AI do gry Urban Rivals: The Rift Mode

(AI for Urban Rivals game: The Rift Mode)

Marcin Martowicz

Praca licencjacka

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

20 września 2023

Streszczenie

Tematem pracy jest stworzenie autonomicznego agenta do trybu Rift karcianej gry Urban Rivals, której istotnymi cechami są losowość oraz niepełna informacja. W pracy opisane jest w jaki sposób bot wykorzystuje informacje dostępne na ekranie oraz numeryczne dane otrzymane od serwera gry do stworzenia obrazu aktualnego stanu gry. Zaprojektowanych zostało kilka parametryzowanych strategii dla każdego z dostępnych etapów rozgrywki. Użytkownik może wpływać na zachowanie bota przez modyfikację pliku konfiguracyjnego. Zostały także przeprowadzone eksperymenty testujące skuteczność dostępnych strategii w losowych rozgrywkach. Projekt został zrealizowany w wysokopoziomowym języku Python.

The goal of the thesis is to create an autonomous agent for the Rift mode of the card game Urban Rivals, whose essential features are randomness and incomplete information. The work describes how the bot uses the information available on the screen and numerical data received from the game server to create an image of the current state of the game. Several parameterized strategies have been designed for each of the available stages of the game. The user can influence the behavior of the bot by modifying the configuration file. Experiments were conducted to test the effectiveness of the available strategies in random games. The project was implemented in a high-level Python language.

Spis treści

1. Wprowadzenie **7**

1.1. Motywacja . . . 7

1.2. Wybór gry . . . 7

1.3. Działanie bota . . . 8

1.4. Plan pracy . . . 8

2. Zasady i etapy gry **9**

2.1. Ekran startowy . . . 9

2.2. Mapa . . . 10

2.3. Walka . . . 10

2.4. Dobieranie kart . . . 11

2.5. Dobór dodatków . . . 12

3. Metody uzyskiwania informacji i ich zastosowania **15**

3.1. Żądania API . . . 15

3.2. Analiza ekranu . . . 16

4. Sztuczna inteligencja **19**

4.1. Strategia dla etapu walki . . . 19

4.1.1. Strategia zamiany kart . . . 20

4.2. Strategia dla etapu mapy . . . 21

4.3. Strategia dla etapu dobierania kart . . . 23

4.4. Strategia dla etapu ulepszania kart . . . 23

4.5. Strategia dla etapu wybierania dodatków . . . 24

5. Wykonanie projektu **25**

5.1. Wybór języka 25

5.2. Logowanie wydarzeń 26

5.3. Struktura projektu 26

6. Część dla użytkownika **29**

6.1. Zastosowanie programu 29

6.2. Uruchmianie programu 29

6.3. Plik konfiguracyjny 30

7. Eksperymenty **33**

8. Podsumowanie **35**

Bibliografia **37**

Rozdział 1.

Wprowadzenie

1.1. Motywacja

Sztuczna inteligencja jest wszechobecna w dzisiejszym świecie. Wiele prac wykonywanych kiedyś przez człowieka może teraz robić maszyna. W kontekście gier, boty można wykorzystać zarówno w celu zautomatyzowania rzeczy monotonnych, jak i próby grania w nie perfekcyjnie. Głównym rodzajem gier, do których napisanie bota stanowi wyzwanie są gry strategiczne, które zawierają zbyt dużo parametrów, by matematyczna funkcja celu była obliczalna w sensownym czasie. W takich przypadkach ludzie osiągają zazwyczaj lepsze wyniki, dzięki intuicyjnemu zrozumieniu konkretnych pozycji. Celem pracy jest stworzenie wszechstronnego bota do trybu Rift strategicznej gry Urban Rivals [1].

1.2. Wybór gry

Wybrana gra jest przeznaczona dla ludzi, to znaczy, że nie jest specjalnie stworzona, by programować do niej boty. W trybie Rift gracz nie ma styczności z żywymi przeciwnikami i przechodzi przez całą rozgrywkę samodzielnie. Elementy w grze mają charakter statyczny, to znaczy nie ma ruszających się obiektów, które byłyby istotne do robienia ruchów i cała interakcja odbywa się poprzez klikanie myszką. Ważnym aspektem trybu jest wiele możliwych celi, które gracz może chcieć osiągnąć, co wymusza możliwość konfiguracji zachowania bota. Gra ma charakter strategiczny i stworzenie bota, który jednocześnie osiąga dobre wyniki i dostosowuje się do potrzeb użytkownika jest nietrywialne. Tryb Rift zawiera kilka różnych etapów, dzięki czemu struktura bota musi być bardziej rozbudowana.

1.3. Działanie bota

Bot do gry Urban Rivals gra zamiast człowieka i obowiązują go takie same zasady jak normalnych graczy. Oprócz zrozumienia jak przebiegają konkretne etapy rozgrywki, musi on również potrafić je rozpoznawać i poruszać się pomiędzy nimi, w szczególności musi być w stanie startować grę po przegranej. Robi to w taki sam sposób jak człowiek, klika w odpowiednie miejsca myszką. Jednak zwykli użytkownicy gry czerpią informację o grze tylko na podstawie tego co widzą na ekranie. Bot będzie również korzystał z danych dostarczanych przez serwer gry. Niektóre z nich nie są bezpośrednio przekładane na ekran, co daje mu możliwość na lepsze zrozumienie pozycji. Tryb Rift charakteryzuje się niejednoznacznym celem rozgrywki, dzięki czemu różne osoby mogą chcieć używać bota do zdobywania różnych nagród w grze. Dlatego dla każdego etapu została zaprojektowana parametryzowana strategia, której parametry użytkownik może ustawiać w pliku konfiguracyjnym.

