Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jakub Sieroń
Nr albumu: 360326

Wybrane heurystyki uczenia ze wzmocnieniem

Praca licencjacka
na kierunku MATEMATYKA

Praca wykonana pod kierunkiem
dr Marcina Szczuki
Instytut Informatyki

Sierpień 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

W pracy przedstawiono zadanie uczenia ze wzmocnieniem na przykładzie dwóch algorytmów - Q-learning i SARSA. Zostały one wykorzystane do rozwiązania problemu wyznaczenia optymalnej strategii w grze w kółko i krzyżyk oraz w problemie Taxi-v1 pochodzącym ze strony OpenAI Gym. Na podstawie wyników obu algorytmów w tych zadaniach dokonano ich analizy porównawczej.

Słowa kluczowe

uczenie ze wzmocnieniem, procesy decyzyjne Markowa, Q-learning, SARSA

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.1 Matematyka

Klasyfikacja tematyczna

68 – Computer Science
68T Artificial Intelligence
68T20 Problem solving (heuristics, search strategies, etc.)

Tytuł pracy w języku angielskim

Selected heuristics for reinforcement learning

Spis treści

Wprowadzenie

Uczenie ze wzmocnieniem jest jedną z odmian uczenia maszynowego. Jest ono metodą pośrednią między uczeniem z nadzorem a uczeniem bez nadzoru. Z jednej strony, w przeciwieństwie do uczenia bez nadzoru, uczeń otrzymuje od środowiska pewne informacje w postaci nagród pozwalające mu oceniać jakość swoich działań. Z drugiej jednak, w przeciwieństwie do uczenia z nadzorem, informacja nie jest pełna ani jednoznaczna. Przyznawane nagrody jedynie sugerują, jaka droga jest słuszna i jak należy postępować.

Powyższy opis jest, rzecz jasna, bardzo ogólny i mało konkretny. By móc skutecznie stosować te idee, niezbędne są ścisłe narzędzia, których dostarczają procesy decyzyjne Markowa. Dzięki nim dowiadujemy się, że aby rozwiązać dany problem, wykorzystując uczenie ze wzmocnieniem, wystarczy poznać wartości pewnej funkcji Q. Od tego momentu będą nas interesować algorytmy, które będą je potrafiły skutecznie wyznaczać, czyli Q-learning i SARSA. Właśnie te algorytmy będziemy analizować na wybranych przykładach.

Praca składa się z pięciu rozdziałów i dodatków. W rozdziale 1 wprowadzimy podstawowe pojęcia umożliwiające skuteczne stosowanie uczenia ze wzmocnieniem w praktycznym rozwiązywaniu problemów. Rodzina algorytmów TD implementujących wprowadzone wcześniej rozwiązania została omówiona w rozdziale 2. Znajduje się tam również opis podstawowych własności i problemów, które wynikają ze stosowania algorytmów z rodziny TD.

W rozdziale 3 zostają szczegółowo zaprezentowane dwa konkretne algorytmy realizujące założenia TD – Q-learning i SARSA. Są one później wykorzystywane w rozwiązaniu dwóch problemów. Pierwszy z nich, czyli problem gry w kółko i krzyżyk, został zaprezentowany w rozdziale 4. Znajduje się tam również opis implementacji oraz analiza działania powyższych algorytmów.

W ostatnim rozdziale zostało przedstawione zadanie Taxi-v1 pochodzące ze strony OpenAI Gym i jego rozwiązanie wykorzystujące algorytmy Q-learning i SARSA. Jest to problem publicznie dostępny, więc może się z nim zmierzyć każdy, dzięki czemu istnieje baza wyników, z którymi można porównać własne rozwiązanie. W rozdziale znajduje się, poza opisem problemu i implementacji, takie porównanie oraz analiza obu użytych algorytmów. W dodatkach umieszczono najistotniejsze fragmenty programów z rozdziałów 4 i 5.

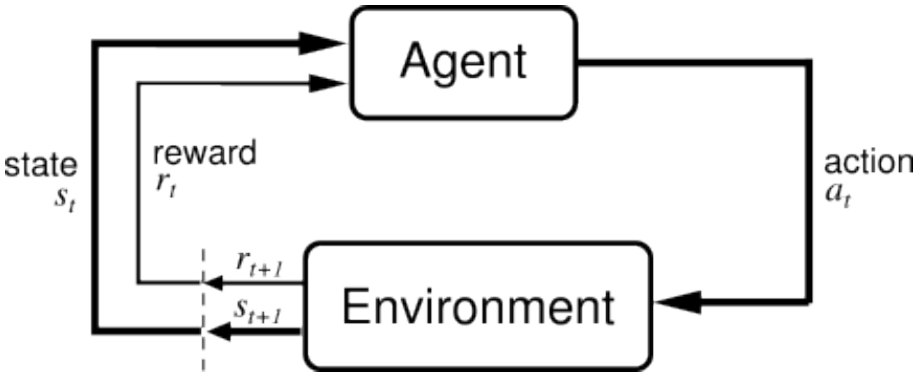
Rozdział 1

Podstawowe pojęcia - zadanie uczenia ze wzmocnieniem, procesy decyzyjne Markowa

Uczenie ze wzmocnieniem jest jedną ze standardowych metod w uczeniu maszynowym. Jego idea jest jednak znana ludzkości od zarania dziejów. Podejście to znajduje się gdzieś na przecięciu uczenia się metodą prób i błędów oraz metody kija i marchewki. Ucząc się ze wzmocnieniem z jednej strony nie znamy w ogóle środowiska, wewnątrz którego się poruszamy – zostanie ono zbadane dopiero, gdy zaczniemy z nim wchodzić w interakcję. Za każde nasze działanie jesteśmy w jakiś sposób nagradzani lub karani i w ten właśnie sposób chcemy uczyć się, jak działa środowisko oraz co powinniśmy robić, aby otrzymywać duże nagrody.

1.1. Uczenie ze wzmocnieniem

W zadaniu uczenia ze wzmocnieniem uczeń (np. program komputerowy) ma rozwiązać postawiony problem, nie znając środowiska, w którym problem jest zadany, jednakże może się z nim w pewien sposób komunikować. Ta komunikacja jest jednak mocno ograniczona - uczeń na podstawie stanu, w jakim obecnie znajduje się środowisko informuje je o podejmowanej przez siebie akcji, w zamian otrzymując nagrodę. Uczeń na podstawie nagród powinien uczyć się podejmowania coraz trafniejszych decyzji.



Rysunek 1.1: Schemat działania ucznia ze środowiskiem

Oczywiście, każdy problem chcielibyśmy rozwiązać w skończonym czasie, jednak ze względu

na brak założeń odnośnie maksymalnego czasu oraz bardzo ogólny schemat załóżmy, że czas jest nieskończony i podzielony na jednostki, z kolei nagrody są liczbami rzeczywistymi. Wprowadźmy w tym miejscu naturalne oznaczenia: s_t – stan, w którym znajduje się środowisko w momencie czasu t, a_t – akcja, którą podejmuje uczeń w momencie czasu t, r_t – nagroda przyznana przez środowisko za akcję a_t. Przy powyższych założeniach możemy powiedzieć, że celem ucznia jest zmaksymalizowanie następującej wielkości:

$$\mathbb{E}\Big[\sum_{t=0}^{\infty}\gamma^t r_t\Big] \tag{1.1}$$

gdzie $\gamma \in [0,1]$ - *współczynnik dyskontowania.* Sensem wprowadzenia parametru γ jest fakt, że zależy nam na szybszym uzyskiwaniu wysokich nagród – gdy $\gamma = 1$ każda nagroda jest tak samo ważna, zaś dla wartości γ bliskich 0 istotne są nagrody wyłącznie za początkowe ruchy, zaś nagrody przyznane „późno" mają niewielkie znaczenie dla wielkości (1.1).

1.2. Procesy decyzyjne Markowa

Matematycznym modelem zadania uczenia się ze wzmocnieniem jest proces decyzyjny Markowa. Wprowadzimy teraz podstawowe definicje i pojęcia z nim związane. Zaznaczmy w tym miejscu, że wszystkie definicje, twierdzenia i dowody pochodzą z [SUS].

Definicja 1.2.1 (Proces Markowa)**.** Proces decyzyjny Markowa *to czwórka* $\langle S, A, \rho, \delta\rangle$*, gdzie:*

- S *- skończony zbiór stanów,*
- A *- skończony zbiór akcji,*
- ρ *- funkcja wzmocnienia,*
- δ *- funkcja przejść stanów.*

Definicja 1.2.2 (Wartość funkcji wzmocnienia)**.** *Dla procesu Markowa* $\langle S, A, \rho, \delta\rangle$ wartością funkcji wzmocnienia $\rho(s,a)$ *dla pary* $\langle s, a\rangle \in S \times A$ *jest rzeczywista zmienna losowa (nagroda po wykonaniu akcji* a *w stanie* s*).*

Definicja 1.2.3 (Wartość funkcji przejścia)**.** *Dla procesu Markowa* $\langle S, A, \rho, \delta\rangle$ wartością funkcji przejścia $\delta(s,a)$ *dla pary* $\langle s, a\rangle \in S \times A$ *jest zmienna losowa o wartościach w* S *(stan po wykonaniu akcji* a *w stanie* s*).*

W tym miejscu zauważmy, że zgodnie z wprowadzonymi wcześniej oznaczeniami $r_t = \rho(s_t, a_t)$ i $a_{t+1} = \delta(s_t, a_t)$. Zauważamy tutaj realizację własności Markowa, tzn. obecny stan, w którym znajduje się środowisko, zależy wyłącznie od stanu poprzedniego i podjętej w nim akcji. Wprowadźmy w tym miejscu dla wygody dodatkowe oznaczenia:

$$R(a,s) = \mathbb{E}[\rho(s,a)]$$
$$P_{s,t}(a) = \mathbb{P}(\delta(s,a) = t)$$

Ważnym z punktu widzenia uczenia ze wzmocnieniem pojęciem jest strategia w procesie decyzyjnym Markowa oraz funkcje przejścia z nią związane.

Definicja 1.2.4 (Strategia)**.** Strategią *dla procesu decyzyjnego Markowa* $\langle S, A, \rho, \delta\rangle$ *nazywamy dowolną funkcję* $\pi : S \to A$*.*

Definicja 1.2.5 (Funkcje wartości dla strategii π)**.** Funkcją wartości stanu *dla strategii* π *nazywamy funkcję:*

$$V^{\pi}(s) = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \big| s_0 = s\Big] \tag{1.2}$$

Funkcją wartości akcji dla *strategii* π *nazywamy funkcję:*

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\Big[\rho(s,a) + \sum_{t=0}^{\infty} \gamma^t r_t \big| s_0 = s, a_0 = a\Big] \tag{1.3}$$

Zapis $\mathbb{E}_{\pi}$ oznacza tutaj warunkową wartość oczekiwaną pod warunkiem (stosowania) strategii π. Sens obu funkcji wartości jest dość jasny - funkcja wartości stanu mówi, jakiej wartości (1.1) spodziewamy się, zaczynając w stanie s, zaś funkcja wartości akcji dodatkowo bierze pod uwagę akcję wykonaną w tym stanie (wówczas pierwsza nagroda jest przyznawana jednoznacznie, stąd $\rho(s,a)$ we wzorze).

Definicja 1.2.6. *Strategia* π' *jest* lepsza *od strategii* π *(zapis* $\pi' \succ \pi$*), jeśli* $V^{\pi'}(s) \geq V^{\pi}(s)$ *dla wszystkich* s *i istnieje* s_0*, taki, że* $V^{\pi'}(s_0) > V^{\pi}(s_0)$*.*

Zauważmy, że powyższą definicję możemy zapisać w równoważnej postaci, kładąc Q w miejsce V.

Definicja 1.2.7 (Strategia optymalna)**.** *Strategia* π *jest* strategią optymalną*, jeśli nie istnieje strategia od niej lepsza.*

Zaznaczmy w tym miejscu, że strategii optymalnych może być więcej niż jedna. Dowolną z nich będziemy oznaczać π^*, zaś odpowiadające jej (optymalne) funkcje wartości stanu i akcji V^* i Q^*. Zauważmy teraz, że uczeń, który będzie się posługiwał strategią optymalną, będzie jednocześnie maksymalizował (1.1) (zgodnie z definicją V i Q). Wobec tego problem uczenia się ze wzmocnieniem możemy sprowadzić do znalezienia optymalnej strategii.

