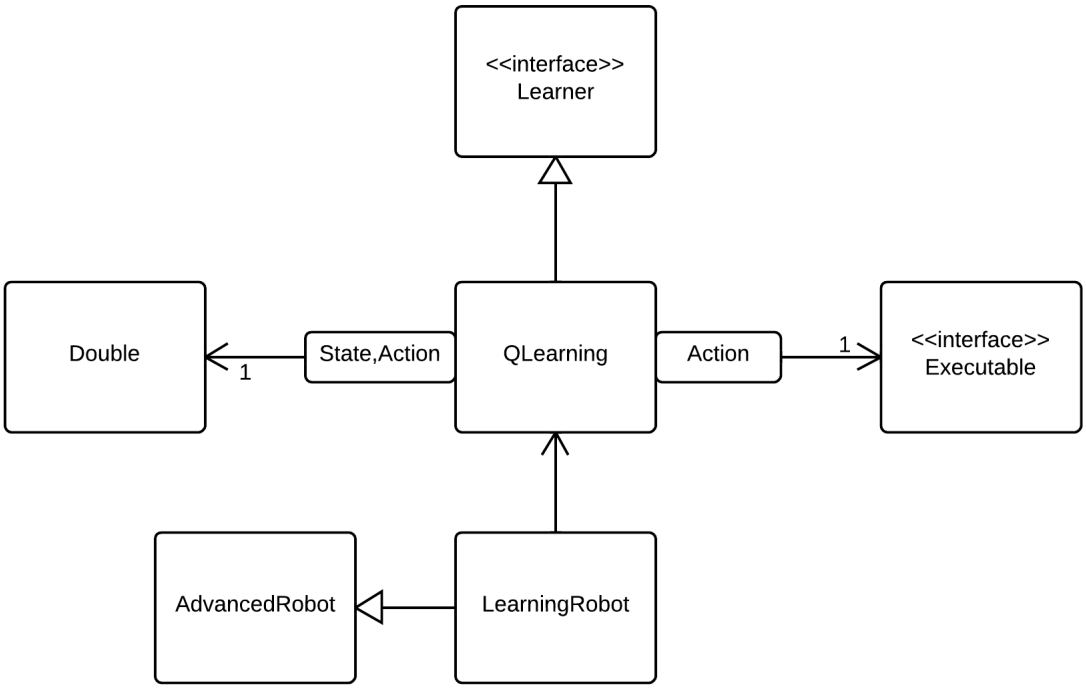
4.3. Implementacja

4.3.1. Architektura

Architekturę programu można przedstawić za pomocą poniższych diagramów (rysunek 4.4 i 4.5).

LearningRobot

Robocode wykorzystuje metodologię programowania sterowanego zdarzeniami. Jest to programowanie, w którym przebieg programu zdefiniowany jest poprzez zdarzenia. Szerokim zastosowaniem tej metodologi są aplikacje graficzne. W Robocode reagowanie na różne sytuacje jest poprzez implementację funkcji z między innymi interfejsu IBasicEvents, który określa podstawowe zdarzenia. Istnieje też możliwość dodawania własnych zdarzeń. Pozowoliło mi to na przykład na dodanie kodu zliczającego tury/tyknięcia (ang. tick).



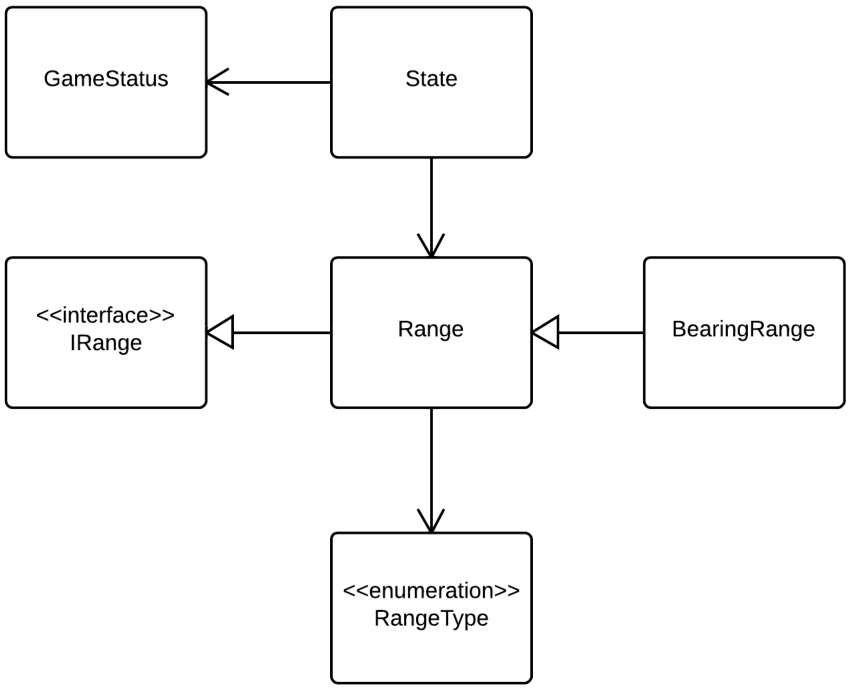
Rysunek 4.4. Architektura systemu (część 1).

```
1  public interface IBasicEvents {
       void onStatus(StatusEvent event);
       void onBulletHit(BulletHitEvent event);
       void onBulletHitBullet(BulletHitBulletEvent event);
5      void onBulletMissed(BulletMissedEvent event);
       void onDeath(DeathEvent event);
       void onHitByBullet(HitByBulletEvent event);
       void onHitRobot(HitRobotEvent event);
       void onHitWall(HitWallEvent event);
10     void onScannedRobot(ScannedRobotEvent event);
       void onRobotDeath(RobotDeathEvent event);
       void onWin(WinEvent event);
   }
```

Wydruk 4.1. Interfejs IBasicEvents.

