Heroes of Might and Magic 3: Bot – opis projektu

Sprawozdanie z przedmiotu

Zintegrowane Systemy Decyzyjne

Grupa 4 – HoM3 Bot

Czerwiec 2022

1. Wstęp

Celem projektu grupowego było stworzenie bota, który byłby w stanie samodzielnie prowadzić rozgrywkę w grze Heroes of Might and Magic 3 grając dowolnym zamkiem oraz bohaterem. HoMM3 należy do gatunku strategicznych gier turowych i polega ona na rozbudowie własnego zamku, rozwoju bohaterów i przejmowaniu zamku/ów przeciwnika, co prowadzi do jego pokonania. Nasz bot docelowo miał za zadanie wygrywać z wrogiem na poziomie medium. Wersja gry użyta do zrealizowania projektu: HoMM3 – Hota HD.

2. Przegląd podobnych rozwiązań

2.1. VCMI

VCMI jest silnikiem typu open-source dla gry Heroes of Might and Magic III napisanym w języku C++. Celem projektu VCMI jest przepisanie całego środowiska HOMM 3 od podstaw, dając mu nowe i rozszerzone możliwości. Projekt aktualnie nie jest ukończony, wciąż wymaga wielu elementów i funkcji do pełnej funkcjonalności.

Udostępniony zestaw deweloperski do rozwoju VCMI jest słabo udokumentowany, ponadto jego niektóre funkcje przeszkadzałyby naszemu zespołowi w rozwoju własnego bota. Z tego powodu zdecydowaliśmy się porzucić pomysł zastosowania wymienionego rozwiązania.

2.2. The Application of Co-Evolutionary Genetic Programming and TD Reinforcement Learning in Large-Scale Strategy Game VCMI (Wilisowski, Dreżewski)

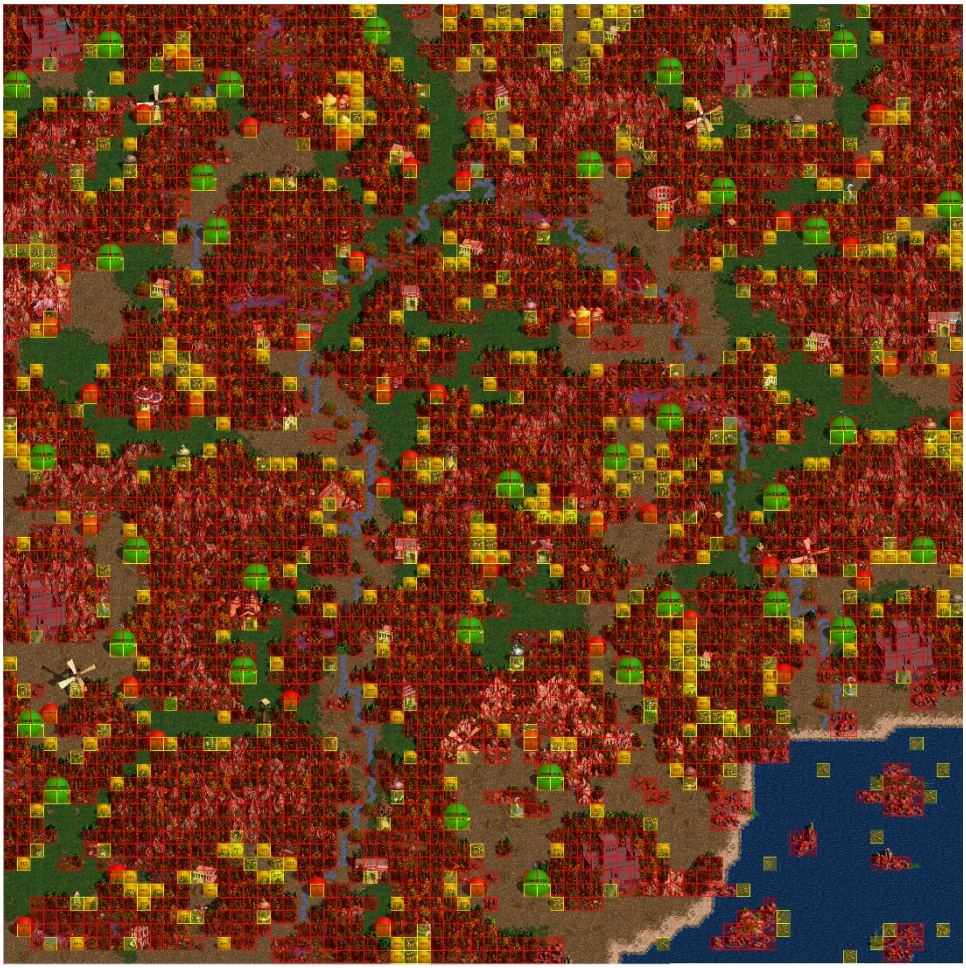
Artykuł proponujący rozwiązanie bitwy poprzez m.in. uczenie ze wzmocnieniem oraz zastosowanie projektu VCMI, który zapewniał środowisko testowe poprzez wprowadzenie modyfikowalnego agenta do bitwy. Ciekawym elementem pracy było stworzenie systemu ewaluacji ruchów na podstawie estymowanych wyników akcji, którymi były na przykład: procent jednostek zasięgowych chronionych czy liczba