Definicja 1.2.8 (Strategia zachłanna)**.** Strategią zachłanną względem V *jest (dowolna) strategia określona dla każdego stanu* s *wzorem:*

$$\pi(s) = \operatorname*{argmax}_a \big[R(s,a) + \gamma \sum_t P_{s,t}(a) V(t)\big] \tag{1.4}$$

Strategią zachłanną względem Q *jest (dowolna) strategia określona dla każdego stanu* s *wzorem:*

$$\pi(s) = \operatorname*{argmax}_a Q(s,a) \tag{1.5}$$

gdzie $\operatorname*{argmax}_a$ *to argument maksymalizujący daną funkcję.*

Innymi słowy, strategia zachłanna względem V (Q) to taka, która w każdym stanie s „wybiera" taką akcję a, dla której $V(s,a)$ ($Q(s,a)$) jest największe (stąd nazwa zachłanna). Przypomnijmy w tym miejscu, że funkcje V i Q wyrażają wartość oczekiwaną wielkości (1.1) przy danej akcji i stanie, czyli jeśli w każdym stanie będziemy wybierać akcję o maksymalnej wartości oczekiwanej, to ostatecznie otrzymamy maksymalną wartość (1.1). Formalizując tę obserwację, możemy zapisać następujące twierdzenie:

Twierdzenie 1.2.9. *Niech* π *będzie dowolną strategią o funkcji wartości stanu* V^{π} *(akcji* Q^{π}*). Jeśli strategia* π' *jest zachłanna względem* V^{π} *(*Q^{π}*), to* π' *jest lepsza od* π *albo obie strategie są optymalne.*

Wniosek 1.2.10. *Dowolna strategia zachłanna względem optymalnej funkcji wartości stanu lub akcji jest strategią optymalną.*

Dowód. Niech π^* będzie dowolną strategią optymalną. Optymalna funkcja wartości stanu V^* jest funkcją wartości stanu względem strategii π^*. Zatem dowolna strategia π zachłanna względem V^* jest na mocy twierdzenia 1.2.9 albo lepsza od strategii π^*, albo również optymalna. Z definicji strategii optymalnej π nie jest lepsza niż π^*. Wobec tego π jest optymalna. Analogicznie dowodzimy dla funkcji wartości akcji. □

Konsekwencją wniosku 1.2.10 jest spostrzeżenie, że aby wyznaczyć strategię optymalną, wystarczy wyznaczyć optymalną funkcję wartości stanu lub akcji. Wówczas strategia zachłanna dla tej funkcji jest jednocześnie strategią optymalną, pozwala nam to sprowadzić zadanie uczenia ze wzmocnieniem do szukania optymalnej funkcji V lub Q.

Rozdział 2

Metoda różnic czasowych TD

2.1. Uczenie się funkcji wartości

Przejdźmy teraz do zagadnień praktycznych, czyli tego, w jaki sposób program (od teraz on będzie naszym „uczniem") ma nauczyć się znajdować właściwą strategię dla problemu. Częstym podejściem (i jedynym zaprezentowanym w pracy) jest nauka funkcji wartości akcji Q. W ogólności: program, nie znając środowiska, będzie się uczył (na podstawie nagród), jaka akcja w danym stanie środowiska przynosi jaki zysk. Jego celem będzie oczywiście wybór akcji, które maksymalizują całościowy zysk, czyli (1.1).

Przyjmijmy w tym miejscu, że zarówno ilość stanów środowiska, jak i ilość możliwych do wykonania akcji jest skończona (biorąc pod uwagę, że będziemy uczyć programy komputerowe to założenie jest oczywiste). Teraz o funkcji wartości akcji Q możemy myśleć jak o tabelce, której wiersze odpowiadają różnym stanom środowiska, zaś kolumny poszczególnym akcjom. Wówczas na przecięciu danego wiersza odpowiadającemu stanowi środowiska s i danej kolumny odpowiadającej akcji a stoi liczba będąca wartością $Q(s, a)$. Oczywiście dla każdego problemu istnieje optymalna funkcja Q, a więc istnieje również odpowiadająca jej tabelka. Zadaniem programu uczącego się ze wzmocnieniem będzie zatem znalezienie tej tabelki.

Warto na koniec dodać, że w analogiczny sposób moglibyśmy poznawać funkcję wartości stanu. W rozważaniach teoretycznych jest to wygodniejsze, dlatego opisując podstawy algorytmów, będziemy się nią posługiwać. Natomiast w praktycznych zastosowaniach lepiej posługiwać się funkcją Q, gdyż algorytmy na niej oparte są „szybsze". Oczywiście funkcje te są ze sobą blisko związane i przy pomocy jednej łatwo wyznaczyć drugą.

2.2. Algorytm TD

Algorytmem wykorzystywanym do uczenia się funkcji wartości akcji jest algorytm różnic czasowych (ang. *temporal differences*), w skrócie TD. W zasadzie jest to rodzina algorytmów bazująca na wspólnej podstawie, dla wygody będziemy jednak mówić o algorytmie TD. Jego podstawową cechą jest w każdym kroku czasu generowanie prognozy pewnej nieznanej wartości i na jej podstawie uzyskanie dodatkowych informacji, dzięki którym możliwe jest wygenerowanie kolejnej prognozy. Z czasem informacja jest coraz bardziej dokładna i pełna, więc stawiane prognozy będą coraz dokładniejsze. W trakcie uczenia się algorytm modyfikuje swoje predykcje przy użyciu błędów obliczanych jako różnice wartości oczekiwanych w dwóch kolejnych krokach czasu.

Powyższa charakterystyka jest bardzo ogólna, przez co nie do końca użyteczna. Rozważmy teraz algorytm TD jako narzędzie uczenia ze wzmocnieniem. Przy takim podejściu

wielkość, którą chcemy prognozować w momencie t i stanie środowiska s_t to *zdyskontowana suma przyszłych nagród* (innymi słowy *dochód* dla kroku t), czyli

$$z_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.1}$$

Zauważmy w tym miejscu, że samo uczenie ze wzmocnieniem ma charakter stochastyczny, wobec czego będziemy jedynie w stanie uzyskać wartość oczekiwaną dochodu (przy danej strategii), czyli $V^\pi(s_t)$.

Aby nauczyć się funkcji V algorytm inicjuje ją jakąś wartością początkową V_0 (w każdym punkcie). Następnie w każdym kroku czasu t wykonuje jakąś akcję i obserwuje nagrodę r_t oraz nowy stan środowiska s_{t+1} po wykonaniu akcji. Na tej podstawie aktualizuje wartość funkcji w tym punkcie zgodnie ze wzorem:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha(r_t + \gamma V_t(s_{t+1}) - V_t(s_t)) \tag{2.2}$$

Występujący we wzorze parametr α ($\alpha \in (0;1]$) to *współczynnik rozmiaru kroku*. Sensem wzoru jest zmiana wartości funkcji V w punkcie s_t w zależności zarówno od otrzymanej nagrody, jak i wartości V w stanie, do którego trafimy. Innymi słowy, jeśli otrzymamy wysoką nagrodę, to znaczy, że wartość $V(s_t)$ wzrośnie, bo chcemy odwiedzać ten stan ze względu otrzymania w nim wysokich nagród. Jeśli zaś otrzymamy niską (ujemną) nagrodę, to wartość $V(s_t)$ zmaleje, co oznacza dla nas, że tego stanu nie chcemy odwiedzać, gdyż nagrody w nim przyznawane nie są atrakcyjne. Analogicznie - jeśli po wizycie w tym stanie znajdziemy się w stanie s_{t+1} o wysokiej wartości V (czyli takim, z którego możemy otrzymać wysokie nagrody), to chcielibyśmy częściej trafiać do stanu s_t, gdyż łatwo się z niego dostać do wartościowego stanu. Wobec tego w naszych oczach również stan s_t jest wartościowy. Jeśli zaś $V(s_{t+1})$ jest niskie (ujemne), to stan s_t nie jest wartościowy, gdyż możemy się z niego łatwo dostać do „złego" stanu - czyli powinniśmy zmniejszyć jego funkcję wartości. ($V(s_{t+1})$ mnożymy przez γ, gdyż ten iloczyn wejdzie do maksymalizowanej sumy (1.1)). Z kolei parametr α ma za zadanie zmniejszyć wpływ pojedynczego kroku na całościową wartość - tzn. jeśli mamy informacje o danym stanie na podstawie wielu kroków, nie chcielibyśmy o niej „zapomnieć" w kolejnym (łatwo zauważyć, że jeśli $\alpha = 1$ to w pełni „zapomnimy", co było przed ruchem). Zaznaczmy jeszcze, że skoro stanów jest skończenie wiele, zaś my poruszamy się w nieskończonym czasie, to wielokrotnie będziemy odwiedzać ten sam stan, więc faktycznie będziemy nabywać o nim coraz więcej informacji.

2.3. Zbieżność TD

Ważnym z matematycznego punktu widzenia jest pytanie o zbieżność algorytmu TD. Granicą w naszym przypadku jest funkcja V^*, czyli funkcja wartości stanu dla strategii optymalnej (strategia optymalna zawsze istnieje, więc jej funkcja wartości stanu również). Okazuje się, że zbieżność taka ma miejsce, jeśli parametr α zmienia się w czasie, zaś ciąg tych parametrów spełnia standardowe wymagania stochastycznej zbieżności. Pominiemy tutaj podanie tych warunków oraz dowód zbieżności, gdyż nie jest to istotne w dalszej części pracy. Z kolei zbieżność, jaką uzyskujemy to zbieżność według prawdopodobieństwa, tzn.:

$$\forall s \in S \; \forall \varepsilon > 0 \lim_{t\to\infty} \mathbb{P}(|V_t(s) - V^*(s)| < \varepsilon) = 1 \tag{2.3}$$

Warto nadmienić, że w przypadku algorytmów uczenia ze wzmocnieniem jest to najsilniejsza możliwa do wykazania zbieżność. Kiedy mówimy, że jakiś algorytm zbiega to mamy na myśli zbieżność według prawdopodobieństwa.

2.4. Eksploatacja i eksploracja

Ostatnim zagadnieniem, które poruszymy, omawiając ogólne algorytmy TD, jest problem eksploatacji i eksploracji. W trakcie uczenia się funkcji wartości stanów, program dowiaduje się, które stany warto odwiedzać, a które powinien omijać. Na tej podstawie może on wygenerować strategię zachłanną (czyli optymalną) i zwykle chcemy, by stosował właśnie tę strategię. Wynika to z faktu, że zazwyczaj bardziej od pełnego poznania środowiska zależy nam na rozwiązaniu danego problemu. Takie podejście nazwiemy eksploatacją, czyli podążaniem za strategią lokalnie optymalną i uczeniem się funkcji wartości stanu tylko w zakresie stanów wykorzystywanych w strategii. Z jednej strony jest to słuszne podejście, bo skoro wiemy, że jakaś strategia daje duże nagrody to znak, że powinna być bliska optymalnej i warto ją stosować. Z drugiej strony może to doprowadzić do sytuacji, w której nie poznamy dokładnie funkcji stanów dla dużej liczby stanów, które są istotne w strategii optymalnej, czyli innymi słowy środowisko nie zostanie dostatecznie poznane.

Jest to dość kluczowy problem, gdyż zależy nam zarówno na eksploatacji strategii, jak i na eksploracji środowiska. Jego rozwiązaniem są algorytmy losowe, tzn. takie, które z danym prawdopodobieństwem wybiorą akcję najlepszą (z punktu widzenia do tej pory wygenerowanej strategii), a z innym (zwykle mniejszym) losową akcję. Gwarantuje nam to, że nawet jeśli mamy lokalnie optymalną strategię na niewielkiej liczbie stanów, inne mogą zostać odwiedzone, co minimalizuje szansę nieodnalezienia optymalnej strategii w długim czasie. Wadą tego rozwiązania jest to, że nawet mając odnalezioną już strategię optymalną, program nie zawsze będzie wybierał optymalne ruchy. Zauważmy również, że o ile na początku uczenia się ważna eksploracja, gdyż chcemy poznać jak najlepiej środowisko, to w miarę nauki wolimy, by program eksploatował optymalną strategię, maksymalizując zyski.