Klasa LearningRobot (rysunek 4.6) jest najważniejszą klasą w programie. Opisuje ona zachowanie robota, które jest określane poprzez nadpisanie niektórych metod klasy AdvanceRobot. Reakcja na zdarzenia polega na przesłonięciu funkcji onXYZ(), gdzie XYZ to nazwa zdarzenia. W funkcji run() opisywane jest cykliczne zachowanie robota (na przykład ruch). Przykładowy kod prostego robota jest pokazany na listingu 4.2.

```
1  public class SimpleRobot extends AdvancedRobot {

       public void run() {
           while (true) {
5            // implementacja zachowania robota
```



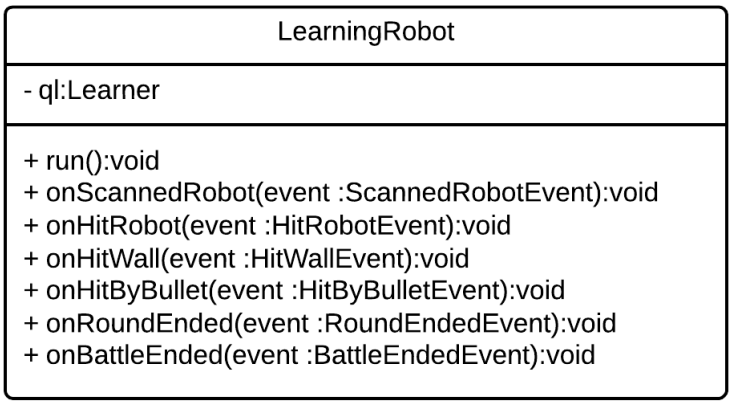
Rysunek 4.5. Architektura systemu (część 2).

```
            ahead(40);
            turnLeft(90);
            (...)
        }
    }

    public void onHitByBullet(HitByBulletEvent e) {
        // reakcja na bycie trafionym przez pocisk
        (...)
    }

    public void onHitRobot(HitRobotEvent e) {
        // reakcja na zderzenie z przeciwnikiem
        (...)
    }
};
```

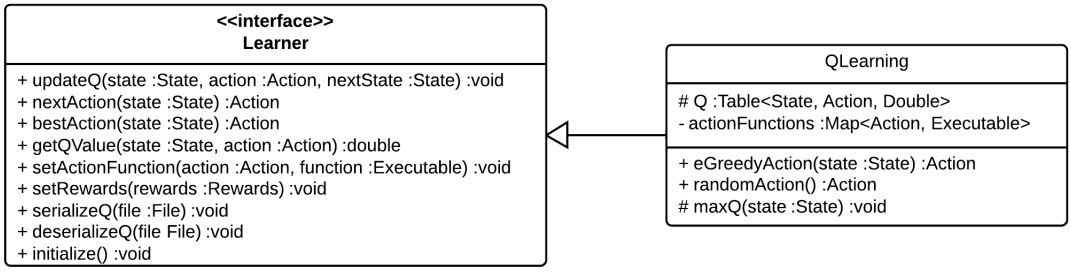
Wydruk 4.2. Prosty przykład robota.

QLearning

Klasa QLearning (rysunek 4.7) jest odpowiedzialna za całą logikę algorytmu uczenia się ze wzmocnieniem. Przechowuje ona tabelę Q posiadającą wartość dla każdej pary stan-akcja. LearningRobot wykorzystuje QLearning poprzez interfejs Learner. Robot pobiera akcję jaką ma wykonać (nextAction() lub bestAction()) i po jej wykonaniu uaktualnia stan tabeli Q



Rysunek 4.6. Klasa LearningRobot.



Rysunek 4.7. Klasa QLearning.

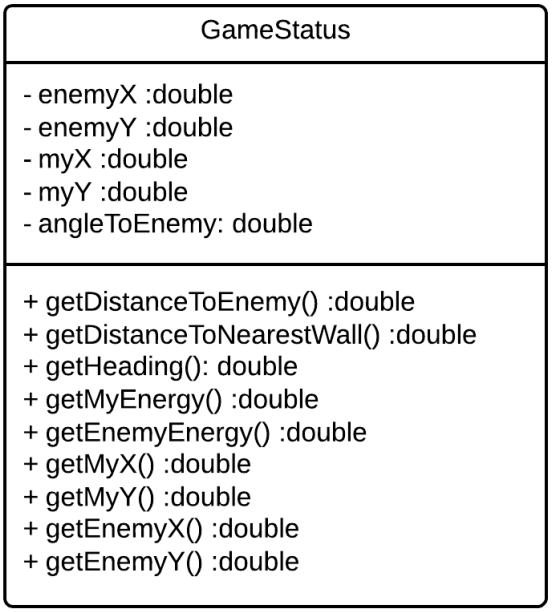
(updateQ()). Interfejs Learner udostępnia również metody pomocne w serializacji tabeli Q. Jeżeli chodzi o metodę setActionFunction() jest to interesujące rozwiązanie. Dzięki tej funkcji LearningRobot przesyła ciało funkcji akcji(wyrażenie lambda), która ma zostać wykonana w przyszłości (np. ruch do przodu). Wewnątrz klasy QLearning zaszyta jest logika wyboru akcji oraz uaktualniania Q. Dzięki interfejsowi Learner późniejsze rozwijanie kodu będzie uproszczone. Dodanie alternatywnego rozwiązania (np. algorytm Q-learning wykorzystujący aproksymator funkcji) będzie polegało tylko na stworzeniu klasy implementującej Learner i podpięciu jej do odpowiedniego ziarna Spring zdefiniowanego w beans.xml:

```
(...)
    <bean id="learner" class="io.github.krris.qlearning.QLearning">
            <property name="rewards" ref="rewards" />
    </bean>
(...)
```