1.4. Plan pracy

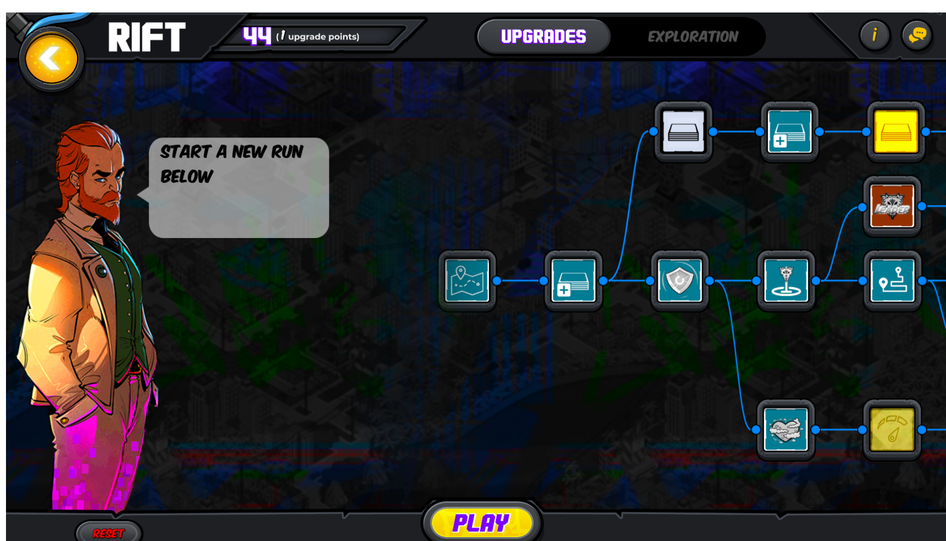
W rozdziale drugim przedstawiono zasady gry i poszczególne etapy występujące w rozgrywce. W rozdziale trzecim opisano dwie metody uzyskiwania informacji o grze - analizę ekranu i zapytania API wraz z ich wykorzystaniem w bocie. W rozdziale czwartym zostały szczegółowo omówione strategie dla każdego etapu rozgrywki. W rozdziale piątym wyjaśniono zalety użytego języka programowania oraz techniczną strukturę projektu. W rozdziale szóstym podano użytkowe instrukcje dotyczące bota, w tym jego uruchomienie oraz wyjaśnienie wszystkich pól w pliku konfiguracyjnym. W rozdziale siódmym przedstawiono wyniki przeprowadzonych eksperymentów. W rozdziale ósmym zawarto ogólne wnioski o projekcie.

Rozdział 2.

Zasady i etapy gry

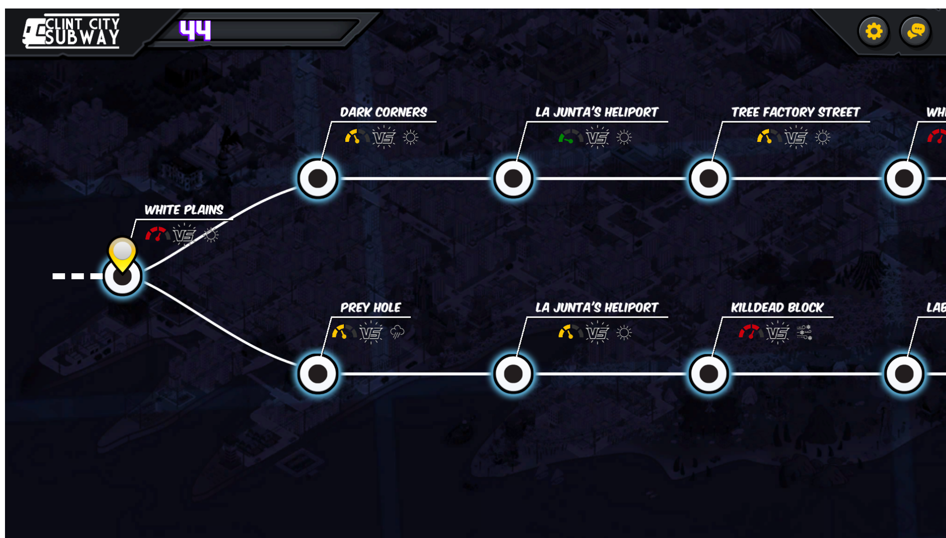
Urban Rivals jest karcianą grą strategiczną, w której większość trybów rozgrywki jest w stylu pvp (player vs player). Dla odmiany, w trybie Rift gracz nie ma styczności z żywymi przeciwnikami i w izolacji przechodzi przez tak zwane biegi. Na początku każdego biegu zaczyna się z losową talią siedmiu kart. Przechodzi się po kolejnych stacjach, w których walczy się z przeciwnikiem w losowym środowisku. Następuje seria rund, aż do momentu, kiedy punkty zdrowia jednej ze stron spadną do zera. Po drodze można ulepszać swoją talię i zbierać dodatkowe nagrody. W kolejnych sekcjach znajduje się szczegółowy opis etapów tej rozgrywki.

2.1. Ekran startowy



Z ekranu startowego możemy rozpocząć nowy bieg. Znajduje się tu również obecny poziom gracza i drzewo talentów. Każdy uzyskany poziom pozwala odblokować jedno ulepszenie w drzewie, które wpływa na bardziej różnorodną rozgrywkę i silniejsze dodatki w kolejnych biegach.

2.2. Mapa



Na mapie trzeba przejść do jednej z kilku sąsiednich stacji. Dla każdej z nich widać niewielkie, osiągalne otoczenie. Na danej stacji trzeba zmierzyć się z przeciwnikiem. Następuje seria walk, aż do momentu, kiedy poziom życia jednej ze stron spadnie do zera. Jeśli gracz przegra, bieg się kończy. W przeciwnym wypadku, wraca się do tego ekranu i wybiera kolejny punkt podróży. Kolory przybliżają trudność środowiska. Czasami na mapie pojawiają się unikatowe ścieżki, których przejście daje rzadkie nagrody. Przejście stacji daje doświadczenie, które wpływa na poziom gracza.