W tym miejscu należy zaznaczyć, że problem eksploatacji i eksploracji nie występuje w ogólnej wersji algorytmu TD. Tam chcemy jedynie nauczyć się funkcji wartości, nie zależy nam natomiast na korzystaniu z optymalnej strategii w trakcie nauki, gdyż rozważamy nieskończony czas. Innymi słowy, ogólny algorytm TD wyłącznie eksploruje środowisko. W praktycznych zastosowaniach nie mamy jednak nieskończonego czasu, a liczba par stan-akcja jest zbyt duża, by je skutecznie eksplorować w rozsądnym czasie, dlatego zależy nam przede wszystkim na jak najlepszym poznaniu strategii optymalnej. Wobec tego niemal każdy algorytm mający rozwiązać konkretny problem musi rozwiązać jakoś zagadnienie eksploatacji i eksploracji.

Rozdział 3

Q-learning i SARSA

W tym rozdziale omówimy szczegółowo dwa algorytmy uczenia ze wzmocnieniem, które porównamy pod względem skuteczności w kolejnych rozdziałach. Oba bazują w dużej mierze na schemacie TD, są też do siebie dość podobne. Oba, w odróżnieniu od ogólnych algorytmów TD, uczą się funkcji wartości akcji Q (stąd nazwa pierwszego z nich).

3.1. Q-learning

Jak już zostało wspomniane wyżej program korzystający z algorytmu Q-learning uczy się funkcji wartości akcji. Przypomnijmy w tym miejscu o reprezentacji funkcji wartości akcji, którą anonsowaliśmy w podrozdziale 2.1 jako tabelki. Sam algorytm wygląda następująco:

Dla każdego kroku czasu t wykonaj:

1. *obserwuj aktualny stan* s_t*;*
2. $a_t = c(s_t, Q_t)$*;*
3. *wykonaj* a_t*;*
4. *obserwuj wzmocnienie* r_t *i następny stan* s_{t+1}*;*
5. *zaktualizuj* $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$*;*

Przeanalizujmy dokładnie punkty 2 i 5 powyższego algorytmu. Zaczniemy od punktu 5, w którym uaktualniamy wartość funkcji Q w danej komórce. Uaktualnienie jest dokonywane zgodnie z równaniem:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{3.1}$$

Zauważmy duże podobieństwo równania (3.1) do równania (2.2). Niemal wszystkie odwołania do funkcji Q w (3.1) są analogiczne jak odwołania do V w (2.2) (tzn.: odwołujemy się do obecnie znanych wartości uczonej funkcji). Jedyna pozorna różnica znajduje się w składniku mnożonym przez współczynnik γ (w przypadku (3.1) jest to $\max_a Q_t(s_{t+1}, a)$, zaś dla (2.2) $V_t(s_{t+1})$). W obu sytuacjach jednak sens tego składnika jest identyczny - dla (2.2) miał on na celu uwzględnienia w ocenie ruchu „jakości" stanu, w jakim znajdziemy się po jego wykonaniu (jest to po prostu wartość oczekiwana zdyskontowanej sumy nagród po znalezieniu się w nim). Analogicznie jest w przypadku (3.1) - $\max_a Q_t(s_{t+1}, a)$ jest swoistą miarą jakości kolejnego stanu. W tym przypadku jednak jest to wartość oczekiwana zdyskontowanej

sumy nagród przy założeniu dokonania najlepszej decyzji (czyli takiej, która maksymalizuje tę sumę) – jest to w praktyce ten sam składnik, jednak w Q-learningu uwzględniamy to, że możemy z góry ocenić jaką decyzję podejmiemy w kolejnym ruchu.

Przejdźmy teraz do analizy punktu 2. Pojawia się w nim funkcja wyboru akcji c. Jej zadaniem jest wskazanie, jaką akcję powinniśmy wykonać w danym stanie, mając dotychczasową wiedzę o funkcji Q. Ona właśnie rozwiązuje problem eksploatacji i eksploracji. Z jednej strony zależy nam na eksploatacji - chcemy więc, by preferowane były akcje o wysokiej wartości funkcji Q - z drugiej chcemy eksplorować środowisko, więc powinna móc również zwrócić inną akcję niż preferowana. W tym celu stosuje się, rzecz jasna, probabilistyczne funkcje wyboru akcji. W pracy ograniczymy się do następującej funkcji c:

$$\mathbb{P}(c(s_t, Q_t) = a) = \begin{cases} \frac{\varepsilon}{|\operatorname{argmax}_a Q(s,a)|} + \frac{1-\varepsilon}{|A|} & \text{, jeśli } a \in \operatorname{argmax}_a Q(s,a) \\ \frac{1-\varepsilon}{|A|} & \text{, w p. p.} \end{cases} \tag{3.2}$$

Powyższa formuła mówi, że z prawdopodobieństwem $1-\varepsilon$ wybrana zostanie losowa akcja, zaś z prawdopodobieństwem ε - akcja o największej wartości funkcji wyboru akcji (jeśli takich akcji jest więcej zostanie wybrana losowa z nich). Taka strategia wyboru akcji jest nazywana strategią ε-zachłanną.

Zauważmy, że przy takiej metodzie wyboru akcji Q-learning nie musi używać strategii, której się uczy. W każdym momencie wie, jaką akcję powinien wykonać w następnej kolejności (tzn. tę o największej wartości funkcji wyboru akcji) i uczy się, zakładając jej wykonanie, jednak może wykonać inną, losową akcję.

Na koniec zaznaczmy, że zaprezentowana tutaj wersja algorytmu Q-learning jest zbieżna (do optymalnej funkcji wartości akcji Q^*) pod takimi samymi warunkami, co algorytm TD (tzn. parametr α musi się zmieniać w czasie, a ciąg tych parametrów musi spełniać standardowe wymagania stochastycznej zbieżności, zbieżność według prawdopodobieństwa). Zbieżność można łatwo wykazać - dowód znajduje się w [ConOfQL].

3.2. SARSA

Nazwa tego algorytmu pochodzi od pierwszych liter słów *State, Action, Reward, State, Action* - są kolejno wykonywane kroki w każdym kroku czasu t. Zanim zaprezentujemy algorytm, przyjmijmy dla uproszczenia zapisu, że pierwsza akcja a_0 jest zdeterminowana przed rozpoczęciem algorytmu (w rzeczywistości jest ona wybierana identycznie jak w Q-learning). Oto algorytm po tym uproszczeniu:

Dla każdego kroku czasu t wykonaj:

1. *wykonaj* a_t*;*
2. *obserwuj wzmocnienie* r_t*;*
3. *obserwuj następny stan* s_{t+1}*;*
4. $a_{t+1} = c(s_{t+1}, Q_t)$*;*
5. *zaktualizuj* $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t))$*;*

Na pierwszy rzut oka różnice między algorytmami SARSA i Q-learning są spore, jednak po krótkiej analizie dostrzegamy, że SARSA wykonuje dokładnie te same operacje co Q-learning, jedynie w nieco zmienionej kolejności. Q-learning najpierw wybierał akcję, następnie

ją wykonywał i na tej podstawie uaktualniał wartość funkcji Q, po czym wybierał kolejną akcję itd.. SARSA z kolei wykonuje zaplanowaną akcję, wybiera kolejną i dopiero wówczas aktualizuje wartość funkcji Q. Ta zmiana kolejności wynika z zasadniczej różnicy między tymi algorytmami, czyli metody uaktualniania funkcji Q. SARSA robi to w następujący sposób:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)), \tag{3.3}$$

zaś Q-learning w sposób opisany w równaniu (3.1). Różnica między tymi metodami jest taka, że Q-learning uwzględnia, aktualizując wartość dla danego stanu i akcji, maksymalną wartość kolejnej akcji (która nie musiała zostać wykonana), podczas gdy SARSA aktualizuje tę wartość, korzystając z wartości dla akcji, która została faktycznie wybrana (stąd najpierw wybór kolejnej akcji). Innymi słowy, Q-learning zakłada, że zostanie wybrana najlepsza akcja i prognozuje w oparciu o to założenie, podczas gdy SARSA przewiduje na podstawie faktycznie wybranej akcji.

Oznacza to, że mimo, iż SARSA również wybiera akcję w oparciu o wzór (3.2), to zawsze używa strategii, której się uczy. Zauważmy, że gdyby wybór akcji nie był losowy, lecz deterministyczny i zachłanny, to Q-learning i SARSA byłyby tym samym algorytmem.

Mimo tak niewielkich różnic w samym algorytmie, SARSA różni się od Q-learningu własnościami. W zastosowaniach praktycznych okazuje się, że zwykle SARSA szybciej uczy się rozwiązania problemu niż Q-learning i jest preferowanym algorytmem. Kolejną różnicą jest fakt, iż dowód zbieżności SARSA jest o wiele trudniejszy niż w przypadku Q-learningu i nie został on jeszcze przeprowadzony w ogólnym przypadku. Dowód częściowy znajduje się w [ConRes].

Rozdział 4

Q-learning i SARSA w problemie gry w kółko i krzyżyk

Przejdźmy teraz do zagadnień praktycznych uczenia ze wzmocnieniem. Pierwszym zadaniem, jakie rozwiążemy, będzie znalezienie optymalnych strategii w grze kółko i krzyżyk w jej najbardziej klasycznej wersji. Mimo, że jest to powszechnie znana gra, przypomnijmy dla porządku jej zasady:

W grze uczestniczy dwóch graczy - jeden używa symbolu kółka, a drugi krzyżyka. Gra toczy się na planszy o wymiarach 3×3 pola. Ruch polega na umieszczeniu swojego symbolu w jednym z niezajętych do tej pory pól. Gracze wykonują ruchy naprzemiennie, zaś gracz grający kółkami zaczyna (tę zasadę wprowadzamy dla wygody - w ogólności oczywiście nie jest istotne, kto zaczyna). Wygrywa osoba, której trzy symbole zajmują rząd poziomy, pionowy lub pola zawierające przekątną. Jeśli wszystkie pola na planszy zostały zajęte, a żaden z graczy nie uzyskał wygrywającej kombinacji, gra kończy się remisem.

Powszechnie znanym faktem jest, że przy rozsądnej grze obu graczy, gra zawsze zakończy się remisem. Tego będziemy wymagać również od naszych uczniów-programów, tzn. jeśli gra on z przeciwnikiem, który nie popełnia błędów, gra powinna zakończyć się remisem, z kolei jeśli przeciwnik popełnia błędy, program powinien zwyciężyć. Na przykładzie tej gry będziemy chcieli porównać dwa zaprezentowane wcześniej algorytmy uczenia ze wzmocnieniem – Q-learning i SARSA. Implementacji dokonamy w języku Python.

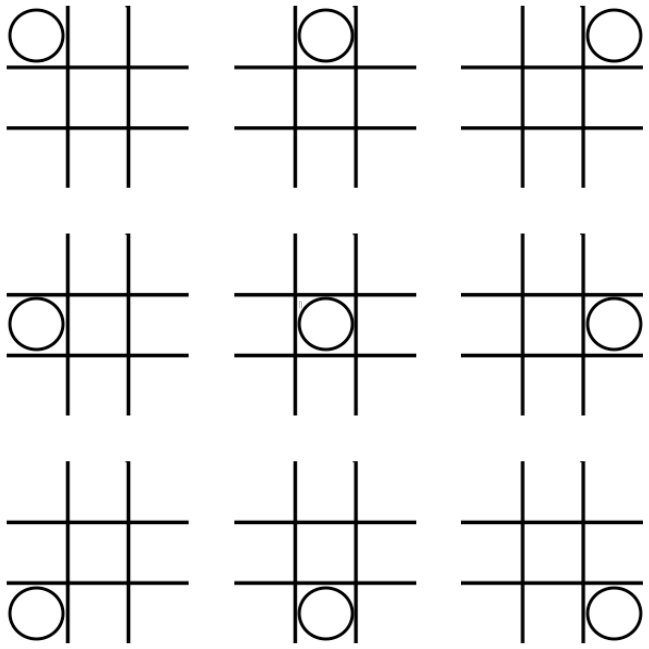
4.1. Reprezentacja problemu w programie

Opiszmy na początku sposób reprezentacji problemu, na której będą bazować oba algorytmy. Zacznijmy od zdefiniowania przestrzeni stanów i przestrzeni akcji. Przestrzeń akcji jest tutaj dość oczywista - składa się ona z 9 elementów odpowiadających polom na planszy, tzn. wybranie akcji odpowiadającej danemu polu oznacza umieszczenie na tym polu swojego symbolu. W programie przestrzeń stanów to po prostu tablica 9-elementowa zawierająca umowne nazwy akcji.