Wydruk 4.3. Scheduler

GameStatus

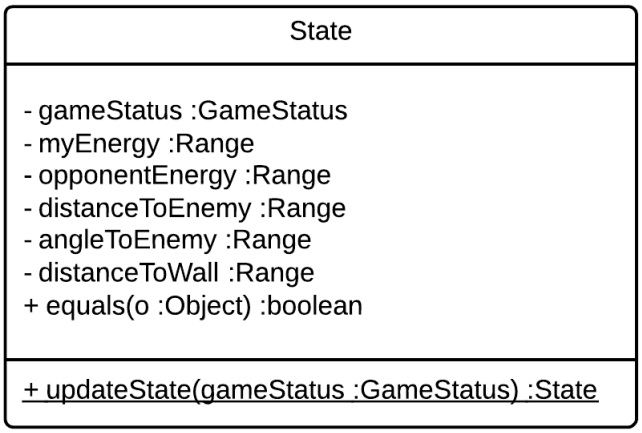
GameStatus (rysunek 4.8) przechowuje informacje odnośnie aktualnego stanu gry, takie jak pozycja robota uczącego się lub przeciwnika, odległość



Rysunek 4.8. Klasa GameStatus.

do najbliższej ściany itp. Wiele metod tej klasy jest oddelegowane do obiektu RobotStatus dostarczonego wraz z implementacją gry.

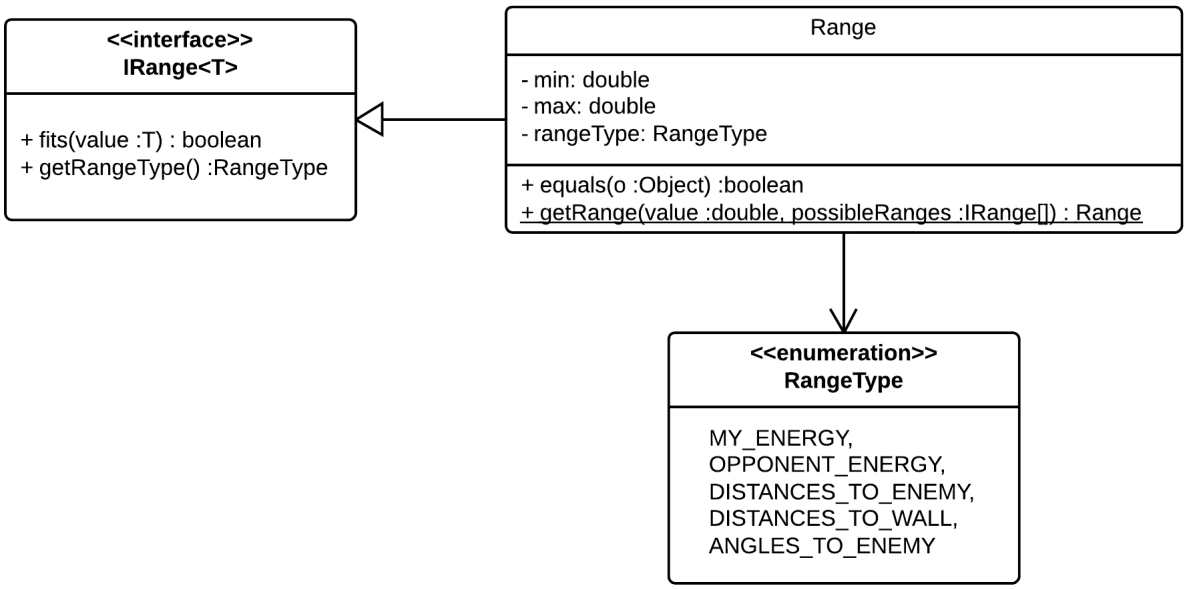
State



Rysunek 4.9. Klasa State.

Klasa state (rysunek 4.9) przechowuje stan środowiska. Przetrzymuje aktualne informacje o stanie gry w danym momencie. Stany są rozróżniane ze względu na odległość od przeciwnika, odległość od ściany, energii robota uczącego się oraz przeciwnika i kąt pod jakim jest wrogi robot względem ucznia.

Range



Rysunek 4.10. Klasa Range.

Stan środowiska jest opisany przez parametry, na przykład odległość od przeciwnika. Taki parametr jest dyskretną wartością opisaną przez pewien zakres, który jest sprecyzowany przez klasę Range (rysunek 4.10).

