UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **Informatyka**
Specjalność: -

Wiktor Wajszczuk
nr albumu: 296534

Zastosowanie uczenia przez wzmacnianie w środowisku agentowym

Application of reinforcement learning in an agent-based environment

Praca licencjacka
napisana w Katedrze Oprogramowania Systemów Informatycznych
pod kierunkiem dr Marcina Denkowskiego

Lublin, rok 2022

Spis treści

Wstęp

Jedną z większych dziedzin uczenia maszynowego jest uczenie przez wzmacnianie (ang. *reinforcement learning*) W odróżnieniu od zarówno uczenia nadzorowanego i nienadzorowanego nie potrzebujemy w tym przypadku żadnych gotowych danych wejściowych i wyjściowych. Zamiast tego, algorytm pozyskuje dane na bieżąco ze środowiska do, którego jest zastosowany. Dzięki temu, że algorytmy uczenia przez wzmacnianie nie mają tego ograniczenia możemy zastosować je do problemów takich jak gra na giełdzie [1], czy nauka grania w gry, w swojej pracy skupię się na tym drugim.

Celem pracy jest zaimplementowanie algorytmu uczenia przez wzmacnianie Q-Learningu. Następnie zoptymalizowaniu go tak by po zastosowaniu go do własnoręcznie zaimplementowanej gry był w stanie osiągnąć w niej możliwie najwyższy wynik. Do zrealizowania tego zostanie użyty język python oraz moduł pygame.

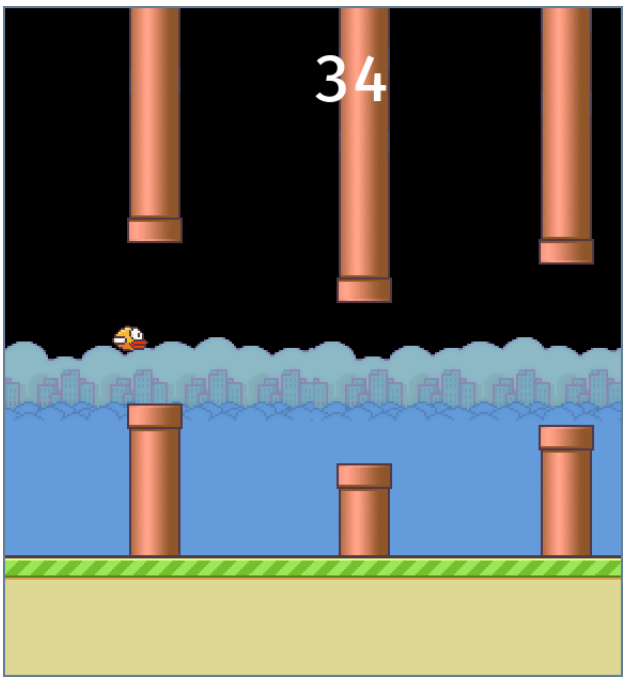
Na początku przedstawię jaką grę wybrałem, opiszę jej zasady oraz zaprezentuję szczegóły jej implementacji. Następnie przybliżę zagadnienie uczenia przez wzmacnianie oraz algorytmu Q-learning. Pod koniec pokażę jak zaimplementowałem wspomniany algorytm, oraz testy po optymalizacjach algorytmu, które zastosowałem.

Rozdział 1

Gra

Problem, który postawiłem przed Q-learningiem to nauczenie się grania w “Flappy Bird” – grę z 2013 autorstwa wietnamskiego developera Dong Nguyena [2]. Zdecydowałem się właśnie na tą grę, gdyż sterowanie w niej jest bardzo proste – jest tylko jeden przycisk do kontrolowania toteż jedna akcja do podjęcia przez algorytm.

1.1 Zasady gry



Rysunek 1.1: Zrzut ekranu z gry

Gracz kontroluje obiekt reprezentowany w grze jako żółty ptak, jego zadaniem jest tak nim sterować by omijać obiekty reprezentowane przez czerwone rury oraz nie pozwolić na to by siła grawitacji doprowadziła do kolizji obiektu ptaka z obiektem ziemi (na rysunku