Przestrzenią stanów natomiast mógłby być zbiór wszystkich możliwych układów elementów na planszy (tzn. jednym ze stanów jest pusta plansza, innym plansza z kółkiem w lewym górnym rogu itd.). Taka przestrzeń byłaby jednak bardzo duża i, w związku z tym, niepraktyczna - jej moc to 3^9, gdyż na każdym z 9 pól może znajdować się kółko, krzyżyk lub może być ono puste. Dostrzegamy od razu jednak, że zdecydowana większość z tych plansz nigdy nie wystąpi w grze, gdyż mają niemożliwą do osiągnięcia liczbę symboli (np. plansza z 9 krzyżykami, plansza z 6 kółkami i 3 krzyżykami, plansza z 4 kółkami i 2 krzyżykami, plansza

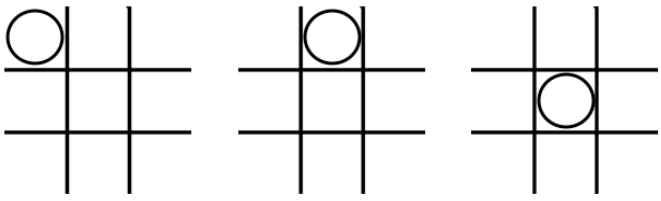
z 5 krzyżykami i 4 kółkami). Zostawiamy zatem te układy, w których liczba kółek jest równa ilości krzyżyków lub jest od nich większa o 1. Zauważamy jednak, że tutaj również znajdują się plansze, które nigdy nie wystąpią w grze tzn. takie, gdzie istnieją dwa niezależne układy wygrywające (np. plansza z 3 kółkami w lewej kolumnie i 3 krzyżykami w środkowej kolumnie). Możemy zatem je wykluczyć - w naszej puli stanów zostały, więc jedynie układy na planszy spełniające poprzedni warunek oraz mające co najwyżej 1 niezależny układ wygrywający.

Zauważmy jeszcze, że niektóre układy są homeomorficzne z innymi tzn. mogą powstać z innego układu poprzez symetrię lub obrót planszy (np. układ, w którym jest tylko jedno kółko znajdujące się w lewym górnym rogu planszy jest homeomorficzny z układem z jedynym kółkiem w prawym górnym rogu). Wobec tego dla każdego zbioru homeomorficznych układów do przestrzeni stanów dodamy tylko jednego reprezentanta tego zbioru. Po powyższych redukcjach nasza przestrzeń stanów liczy jedynie 765 elementów.



Rysunek 4.1: Wszystkie możliwe pierwsze ruchy

Zauważmy jednak, że powoduje to, iż niektóre różne akcje prowadzą do tego samego stanu. Rozważając np. pierwszy ruch w grze - gracz grający kółkami ma 9 akcji do wyboru (gdyż każde pole jest wolne), jednak po jego ruchu będzie jeden z trzech stanów: kółko ustawione w narożniku, kółko ustawione w środkowym polu „krawędzi" planszy, kółko ustawione w środkowym polu planszy (innymi słowy są 4 różne akcje wyboru narożnika, ale prowadzą one do tego samego stanu).



Rysunek 4.2: Wszystkie możliwe stany po pierwszej akcji

W programie dany układ na planszy będziemy reprezentować jako macierz 3×3, gdzie jeśli na danym polu planszy znajduje się kółko, na odpowiadającym mu polu w macierzy znajduje się 1, krzyżyk – -1, zaś jeśli pole jest puste – 0. Z kolei przestrzeń stanów jest reprezentowana jako 10-elementowa tablica (indeksowana od 0), gdzie pod n-tym indeksem znajduje się lista stanów zawierająca dokładnie n elementów (w tym oczywiście $[\frac{n+1}{2}]$ kółek i $[\frac{n}{2}]$ krzyżyków) tzn. pod indeksem 0 jest jednoelementowa lista, której jedynym elementem jest macierz samych zer (czyli odpowiadająca pustej planszy), zaś pod indeksem 1 – lista 3-elementowa, której każdy element odpowiada jednemu stanowi z jednym kółkiem (wymieniliśmy je wyżej).

Ostatnim elementem problemu, który musimy przedstawić, jest funkcja Q. Oczywiście chcielibyśmy, aby była to tabela taka jak opisaliśmy to w podrozdziale 2.1. Ze względu jednak na opisaną wyżej reprezentację przestrzeni stanów musimy wprowadzić drobną modyfikację. W programie funkcja Q będzie 10-elementową tabelą, której każdy n-ty element to tabela, której każdy wiersz odpowiada jednemu ze stanów zawierającemu dokładnie n symboli, zaś kolumny odpowiadają poszczególnym akcjom. Oznacza to, rzecz jasna, że na przecięciu danego wiersza s i kolumny a stoi wartość funkcji $Q(s, a)$.

Taka reprezentacja ma swoje wady. Po pierwsze ze względu na to, że rozróżniamy akcję, nawet jeśli prowadzą do tego samego stanu, program będzie musiał się uczyć wartości Q dla każdej z tych akcji z osobna, co wydłuża czas nauki. Innymi słowy, np. w pierwszym ruchu są 4 akcje wyboru narożnika i – mimo, że prowadzą do tego samego stanu – program będzie musiał się uczyć osobno wartości dla każdej z tych akcji. Kolejnym problemem jest fakt, że taka postać tabeli teoretycznie pozwala wartościować ruchy niedozwolone. Jeśli np. jesteśmy w stanie s, w którym środkowe pole jest zajęte, to akcja a wyboru środkowego pola jest niedopuszczalna w myśl zasad gry. Istnieje jednak komórka odpowiadająca wartości $Q(s, a)$, co mogłoby sugerować programowi, że warto wykonać w określonych sytuacjach tę akcję. Z problemem tym można sobie jednak łatwo poradzić, tworząc funkcję, która dopuszcza do puli, z której program wybiera akcje, jedynie legalne ruchy, dzięki czemu ta akcja nigdy nie będzie brana pod uwagę oraz wartościowana.

4.2. Podejście do problemu

Aby stosować uczenie ze wzmocnieniem, należy podjąć decyzję o początkowej wartości funkcji Q oraz, co istotniejsze, o wysokości przyznawanych nagród. Jeśli chodzi inicjację Q, to standardowo przyjmiemy wartość w każdym punkcie na 0. Ma to tę dodatkową zaletę, że nie obciąża nas w „fałszywych" punktach (czyli tych, które odpowiadają nielegalnej akcji). Jeśli chodzi o nagrody, to przyjmiemy, że za ruch, który nie doprowadził do wygranej przyznawana jest nagroda 0, za ruch prowadzący do wygranej 1, zaś za ruch prowadzący do przegranej -1. Oczywiście gracz w swoim ruchu nie może doprowadzić do własnej przegranej(po narysowaniu kółka nie pojawi się wszak układ 3 krzyżyków), jednak chcemy jakoś karać za ruchy, które doprowadziły do zwycięstwa przeciwnika. Okazuje się, że jest to skutek większego problemu, który występuje w przypadku uczenia się jakiejkolwiek gry więcej niż jednoosobowej, więc w szczególności kółka i krzyżyka.

Problem, o którym mowa, to brak możliwości przewidzenia, w jakim stanie będziemy podejmować kolejną decyzję. Wykonując daną akcję, wiemy, w jakim stanie będzie gra, jednak w tym nowym stanie decyzję o akcji podejmuje nasz przeciwnik. Nie możemy jednak tej decyzji przewidzieć (w najlepszym przypadku możemy szacować, jaka ona będzie), a właśnie ta decyzja determinuje, w którym stanie znajdziemy się przed naszym kolejnym ruchem, a jak wiemy, ten stan istotnie wpływa na wartościowanie wykonanego ruchu.

Rozwiązanie tego problemu zaproponowane przeze mnie i użyte w programie polega na

uczeniu zarówno na podstawie swojego ruchu, jak i przeciwnika. Program uczy się naraz gry jako pierwszy i drugi gracz, dla każdego z nich starając się optymalizować strategię. Można też powiedzieć, że program gra sam ze sobą, gdyż w każdym momencie zna całą tabelę wartości Q i wykorzystuje ją w ruchach obu graczy. Zauważmy tutaj, że ze względu na to, że kółko wykonuje akcje tylko w ruchach nieparzystych, a krzyżyk w parzystych gracze nie ingerują w swoje strategie (tzn. nie ma pola w tabeli, które determinuje ruch obu graczy naraz). Oczywiście każdy z graczy dąży do maksymalizacji swoich zysków, w związku z tym akcje muszą być nagradzane przeciwnie.

W praktyce w programie zawsze pamiętamy zarówno obecnie wykonany ruch, jak i poprzedni. Każda akcja z kolei powoduje uaktualnienie dwóch komórek w tabeli - tej związanej z nią oraz z jej poprzedniczką. Przywołajmy tutaj wzór ogólny (2.2), według którego uaktualniamy funkcję wartości:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha(r_t + \gamma V_t(s_{t+1}) - V_t(s_t))$$

Modyfikujemy go w taki sposób, by wartość danej akcji była faktycznie zwiększana przez nagrodę (gdyż ta akcja spowodowała jej przyznanie) i zmniejszana przez jakość kolejnej akcji, gdyż jej wartość działa na korzyść przeciwnika, a co za tym idzie – na naszą niekorzyść. Oto zmodyfikowany wzór:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha(r_t - \gamma V_t(s_{t+1}) - V_t(s_t)) \tag{4.1}$$

Z kolei w sytuacji, kiedy oceniamy odpowiedź przeciwnika na naszą akcję, chcemy, by jego nagrody były dla nas „karą", zaś naszym zyskiem z tej akcji będzie jakość stanu, w jakim się znajdziemy przed kolejną akcją. Oto zmodyfikowana wersja wzoru (2.2) dla tej sytuacji (zwróćmy uwagę na zwiększone indeksy, gdyż oceniamy w następnej jednostce czasu):

$$V_{t+2}(s_t) = V_{t+1}(s_t) + \alpha(-r_{t+1} + \gamma V_{t+1}(s_{t+2}) - V_{t+1}(s_t)) \tag{4.2}$$

Zapiszmy jeszcze odpowiednie wersje tych wzorów dla Q-learningu i SARSA. Zauważmy przy okazji, że wzory (4.1) i (4.2) opisują, co dzieje się z wartością dla stanu s_t w kolejnych jednostkach czasu. W programie natomiast preferowane jest podejście mówiące o zmianach w różnych stanach w danej jednostce czasu t (gdyż wolimy symulować jedną jednostkę czasu na raz). Oczywiście jest to wyłącznie manipulacja indeksami, jednak dla wygody podamy wzory dla Q-learningu i SARSA zarówno w wersji dla danego stanu (czyli zgodnej z ogólną), jak i dla danego czasu (czyli użytej w programie).

Q-learning wersji dla stanu:

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= Q_t(s_t, a_t) + \alpha(r_t - \gamma \widehat{\max_a} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \\ Q_{t+2}(s_t, a_t) &= Q_{t+1}(s_t, a_t) + \alpha(-r_{t+1} + \gamma \widehat{\max_a} Q_t(s_{t+2}, a) - Q_{t+1}(s_t, a_t)) \end{aligned} \tag{4.3}$$

Q-learning wersji dla czasu:

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= Q_t(s_t, a_t) + \alpha(r_t - \gamma \widehat{\max_a} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \\ Q_{t+1}(s_{t-1}, a_{t-1}) &= Q_t(s_{t-1}, a_{t-1}) + \alpha(-r_t + \gamma \widehat{\max_a} Q_t(s_{t+1}, a) - Q_t + (s_{t-1}, a_{t-1})) \end{aligned} \tag{4.4}$$

Użyte powyżej oznaczenie $\widehat{\max}$ to maksymalna wartość dla akcji możliwych do wykonania. Rozróżnienie to wynika z faktu, że tabela wartości funkcji Q jest określona (jak zostało powiedziane wcześniej) dla każdej pary stan-akcja, ale nie każdą akcję można wykonać, a nie chcemy

zaburzać wyników (akcja niemożliwa do wykonania zawsze ma wartość 0, więc wybierając akcję prowadzącą do stanu, w którym możliwe wszystkie do wykonania akcje mają wartości ujemne zwykłe maksimum zwróci nam to „fałszywe" 0, podczas gdy faktyczne maksimum jest ujemne).