4.3.2. Konfiguracja

Do konfiguracji gry i algorytmu Q-learning wykorzystywany jest plik application.conf. Możliwe jest w nim dokładne określenie parametrów algorytmu i ustawień gry. Dokładny opis głównych parametrów konfiguracyjnych jest opisany w tabeli 4.1.

```
# Parameters for qlearning algorithm
alpha:          0.2
gamma:          0.8
epsilon:        0.05

# Rewards
hitAWall:                       0
collisionWithEnemy:             10
collisionAndKillEnemy:          100
livingReward:                   0
hitByBullet:                    0
distanceToEnemyLessThan50:      5
distanceToEnemyLessThan200:     2

learningRounds:         500
optimalPolicyRounds:    0

# Game configuration
battlefieldWidth:       600
```

Parametr	**Opis**
alpha	współczynnik uczenia się
gamma	współczynnik dyskontowania
epsilon	współczynnik określający prawdopodobieństwo wyboru losowej akcji
hitAWall	nagroda za uderzenie w ścianę
collisionWithEnemy	nagroda za uderzenie w przeciwnika
collisionAndKillEnemy	nagroda za uderzenie, które niszczy przeciwnika
livingReward	nagroda za przeżycie jednej tury
hitByBullet	nagroda za bycie postrzelonym
distanceToEnemyLessThan50	nagroda za bycie w odleglości od przeciwnika < 50 pikseli
distanceToEnemyLessThan200	nagroda za bycie w odleglości od przeciwnika pomiędzy (200, 50) pikseli
learningRounds	liczba rund nauki
optimalPolicyRounds	liczba rund w których robot będzie podążał za optymalną strategią
battlefieldWidth	szerokość planszy
battlefieldHeight	wysokość planszy
selectedRobots	roboty, które biorą udział w grze
initialPositions	początkowa pozycja robotów

Tablica 4.1. Opis parametrów wykorzystywany w pliku konfiguracyjnym.

```
20 battlefieldHeight:      600

   selectedRobots:         "LearningRobot*,sample.SpinBot"
   initialPositions:       "(425, 425, 0), (25, 50, 0)"
```

Wydruk 4.4. Scheduler

5. Eksperymenty

W tym rozdziale przedstawiłem wyniki jakie otrzymałem w czasie uczenia się przez robota kilku strategii. Warto odnotować, że nacisk był położony na poruszanie się w danym środowisku. Czołg nie uczy się jak strzelać do przeciwników i jak operować radarem. Logika oddawania strzałów nie jest zaimplementowana w programie. Radar natomiast stale się obraca dookoła swojej osi, szukając przeciwników. Warte odnotowania jest również to, że podczas każdej tury czołgi nie startują z tej samem pozycji.

5.1. Procedura eksperymentalna

Postęp uczenia się był sprawdzany w dwóch przypadkach:

1. Konkurując z wymagającym robotem dla danej strategii.
2. Konkurując z robotami dostarczonymi wraz z grą Robocode. Następnie wyniki były porównywane do wyników uzyskanych przez robota, który posiada odgórnie zaimplementowaną zdroworozsądkową sztuczną inteligencję.

Robot miał do dyspozycji 500 rund/walk uczenia się. Walka kończy się, kiedy na polu bitwy zostaje jeden czołg. Każda walka może zawierać różną liczbę tur (wszystko zależy jak długo jest w stanie przetrwać dany bot).

W pierwszym rozważanym przypadku po 50, 100, 300, oraz 500 rundach walki, uczenie się było zatrzymywane i była testowana aktualna strategia optymalna. Strategia optymalna była testowana poprzez przeprowadzenie 100 bitew. Na samym końcu wyniki zostały uśrednione z 5 niezależnych uruchomień.

W drugim przypadku robot również miał do dyspozycji 500 rund. Natomiast jakość działania była sprawdzana poprzez przeprowadzenie 100 bitew korzystając z uzyskanej optymalnej strategii. Następnie wyniki zostały porównane z robotem, który posiadał zdroworozsądkową logikę i który również przeprowadził 100 bitew z czołgiem dostarczonym wraz z Robocode. Tutaj również wyniki zostały uśrednione z 5 niezależnych uruchomień.

5.2. Strategia agresywna

5.2.1. Wyniki eksperymentu

Eksperyment polegał na sprawdzeniu czy robot jest w stanie nauczyć się agresywnego zachowania na polu bitwy. W tym celu funkcja nagrody została zdefiniowana następująco:

- kolizja z przeciwnikiem = 5
- kolizja z przeciwnikiem powodująca jego śmierć = 100
- zderzenie ze ścianą = -1
- nagroda za przeżycie tury = -1
- śmierć = -5
- pozostałe przypadki = 0

Parametry algorytmu Q-learning:

- α = 0.4
- γ = 0.8
- ϵ = 0.05

Współczynniki zostały dobrane podczas przeprowadzania eksperymentów na takiej podstawie, aby uzyskać możliwie najlepsze wyniki. Współczynnik α oznacza tempo uczenia się. Wartość współczynnika dyskontowania γ jest równa 0.8, ponieważ robot ma zwracać uwagę na długoterminowe nagrody. Wartość ϵ zapewnia balans pomiędzy eksploracją, a eksploatacją, zmuszając robota w większym stopniu do eksploatacji.