pokonanych przeciwników. W podobny sposób my zrealizowaliśmy bitwę w naszym projekcie.

3. Nasze rozwiązanie

3.1. Przetwarzanie obrazu

Nasze rozwiązanie opiera się przede wszystkim na przetwarzaniu obrazu z powodu decyzji tworzenia całego środowiska od podstaw. W konsekwencji częścią naszej pracy było wyciągnięcie obrazów z gry, stworzenie dataset'u oraz ustalenie konkretnych algorytmów do rozpoznawania ich w oknie gry.



Rys. 1. Mapa "Darwin's Prize" używana w projekcie

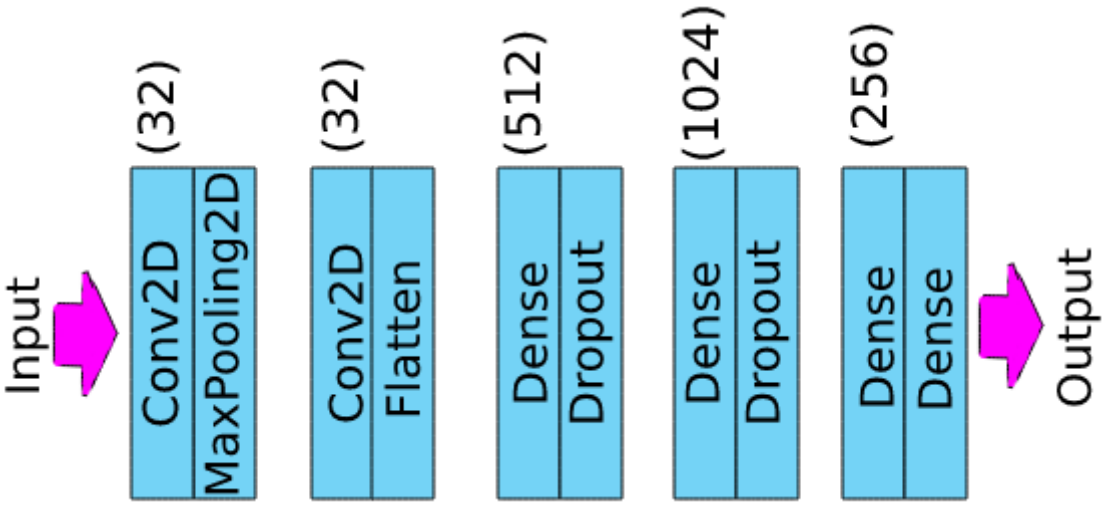
Na rysunku 1 znajduje się mapa, której lekko zmodyfikowaną wersję używamy do wypełniania i odkrycia niektórych elementów na naszej wewnętrznej mapie, która jest reprezentowana przez macierz 2D obiektów. Rozpoznajemy tylko obiekty, które nie są czerwone (przeszkody) oraz losowe części mapy, czyli te, które zmieniają się przy każdym resecie rozgrywki.