Zapiszmy teraz analogiczne wzory dla SARSA:

SARSA wersji dla stanu:

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= Q_t(s_t, a_t) + \alpha(r_t - \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) \\ Q_{t+2}(s_t, a_t) &= Q_{t+1}(s_t, a_t) + \alpha(-r_{t+1} + \gamma Q_t(s_{t+2}, a_{t+2}) - Q_{t+1}(s_t, a_t)) \end{aligned} \tag{4.5}$$

SARSA wersji dla czasu:

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= Q_t(s_t, a_t) + \alpha(r_t - \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) \\ Q_{t+1}(s_{t-1}, a_{t-1}) &= Q_t(s_{t-1}, a_{t-1}) + \alpha(-r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t + (s_{t-1}, a_{t-1})) \end{aligned} \tag{4.6}$$

Zaznaczmy jeszcze w tym miejscu, że pozostałe parametry tj. α, γ, ε są identyczne zarówno dla obu graczy. Program został przetestowany na różnych wartościach parametrów - szczegóły znajdują się dalej.

4.3. Gracze wzorcowi

Celem tego rozdziału jest zbadanie, jak szybko jesteśmy w stanie uzyskać strategię optymalną w grze w kółko i krzyżyk, wykorzystując algorytmy Q-learning i SARSA w zależności od różnych parametrów. Zanim jednak przejdziemy do tej analizy, uzyskamy *graczy wzorcowych*. Gracze wzorcowi to odpowiednio wytrenowane programy (jeden przy użyciu Q-learningu, a drugi SARSA), z którymi będą porównywane programy dopiero się uczące.

W naszym przypadku obaj gracze wzorcowi zostali wygenerowani przy następującym zestawie parametrów: $\varepsilon = 0,9$, $\alpha = 0,5$, $\gamma = 0,9$. Każdy z graczy wzorcowych uczył się na podstawie 20.000 gier. Po procesie uczenia obaj gracze wzorcowi rozegrali szereg partii sprawdzających ich poziom. W przeciwieństwie do gier uczących, w których posługiwali się strategią ε-zachłanną, w grach sprawdzających stosowali strategię zachłanną wynikającą z ich tabeli wartości funkcji Q (a dokładnie zmodyfikowaną strategię zachłanną, opisaną niżej). Gracze wzorcowi nigdy nie przegrywali swoich gier, jednak w pojedynczych sytuacjach nie wygrywali swoich gier, które można wygrać. Takie odstępstwa od ideału były jednak bardzo rzadkie i w zdecydowanej większości sytuacji ci gracze byli nieomylni. Wyniki te osiągali niezależnie (tzn. czasem gracz Q-learning wygrał grę, którą SARSA zremisował i odwrotnie). Tacy niemal bezbłędni gracze stali się wzorcami testującymi uczące się programy.

Warto zauważyć, że gdy gracze posługują się strategią zachłanną, to grają w pełni deterministycznie. Oznacza to, że gdyby gracz taki grał sam ze sobą (tzn. posługiwał się strategią dla obu graczy, której się nauczył), to każda partia byłaby identyczna. Analogicznie gdyby dwóch graczy o deterministycznych ruchach grało ze sobą, istniałyby tylko dwa schematy rozgrywki zależące od tego, który gracz rozpoczyna. Zauważmy w tym momencie, że żaden pierwszy ruch w kółku i krzyżyku nie prowadzi do porażki (przy założeniu idealnej gry pierwszego gracza). Bazując na tej obserwacji, możemy stworzyć *zmodyfikowaną strategię zachłanną* tzn. taką, w myśl której pierwszy ruch jest losowy, zaś kolejne są wykonywane zgodnie ze strategią zachłanną. Gracz, który stosuje zmodyfikowaną strategię zachłanną, grając sam ze sobą, wygeneruje więc trzy schematy rozgrywki, zaś dwóch graczy - sześć schematów. Czyni to ze zmodyfikowanej strategii zachłannej zdecydowanie lepszy test do badania jakości wiedzy gracza.

Wnioskiem z powyższej obserwacji o pierwszym ruchu jest spostrzeżenie, że w tabeli idealnie wytrenowanego gracza każda pierwsza akcja ma wartość funkcji Q równą 0 (bo żaden ruch początkowy nie gwarantuje przegranej, a przy idealnej grze partia zawsze skończy się remisem). Rzecz jest w tym, że program uczący się nigdy nie osiągnie tam takiej wartości (rzecz jasna z czasem wartości te powinny być bliskie 0, ale od niego różne). Pozwala nam to zdefiniować pojęcie *luki gracza*. Luką gracza nazwiemy różnicę między maksymalną wartością i minimalną wartością funkcji Q tego gracza dla pierwszej akcji. Spodziewamy się, że w miarę uczenia się luka będzie maleć do 0.

W oparciu o lukę możemy zdefiniować jeszcze jeden typ strategii dla gracza, czyli *strategię zachłanną z luką*. Grając tą strategią, gracz losuje akcję spośród tych, dla których wartość funkcji Q jest mniejsza co najwyżej o lukę od akcji o największej wartości. Oznacza to w szczególności, że akcja o największej wartości funkcji Q (czyli ta, która zostałaby wybrana w strategii zachłannej) zawsze może zostać wylosowana. Dodatkowo każdy z pierwszych ruchów może zostać wylosowany.

Stosowanie przez gracza strategii zachłannej z luką powoduje, że musi mieć zdecydowanie dokładniejszą tabelę wartości funkcji Q, by grać dobrze. Gdyby akcje korzystne miały wartości zbliżone do niekorzystnych, to taki gracz będzie wybierał akcję niekorzystną równie często co korzystną, przez co przegra duży odsetek gier. Strategia taka premiuje więc graczy o dobrej strategii zachłannej i małej luce lub o wyraźnie zróżnicowanych akcjach korzystnych i niekorzystnych pod względem wartości funkcji Q, gdyż wówczas nawet duża luka nie spowoduje wylosowania akcji niekorzystnej. Na koniec podamy wartości luki naszych graczy wzorcowych: luka gracza Q-learning wynosi $0,0901877307633$, zaś gracza SARSA - $0,198408762693$. Widzimy więc, że ten drugi ma lukę ponad 2 razy większą.

4.4. Porównanie uczenia algorytmem Q-learning i SARSA

W tym podrozdziale przeanalizujemy oba algorytmy pod względem tempa uczenia się gry w kółko i krzyżyk, w zależności od różnych wartości parametrów. Dla każdego zestawu parametrów z puli podanej niżej naukę rozpoczynają dwaj gracze – jeden używa Q-learningu, a drugi SARSA. Używając danego zestawu parametrów, każdy z graczy rozgrywa („sam ze sobą") 20.000 gier. Co 250 partii obaj zostają poddani *testowi walidacyjnemu* składającemu się z czterech etapów.

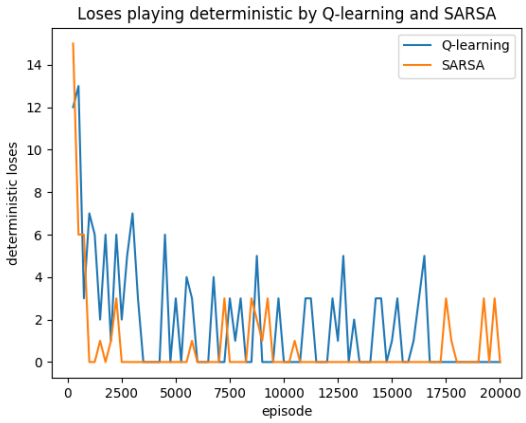
Pierwszy etap polega na rozegraniu z każdym z graczy wzorcowych 6 partii, stosując zmodyfikowaną strategię zachłanną (każda partia to inna kombinacja gracza rozpoczynającego i pierwszego ruchu). Etap ten jest kluczowy, gdyż jeśli program przegra choć jedną partię, oznacza to, że nie wypracował jeszcze poprawnej strategii zachłannej i nie powinien teraz przerywać procesu uczenia.

Drugi etap z kolei składa się ze 100 partii. W każdej z nich najpierw losowany jest jeden z graczy wzorcowych jako przeciwnik, a następnie gracz rozpoczynający. Zarówno gracz wzorcowy, jak i uczący się program stosują strategię zachłanną z luką.

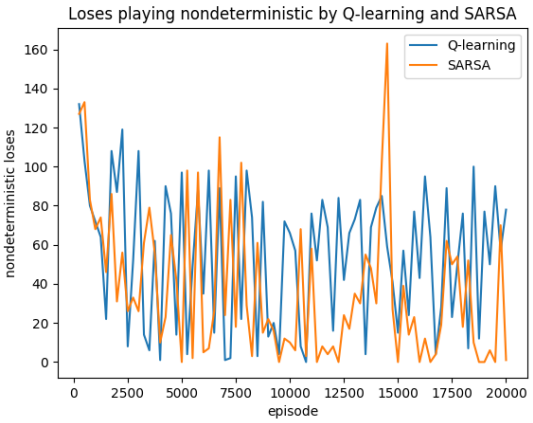
Kolejne dwa etapy są analogami pierwszych dwóch, jednak zamiast graczy wzorcowych uczące się programy grają między sobą. W trzecim etapie stosują zmodyfikowaną strategię zachłanną i rozgrywają między sobą 6 partii (po jednej dla każdego układu otwierającego). Z kolei w czwartym również grają miedzy sobą, jednak tutaj korzystają ze strategii zachłannej z luką i rozgrywają 100 partii.

Poniżej znajdują się dwa komplety wykresów obrazujących część efektów. W obu zestawach pierwszy wykres przedstawia liczbę przegranych przez program gier (w zależności od czasu), gdy stosował zmodyfikowaną strategię zachłanną (a zatem maksymalnie mógł prze-

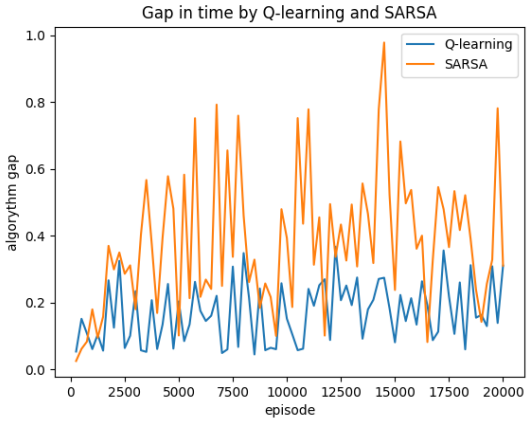
grać 18 gier – tj. 6 z każdym z graczy wzorcowych i 6 z drugim programem uczącym się). Jest to kluczowy wykres, gdyż obrazuje, kiedy możemy podejrzewać, że program się nauczył już grać tzn. liczba przegranych równa 0.



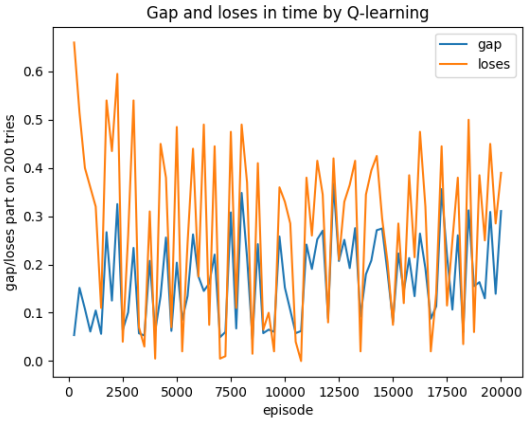
(a) Przegrane przy zmodyfikowanej strategii zachłannej



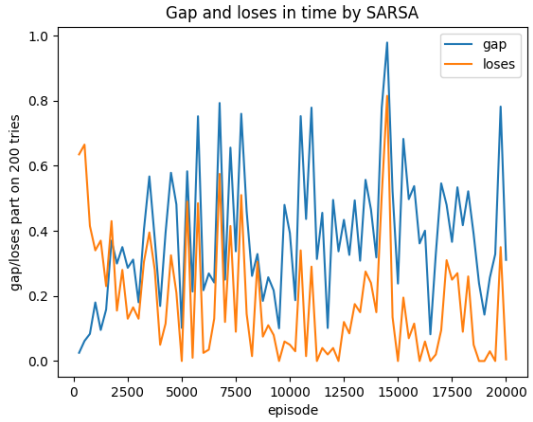
(b) Przegrane przy strategii zachłannej z luką