Konkurencja z robotem Crazy

W tym eksperymencie przeciwnikiem robota był czołg Crazy. Został on wybrany ze względu na to, że jest to wymagający przeciwnik dla czołgu taranującego. Robot Crazy, którego implementacja jest dostarczona wraz z grą, charakteryzuje się szybkim ruchem i dużą ilością skrętów. Jego poruszanie się jest ciężkie do przewidzenia. Jest on też wymagającym przeciwnikiem ze względu na jego szybkość. Tabela 5.1 przedstawia wyniki uzyskane podczas podążania za optymalną strategią po pewnej ilości rund nauki.

Porównania

W tym przypadku sprawdzona była jakość działa z czterema robotami dostarczonymi wraz z grą Robocode w porównaniu z robotem zdroworozsądkowym RamFire. Jest to czołg taranujący, niestrzelający. Tabele 5.2 i 5.3 prezentują wyniki walk.

Robot	Total	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
Po 50 rundach										
Crazy	12911 (69%)	3250	650	5158	561	2798	494	66	34	0
LearningRobot	5746 (31%)	1700	340	0	0	3666	40	35	65	0
Po 100 rundach										
Crazy	13298 (71%)	3250	650	5474	639	2767	517	65	35	0
LearningRobot	5455 (29%)	1750	350	0	0	3320	35	35	65	0
Po 300 rundach										
Crazy	11011 (73%)	2450	490	6395	704	898	75	50	50	0
LearningRobot	4133 (27%)	2500	500	0	0	1126	8	51	49	0
Po 500 rundach										
Crazy	12689 (69%)	2600	520	5191	446	3397	535	53	47	0
LearningRobot	5757 (31%)	2350	470	0	0	2880	57	48	52	0

Tablica 5.1. Wyniki walki robota uczącego się z robotem Crazy uzyskane podążając za optymalną strategią uzyskaną po danej liczbie rund nauki.

Robot	Total	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
Walka z Corners										
Corners	17598 (97%)	5000	1000	9658	1932	8	0	100	0	0
LearningRobot	618 (3%)	0	0	0	0	618	0	0	100	0
Walka z Crazy										
Crazy	12689 (69%)	2600	520	5191	446	3397	535	53	47	0
LearningRobot	5757 (31%)	2350	470	0	0	2880	57	48	52	0
Walka z SpinBot										
SpinBot	15625 (73%)	5000	1000	6056	1047	1979	543	100	0	0
LearningRobot	5766 (27%)	0	0	0	0	5766	0	0	100	0
Walka z Walls										
Walls	17091 (98%)	4700	940	9539	1863	48	0	94	6	0
LearningRobot	378 (2%)	300	60	0	0	18	0	6	94	0

Tablica 5.2. Wyniki walki robota uczącego się LearningRobot z różnymi czołgami dostarczonymi wraz z grą.

Robot	Total	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
Walka z Corners										
Corners	15807 (82%)	5000	1000	8169	1634	4	0	100	0	0
RamFire	3562 (18%)	0	0	0	0	3562	0	0	100	0
Walka z Crazy										
Crazy	14763 (79%)	3750	750	6177	780	2693	614	75	25	0
RamFire	3872 (21%)	1250	250	0	0	2315	57	25	75	0
Walka z SpinBot										
SpinBot	17198 (82%)	5000	1000	7021	1169	2309	699	100	0	0
RamFire	3659 (18%)	0	0	0	0	3659	0	0	100	0
Walka z Walls										
Walls	16714 (98%)	5000	1000	8894	1783	37	0	100	0	0
RamFire	260 (2%)	0	0	0	0	260	0	0	100	0

Tablica 5.3. Wyniki walki robota RamFire z różnymi czołgami dostarczonymi wraz z grą.

5.2.2. Wnioski

Na podstawie tabeli 5.1 wyraźnie widać efekty nauki. Robot uczący się agresywnej strategii jest w stanie rywalizować z czołgiem Crazy. Początkowo po małej liczbie rund uczenia się wygrywał około 35 walk (na 100 odbytych). Pod koniec nauki jego wyniki oscylowały wokół 50 wygranych. Tak samo całkowita liczba zdobytych punktów uległa poprawie. Obserwując wyniki uczenia się, zauważyłem, że robot jest w stanie zlokalizować wroga na planszy i precyzyjnie w niego uderzyć.

Po porównaniu wyników z 5.2 i 5.3 widać, że robot uczący się uzyskuje o wiele lepsze rezultaty. Jest on w stanie wygrać więcej rund i także uzyskać przy tym więcej punktów. Jedynym przypadkiem, w którym robot uczący się jest gorszy, to walka z Corners. Jest to robot, który chowa się w rogu planszy i cały czas oddaje strzały. Przyczyną gorszych wyników jest to, że robot RamFire jest znacznie szybszy od LearningRobot. Robot uczący pokonuje odcinki 50 pikseli i nie jest w stanie się dobrze rozpędzić, a już musi hamować. RamFire natomiast od początku pokonuje dłuższe dystanse.