2.3. Walka

W fazie walki odbywa się siedem pojedynków przeciwległych kart. W każdym

pojedynku jest określona szansa na wygraną. Generalnie wzór procentowy na zwycięstwo karty o mocy X z kartą o mocy Y to $50+X-Y$, ale zmienne środowiskowe mogą na niego wpływać. Jeśli wartość tego wyrażenia jest mniejsza od 0 to prawdopodobieństwo wynosi 0, symetrycznie jest gdy wartość jest większa od 100. Strona, która wygra większość pojedynków zadaje obrażenia stronie przegranej zależne od liczby wygranych i jakości kart. Jeśli przegrany posiada punkty tarczy, to one w pierwszej kolejności niwelują część obrażeń. Celem gracza jest optymalne ułożenie jego talii. Jest w stanie zamieniać miejscami dowolną parę swoich kart. Zmienne środowiskowe działające na określone pozycje mogą wprowadzać dodatkowe efekty:

- modyfikacja mocy karty
- szansa wygrania danego pojedynku jest stała
- szansa wygrania jest ukryta
- unieważnienie bonusu lidera, jeśli karta jest liderem
- karty nie można zamieniać

2.4. Dobieranie kart

Po fazie walki gracz ma możliwość ulepszenia swojej talii. Może wyrzucić jedną ze swoich kart i zastąpić ją jedną z oferowanych. Każda karta ma określoną moc, która wpływa na wyniki w pojedynkach. Dodatkowo posiada obecny i maksymalny poziom, każdy dający dodatkowe 20 mocy. Liczba oferowanych kart w biegu jest ograniczona i informacja o ilości pozostałych kart do wyboru jest dostępna w lewej części ekranu. Każda karta należy do jednej z sześciu frakcji. Posiadanie 3 i 6 kart z danej frakcji daje dodatkowy bonus:

frakcja	co najmniej 3 karty	co najmniej 6 kart
Guardians	+30 tarczy po przegranej	+30 maksymalnego życia po przegranej
Urbans	+10% wygrania dla kart z tej frakcji	+10 mocy dla kart tej frakcji
Activists	+1 dodatek do wyboru	+1 dodatek do wyboru
Psychos	-50 życia przeciwnika po wygranej	-50 maks. życia przeciwnika po wygranej
Supernaturals	+1 poziom dla sąsiednich kart	+30 maksymalnego życia
Technophiles	+5% wygrania dla sąsiednich kart	+1 dodatek do wyboru

Dodatkowo istnieją specjalne karty, zwane liderami, których posiadanie również daje bonus:

lider	bonus
Timber	+1 poziom do wszystkich kart
Eklore	+15% na lepszą rzadkość dodatków
Hugo	+10% szansy wygrania dla wszystkich kart
Vholt	×2 punktów tarczy
Vansaar	×1.5 doświadczenia
Morphun	+30 kart do wyboru
Kate	×2 nagród na koniec biegu

2.5. Dobór dodatków

Po fazie zamiany kart gracz wybiera jeden z oferowanych dodatków. Każdy z nich ma jeden z pięciu poziomów rzadkości, który wpływa na ilość dodawanej cechy. Niektóre z nich trzeba odblokować w drzewie ulepszeń. Możliwe dodatki to:

- dodanie poziomu do wybranej karty

- dodanie poziomu do losowej karty
- zwiększenie mocy wybranej karty
- zwiększenie mocy losowej karty
- zwiększenie punktów tarczy
- zebranie nagrody, których ilość wpływa na końcowe nagrody po zakończeniu biegu
- dodanie punktów zdrowia
- zabranie punktów zdrowia przeciwnikowi
- dodanie punktów zdrowia i zwiększenie maksymalnego zdrowia
- odblokowanie w bieżącym biegu specjalnej stacji `ARCADES`, której przebycie daje dodatkowe nagrody

Przeciwnik na danej stacji również wybiera w tej fazie dodatek, przez co jego talia też jest ulepszana przed kolejną rundą.

Rozdział 3.

Metody uzyskiwania informacji i ich zastosowania

Trzema głównymi encjami występującymi w grach internetowych są gracz, aplikacja i serwer gry. Gracz komunikuje się tylko z aplikacją, poprzez odczytywanie stanu gry z ekranu oraz wykonywanie ruchów myszką i klawiaturą. Aplikacja, która jest na komputerze gracza komunikuje się również z serwerem gry przez sieć. Wysyła ona ruchy gracza na serwer i otrzymuje od niego nowy stan gry, który następnie wyświetla na ekranie. Opisywany w pracy bot wykorzystuje zarówno informacje wyświetlane na ekranie, jak i dane dostarczone od serwera gry.

3.1. Żądania API

Każdy serwer internetowy wystawia zbiór akcji, które może obsłużyć tzw. API (Application Programming Interface). Gra Urban Rivals oferuje wewnętrzne oraz zewnętrzne API. Zewnętrzne API służy do uzyskiwania ogólnych informacji, takich jak statystyki gracza i ceny kart na rynku. Wewnętrzne API jest zdefiniowane w celu obsługiwania postępu w rozgrywce i jest używane przez aplikacje w dwie strony – żeby informować serwer o działaniach gracza i dostawać od niego szczegółowe informacje na temat etapu, na którym gracz się znajduje. Do tego używany jest elastyczny protokół HTTP warstwy aplikacji.

Bot używa jednego żądania API w celu pobierania informacji o obecnym stanie gry. Odpowiedź jest dostarczana w formacie `json` i zawiera szczegółowe informacje i statystyki o obecnym biegu. Dane z formatu `json` przekształcane są do klas w Pythonie w celu wyodrębnienia danych, które rzeczywiście są używane, jak i zamienienia danych liczbowych na napisy czytelne dla człowieka (np. identyfikatory frakcji na ich nazwy). Na ich podstawie wykonywane są ruchy w konkretnych etapach rozgrywki.

Zazwyczaj znaczenie otrzymanych pól można wywnioskować przez analizę konkretnych pozycji. W przypadku, gdy nie do razu wiadomo jaki jest dokładny związek pomiędzy obiektami na ekranie a otrzymanymi danymi, można wykorzystać bota do zapisywania szczegółowych informacji o interesujących sytuacjach do pliku, puszczając program na kilka godzin i następnie ręcznie przeanalizować zebrane dane.

3.2. Analiza ekranu

Gracz będący człowiekiem wszystkie informacje o grze odczytuje z ekranu. Program komputerowy nie ma zdolności rozumienia co się na nim znajduje. Dla niego ekran jest dwuwymiarową tablicą pikseli, każdy mający określony kolor zakodowany jako trójka liczb całkowitych w przedziale $[0, 255]$ (format RGB). Biblioteka `Open Computer Vision` [2] oferuje wiele algorytmów związanych z grafiką komputerową i jest tam dostępna metoda wyszukiwania wzorca w obrazie. Algorytm polega na przyłożeniu wzorca w każdy możliwy obszar obrazu i obliczenia ich podobieństwa. Używając tej metody można stwierdzić czy obecnie na ekranie występuje dany obraz. Szukany wzorzec musi mieć dokładnie takie same wymiary, co wymusza, że użytkownicy bota muszą grać w trybie pełnoekranowym na ustalonej rozdzielczości. Jeśli wiadomo, że wzorzec zawsze znajduje się w tym samym miejscu (np. jak przyciski w grze), można przyspieszyć jego wyszukiwanie, szukając go w zawężonym obszarze ekranu.