1.1 zielona część na dole). Na obiekt ptaka działa jedynie siła grawitacji a obiekty rur przesuwane są w lewo w jego stronę ze stałą prędkością. Gracz ma możliwość kontrolować jedynie czy obiekt, którym steruje zwiększy swoją wysokość o stałą, zakodowaną ilość jednostek czy tego nie zrobi i obiekt reprezentowany przez ptaka zmniejszy swoją wysokość. Gra kończy się gdy dojdzie do kolizji obiektu ptaka z, którymkolwiek obiektem rur – dolnej bądź górnej albo z obiektem ziemi. Punkty przyznawane są jeżeli graczowi uda się ominąć nadchodzące przeszkody przelatując przez szczelinę między nimi. Za każde ominięcie przyznawany jest jeden punkt. Na górze rysunku 1.1 białymi cyframi wypisana jest ilość osiągniętych przez gracza punktów. Jak widać gracz ominął już trzydzieści czery pary rur. Podsumowując – gra sprowadza się do kontrolowania wysokości ptaka tak by ten przelatywał między nadchodzącymi rurami.

1.2 Wykorzystane technologie

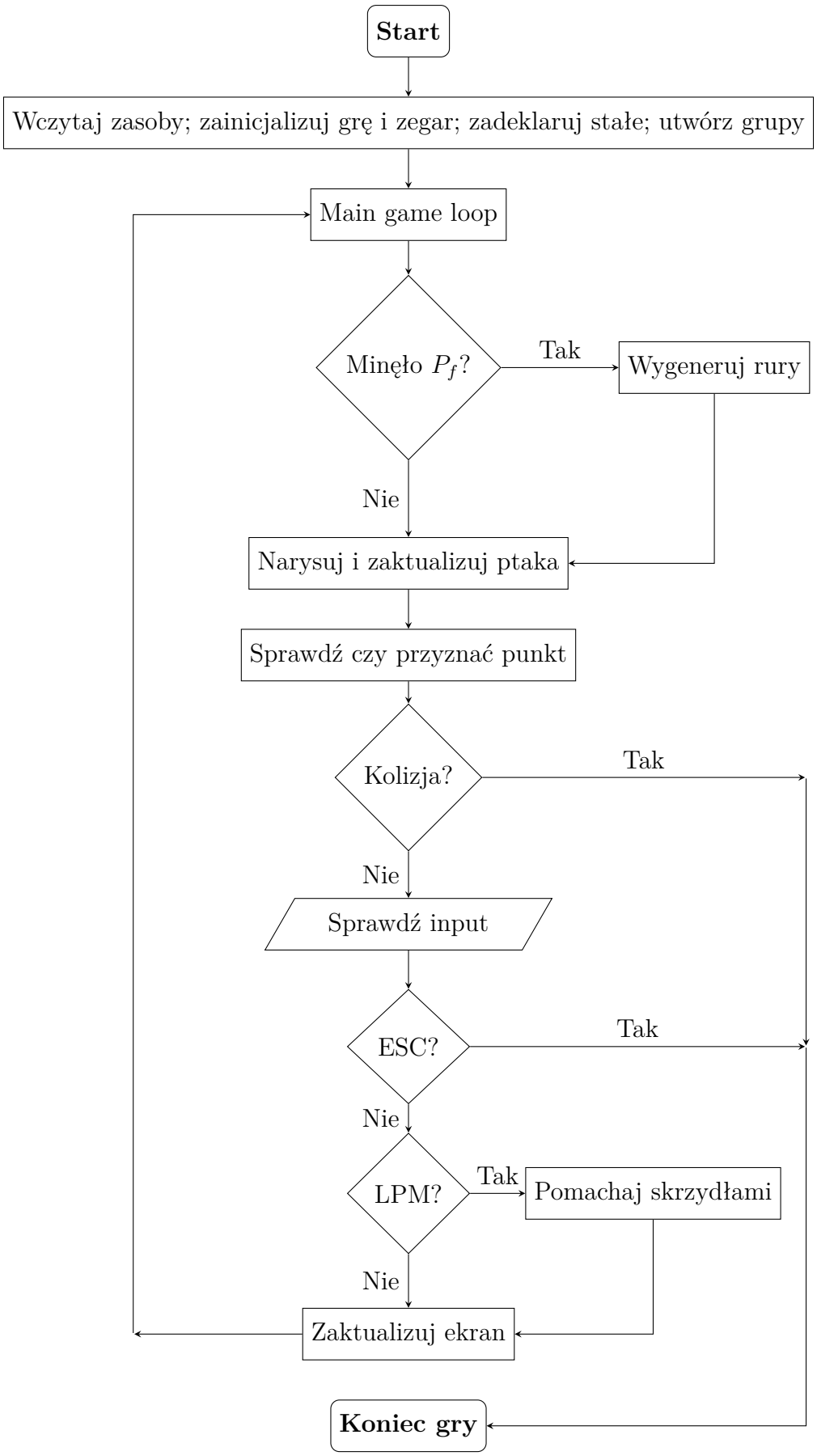
Do zaimplementowania tej gry zdecydowałem się na użycie wysoko poziomowego interpretowanego języka **python** w wersji 3.8. Python jest dostępny na większości współczesnych platform oraz jest to język open-source. Dodatkowo wykorzystałem moduł **pygame**.