5.3. Strategia przetrwania

5.3.1. Wyniki eksperymentu

Eksperyment polegał na sprawdzeniu czy robot jest w stanie nauczyć się agresywnego zachowania na polu bitwy. Parametry algorytmu Q-learning zostały takie same jak w poprzednim doświadczeniu. Kary i nagrody były następujące:

- śmierć = -100
- obrażenia od pocisku = -1
- kolizja z przeciwnikiem = -1
- zderzenie ze ścianą = -1
- nagroda za przeżycie tury = 1
- pozostałe przypadki = 0

Konkurencja z robotem Walls

Ten eksperyment miał na celu sprawdzić czy czołg jest w stanie nauczyć się takiej strategii, która umożliwi mu jak najdłuższe przetrwanie na polu bitwy. W pierwszym przypadku eksperymentu, postęp uczenia się był badany podczas walki z czołgiem Walls. Został on wybrany, ze względu na to, że jest to najtrudniejszy robot dostarczony wraz z grą. Jest to robot charakteryzujący się jazdą przy ścianie. Tabela 5.4 przedstawia wyniki uzyskane podczas podążania za optymalną strategią po pewnej ilości rund nauki.

Liczba rund	Liczba przetrwanych tur robota uczącego się
50	611
100	891
300	737
500	890

Tablica 5.4. Tabela przedstawiająca liczbę tur jaką był w stanie przetrwać robot uczący się w walce z Walls.

Porównania

W tym przypadku sprawdzona była jakość działa z czterema robotami dostarczonymi wraz z grą Robocode w porównaniu z robotem zdroworozsądkowym CrazyTicks. Wybrałem ten czołg ze względu na to, że jest trudnym celem (szybki ruch, częste skręty, nie strzela). Aby ocenić jakość działania algorytmu wykorzystałem pomiar tur (tyknięć). Im dłużej robot jest w stanie się utrzymać, tym więcej tur przetrwa. Tabela 5.5 prezentuje uzyskane wyniki.

Walka z przeciwnikiem	Liczba przetrwanych tur robota uczącego się	Liczba przetrwanych tur robota CrazyTicks
Corners	517	2929
SpinBot	425	1426
CrazyTicks	2362	2053
Walls	890	1869

Tablica 5.5. Tabela przedstawiająca liczbę tur jaką był w stanie przetrwać robot uczący się i robot CrazyTicks.

5.3.2. Wnioski

Na podstawie uzyskanych wyników widać od razu, że robot CrazyTicks uzyskuje o wiele lepsze wyniki. Czołg, który uczy się, nie jest w stanie dorównać robotowi, który porusza się losowo. Spowodowane jest to szybkością ruchów. LearningRobot porusza się za wolno i nie jest się w stanie odpowiednio rozpędzić (ze względu na małe dystansy jakie pokonuje jednorazowo, czyli 50 pikseli). CrazyTicks natomiast porusza się praktycznie cały czas z maksymalną prędkością, dlatego też jest go trudno trafić i dzięki temu jest w stanie przetrwać o wiele więcej tur na polu bitwy.

Zaobserwowałem, że LearningRobot bardzo często stara się oddalić od przeciwnika i gdy jest już przy ścianie, zawsze zostaje zablokowany i nie ma możliwości ucieczki. Oglądając przebieg walki, udało mi się zauważyć ciekawą strategię podczas walki z robotem SpinBot (jest to przeciwnik jeżdżący po małym okręgu i oddający strzały). Robot nie utrzymywał dużej odległości od przeciwnika. Bardziej opłacalne było dla niego pozostanie na krótkim

dystansie i jeżdżenie cały czas po prostej linii, raz do przodu, raz do tyłu. Dzięki temu był w stanie uniknąć większości kul.

5.4. Transfer strategii do innych środowisk

5.4.1. Wyniki eksperymentu

Ten eksperyment miał na celu sprawdzenie czy jest możliwy transfer umiejętności do innych instancji tego samego zadania. Badania zostały przeprowadzone na robocie uczącym się strategii agresywnej. Na początku uczył się podczas bitwy wraz z robotem Crazy. Miał do dyspozycji 500 rund. Następnie korzystając z posiadanej wiedzy brał udział w walkach z kilkoma innymi czołgami. Wyniki zaprezentowane są w tabeli 5.6.

Robot	Total	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
Walka z Corners										
Corners	16873 (92%)	5000	1000	9052	1811	10	0	100	0	0
LearningRobot	1415 (8%)	0	0	0	0	1415	0	0	100	0
Walka z SpinBot										
SpinBot	16185 (76%)	5000	1000	6110	1178	2462	435	100	0	0
LearningRobot	5234 (24%)	0	0	0	0	5234	0	0	100	0
Walka z Walls										
Walls	12813 (82%)	3100	620	7756	1143	161	33	62	38	0
LearningRobot	2756 (18%)	1900	380	0	0	467	9	38	62	0

Tablica 5.6. Wyniki dla przenoszenia umiejętności do innych instancji gry. Robot uczył się z czołgiem Crazy, a wiedzę wykorzystał z innymi robotami dostarczonymi wraz z grą.

5.4.2. Wnioski

Wyniki uzyskane w przeprowadzonym eksperymencie są bardzo obiecujące. Robot jest w stanie uczyć się walki z jednym czołgiem, a następnie wykorzystać swoją wiedzę podczas bitew z innymi robotami. Ciekawe jest to, że robot uczący się w ten sposób uzyskał lepsze wyniki niż robot w pierwszym eksperymencie 5.2, gdzie uczył się i wykorzystywał wyznaczoną strategię w starciu z tym samym robotem. Przyczyną tego może być, że podczas nauki z czołgiem Crazy, robot uczący się ódwiedził"większą liczbę stanów, ze względu na losowość ruchów Crazy. Dzięki temu odpowiednie wartości dla danych par stan-akcja były częściej uaktualniane co owocowało lepszymi rezultatami.