Wcześniejsza wersja bota korzystała tylko z zapytań API. W większości przypadków z otrzymanych danych można było wywnioskować jaki obecnie jest etap (np. jeśli pole `picks` było niepuste, wtedy na pewno był etap doboru kart). Wyjątkiem była faza z mapą i walką – podczas obu faz zarówno pole `map` i `player_hand` było niepuste, przez co nie można było bezpośrednio stwierdzić jaki jest obecnie etap. Rozwiązaniem tego problemu było przeanalizowanie przebiegu rozgrywki i wyłapanie warunków, po których był etap mapy. Początkowo tymi sytuacjami były: bezpośrednio po doborze lidera i po etapie wyboru dodatków, jeśli `ai_life` to 0. Później okazało się, że jest inna rzadka sytuacja, która występowała w przypadku zajścia daleko w biegu. Pokazuje to ogólny problem nieznajomości niezmienników, które spełniają odebrane dane. W trybie Rift każdy etap można rozpoznać po unikalnym obiekcie na ekranie i obecna wersja bota wykorzystuje to, by dać użytkownikom gwarancję poprawnego działania.

Bot po zrobieniu ruchu musi odczekać pewną ilość czasu, aż do momentu zakończenia animacji i załadowania nowego etapu. W pierwotnej wersji bota, odczekiwał on określoną ilość sekund zależną od etapu, która na pewno to gwarantowała. Czekając na unikalny obrazek charakterystyczny dla danej fazy możemy czekać precyzyjnie tyle ile trzeba. To rozwiązanie działa niezależnie od tego czy gra się na aplikacji czy przeglądarce internetowej, gdzie mogą występować różnice. Dodatkowo uodpornia to bota w przypadku, gdy gra się zatnie, np. z powodu dużej ilości procesów w systemie

lub problemu z internetem.

Kolejnym zastosowaniem jest możliwość poznania koloru obiektu na ekranie. Po kolorze jesteśmy w stanie rozpoznać frakcję, do której należy karta, poziom rzadkości dodatku oraz poziom trudności stacji na mapie. Ostatniej z wymienionych informacji nie ma w danych API i obecnie bot sczytuje ją z ekranu.

Rozdział 4.

Sztuczna inteligencja

Kiedy wiadomo jaka jest obecnie faza i stan rozgrywki, bot musi wybrać jeden z wielu możliwych ruchów. Jako, że użytkownik może chcieć osiągnąć jeden z kilku celi gry, bot nie powinien zawsze zachowywać się tak samo na danym etapie. Z drugiej strony, losowe ruchy nie są na ogół dobre i niekoniecznie prowadzą do ustalonego celu. Obecnie każdy etap ma zaimplementowaną jedną parametryzowaną strategię, która umożliwia elastyczne sterowanie botem. Użytkownik modyfikuje parametry w pliku `config.json`. Bot po uruchomieniu czyta ten plik i gra zgodnie z zaleceniami.

4.1. Strategia dla etapu walki

Przy ustalonych wynikach pojedynków, to znaczy wiedząc kto wygrał, nie wiadomo jaki jest dokładnie wzór na zadawane obrażenia. Dlatego celem w tej fazie jest maksymalizowanie prawdopodobieństwa wygrania co najmniej 4 pojedynków. Przeiterujmy się po wszystkich permutacjach naszych kart, licząc dla każdej powyższe prawdopodobieństwo. Odrzucamy wszystkie permutacje, które nie zgadzają się ze zmiennymi środowiskowymi, dotyczącymi zakazu zamiany karty i anulowaniu bonusu lidera. Po nałożeniu efektów reszty zmiennych środowiskowych oraz bonusów frakcji i lidera, możemy obliczyć ostateczną moc każdej karty i otrzymać szansę wygrania dla każdego pojedynku. Niech $winrate_i$ to prawdopodobieństwo zwycięstwa w i-tym pojedynku. Rozważmy problem wygrania większości pojedynków dla liczby kart n.

Prawdopodobieństwo wygrania większości pojedynków można wyrazić wprost z definicji jako:

$$P = \sum_{\substack{X \subseteq \{1,2,\ldots,n\} \\ |X| > \frac{n}{2}}} \prod_{i \in X} winrate_i \prod_{j \notin X} (1 - winrate_j)$$

Korzystając bezpośrednio z podanego wzoru, czas obliczenia P to $O(2^n n)$, co nie jest zadowalające, ponieważ musielibyśmy to powtórzyć dla każdej permutacji

kart. Kluczową obserwacją do szybszego wyznaczenia P jest fakt, że znając wyniki w i pierwszych pojedynkach, nie jest istotny dokładny zbiór zwycięzców, tylko ich liczba. Z pomocą przychodzi technika programowania dynamicznego. Niech $p[i][k]$ to prawdopodobieństwo, że spośród i pierwszych pojedynków zwycięży k naszych kart. Będziemy wypełniać tę tablicę dla kolejnych i od 1 do n. Obliczając $p[i][k]$ musimy rozważyć dwa przypadki:

- i-ta karta przegrała, wtedy spośród $i-1$ pierwszych pojedynków musi być k zwycięstw
- i-ta karta wygrała, wtedy spośród $i-1$ pierwszych pojedynków musi być $k-1$ zwycięstw

Dostajemy zatem następujący wzór:

$$p[i][k] = \begin{cases} 1 & i = 0, k = 0 \\ 0 & i = 0, k > 0 \\ winrate_i \cdot p[i-1][k-1] + (1 - winrate_i) \cdot p[i-1][k] & i > 0, k > 0 \\ (1 - winrate_i) \cdot p[i-1][k] & i > 0, k = 0 \end{cases}$$

Korzystając z tego obliczamy P:

$$P = \sum_{k=\lceil \frac{n}{2} \rceil}^{n} p[n][k]$$

Złożoność programowania dynamicznego to $O(n^2)$, więc czas całego algorytmu to $O(n!n^2)$. W naszym przypadku n to 7, więc cały algorytm jest wystarczająco szybki.