Pygame dodaje funkcjonalność do biblioteki SDL. Simple DirectMedia Layer(SDL) jest to wieloplatformowa biblioteka zapewniająca nisko poziomowy dostęp do audio, urządzeń peryferyjnych takich jak mysz, klawiatura, joystick oraz sprzętu graficznego (za pomocą OpenGL i Direct3D). Technologia open source zaimplementowana w C. Używana w grach firm takich jak *Valve Software* czy *SuperTuxCart*[3]. To za pomocą tej biblioteki pygame może odczytywać wejście czy rysować wyjście na ekranie.

Pygame pozwala tworzyć w pełni funkcjonalne gry oraz programy multimedialne w pythonie. Zaletami pygame są między innymi : podstawowe funkcje używają zoptymalizowanego kodu w C oraz Assembly, dostępny podobnie jak python na większości współczesnych platform. Ponadto jest modularny co pozwala programiście używać tylko tych komponentów, których naprawdę potrzebuje. Gier stworzonych z użyciem pygame na samej stronie projektu jest ponad 660 [4].

1.3 Implementacja

Głównym elementem implementacji jest nieskończona pętla, w której sprawdzamy zdarzenia, które zaszły, sprawdzamy czy mamy wejście od gracza, i aktualizujemy obiekty gry oraz ekran. Na rysunku 1.2 opisana jako *Main game loop*.



Rysunek 1.2: Uproszczony diagram przepływu gry

Program zaczyna od wczytania wczytania zasobów gry – obrazków reprezentujących tło, rurę, ziemię, ptaka. Każdemu z tych zasobów tworzy obiekt klasy *pygame.image*, wczytywane są za pomocą metody *load*. Do śledzenia czasu w grze wykorzystywany jest obiekt klasy *pygame.time.Clock*.

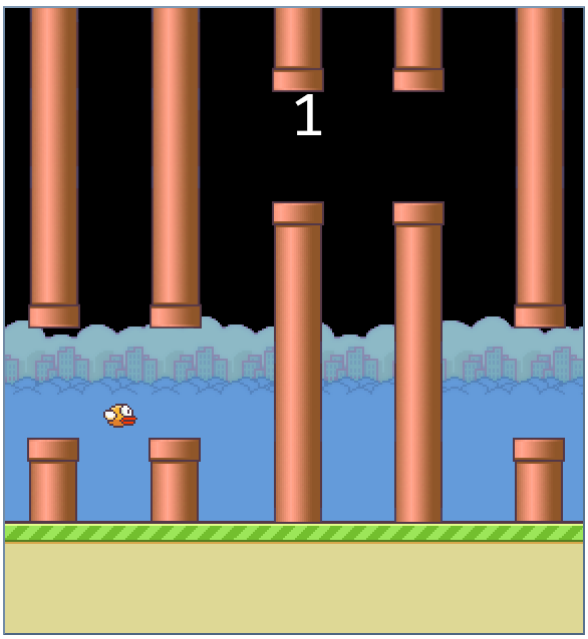
Stałe deklarowane na początku programu to na przykład: ilość jednostek o ile przesunąć w lewo obiekty rur przy każdym tyknięciu zegara, szerokość szczeliny między obiektami rur, czy czas po jakim generować nowe rury. W tym miejscu przypisujemy wartość stałej opisującej co jaki czas mamy generować nowe obiekty rur. Jest wyliczana ze wzoru:

$$P_f = 900 * (30 \div F) \tag{1.1}$$

Gdzie P_f to czas po jakim generowane są nowe obiekty rur (ang. *pipe frequency*), F to ilość klatek na sekundę (ang. *frames per second*). Potrzeba zmiennej częstotliwości generowania obiektów rur wzięła się z okoliczności, które opisuję w rozdziale Implementacja Q-learningu.