6. Podsumowanie

Przeprowadzone badania wskazują, że jest możliwe tworzenie sztucznej inteligencji dla gier komputerowych korzystając z algorytmów uczenia się ze wzmocnieniem. Robot był w stanie nauczyć się strategii agresywnej i potrafił wykorzystać swoją wiedzę w działaniu, co owocowało lepszymi wynikami niż robot z odgórnie zaimplementowaną logiką. Na podstawie zaobserwowanych zachowań robota uczącego się strategii ostrożnej, można zauważyć, że niekiedy robot jest w stanie znaleźć takie rozwiązanie problemu, które nie jest intuicyjne dla człowieka, a mimo to, wciąż daje dobre rezultaty. Eksperymenty potwierdziły również, że jest możliwy transfer wiedzy do innych instancji tego samego problemu. Dzięki temu bot może się uczyć tylko z jednym wrogiem, a następnie gdy będzie prowadził walkę z dotąd nieznanym przeciwnikiem, będzie wiedział jakie akcje podjąć.

Tworzenie logiki botów przy pomocy algorytmu Q-learning ma swoje zalety. Dużym plusem jest trudna do przewidzenia strategia jakiej robot się nauczy. Kolejną zaletą jest to, że można swobodnie manipulować poziomem zaawansowania, między innymi poprzez liczbę epizodów nauki. Wadą takiego sposobu konstruowania sztucznej inteligencji jest na pewno czasochłonność doboru odpowiednich nagród, parametrów algorytmu, przestrzeni stanów i konieczność przeprowadzenia wielu eksperymentów.

Biorąc pod uwagę przyszłe prace w tym obszarze, ciekawe byłoby zbadanie, czy jest możliwe nauczenie robota jak operować radarem lub jak oddawać strzały. Strzelanie jest ciężkim zadaniem, ze względu na to, że trzeba brać pod uwagę kilka czynników, takich jak: ruch przeciwnika, przegrzewanie się działa, tracenie energii po oddaniu strzału. Ciekawe również by było sprawdzenie, jak robot korzystający z uczenia ze wzmocnieniem radzi sobie w zawodach RoboRumble [1].

Jeżeli chodzi o samą grę, Robocode świetnie spisuje się jako miejsce badań nad sztuczną inteligencją. Potwierdzają to między innymi prace [20], [14]. Plusem również jest łatwość z jaką można zaimplementować logikę czołgu. Dokumentacja jest bardzo dobrze napisana i API jest zrozumiale wytłumaczone. Istnieje również bardzo strona[2] z prostymi samouczkami i innymi pomocnymi informacjami odnośnie samej gry. Sam kod gry jest cały czas utrzymywany przez społeczność. Kolejne wersje programu są wydawane kilkakrotnie w przeciągu roku. Społeczność, która pracuje nad

[1] http://robowiki.net/wiki/RoboRumble
[2] http://robowiki.net/

programem działa bardzo sprawnie. Pracując nad implementacją pracy inżynierskiej, napotkałem się na pewien problem, który całkowicie blokował pracę nad kodem. Po zgłoszeniu błędu, w przeciągu kilku dni został on naprawiony i kolejna wersja Robocode została wydana.

Warto zwrócić uwagę, że gra Robocode jest polecana również dla osób chcących nauczyć się programować [6]. Robocode jest napisany w języku Java, który posiada prostą składnię. Aby zacząć zabawę wystarczy dosłownie kilka linii kodu i już można obserwować zachowanie swojego robota rywalizującego z innymi na polu bitwy.

A. Wykorzystane technologie

Gra Robocode jest napisana w języku programowania Java. Do jej uruchomienia wymagana jest Java Standard Edition w wersji 6 lub nowszej. Do korzystania z gry potrzebna jest Java Runtime Environment (JRE) lub Java Developer Kit (JDK). Warto zauważyć, że JRE, w przeciwieństwie do JDK nie zawiera kompilatora Javy (javac). JRE jest wystarczające, aby móc w pełni korzystać z gry Robocode, ponieważ gra jest dostarczana z wbudowanym kompilatorem Eclipse Java Compiler.

Środowisko pracy

Bardzo ważnym dla mnie apektem było stworzenie wygodnego środowiska pracy. Mimo to, że Robocode dostarcza własny edytor, zdecydowałem się na użycie innego zintegrowanego środowiska programistycznego, ze względu na łatwiejszą integrację z zewnętrznymi narzędziami. Wybór padł na Intellij IDEA.

Do automatyzacji wykorzystałem następujące narzędzia:

- **Gradle**[1] - jest to narzędzie to automatycznego budowania oprogramowania. Jednym z jego głównych zastosowań w moim projekcie było zarządzanie zależnościami.
- **Typesafe config**[2] - biblioteka ułatwiająca pisanie własnych plików konfiguracyjnych i ich parsowanie.
- **Skrypty powłoki bash** - głównym zastosowaniem było uruchamianie wcześniej skonfigurowanych walk robotów.