(c) Luka w czasie

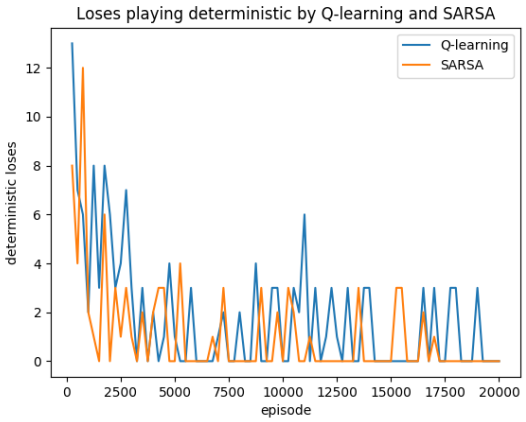


(d) Odsetek przegranych a luka w Q-learning

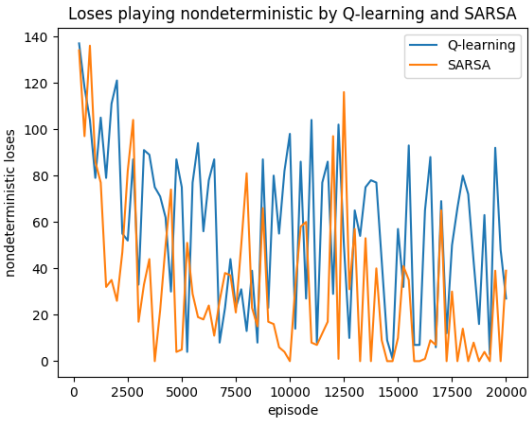


(e) Odsetek przegranych a luka w SARSA

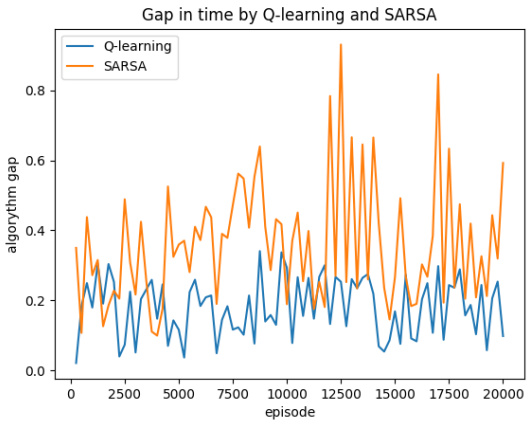
Rysunek 4.3: Wykresy dla zestawu parametrów $\varepsilon = 0,8$ $\alpha = 0,55$ $\gamma = 0,8$



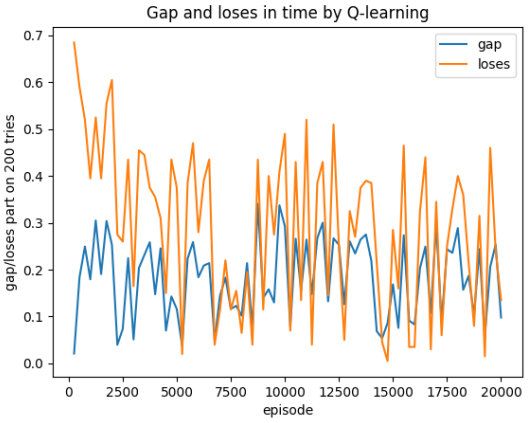
(a) Przegrane przy zmodyfikowanej strategii zachłannej



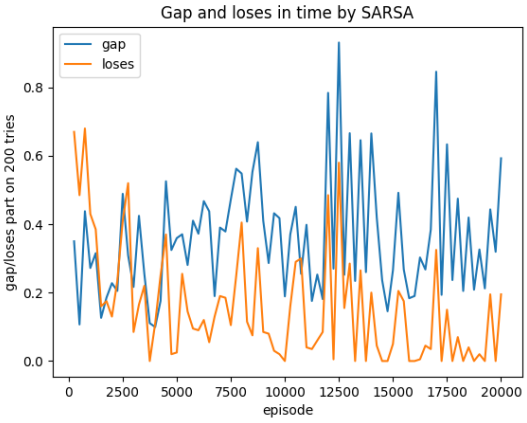
(b) Przegrane przy strategii zachłannej z luką



(c) Luka w czasie



(d) Odsetek przegranych a luka w Q-learning



(e) Odsetek przegranych a luka w SARSA

Rysunek 4.4: Wykresy dla zestawu parametrów $\varepsilon = 0,8$ $\alpha = 0,4$ $\gamma = 0,9$

Pierwszym wnioskiem, jaki wyciągamy z wykresów, jest kluczowy fakt, że oba programy faktycznie się uczą. Jest to widoczne zwłaszcza na pierwszym wykresie w każdym zestawie – w miarę upływu czasu uczeń przegrywa coraz mniej gier i ma coraz dłuższe okresy bez porażek. Łatwo również zauważyć, że SARSA jest skuteczniejsza niż Q-learning - ma mniejsze wahania od sytuacji idealnej i dłuższy czas gra bez porażek. Widzimy również wyraźnie, że co jakiś czas występuje odstępstwo od sytuacji idealnej, czyli 0 porażek (dla Q-learningu nawet stosunkowo często). Może być to efekt *przeuczenia*, czyli faktu, że mimo posiadania dostatecznie wielu informacji o środowisku program próbuje uczyć się dalej, przez co przypadkowe akcje przynoszą w pewnych (zwykle wylosowanych) sytuacjach korzyść, która jest traktowana jak efekt pożądany. Innym wytłumaczeniem jest *niedouczenie* – program działał zbyt krótko, by nauczyć się doskonale środowiska i dalej popełnia błędy. Niestety efekt nie znikał podczas symulacji dla większych liczb gier (największa sprawdzona to 200.000, a jej czas to ponad 10 godzin, więc zrezygnowałem z dalszych prób dla równie dużej liczby gier).

Kolejna obserwacja to fakt, że w przypadku stosowania strategii zachłannej z luką to nie liczba rozegranych gier do tej pory gier jest istotna, ale wielkość luki. Widzimy to, porównując wykres pierwszy i drugi, gdzie ciężko dostrzec prawidłowość między sukcesami w grach o strategiach czysto deterministycznych i nieco losowej z luką. Jednak najlepiej możemy to dostrzec na wykresie czwartym i piątym, gdzie większa luka idzie w parze z większym odsetkiem porażek, zwłaszcza po większej ilości rozegranych gier.

Następnym wnioskiem jest fakt, że luka nie wykazuje zbieżności do postulowanego wcześniej 0, ani żadnej innej wartości. Być może jest to wynik zbyt małej liczby prób lub, ze względu na mały wpływ pierwszej akcji, wielokrotnie podlega przeuczeniu. Warto natomiast zauważyć, że SARSA wykazuje większą lukę niż Q-learning, mimo że jest skuteczniejsza w grze strategią zachłanną z luką.

Ogólnie widzimy, że SARSA jest skuteczniejsza niż Q-learning – skuteczniej gra zarówno deterministycznie, jak i z luką oraz ma lepiej odseparowane wartości dobrych i słabych akcji, bo nawet mimo dużej luki wybiera lepsze niż Q-learning z mniejszą.

Analizując większą liczbę wykresów, które są dołączone do pracy, dochodzimy również do wniosku, że większy ε zmniejsza przewagę SARSA nad Q-learning (co jest spodziewanym wynikiem o tyle, że częściej wybierane są akcje o większej wartości, czyli te, których „spodziewa się" Q-learning), ale również ogólnie większy ε poprawia wyniki dla obu algorytmów.

Okazuje się również, że lepiej sprawdzają się małe wartości γ oraz optymalna α ma wartość rzędu $0,55$. Widać to zarówno na wykresach zamieszonych w pracy (te o parametrach podanych powyżej wydają się szybciej i trwalej zbiegać do optymalnych strategii) oraz na ogóle wykresów uzyskanych w programie.

Zaznaczmy na koniec, że zakresy sprawdzanych parametrów to: $\varepsilon \in [0,8;\ 0,95]$, $\alpha \in [0,3;\ 0,6]$ i $\gamma \in [0,8;\ 0,9]$, w każdym przypadku z krokiem co $0,05$, czyli łącznie 84 zestawy parametrów.

Rozdział 5

Algorytmy Q-learning i SARSA w problemie Taxi-v1

W ostatnim rozdziale pracy zajmę się porównaniem algorytmów Q-learning i SARSA w zadaniu Taxi-v1 pochodzącym ze strony OpenAI Gym. Strona zawiera wiele przykładowych problemów do rozwiązania algorytmami uczącymi się ze wzmocnieniem wraz z narzędziami do ich obsługi i badania. Problem Taxi-v1 jest jednym z prostszych problemów na stronie, dzięki temu ma on wiele rozwiązań, z którymi będzie można skonfrontować przestawione w pracy rezultaty. Prezentowane poniżej kryteria oceny są identyczne jak w oryginalnym problemie (link do strony z opisem: `https://gym.openai.com/envs/Taxi-v1`).

5.1. Opis problemu

W zadaniu Taxi-v1 celem programu jest nauczenie się najefektywniejszego „poruszania się taksówką po mieście". Miasto to plansza 5×5 z czterema wyróżnionymi polami-stacjami oraz zablokowanymi krawędziami między niektórymi polami. Taksówka porusza się po polach planszy, może przechodzić jedynie przez niezablokowane krawędzie. Na początku zadania z pól-stacji losujemy punkty początkowy i końcowy (niekoniecznie różne), a ze wszystkich pól losujemy lokalizację początkową taksówki.

Rysunek 5.1: Przykładowy układ początkowy

W naszym zadaniu układ planszy jest taki, jak na rysunku powyżej - pola-stacje są oznaczone literami „R", „G", „Y" i „B", krawędzie zablokowane - „|", niezablokowane pionowe - „:". Wszystkie krawędzie poziome są niezablokowane, więc nie będziemy ich oznaczać. Układ tych

pól nigdy się nie zmienia, tzn.: każda plansza ma te same pola-stacje i zablokowane krawędzie. Jedyne co może się zmienić to układ początkowy. Na planszy punkt początkowy oznaczamy kolorem czerwonym (w naszym przypadku „G"), punkt końcowy kolorem niebieskim (u nas „R"), zaś lokalizacja taksówki jest oznaczona poprzez żółte tło pola, na którym przebywa (w naszym wypadku pole w trzecim wierszu i czwartej kolumnie).

Taksówka ma zawsze dostępne 6 akcji: ruch w górę, ruch w dół, ruch w prawo, ruch w lewo, zabranie pasażera, wysadzenie pasażera. Ponieważ taksówka ma wszystkie akcje zawsze dostępne, może się zdarzyć, że stojąc np. przy krawędzi wykona akcję ruchu „w krawędź", wówczas układ na planszy nie zmienia się, ale akcja liczy się jako wykonana.

Zadaniem taksówki jest przejechanie do punktu początkowego (tam znajduje się pasażer), zabranie go, przejechanie do punktu końcowego i wysadzenie pasażera w tym miejscu (przypomnijmy w tym miejscu, że możliwy jest układ, w którym punkt początkowy i końcowy znajdują się na tym samym polu-stacji – wówczas taksówka musi dostać się na to pole, zabrać pasażera, a następnie wysadzić go w tym samym miejscu). Dodatkowo wprowadzona została punktacja poszczególnych sytuacji: za wykonanie zadania przyznawane jest 20 punktów, za każdą wykonaną akcję odbierany jest 1 punkt, zaś za wykonanie akcji zabrania lub wysadzenia pasażera w niedozwolony sposób odbierane jest 10 punktów. Wprowadzenie punktacji ma na celu ocenę szybkości wykonania zadania, czyli poza samym dowiezieniem pasażera do celu chcemy to zrobić możliwie szybko. Wspomniany wyżej niedozwolony sposób wykonania akcji zabrania pasażera jest rozumiany tutaj jako próba wywołania jej w innym miejscu niż punkt startowy lub w punkcie startowym po wcześniejszym zabraniu pasażera, zaś niedozwolone wykonanie akcji wysadzenie to wywołanie jej bez uprzedniego wzięcia pasażera lub, mając pasażera, w innym punkcie niż końcowy.

Maksymalna liczba punktów, jakie można otrzymać w przykładowym układzie (rys. 5.1) to 9 – najpierw w trzech akcjach docieramy do pola-stacji „G" (-3 punkty), następnie wykonujemy akcję zabrania pasażera (-1 punkt), później w sześciu akcjach (musimy ominąć przeszkodę między polami 2 i 3 w pierwszym wierszu) przejeżdżamy do pola-stacji „R" (-6 punktów) i wykonujemy akcję wysadzenie pasażera (-1 punkt), za co otrzymujemy 20 punktów, zatem sumarycznie mamy $20 - 3 - 1 - 6 - 1 = 9$ punktów. Zauważmy jeszcze, że maksymalna liczba punktów to 18, którą możemy otrzymać tylko, gdy punkt startowy, końcowy i początkowa lokalizacja taksówki są na tym samym polu (wówczas i tak musimy wykonać akcję zabrania i wysadzenia pasażera, ale nie trzeba podróżować).