Przy uruchamianiu programu użytkownik ma opcję wybrać wartość parametru klatek na sekundę : 30, 60 bądź 120. Domyślnie jest to 30, stąd we wzorze znajduje się ta liczba. W ten sposób przy 30 klatkach na sekundę nowe obiekty rur będą generowane co 900 milisekund, przy 60 co 450 milisekund i co 225 milisekund w przypadku 120 klatek na sekundę. Po kilku próbach stwierdziłem, że częstsze generowanie obiektów rur sprawiało, że gra była zbyt trudna dla gracza lub w najgorszym wypadku sprawiało, że gra była nie do przejścia jak na przykład w scenariuszu przedstawionym na rysunku 1.3 gdzie obiekt ptaka nie będzie miał wystarczająco czasu by wznieść się na odpowiednią wysokość a potem opaść by ominąć rury. Częstsze generowanie obiektów rur nie ma wpływu przy większej ilości klatek na sekundę, gdyż te przemieszczają się w lewo co tyknięcie zegara o stałą ilość jednostek, a tykanie zegara uzależnione jest od ilości klatek na sekundę – większa ilość klatek na sekundę oznacza częstsze tykanie zegara przez co rury przemieszczają się z większą prędkością. Rozgrywka staje się zdecydowanie dynamiczniejsza i przez to również trudniejsza, ale tylko dla ludzi. Q-learningowy agent dostaje informacje o grze co tyknięcie zegara i jeszcze w tym samym tyknięciu podejmuje decyzje o ruchu, ale więcej o tym w rozdziale Implementacja Q-learningu

Po określeniu wartości stałych tworzymy trzy instancje klasy *pygame.sprite.Group*, jedną dla obiektu ptaka, drugą dla obiektów rur dolnych oraz jedną dla obiektów rur górnych. Będą nam potrzebne później przy sprawdzaniu czy zaszła kolizja między obiektem ptaka a obiektami rur. Sam obiekt ptaka jest instancją klasy *Bird* dziedziczącej po *pygame.sprite.Sprite*. Wspomniana klasa jest przeznaczona dla obiektów, które mają być widoczne na ekranie, zawiera między innymi pole na obrazek reprezentujący obiekt oraz metodę *update*[5], którą będę wywoływał co każdy obrót głównej pętli programowej. W przypadku obiektu ptaka w przeładowanej metodzie *update* znajduje się logika “latania”



Rysunek 1.3: Często generowane obiekty rur

– co wywołanie metody update inkrementuję zmienną która mówi o ile ma być przesunięty obiekt ptaka w dół do maksymalnej wartości ośmiu jednostek. Jeżeli został wciśnięty lewy przycisk myszy to wartość ta jest zmniejszana o dziesięć (minusowe wartości tej zmiennej powodują, że obiekt zostanie przesunięty w górę).

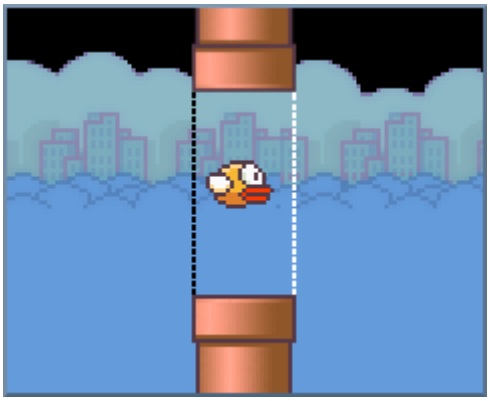
Po tym zaczyna się główna pętla programowa. Co P_f czasu generujemy nowe rury – polega to na stworzeniu dwóch instancji klasy *Pipe* dziedziczącej po *pygame.sprite.Sprite*. Generuję liczbę losową r z zakresu $(-100, 100)$, która będzie potrzebna w ustaleniu wysokości, na której wygeneruję obiekty rur. W konstruktorze nadaję obiektom odpowiedni obrazek. Po wczytaniu obrazku używam *get_rect* będącej metodą klasy *pygame.Surface*, której instancją jest nadany w konstruktorze obrazek. Zwrócny obiekt przypisuję do zmiennej klasy *rect*. Metoda ta mapuje obrazek na najmniejszy prostokąt, w którym zmieści się wczytany obrazek. Mamy dzięki temu dostęp do jego krawędzi, którym możemy nadawać współrzędne na ekranie, w których ma zostać narysowany. Wszystkie obiekty rysowane są w przestrzeni kartezjańskiej, gdzie punkt $(0, 0)$ jest lewym górnym rogiem tła. Górnej lewej krawędzi obiektu dolnej rury nadajemy współrzędne:

$$(W, H \div 2 + r + G \div 2)$$

Natomiast dolnej lewej krawędzi obiektu górnej rury nadajemy współrzędne:

$$(W, H \div 2 + r - G \div 2)$$

Gdzie W to szerokość tła ustalona na początku, H to wysokość tła, a G to stała oznaczająca odległośc między dolną a górną rurą, przypisywana na początku programu. Tak