4.1.1. Strategia zamiany kart

Wiedząc jakie powinno być ustawienie kart, musimy wykonać serię zamian, by je tak ustawić. Jako, że każda zamiana kart nie jest natychmiastowa i powoduje animacje, powinniśmy zminimalizować ich liczbę. Niech p_i będzie pozycją, na której powinna znaleźć się karta znajdująca się na i-tej pozycji; p jest permutacją. Wprost z definicji, zamiana kart na pozycjach i i j odpowiada zamianie p_i i p_j. Celem jest doprowadzenie p do permutacji identycznościowej. Rozważmy ten problem w ogólności dla dowolnej liczby kart n. W rozwiązaniu skorzystamy z faktu, że każda permutacja rozkłada się na cykle.

Lemat 1. *Cykl $(x_1\ x_2\ \dots\ x_n)$ o długości n można doprowadzić do permutacji identycznościowej w $n-1$ ruchach.*

Dowód. Dowód przez indukcję matematyczną.
Dla $n = 1$ jest to oczywiste.

Dla $n > 1$ zamieńmy ze sobą elementy x_1 i x_2. Powstaną dwa cykle: (x_2) o długości 1 i $(x_1\ x_3\ ...\ x_n)$ o długości $n-1$. Z założenia indukcyjnego cykl o długości $n-1$ można doprowadzić do permutacji identycznościowej w $n-2$ ruchach, więc cały cykl można w $n-1$ ruchach. □

Twierdzenie 1. *Optymalna liczba ruchów to $n - |cykle(p)|$, gdzie $cykle(p)$ to zbiór cykli permutacji p.*

Dowód. Po pierwsze udowodnijmy, że w takiej liczbie ruchów permutację p można doprowadzić do permutacji identycznościowej. Aplikując powyższy lemat do każdego cyklu otrzymamy permutację identycznościową. Liczba ruchów to $\sum_{c \in cykle(p)}(|c| - 1) = n - |cykle(p)|$.
Po drugie udowodnijmy, że jest to ograniczenie dolne. Zauważmy, że zamieniając i i j, które należą do tego samego cyklu, liczba cykli zwiększa się o 1, a w przeciwnym przypadku zmniejsza się o 1. Zatem liczba cykli rośnie o co najwyżej 1 po jednym ruchu. Permutacja identycznościowa ma n cykli, a początkowa $|cykle(p)|$, zatem każda sekwencja operacji musi mieć co najmniej rozmiar $n - |cykle(p)|$. □

Twierdzenie daje prostą konstrukcję ciągu zamian, która jest wykorzystywana w bocie.

4.2. Strategia dla etapu mapy

Definicja 1. *Drzewo to graf, w którym pomiędzy dowolną parą wierzchołków istnieje dokładnie jedna ścieżka prosta.*

Graf na mapie jest skierowanym drzewem, gdzie korzeń jest obecną lokalizacją i ruchy można wykonywać tylko do swoich dzieci. Mapę zakodowaną w formacie `json` można sparsować do postaci, gdzie wierzchołek zawiera atrybuty i listę swoich sąsiadów. Mimo tego, że czasami wszystkie właściwości stacji na ekranie są ukryte, dane API zawsze je zawierają, co daje botowi dodatkowe informacje, które nie są dostępne dla zwykłych graczy.

Pierwszą własnością stacji jest jej rzadkość. Ta informacja jest bezpośrednio dostępna w danych API pod nazwą `rarity`. Przejście stacji o dodatniej rzadkości daje dodatkowe nagrody, więc mając taką możliwość zawsze opłaca się do nich udać.

Drugą własnością jest trudność stacji, którą przybliża kolor znajdujący się koło jej nazwy. Jest to jedna z niewielu informacji, której nie ma w danych API. Dodatkowo cała mapa nie mieści się na ekranie i trzeba się po niej przesuwać myszką, co utrudnia bezpośrednie sczytanie kolorów wszystkich stacji. Sprawdziłem kilka hipotez, dotyczących powiązania trudności stacji z dostępnymi danymi i wyszło, że trudność jest zawsze taka sama dla danego zbioru identyfikatorów elementów tablicy `stage_variable_sets` przypisanej do stacji w danych API. Sprawdzenie tego

przypuszczenia zostało zrealizowane przez zapisywanie przez bota powiązań kolorów ze zbiorem identyfikatorów dla bieżącej stacji do pliku i następnie przeanalizowanie otrzymanych danych. Bot dalej zbiera te dane, ponieważ mapowanie to jest duże i z każdym przebytym biegiem dochodzą nowe informacje. Dzięki temu mapowaniu możemy otrzymać atrybut `color` dla każdego wierzchołka w grafie. Obecnie informacje te są zapisywane i odczytywane z pliku, ale można by było użyć bazy danych.

Gdy zaczynałem implementować powyższą memoizację myślałem, że obiekt z kolorem obecnej stacji jest zawsze na tej samej pozycji względem jej symbolu. Testując napisany program, napotkałem niepoprawne kolory przechodzonych stacji. Pomocne było poruszenie kursora w momencie, kiedy jest pobierany kolor, dokładnie w obserwowany piksel. Okazało się, że w zależności od długości nazwy stacji obiekt może być lekko przesunięty. Rozwiązaniem tego problemu jest analizowanie kolorów pikseli w prostokącie, który na pewno zawiera szukany kolor.