Biblioteki wykorzystane do realizacji projektu:

- **Spring**[3] - jest to platforma, która dostarcza rozwiązania do często spotykanych problemów technicznych. Głównym zastosowaniem tego oprogramowania w projekcie było zarządzanie cyklem życia obiektów i wstrzykiwanie zależności (ang. Dependency Injection)

[1] http://www.gradle.org/
[2] https://github.com/typesafehub/config
[3] https://spring.io/

- **JUnit**[4] i **Mockito**[5] - biblioteki zostały wykorzystane do testowania aplikacji oraz tworzenia atrap obiektów (ang. mock objects)
- **Log4j**[6] i **SLF4J**[7] - biblioteki wykorzystane do logowania informacji z przebiegu działania aplikacji.
- **Google Guava**[8] - biblioteka dostarczające pewne użyteczne funkcje i kolekcje, które nie występują w podstawowych bibliotekach Javy.

[4] http://junit.org/
[5] https://code.google.com/p/mockito/
[6] http://logging.apache.org/log4j/2.x/
[7] http://www.slf4j.org/
[8] https://code.google.com/p/guava-libraries/

B. Konfiguracja środowiska

Aby uruchomić program należy postępować zgodnie z poniższymi krokami:

1. Pobranie i instalacja Robocode (wymagana wersja 1.9.2.1 lub wyższa) `http://robocode.sourceforge.net/`.
2. Pobranie i instalacja Gradle `http://www.gradle.org/`.
3. Przejście do głównego katalogu z kodem i wywołanie:
   $ gradle build
   $ gradle copyDeps
4. W pliku *scripts/config.sh* proszę zmienić ścieżkę wskazującą na lokalną instalację Robocode.
5. W pliku *src/main/resources/application.conf* proszę zmienić"
   — *rewardsPath* - ścieżka do pliku, w którym będą zapisywane nagrody uzyskane w każdej rundzie,
   — *chartPath* - ścieżka do generowanego wykresu nagród,
   — *battleconfigPath* - ścieżka do pliku *battles/generate.battle*.
6. Proszę uruchomić grę Robocode i przejść do Options->Preferences->Development Options i dodać ścieżkę do folderu, w którym znajduje się kod robota.
7. Gra jest uruchamiana poprzez skrypt *scripts/run.sh*.

Bibliografia

[1] Classical conditioning. http://en.wikipedia.org/wiki/Classical_conditioning. Dostęp z: 2014-09-08.

[2] Coursera: Machine learning. https://www.coursera.org/course/ml. Dostęp z: 2014-05-13.

[3] Designing artificial intelligence for games. https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1. Dostęp z: 2014-08-28.

[4] Drivatar™ in forza motorsport. http://research.microsoft.com/en-us/projects/drivatar/. Dostęp z: 2014-08-03.

[5] Ibm research: Why games matter to artificial intelligence. http://ibmresearchnews.blogspot.fr/2013/08/why-games-matter-to-artificial.html. Dostęp z: 2014-09-08.

[6] Robocode. http://robocode.sourceforge.net/. Dostęp z: 2014-08-28.

[7] Udacity: Reinforcement learning. https://www.udacity.com/course/ud820. Dostęp z: 2014-08-28.

[8] Udacity: Supervised learning. https://www.udacity.com/course/ud675. Dostęp z: 2014-08-28.

[9] Udacity: Unsupervised learning. https://www.udacity.com/course/ud741. Dostęp z: 2014-08-28.

[10] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[11] Christopher Amato, Guy Shani. High-level reinforcement learning in strategy games. *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, strony 75–82. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[12] Luigi Cardamone, Daniele Loiacono, Pier Luca Lanzi, Alessandro Pietro Bardelli. Searching for the optimal racing line using genetic algorithms. *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, strony 388–394. IEEE, 2010.

[13] Paweł Cichosz. *Systemy uczące się*. Wydawnictwa Naukowo-Techniczne, 2000.

[14] J Eisenstein. Evolving robot tank fighters. Raport instytutowy, Technical Report AIM-2003-023, AI Lab, MIT. 2003. http://www. ai. mit. edu/people-/jacobe/research/robocode/genetic_tanks. pdf, 2003.

[15] Thore Graepel, Ralf Herbrich, Julian Gold. Learning to fight. *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, strony 193–200, 2004.

[16] Stefano Lecchi. Artificial intelligence in racing games. *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009.

[17] Tom M Mitchell. Machine learning. wcb, 1997.

[18] Katharina Mülling, Jens Kober, Oliver Kroemer, Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[19] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, strony 363–372. Springer, 2006.

[20] J Nielsen, B Jensen. Modern ai for games: Robocode. `http://www.jonnielsen.net/RoboReportOfficial.pdf`, 2011. Dostęp z: 2014-08-28.

[21] Richard S Sutton, Andrew G Barto. *Introduction to reinforcement learning*. MIT Press, 1998.

[22] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[23] Julian Togelius, Sergey Karakovskiy, Robin Baumgarten. The 2009 mario ai competition. *Evolutionary Computation (CEC), 2010 IEEE Congress on*, strony 1–8. IEEE, 2010.

[24] JMP Van Waveren, Léon JM Rothkrantz. Artificial player for quake iii arena. *International Journal of Intelligent Games & Simulation*, 1(1), 2002.