Rysunek 1.4: Przyznawanie punktów

narsyowane obiekty rur zostaną narysowane zaraz za ekranem z prawej strony, przesunięte od środku ekranu x oraz połowę odległości ustalonej na początku działania programu. Tak wygenerowane obiekty dodaję do odpowiednich instancji *pygame.sprite.Group*. Jako, że klasa *Pipe* dziedziczy po *pygame.sprite.Sprite* (tak samo jak klasa *Bird*), to powinna przeładowywać metodę *update*[5]. W przypadku klasy reprezentującej rury jest tam tylko logika przesuwania obiektu w lewo co sprowadza się do zmniejszenia współrzędnej x zmiennej *rect* oraz ewentualnym usunięciu go jeżeli współrzędna ta po zmniejszeniu staje się ujemna (oznacza to, że jest po za lewą krawędzią ekranu przez co jest nie widocznia i nie potrzebna). Usuwanie polega na wywołaniu metody *kill*, co powoduje, że obiekt zostaje usunięty z każdej grupy, której był elementem.

W następnym kroku rysuję na ekranie obiekt ptaka za pomocą metody *draw* wywoływanej na grupie, do której ten obiekt należy. Metoda ta korzysta z nadanego w konstruktorze obrazka, oraz zmapowanego obiektu prostokąta (*sprite.Rect*) do określenia pozycji, na której ma zostać narysowany[6]. Rysowanie obiektów rur odbywa się w ten sam sposób, z tym, że wspomniana metoda jest wywoływana na odpowiednio grupie obiektów rur dolnych i górnych. Następnie na grupie, do której należy obiekt ptaka wywołuję metodę *update*, która obsługuje logikę "latania".

Najbliższym z prawej strony obiektem rury jest zawsze obiekt o indeksie zero z grupy dolnych bądź górnych rur, jest tak dzięki logice metody *update* klasy *Pipe*. Aby sprawdzić, czy przyznać graczowi punkt sprawdzamy najpierw, czy obiekt ptaka znalazł się w strefie między dwoma rurami, tak jak jest to przedstawione na rysunku **??**.

Rozdział 2

Uczenie przez wzmacnianie

Tutaj do napisania o tym czym jest uczenie przez wzmacnianie pare slow jakie są algorytmy jak działa, schemat środowisko - agent - akcja - reward

2.1 Q-learning

Tutaj wszystko związane z qlearningiem, wzory, itp :)

Rozdział 3

Implementacja Q-learningu

Tutaj fragmenty kodu, jakie zmiany dokonałem względem książkowego algorytmu

3.1 Podejście pierwsze grid 10 na 10

Po kolei wyjaśnienie co to grid 10 na 10 jak to jest implementowane itp

3.2 Podejście drugie grid 10 na 10 i flaga za śmierć od wysokiej rury

Jak wyżej

3.3 Podejście trzecie grid 5 na 5 z flagą

Jak wyżej

Rozdział 4

Podsumowanie

Wyniki osiągane są w miarę ok ale mogło by być lepiej bo są na internecie lepsze podejścia do tego problemu – być może wynika to z tego ze sam implementowałem grę i nie jest ona tak dobrze przetestowana jak ta dostępna na GitHub?

4.1 Co można by ulepszyć

Tutaj będę pisał co mogłem zrobić gdybym poświęcił więcej czasu np:

- uczenie bez wizualizacji pygamowej
- zrównoleglenie by paru agentów na raz mogło się uczyć
- doszlifowanie algorytmu??

Bibliografia

[1] K. Dabérius, E. Granat, and P. Karlsson, "Deep execution - value and policy based reinforcement learning for trading and beating market benchmarks," 2019. [Online]. Available: http://dx.doi.org/10.2139/ssrn.3374766

[2] "Strona internetowa dotgears." [Online]. Available: https://dotgears.com/games/flappybird

[3] "Strona internetowa sdl." [Online]. Available: https://www.libsdl.org/

[4] "Strona about projektu pygame." [Online]. Available: https://www.pygame.org/wiki/about

[5] *Dokumentacja klasy pygame.sprite.Sprite.* [Online]. Available: https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite

[6] *Dokumentacja metody pygame.sprite.Group.Draw.* [Online]. Available: https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.draw