5.2. Reprezentacja problemu

Jak wspomnieliśmy na początku rozdziału, strona OpenAI Gym dostarcza narzędzia umożliwiające łatwą implementację zadania, jednak ze względu na ich fakt, że są one zamknięte, ciężko mierzyć za ich pomocą niektóre parametry. W związku z tym będziemy się posługiwać własną implementacją, wiernie odwzorowującą oryginalny problem.

Rozpocznijmy od opisania, w jaki sposób będziemy reprezentować planszę. Oczywistym pomysłem wydaje się być reprezentacja w postaci macierzy 5×5. Pola będziemy kodować następująco: jeśli pole jest puste odpowiadająca mu wartość w macierzy to 0, jeśli na polu znajduje się taksówka wartość jest zwiększana o 1, jeśli jest tam punkt startowy, zwiększamy jego wartość o 4, zaś jeśli punkt końcowy - o 2. Dzięki temu pola kodujemy poniekąd w systemie ósemkowym np. pole zwierające tylko taksówkę ma wartość 1, zawierające jednocześnie punkt startowy i końcowy - 6, zaś punkt startowy, końcowy i taksówkę - 7. Oczywiście przy takiej reprezentacji nie kodujemy w żaden sposób krawędzi między polami – tego, czy są zablokowane, czy nie. Wiemy jednak, które krawędzie są zablokowane (zawsze są to te

same krawędzie), więc sprawdzanie, czy można daną krawędź pokonać, czy zaprogramujemy w osobnej funkcji, która będzie pilnować zasad poruszania się po planszy.

Przejdźmy teraz do opisania przestrzeni akcji i przestrzeni stanów. Przestrzeń stanów to, podobnie jak w poprzednim problemie, zbiór wszystkich możliwych plansz (w tym wypadku jednak nie mamy plansz homeomorficznych). Innymi słowy, przestrzeń stanów to zbiór macierzy 5×5, których pola przyjmują wartości ze zbioru $\{0, 1, \ldots, 7\}$, przy czym tylko 4 wyróżnione pola mogą mieć wartość większą od 1 (są to rzecz jasne pola-stacje, bo tylko tam mogą być punkty startowe i końcowe). Wobec tego przestrzeń akcji składa się z $25 \cdot 5 \cdot 4 + 1 = 501$ elementów. W powyższej równości 25 to liczba możliwych lokalizacji taksówki, 5 to liczba możliwych punktów startowych – 4 odpowiadające polom-stacjom i 1 w przypadku braku punktu startowego, czyli po zabraniu przez taksówkę pasażera, 4 zaś to liczba możliwych punktów końcowych, dodane 1 odpowiada stanowi ostatecznemu, który nie odpowiada żadnej planszy i następuje po wysadzeniu pasażera w punkcie końcowym. Z przyczyn formalnych musi się on pojawić, jednak w programie nie będzie występował, czyli będziemy mieć 500 stanów.

Zauważmy w tym miejscu, że jeśli ponumerujemy (od 0, zgodnie ze standardami informatycznymi) wiersze, kolumny i pola-stacje (przy czym wprowadzimy dodatkowe, wirtualne pole-stację o numerze 4, odpowiadające niewybraniu żadnego prawdziwego pola stacji), to każdą planszę możemy utożsamiać z czwórką (r, c, s, e), gdzie r to numer wiersza, w którym znajduje się taksówka, c - numer kolumny, s - numer pola-stacji, na którym znajduje się punkt startowy, e - numer pola-stacji, na którym znajduje się punkt końcowy. Zgodnie z tym, co powiedzieliśmy wcześniej $r, c, s \in \{0, 1, 2, 3, 4\}$, $e \in \{0, 1, 2, 3\}$. Wobec tego, w programie przestrzeń stanów będziemy reprezentować jako tablicę czterowymiarową $5 \times 5 \times 5 \times 4$, której każdym elementem jest macierz odpowiadająca planszy kodowanej za pomocą współrzędnych.

Przestrzeń akcji to zbiór 6-elementowy, którego każdy element odpowiada jednej z wymienionych w poprzednim podrozdziale akcji. Warto zauważyć, że każda akcja może zmienić co najwyżej jedną współrzędną obecnego stanu.

Opiszmy jeszcze reprezentację tabeli wartości funkcji Q. Oczywistym pomysłem jest: $przestrzen\ stanow \times przestrzen\ akcji$, co oznaczałoby, że będzie wymiaru $5 \times 5 \times 5 \times 4 \times 6$. Pamiętajmy jednak, że nie mamy tutaj reprezentacji stanu ostatecznego, zaś zarówno Q-learning, jak i SARSA potrzebują wartości funkcji Q dla stanu po wybraniu akcji (czyli zabraknie nam w tabeli wartości dla stanu po ostatnim ruchu). Wobec tego dołożymy sztucznie ten stan, czyli współrzędna e również może przyjąć wartość 4, podobnie jak s, zaś tabela Q jest kształtu: $5 \times 5 \times 5 \times 5 \times 6$, przy czym nigdy nie zostanie zmieniona w niej wartość dla stanu, w którym $e = 4$.

5.3. Podejście do problemu

By sprawdzać skuteczność naszego rozwiązania, skorzystamy z tej metody zaproponowanej przez twórców. Zakłada ona, że algorytm nauczył się rozwiązywać zadanie Taxi-v1, gdy średni wynik ze 100 kolejnych prób jest nie mniejszy niż 9,7. Algorytmy użytkowników spełniają ten warunek w mocno zróżnicowanej liczbie prób - najlepszy wynik to 360 prób, najgorszy zaś to 177340. Przeciętny wynik to pomiędzy 800 a 1500 prób. Godne odnotowania jest spostrzeżenie, że oceniana jest tutaj wartość nagród otrzymywanych przez faktycznie działający algorytm. Ponieważ w naszym wypadku będzie on korzystać ze strategii ε-zachłannej, to niektóre akcje będą wybierane losowo, co może mocno wpływać na wyniki, musimy więc bardzo rozsądnie dobrać wartość ε.

Zanim jednak przejdziemy do faktycznego algorytmu, przeanalizujmy problem czysto teo-

retycznie i odpowiedzmy na pytanie, jaka jest średnia suma nagroda otrzymywana w pojedynczej, bezbłędnej próbie rozwiązania problemu. Rozważamy najpierw dla każdego pola na naszej planszy sumę najkrótszych dróg do każdego pola-stacji (np. dla pola w pierwszym wierszu i pierwszej kolumnie, czyli odpowiadającemu polu-stacji „R" suma ta wyniesie 17, bo mamy 0 do pola-stacji „R", 6 do pola-stacji „G", 4 do pola-stacji „Y" i 7 do „B" - w sumie 17). Wszystkie wyniki są zaprezentowane w poniższej tabeli:

17	17	17	17	19	= 87
15	15	15	15	17	= 77
15	15	15	15	17	= 77
17	19	19	17	19	= 91
19	23	23	19	21	= 105

Jak łatwo obliczyć, suma na wszystkich polach wynosi 437, wobec tego wartość oczekiwana liczby ruchów od lokalizacji początkowej do punktu startowego wynosi: $\frac{1}{4} \cdot \frac{1}{25} \cdot 437 = 4{,}37$ (mamy 4 możliwe punkty startowe i 25 możliwych lokalizacji startowych). Wyznaczmy teraz wartość oczekiwaną liczby ruchów potrzebnych na dotarcie z punktu początkowego do punktu końcowego. W tym celu wystarczy rozważyć sumę z pól-stacji (oznaczonych w tabeli na szaro). Jest to 74, zatem wartość oczekiwana dotarcia z punktu początkowego do końcowego wynosi $\frac{1}{4} \cdot \frac{1}{4} \cdot 74 = 4{,}625$ (mamy 4 możliwe punkty początkowe i 4 możliwe punkty końcowe). Wobec tego wartość oczekiwana liczby akcji na całej trasie to $4{,}37 + 4{,}625 = 8{,}995$. Dodajmy jeszcze 2 akcje, które musimy wykonać, czyli zabranie i wysadzenie pasażera. Wobec tego średnio musimy wykonać $8{,}995 + 2 = 10{,}995$ akcji w jednej próbie. Zatem średnio z jednej próby otrzymujemy nagrodę równą $20 - 10{,}995 = 9{,}005$. Jest to wartość istotnie mniejsza niż 9,7, która jest wymagana do zaakceptowania algorytmu jako poprawnego. Oznacza to, że nawet jeśli algorytm postępuje bezbłędnie, to i tak musi czekać na serię 100 prób, w których przeciętna minimalna liczba akcji jest mniejsza od średniej iliczby akcji w przeciętnej próbie. Oczywiście sytuacja taka musi nastąpić, jednak w przypadku dłuższego jej braku, może to poważnie zaburzyć ocenę w pełni wyuczonego algorytmu. Dodając do powyższej obserwacji fakt, że posługujemy się strategią ε-zachłanną, sugeruje, że powinniśmy dobrać ε bardzo zbliżony do 1, gdyż kara za wykonanie nieoptymalnej akcji jest bardzo wysoka (w przypadku błędnego wykonania akcji zabrania lub wysadzenia pasażera straty wynoszą 11 punktów - 10 za tę akcję i 1 za wybranie jakiejkolwiek akcji), co może serii, która potencjalnie będzie miała wymaganą średnią, obniżyć ją o zbyt wiele, aby stała się serią akceptującą.

5.4. Porównanie Q-learning i SARSA

Przejdźmy teraz do porównania algorytmów Q-learning i SARSA w zadaniu Taxi-v1. Na początku przypomnijmy wzory uaktualniające wartości funkcji Q dla Q-learningu (3.1):

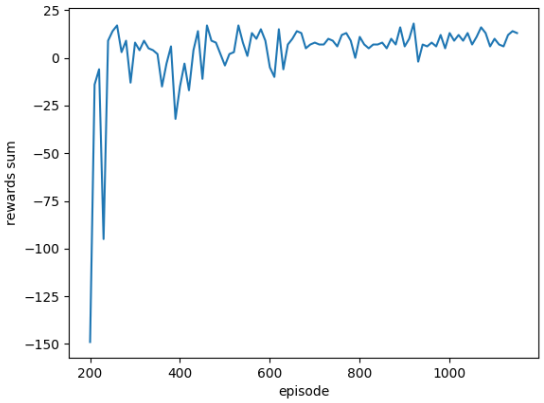
$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

i SARSA (3.3):

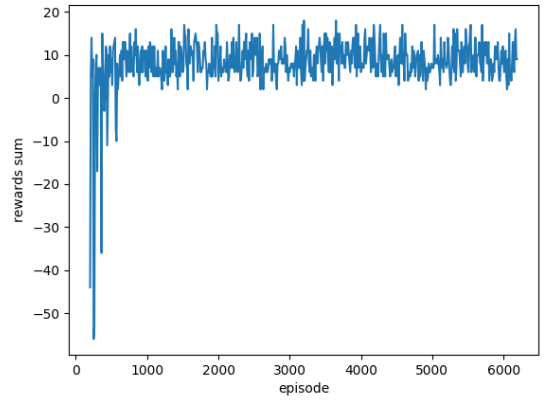
$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t))$$

Taxi-v1 jest problemem dość typowym i powyższe wzory stosujemy bez żadnych korekt. Przypomnijmy, że definiując tabelę funkcji Q, zadbaliśmy, by powyższe funkcje zawsze były dobrze określone.