3.1.1. OCR (Optyczne rozpoznawanie znaków)

Jest to rozwiązanie fundamentalne dla naszego projektu. Wszelka obsługa okienek i pop-upów wymagała od nas rozpoznawania znaków, żeby wyciągnąć informacje na temat stanu gry. Do tego zadania użyliśmy biblioteki **pytesseract** - po każdym ruchu naszego agenta cyklicznie wywołujemy skrypty używające OCR i na podstawie otrzymanych informacji podejmujemy decyzje. W niektórych przypadkach OCR wspieramy również techniką **template matching** z biblioteki **OpenCV** do innych części okienka z interesującą nas częścią obrazu gry. Godnym uwag skryptem używającym technologię OCR jest **hero_filling.py**, który zczytuje z okna bohatera dane i uzupełnia je w naszym środowisku (wypełnianie naszej klasy typu Hero).

Same losowo odkrywane części mapy również wykorzystują OCR, ponieważ wiemy z naszej mapy gdzie znajdują się obiekty, ale nie wiemy czym są. Ustalenie obiektu wygląda następująco - najeżdżamy na dane pole myszką (zasługa biblioteki **pyautogui**) i za pomocą OCR odczytujemy nazwę obiektu, która wyświetla się na dole ekranu.

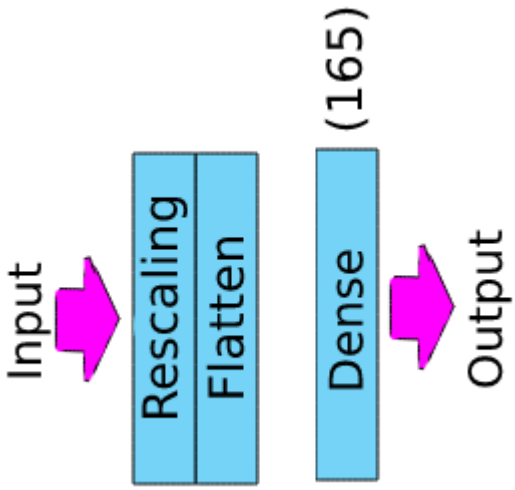
3.1.2. CNN (Konwolucyjna sieć neuronowa)



Rys. 2. Struktura sieci CNN

W naszym projekcie używamy CNN do wykrywania przeszkód. Struktura sieci została przedstawiona na rysunku 2.