Na podstawie koloru i rzadkości trzeba podjąć decyzje, do której stacji się udać. Dla każdego wierzchołka wyznaczymy najbardziej opłacalną ścieżkę z niego wychodzącą. Po pierwsze interesuje nas największa rzadkość występująca na ścieżce, w przypadku remisów chcemy wybrać ścieżkę z łatwiejszymi stacjami. Pierwszym pomysłem na wagę ścieżki jest suma wag trudności jej stacji. Użytkownik mógłby dla każdego koloru dobrać wagę przez plik konfiguracyjny. Jako, że po przejściu stacji kolejne są odsłaniane, więc naturalnym spostrzeżeniem jest, że istotniejsze są wagi, które występują wcześniej. Obecnie wzór na wagę ścieżki $(u_0, u_1, ..., u_n)$ jest następujący

$$W(u_0, u_1, ..., u_n) = \sum_{i=0}^{n} w(u_i) \cdot C^i,$$

gdzie C to liczba w $[0, 1]$ ustalana w pliku `config.json`. Przydatna jest następująca własność

$$W(u_0, u_1, ..., u_n) = w(u_0) + C \cdot W(u_1, ..., u_n).$$

Procedura chodząca po drzewie wygląda następująco.

```
def dfs(self, node):
    # max_rarity, path_weight, who_is_next
    best = None

    for i, next in enumerate(node.edges):
        path = self.dfs(next)
        current = (path[0], path[1], i)
        if best is None:
            best = current
        else:
            best = min(best, current)

    if best is None:
        best = (0, 0, 0)
    best = (
        min(best[0], -node.rarity),
        self.c_constant * best[1] + self.weight(node),
        best[2],
    )
    return best
```

4.3. Strategia dla etapu dobierania kart

Użytkownik specyfikuje jaką talią chce, żeby bot grał. Określa lidera i liczności kart dla każdej z frakcji. Podczas dobierania kart, sprawdzana jest każda możliwa podmiana oraz pominięcie fazy. Dla każdej wynikowej talii jest obliczana jej siła. W tym celu zdefiniujmy potencjał karty:

$$potential(card) = power(card) + 10 \cdot (max_level(card) - level(card))$$

Wybrana została stała 10, ponieważ zwiększenie poziomu karty daje o 10 więcej mocy niż wybranie zwykłego dodatku dodającego moc. Niech n_i oznacza ile kart i-tej frakcji użytkownik oczekuje. Podzielmy karty należące do każdej frakcji na dwa zbiory. Niech $wanted_i$ to zbiór n_i kart o największym potencjale występujących w talii i należących do i-tej frakcji, a $unwanted_i$ to reszta kart tej frakcji.

Siła talii jest zdefiniowana jako krotka czterech elementów:

1. czy talia posiada oczekiwanego lidera
2. $\sum_i |wanted_i|$
3. $\sum_i \sum_{card \in wanted_i} potential(card)$
4. $\sum_i \sum_{card \in unwanted_i} potential(card)$

Strategia wybiera krotkę największą leksykograficznie.

4.4. Strategia dla etapu ulepszania kart

W przypadku, kiedy zostanie wybrany dodatek dodający poziom lub moc, należy wybrać kartę, którą się ulepsza. Strategia pozwala wybrać użytkownikowi liczbę kart, które będą ulepszane. Nazwijmy tę liczbę *boost_size*. Strategia została zainspirowana taktyką najlepszych graczy, ulepszających jedynie 4 najmocniejsze karty, które same wygrywały walki. Z założenia będę one mieć dużo mocy i powinny wygrywać pojedynki. Zdefiniujmy

$$wanted = \sum_i wanted_i,$$

gdzie $wanted_i$ zostało zdefiniowane w poprzedniej sekcji. Skoro karty nienależące do tego zbioru zostaną w przyszłości zamienione, nie opłaca się ich ulepszać. Niech *better_wanted* to zbiór *boost_size* kart z *wanted* o największym potencjale, a *worse_wanted* reszta kart z *wanted*.

W przypadku kiedy dodatek to zwiększenie mocy wybieramy kartę o najmniejszym potencjale z *better_wanted*, jako że chcemy żeby wszystkie te karty były o zbliżonej mocy.

W przypadku kiedy dodatek to zwiększenie poziomu mamy trzy przypadki:

- jeśli *better_wanted* posiada kartę, która nie ma maksymalnego poziomu, wybieramy taką o największym potencjale, ponieważ z dużą szansą zostanie ona w talii na zawsze
- jeśli *worse_wanted* posiada kartę, która nie ma maksymalnego poziomu, wybieramy taką o największym potencjale, ponieważ skoro wszystkie karty z *better_wanted* mają maksymalne poziomy, to minęła już duża ilość rund i karty w tym zbiorze też nie powinny się zmieniać; warto wybierać ten dodatek, bo gdy wszystkie karty mają maksymalne poziomy to dodatki zwiększające poziom nie będę oferowane
- jeśli oba zbiory są puste, to strategia wybierania dodatków nie wybierze dodatku zwiększającego poziom

4.5. Strategia dla etapu wybierania dodatków

Użytkownik przypisuje dla każdego dodatku wagę oznaczającą jak często ten dodatek powinien być wybierany. Jest to metoda dająca dużą swobodę. Niech $w(d)$ będzie wagą dodatku d. Jako, że każdy dodatek może mieć kilka poziomów rzadkości, ostateczna waga konkretnego dodatku wyraża się wzorem:

$$w'(d) = w(d) \cdot 10^{rarity(d)}$$

Niech D będzie zbiorem oferowanych dodatków. Prawdopodobieństwo wybrania i-tego wyraża się wzorem:

$$p_i = \frac{w'(d_i)}{\sum_{d \in D} w'(d)}$$

Dodatkowo z początkowego zbioru dodatków odrzucane jest dodawanie mocy i poziomu, jeśli strategia ulepszania kart ocenia, że jest to nieopłacalne. Poza tym, w pliku konfiguracyjnym jest pole `health_treshold`, oznaczające że dodatki dodające zdrowie są odrzucane, gdy procentowa ilość obecnego zdrowia jest większa od `health_treshold`.

Rozdział 5.

Wykonanie projektu

5.1. Wybór języka

Projekt został napisany w wysokopoziomowym języku Python [3]. Charakteryzuje się on prostą i logiczną składnią, przez co ilość kodu potrzebna do wykonania danego zadania jest mniejsza niż w przypadku używania innych języków. Jest to najpopularniejszy język ogólnego przeznaczenia i posiada obszerną kolekcję bibliotek pokrywającą praktycznie wszystko, czego można chcieć użyć.