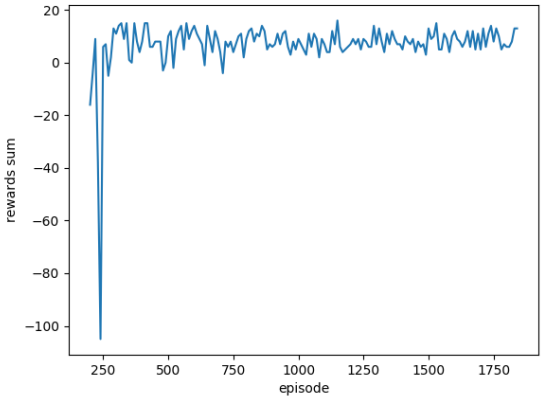
Ze względu na efekt omówiony w poprzednim rozdziale będzie bardzo ciężko doszukać się związku między wartościami parametrów innymi niż ε a tempem nauczenia się problemu (w myśl przyjętego kryterium), ani wskazać istotnych różnic, gdyż kluczowy dla spełnienia warunku „nauczenia się" będzie moment pojawienia się odpowiedniej serii warunków początkowych. W związku jednak z tym, że wartości ε są bardzo bliskie 1, nie należy się spodziewać istotnych różnic między Q-learningiem i SARSA (najmniejsza użyta wartość ε to 0,9999). Przyjrzyjmy się jednak trzem wybranym wykresom uzyskanym dla Q-learningu i trzem dla SARSA (na wykresach mamy przedstawioną sumę nagród dla próby w czasie). Dla zwiększenia przejrzystości wartości były obliczane co wyłącznie dla prób o numerze będącym wielokrotnością 10 oraz rozpoczynają się 200 próby. Wcześniejsze próby miały bardzo niskie sumy nagród, przez co niepotrzebnie zaciemniały wykres. Największy argument przedstawiony na wykresie odpowiada próbie, w której spełniono kryterium.



(a) Paramtery: $\varepsilon = 0{,}99993$, $\alpha = 0{,}44$, $\gamma = 0{,}75$

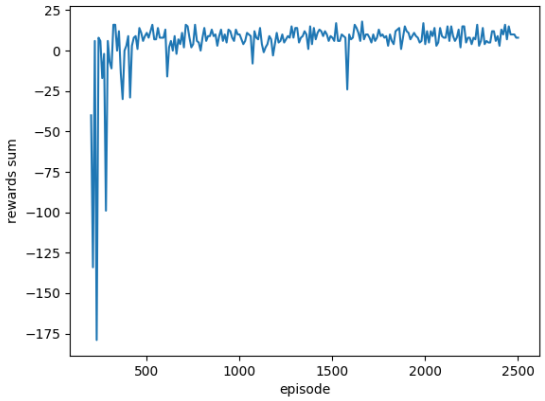


(b) Paramtery: $\varepsilon = 0{,}99995$, $\alpha = 0{,}6$, $\gamma = 0{,}7$

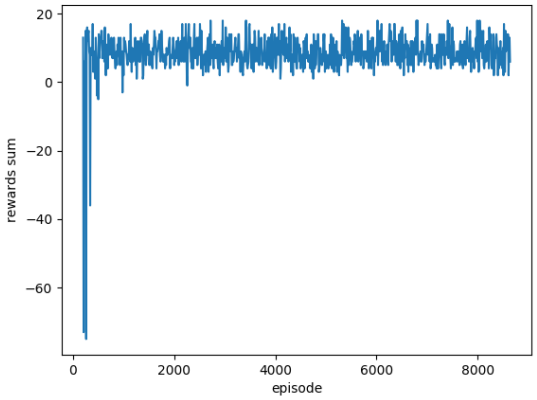


(c) Paramtery: $\varepsilon = 0{,}99995$, $\alpha = 0{,}6$, $\gamma = 0{,}85$

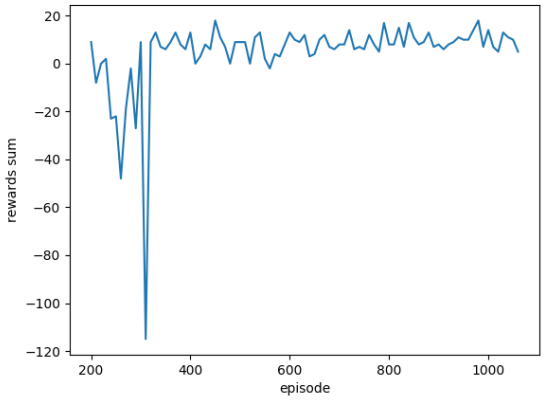
Rysunek 5.2: Wykresy sumy nagród od czasu dla Q-learningu



(a) Paramtery: $\varepsilon = 0{,}99992$, $\alpha = 0{,}4$, $\gamma = 0{,}8$



(b) Paramtery: $\varepsilon = 0{,}99993$, $\alpha = 0{,}52$, $\gamma = 0{,}92$



(c) Paramtery: $\varepsilon = 0{,}99995$, $\alpha = 0{,}6$, $\gamma = 0{,}9$

Rysunek 5.3: Wykresy sumy nagród od czasu dla SARSA

Mimo opisanego wyżej braku związku parametrów i tempa nauki, możemy z wykresów uzyskać pewne informacje. Pierwszą i najważniejszą jest fakt, że oba algorytmy się uczą i są raczej stabilne, jeśli chodzi o jakość rozwiązywania problemu. Zauważmy, że w zależności od układu początkowego, maksymalna wartość nagród waha się od 2 do 18, wobec czego wahania wykresu, zwłaszcza dla późniejszych prób, powinniśmy tłumaczyć raczej tą dużą rozbieżnością (zauważmy, że wartości wahają się właśnie w tym przedziale), nie zaś niedouczeniem programu. Co więcej, możemy powiedzieć, że począwszy od próby o numerze między 600 a 1000, program rozwiązuje problem bardzo skutecznie i optymalnie, a ewentualne, lokalne spadki sumy nagród powinniśmy tłumaczyć stosowaniem strategii ε-zachłannej i wynikającym z niej losowym (tutaj niekorzystnym) wyborem akcji.

Podsumowując, możemy stwierdzić, że zaproponowane programy są porównywalnie skuteczne z przeciętnymi rozwiązaniami użytkowników OpenAI Gym. Zauważmy, że stosowaliśmy tutaj najprostszą wersję Q-learningu i SARSA, co pozwalało już na początku sądzić, że nie uzyskamy tymi klasycznymi algorytmami równie dobrych wyników, co użytkownicy używający nowocześniejszych rozwiązań. Można również zastanowić się, czy nie należałoby obniżyć progu akceptacji algorytmu jako skutecznego do rzeczywistej średniej wartości $9,005$ lub nawet nieco niższej, ze względu na strategię ε-zachłanną.

Podsumowanie

Podsumowując uzyskane w pracy rezultaty, możemy uznać, że zarówno Q-learning, jak i SARSA są skutecznymi rozwiązaniami pewnych problemów z zakresu uczenia ze wzmocnieniem. Wspólną cechą obu problemów zaprezentowanych w pracy była mała ilość stanów, w których może znajdować się środowisko, w związku z tym wyznaczenie deterministycznie strategii postępowania w obu sytuacjach jest bardzo łatwe. Problemy te jednak miały być raczej próbą, na której sprawdzimy, czy teoretyczne algorytmy działają w praktyce. Pozwoliły one również sprawdzić, jaki jest sens poszczególnych parametrów w nich występujących oraz jakie skutki może mieć ich nieumiejętne dobranie.

Oczywistym jest, że uzyskane tutaj wyniki nie są czymś przełomowym, można by wręcz powiedzieć, że nie są one zbyt potrzebne, skoro te problemy można łatwo rozwiązać na kartce, bez wymyślnych algorytmów i teorii. Idąc krok dalej, można by pomyśleć, że cały koncept uczenia ze wzmocnieniem jest niepotrzebny, skoro rozwiązuje takie trywialne problemy. Okazuje się jednak, że uczenie ze wzmocnieniem, często w połączeniu z innymi działami uczenia maszynowego, daje skuteczne rozwiązania trudnych problemów, które przez lata wydawały się wręcz niemożliwe do rozwiązania. W miarę świeżym i bardzo spektakularnym przykładem może być program AlphaGo, który jest pierwszym w historii programem komputerowym, który pokonał arcymistrza w Go.

Oczywiście zaprezentowane tu algorytmy nie mogą się równać z AlphaGo, jednak fakt, że wychodzą ze wspólnego korzenia, jakim jest uczenie ze wzmocnieniem, czyni je godnymi uwagi. Są one bardzo blisko podstaw tej dziedziny i poznanie ich pozwala dobrze zrozumieć idee, które są w niej kluczowe. Zatem mimo, że nie posłużyły nam do rozwiązania wielkich problemów, pozwoliły dobrze poznać i przybliżyć jeden z ważniejszych konceptów w uczeniu maszynowym.

Dodatek A

Implementacja funkcji uaktualniającej wartość funkcji Q w problemie gry w kółko i krzyżyk w języku Python

- Q-learning:

```
def q_update(prev_action, action, mv_num, new_pos, curr_pos, prev_pos,
reward, qtable):
    curr_act = ACTIONS.index(action)
    prev_act = ACTIONS.index(prev_action)

    q_curr_mv_val = qtable[mv_num - 1][curr_pos][curr_act]
    q_prev_mv_val = qtable[mv_num - 2][prev_pos][prev_act]

    q_opt_val = cf.true_max(mv_num, new_pos, qtable)

    q_change_curr = reward + (-1) * GAMMA * q_opt_val
    q_change_prev = (-1) * reward + GAMMA * q_opt_val

    qtable[mv_num - 1][curr_pos][curr_act] += ALPHA *
    (q_change_curr - q_curr_mv_val)

    qtable[mv_num - 2][prev_pos][prev_act] += ALPHA *
    (q_change_prev - q_prev_mv_val)
```

oraz funkcja pomocnicza `true_max`:

```
def true_max(mv_num, pos_num, qtable):
    maxi = 0
    posibble_act = give_possible_actions(mv_num + 1, pos_num)
    for act in posibble_act:
        if qtable[mv_num][pos_num][act] > maxi:
            maxi = qtable[mv_num][pos_num][act]
    return maxi
```

- SARSA:

```
def q_update(prev_action, action, mv_num, new_pos, curr_pos, prev_pos,
reward, qtable):

    curr_act = ACTIONS.index(action)
    prev_act = ACTIONS.index(prev_action)

    next_act = cf.choose_action(mv_num + 1, new_pos, EPSILON, qtable)
    next_act_num = ACTIONS.index(next_act)
    next_act_val = qtable[mv_num][new_pos][next_act_num]

    q_curr_mv_val = qtable[mv_num - 1][curr_pos][curr_act]
    q_prev_mv_val = qtable[mv_num - 2][prev_pos][prev_act]

    q_change_curr = reward + (-1) * GAMMA * next_act_val
    q_change_prev = (-1) * reward + GAMMA * next_act_val

    qtable[mv_num - 1][curr_pos][curr_act] += ALPHA *
    (q_change_curr - q_curr_mv_val)
    qtable[mv_num - 2][prev_pos][prev_act] += ALPHA *
    (q_change_prev - q_prev_mv_val)

    return next_act
```

gdzie funkcja `choose_action` to funkcja wybierająca akcję.

Dodatek B

Implementacja funkcji uaktualniającej wartość funkcji Q w problemie Taxi-v1 w języku Python

- Q-learning:

```
def update_qtable(qtable, pos_row, pos_col, start, end, reward, act_num,
pos_row_new, pos_col_new, start_new, end_new, alpha, gamma):

    q_act_val = qtable[pos_row][pos_col][start][end][act_num]
    q_opt_val = cf.qtab_maxi(qtable, pos_row_new, pos_col_new,
    start_new, end_new)

    q_change = reward + gamma * q_opt_val

    qtable[pos_row][pos_col][start][end][act_num] += alpha *
   (q_change - q_act_val)
```

- SARSA:

```
def update_qtable(qtable, pos_row, pos_col, start, end, reward, act_num,
pos_row_new, pos_col_new, start_new, end_new, epsilon, alpha, gamma):

    q_act_val = qtable[pos_row][pos_col][start][end][act_num]
    act_num_new = cf.choose_action(pos_row_new, pos_col_new, start_new,
    end_new, epsilon, qtable)

    q_opt_val = qtable[pos_row_new][pos_col_new][start_new][end_new]
    [act_num_new]
    q_change = reward + gamma * q_opt_val

    qtable[pos_row][pos_col][start][end][act_num] += alpha *
    (q_change - q_act_val)
    return act_num_new
```

gdzie funkcja `choose_action` to funkcja wybierająca akcję.

Bibliografia

[SUS] Paweł Cichosz, *Systemy uczące się*, Wydawnictwo Naukowo-Techniczne, Warszawa 2007.

[RLAI] Richard S. Sutton, Andrew G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Massachusetts, London, England, 2012.

[ConOfQL] Francisco S. Melo *Convergence of Q-learning: a simple proof* , Institute for Systems and Robotics, rękopis.

[ConRes] Satinder Singh, Tommi Jaakkola, Michaeel L. Littman, Csaba Szepesvari, *Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms*, Machine Learning, March 2000, Volume 38, Issue 3.