3.1.3. QNN (Sieć neuronowa QNN)



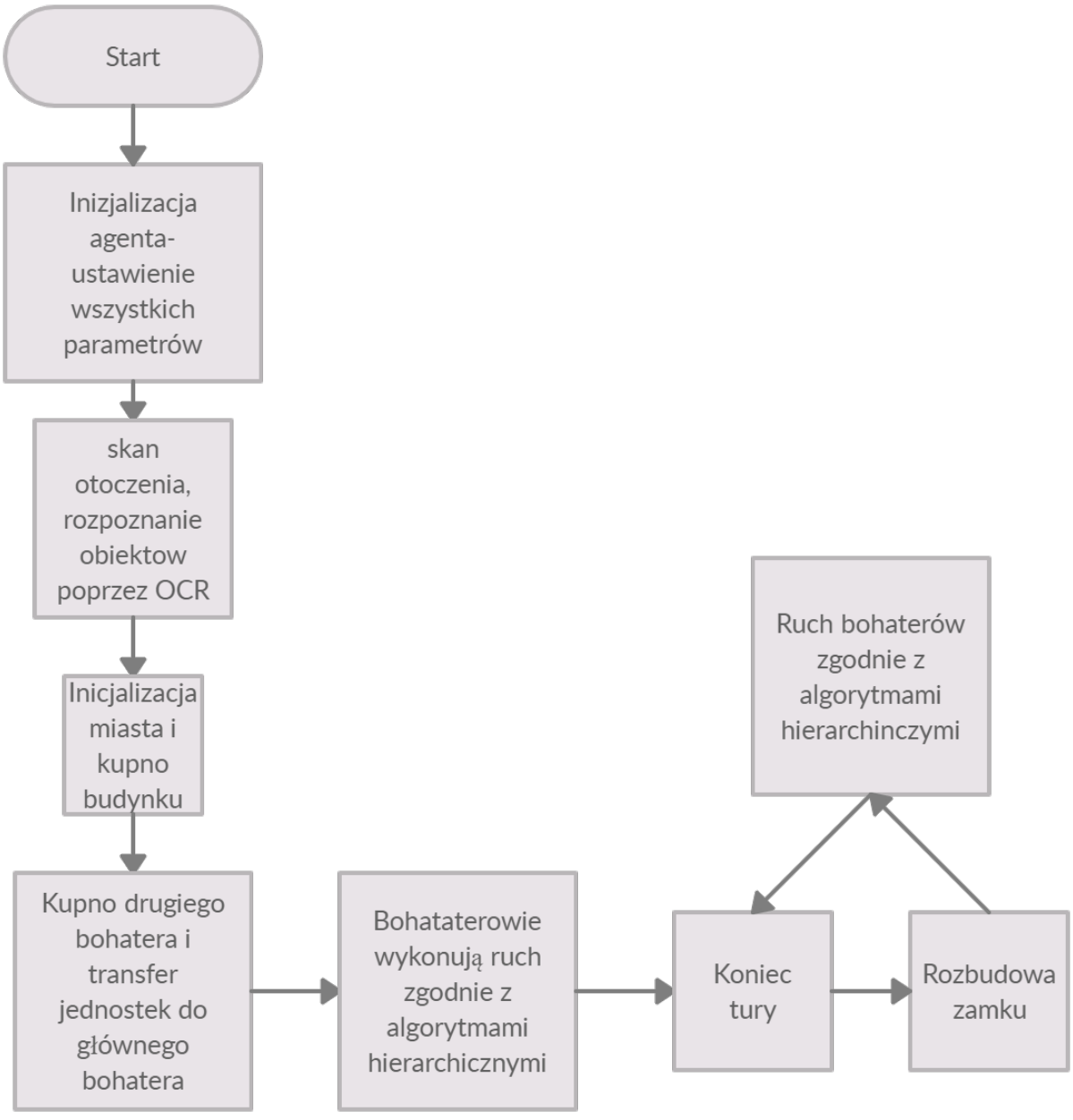
Rys. 3. Struktura sieci QNN

Natomiast QNN wykorzystujemy do wykrycia i ustalenia typu jednostki za pomocą rozpoznawania portretów znajdujących się w kolejce (rys. 7.). Strukturę sieci przedstawiono na rysunku 3.