Najważniejsze biblioteki użyte w projekcie:

- `requests` umożliwia prostą obsługę żądań HTTP; gwarantuje, że w jednej sesji kolejne żądania HTTP są robione w jednym połączeniu TCP, co jest ważne, bo bot wysyła żądania co kilka sekund
- `opencv`[2] oferuje efektywne algorytmy związane z grafiką komputerową; bot wykorzystuje metodę znajdowania wzorca w obrazie
- `keyboard` daje dostęp do klawiatury i umożliwia czekanie na wciśnięcie konkretnych klawiszy
- `pyautogui` pozwala na zmianę pozycji kursora i klikanie myszką
- `mvss` umożliwia zrobienie zrzutu bieżącego ekranu i zamienienie go na format RGB
- `pyinstaller` daje możliwość stworzenia pliku wykonywalnego z projektu napisanego w Pythonie; może to być przydatne do prostej dystrybucji bota

Projekt został sformatowany zgodnie z `PEP 8` [4] - oficjalnym dokumentem twórców Pythona odnośnie wyglądu kodu, przy pomocy narzędzia `black`. Dodatkowo zostały użyte anotacje typowe, które pozwalają na wykrycie błędów przed uruchomieniem programu. Było to szczególnie użyteczne, ponieważ wiele komponentów w projekcie trudno jest testować w izolacji.

5.2. Logowanie wydarzeń

Bot wypisuje na standardowe wyjście zdarzenia wykonane podczas grania. Pojedyncza wiadomość zawiera czas, poziom zdarzenia i treść. Poziom `INFO` odnosi się wysokopoziomowo do wykonywanych ruchów i etapów w grze, zaś poziom `DEBUG` zawiera szczegółowe informacje działań wykonywanych przez konkretne strategie. W trakcie gry, twórca bota może na drugim ekranie śledzić wypisywane wiadomości, upewniając się, że wykonywane ruchy zgadzają się z wybranymi strategiami.

```
→ ~/uwr/thesis/UrbanRivals/rift_api_player git:(main) ✗ sudo -E python3 rift_player.py
08/18/2023 03:44:48 [DEBUG] Login https done.
08/18/2023 03:44:49 [DEBUG] Login API done.

      Urban Rivals Rift Player
      Required resolution: 1920x1080
      Press 's' to start a bot.
      Hold 'q' to quit.

08/18/2023 03:44:49 [INFO] Config loaded.
08/18/2023 03:45:12 [INFO] Bot started by 's' key press.
08/18/2023 03:45:15 [INFO] New run.
08/18/2023 03:45:16 [INFO] Card(HATICE, ACTIVISTS, 30, 1/3) was exchanged to Card(HUGO, LEADERS, 50, 1/5)
08/18/2023 03:45:20 [DEBUG] Map memoizator: station_id = '', color_name = 'YELLOW'.
08/18/2023 03:45:20 [DEBUG] Map strategy: rarity = 0, size = 0, where_to_go = 0
08/18/2023 03:45:24 [INFO] First road was choosen on the map.
08/18/2023 03:45:25 [INFO] Fight.
08/18/2023 03:45:25 [DEBUG] Fight strategy: [100, 100, 80, 80, 90, 80, 90]
08/18/2023 03:45:25 [DEBUG] Permuter: Swaping card 0 and 1.
08/18/2023 03:45:27 [DEBUG] Permuter: Swaping card 0 and 5.
08/18/2023 03:45:29 [DEBUG] Permuter: Swaping card 2 and 6.
08/18/2023 03:45:30 [DEBUG] Permuter: Done.
08/18/2023 03:45:39 [INFO] Card(WYRE, GUARDIANS, 30, 1/3) was exchanged to Card(HYDRAEREVA, TECHONOPHILES, 72, 2/5)
08/18/2023 03:45:43 [DEBUG] Perk choice: [Perk(MAX_HEALTH_POINTS, 30), Perk(POWER, 10), Perk(SHIELD, 60), Perk(REWARDS, 1)]
08/18/2023 03:45:43 [INFO] Chose Perk(POWER, 10).
08/18/2023 03:45:44 [INFO] Boosted Card(LOOCIO, URBANS, 30, 1/3).
08/18/2023 03:45:47 [INFO] Fight.
08/18/2023 03:45:48 [DEBUG] Fight strategy: [100, 80, 90, 100, 100, 90, 90]
08/18/2023 03:45:48 [DEBUG] Permuter: Swaping card 0 and 4.
08/18/2023 03:45:50 [DEBUG] Permuter: Swaping card 0 and 6.
08/18/2023 03:45:51 [DEBUG] Permuter: Swaping card 0 and 2.
08/18/2023 03:45:53 [DEBUG] Permuter: Swaping card 0 and 1.
08/18/2023 03:45:55 [DEBUG] Permuter: Swaping card 0 and 3.
08/18/2023 03:45:57 [DEBUG] Permuter: Done.
08/18/2023 03:46:05 [INFO] Card(JONATTAN, PSYCHOS, 40, 1/2) was exchanged to Card(DAN, GUARDIANS, 34, 1/3)
08/18/2023 03:46:09 [DEBUG] Perk choice: [Perk(RANDOM_STARS, 1), Perk(POWER, 10), Perk(RANDOM_POWER, 10), Perk(REWARDS, 1)]
08/18/2023 03:46:09 [INFO] Chose Perk(RANDOM_POWER, 10).
08/18/2023 03:46:13 [DEBUG] Map memoizator: station_id = '', color_name = 'YELLOW'.
08/18/2023 03:46:13 [DEBUG] Map strategy: rarity = 0, size = 0, where_to_go = 0
08/18/2023 03:46:16 [INFO] First road was choosen on the map.
08/18/2023 03:46:17 [INFO] Bot finished by 'q' key press.
→ ~/uwr/thesis/UrbanRivals/rift_api_player git:(main) ✗
```

5.3. Struktura projektu

W głównym katalogu projektu znajdują się:

- `images/` - katalog ze zdjęciami obiektów w grze
- `map_memoization/` - katalog zawierający logikę do memoizowania kolorów stacji na mapie
- `strategies/` - katalog zawierający strategie do poszczególnych etapów
- `experiments/` - katalog zawierający logikę do tworzenia eksperymentów
- `clicker.py` - moduł oferujący funkcjonalność klikania w obiekty z gry
- `game_state.py` - moduł parsujący dane API do klas w Pythonie

- `permute_cards.py` - moduł zajmujący się wykonaniem serii zamian kart
- `rift_player.py` - moduł, który uruchamia bota
- `screen_recognition.py` - moduł oferujący funkcjonalność czekania na określone obiekty na ekranie
- `ursession.py` - moduł do wykonywania zapytań API

Rozdział 6.