3.2. Schematy blokowe

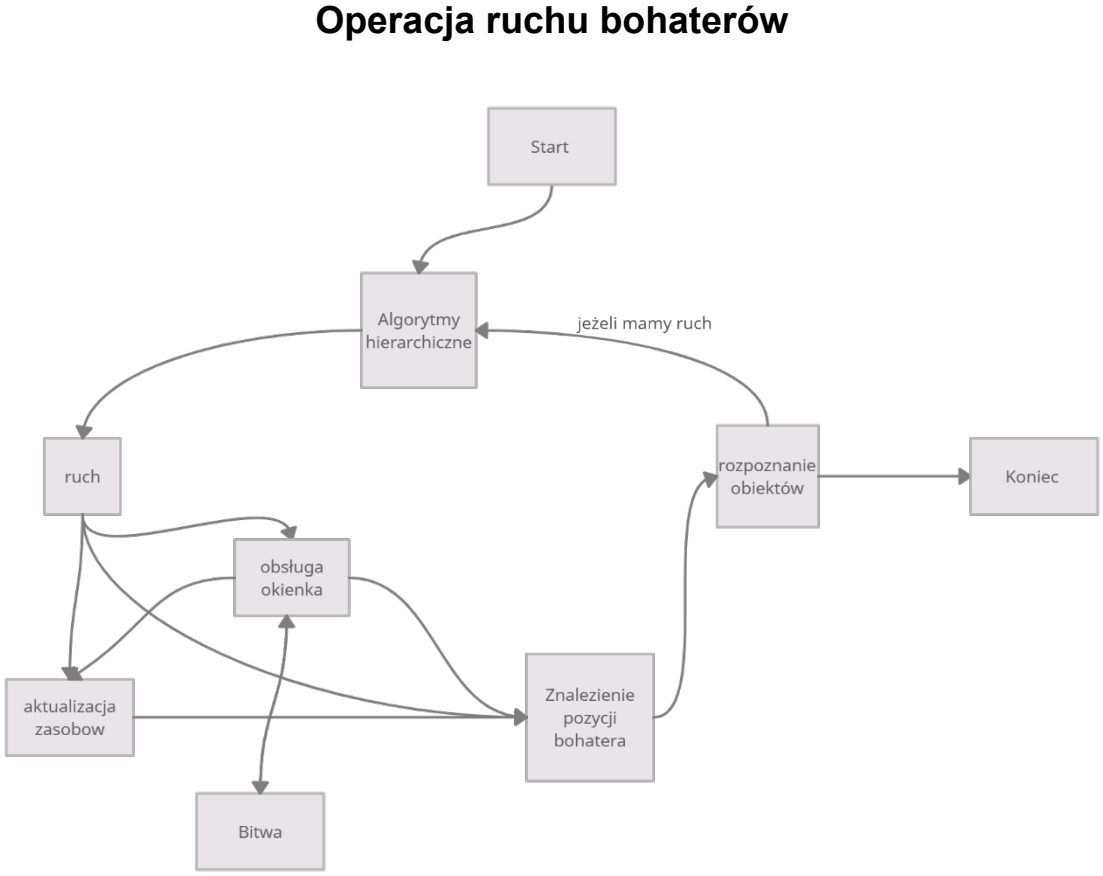
Poniższe schematy przedstawiają sposób działania bota.

Generalny schemat działania agenta



Rys. 4. Generalny schemat działania projektu

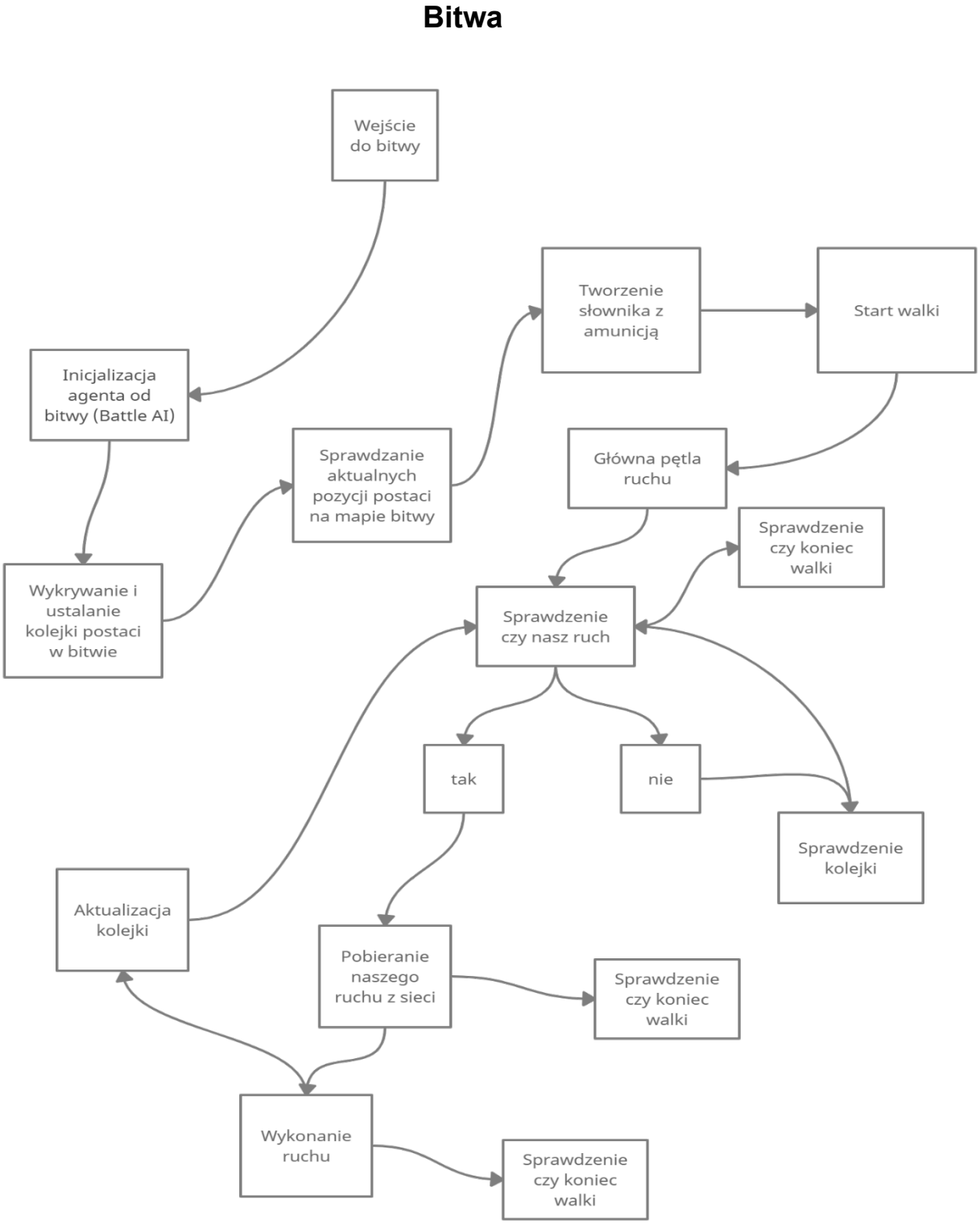
Powyższy graf przedstawia generalny schemat blokowy działania programu. Grafy poniżej będą się zgłębiały w poszczególne bloki. Na początku działania programu inicjalizujemy wszystkie parametry agenta i skanujemy otoczenie w celu rozpoznania obiektów w swoim otoczeniu. Następnymi operacjami jest obsługa miasta i drugiego bohatera. Następnym stopnieniem jest ruch wszystkich bohaterów.



Rys. 5. Schemat działania ruchu agenta po mapie

Powyższy graf przedstawia to co się dzieje w bloku "Bohaterowie wykonują ruch zgodnie z algorytmami hierarchicznymi" znajdującym się na rysunku 5. Za pomocą algorytmów hierarchicznych (podpunkt 3.3.1.) obliczamy wartość obiektu do którego agent pójdzie. Po ruchu bohatera jest parę scenariuszy, które mogą nastąpić, ale zawsze po ruchu sprawdzana jest pozycja bohatera. Potem znowu rozpoznajemy obiekty (o ile odkryliśmy nowy teren). W końcu, o ile agent ma ruch, powtórzy całą operację lub zakończy algorytm w przypadku gdy go nie ma.

W trakcie tego algorytmu możemy wejść do bitwy. Poniższy graf przedstawia cały cykl bitwy.



Rys. 6 – Schemat działania algorytmu odpowiadającego za walkę

Cykl bitwy szczegółowo opisany za pomocą rysunku 6 sprowadza się do kilku kluczowych momentów i ich rozgałęzień. Na początku inicjalizujemy naszego agenta

od bitwy, który był trenowany za pomocą Deep Q-Learning w naszym specjalnym, napisanym od podstaw środowisku imitującym walki w Heroes 3. Ostateczny model przeszedł 182,32 godzin trenowania. Następnym krokiem jest wykrycie i sprawdzenie kolejki postaci w bitwie (szczegółowo opisane w punkcie 3.4), a także ustalenie ich pozycji na mapie bitwy. Kolejnym etapem jest rozpoczęcie pętli ruchu, w której cyklicznie sprawdzamy stany typu czy czas na nasz ruch, czy walka się zakończyła, sprawdzanie kolejki. W przypadku etapu wykonania ruchu pobieramy decyzję z naszej sieci.

3.3. Mapa przygody

Mapa przygody jest częścią gry na której odbywają się procesy mikrozarządzania. Na planszy znajdziemy wiele lokacji zawierających surowce, skarby, wędrowne stworzenia, cenne surowce i ważne miejsca, takie jak kopalnie, które zapewnią stały dopływ surowców lub mieszkania stworzeń, w których bohater może rekrutować żołnierzy do swojej armii.