Część dla użytkownika

6.1. Zastosowanie programu

Program oferuje autonomicznego bota do trybu Rift gry Urban Rivals. Użytkownik może specyfikować używaną strategię za pomocą pliku konfiguracyjnego `config.json`. Bot umożliwia opcję restartu biegu po jego zakończeniu, przez co można zostawiać go samego na dowolny okres czasu. Podstawowym zastosowaniem jest zautomatyzowanie zdobywania poziomów w celu odblokowania wszystkich ulepszeń w drzewie talentów. Można go również wykorzystywać do zbierania nagród lub po prostu do przechodzenia po jak największej liczbie stacji, by zdobywać punkty eksploracji.

6.2. Uruchmianie programu

Na początku należy zainstalować język Python, minimalnie wersję 3.9. Następnie stworzyć środowisko wirtualne, używając polecenia `python3 -m venv .venv` i je uruchomić używając `. .venv/bin/activate`. Zainstalować w nim zależności za pomocą `pip install -r requirements.txt`. W przypadku Linuxa z X Window System może być konieczne użycie polecenia `xhost +`. W zależności od systemu operacyjnego bota uruchamiamy poleceniem:

- Windows: `python3 ./rift_player.py`
- Linux: `sudo -E python3 ./rift_player.py`

Następnie należy upewnić się, że rozdzielczość jest ustawiona na 1920×1080, wejść w pełnoekranowy tryb gry i wystartować bota klikając klawisz `s`. Bota można zatrzymać przytrzymując klawisz `q` przez kilka sekund.

6.3. Plik konfiguracyjny

Przykładowy plik konfiguracyjny `config.json` wygląda następująco:

```
{
    "USERNAME": "username",
    "PASSWORD": "password",
    "RESTART": true,
    "AUTOSTART": false,
    "LEADER": "HUGO",
    "PICK_FACTIONS": {
        "URBANS": 6
    },
    "PERK_WEIGHTS": {
        "REWARDS": 1,
        "STARS": 40,
        "RANDOM_STARS": 5,
        "POWER": 20,
        "RANDOM_POWER": 20,
        "HEALTH_POINTS": 10,
        "OPP_HEALTH_POINTS": 0,
        "MAX_HEALTH_POINTS": 10,
        "SHIELD": 0,
        "ARCADES_WAY": 10000000
    },
    "HEALTH_TRESHOLD": 40,
    "BOOSTING_SIZE": 4,
    "MAP_C_CONSTANT": 0.95,
    "MAP_COLOR_WEIGHTS": {
        "GREEN": 1,
        "YELLOW": 5,
        "RED": 30
    }
}
```

Znaczenie pól:

- `USERNAME i PASSWORD` - nazwa i hasło użytkownika, używane w celu autentykacji do zapytań API o stan gry
- `RESTART` - czy automatycznie startować kolejne biegi
- `AUTOSTART` - czy wystartować bota bez czekania na wciśnięcie klawisza `s`
- `LEADER` - nazwa lidera, którym chcemy grać lub `NONE`

- `PICK_FACTIONS` - określenie ile kart danej frakcji chcemy (domyślna ilość to 0)
- `PERK_WEIGHTS` - określenie wag dla dodatków (domyślna waga to 0)
- `HEALTH_TRESHOLD` - procent życia, poniżej którego brane są po uwagę dodatki zwiększające zdrowie
- `BOOSTING_SIZE` - ilość ulepszanych kart, zgodnie ze strategią do etapu ulepszania kart
- `MAP_C_CONSTANT` - stała używana do obliczania wagi ścieżki w etapie mapy
- `MAP_COLOR_WEIGHTS` - wagi dla kolorów do obliczania wagi ścieżki w etapie mapy

Rozdział 7.

Eksperymenty

Skuteczność bota można mierzyć różnymi metrykami. W tym eksperymencie celem było przejście jak największej liczby stacji, ponieważ im dalej w rozgrywce tym większe rzadkości dodatków, lepsze karty z unikatowych ścieżek oraz większe doświadczenie po przejściu stacji. Dla wybranych dwóch talii kart z ustalonym liderem Hugo zostały przetestowane wartości `boosting_size` od 5 do 7. Reszta parametrów konfiguracyjnych była taka sama dla wszystkich wariantów i jest dostępna w katalogu `experiments/`. Dla każdej konfiguracji został przeprowadzony jeden bieg.

talia kart	`boosting_size`	liczba stacji	liczba walk	czas
6×Urbans	5	186	1378	5h 7m
6×Urbans	6	104	435	1h 48m
6×Urbans	7	78	283	1h 17m
3×Guardians + 3×Activists	5	248	1884	6h 54m
3×Guardians + 3×Activists	6	247	1360	5h 23m
3×Guardians + 3×Activists	7	264	1068	4h 16m

Tabela 7.1: Wyniki eksperymentów

Zgodnie z oczekiwaniami konfiguracje z mniejszymi wartości `boosting_size` potrzebują więcej walk na danej stacji i przez to spędzają więcej czasu na ich przejście. W tabeli nie została pokazana ilość unikatowych ścieżek które zostały przebyte, ponieważ na ten moment bot nie jest w stanie rozróżnić, jaką rzadkość ma dana ścieżka, więc sama ilość mogłaby być myląca. Najlepsza konfiguracja z talią 3×Guardians + 3×Activists z `boosting_size` równym 7 doszła do najbardziej rzadkiej mitycznej ścieżki i przeszła łączenie 10 unikatowych ścieżek, co dało 40 nowych kart. Oznacza to, że z każdym takim biegiem dostępne karty będą coraz lepsze i otrzymywane wyniki również będą się poprawiać.

Rozdział 8.

Podsumowanie

Zamierzone funkcjonalności agenta zostały zrealizowane. Zaprojektowane strategie dają oczekiwane rezultaty i bot osiąga dobre wyniki, porównywalne z graczami ludzkimi, którzy nie specjalizują się w trybie Rift. Program jest gotowy do używania przez graczy, a jego dalszy rozwój może być motywowany przez ich propozycje ulepszeń.

Bibliografia

[1] Acute Games. Urban Rivals. https://www.urban-rivals.com, 2006.

[2] Intel. Open Computer Vision. https://opencv.org/, 2000.

[3] Guido van Rossum. Python Programming Language. https://www.python.org/, 1991.

[4] Guido van Rossum. PEP 8. https://peps.python.org/pep-0008/, 2001.