Wszystkie procesy mikrozarządzania w trybie przygody wymagały od nas przygotowania ich obsługi. Najważniejsze algorytmy zostały opisane w poniższych punktach.

3.3.1. Algorytmy hierarchiczne

Jednym z typowych zagadnień większości botów zarządzającymi grami są wybory celów. W Heroes 3 mapa składa się z siatki zawierającej obiekty. Z tej przyczyny zdecydowaliśmy się rozwiązać ten problem nadając każdemu obiektowi odpowiednią wartość wyliczaną przez nasze **algorytmy hierarchiczne**.

W ich skład wchodzą skrypty:

- artifact_value.py (artefakty),
- creature_banks_value.py (budynki ze stworami),
- enter_city_value.py (wchodzenie do miasta),
- ExplorationBoost.py (zachęta do eksploracji nieodkrytych miejsc),

- group_value.py (grupa obiektów posiada większą wartość niż pojedyncze),
- habitats_value_evaluation.py (siedliska stworzeń),
- mines_and_resources_value.py (kopalnie oraz zasoby),
- MiscFunc.py (pozostałe budynki ze specyficzną specjalnością),
- neutral_enemies_value.py (wrogowie stojący na mapie),
- power_evaluation.py (skrypt do oceniania naszych sił w porównaniu do przeciwnika).

Każdy wymieniony algorytm zwraca wartości obliczane na podstawie obecnego stanu naszej rozgrywki - na przykład 'artifact_value.py' ocenia opłacalność pola na podstawie zawartego na nim artefaktu, mnożnik polega między innymi na jego cenie, poziomie oraz sterowanym bohaterze.

W ten sposób wyznaczone wartości są następnie sumowane oraz dodawane (lub nie) do listy celów. Następnie podczas rozgrywki nasz agent wybiera cel o najwyższym priorytecie, tj. największej wartości.

3.3.2. Wyszukiwanie optymalnej ścieżki.

Istnieje wiele algorytmów do wyznaczania optymalnych ścieżek, na naszym projekcie testowaliśmy między innymi algorytm Dijkstry oraz A* (czyt. 'a z gwiazdką'). Ostatecznie zdecydowaliśmy się wykorzystać A* ze względu na jego kompletność, optymalność i optymalną wydajność.

Algorytm A* od wierzchołka początkowego tworzy ścieżkę, za każdym razem wybierając wierzchołek *x* z dostępnych w danym kroku niezbadanych wierzchołków tak, by minimalizować funkcję *f(x)* zdefiniowaną:

f(x) = g(x) + h(x), gdzie:

g(x) - droga pomiędzy wierzchołkiem początkowym a *x.* Dokładniej: suma wag krawędzi, które należą już do ścieżki plus waga krawędzi łączącej aktualny węzeł z *x.*

h(x) - przewidywana przez heurystykę droga od *x* do wierzchołka docelowego.

W naszym projekcie posiadamy mapę na której są zdefiniowane pola, po których możemy chodzić oraz te, które są dla nas przeszkodami lub celami. Po polach możemy poruszać się w osiem kierunków. Na naszej mapie uruchamiamy algorytm A*, który zwraca nam ścieżkę w postaci listy krotek z kolejnymi koordynatami. Następnie iterujemy po otrzymanej ścieżce - w zależności od kierunku ruchu (skos lub linia prosta) dodajemy do kosztu odpowiednią wartość.

Założenie projektu polegające na ‘patrzeniu parę tur wprzód’ w przypadku wykonania następnego ruchu zostało zrealizowane w sposób nadania większej wartości obiektom, które mają sąsiadów.

3.4. Bitwa

Obsługa bitwy polega na wykorzystaniu uczenia ze wzmocnieniem, a konkretnie Deep-Q-Learningu. Stan bitwy, niezbędny do określenia nagrody, jest otrzymywany przez przetwarzanie obrazu. Pozycja jednostek jest określona przez najechanie na nie myszką i sprawdzenie zmian kolorów kolejki, która znajduje się w dolnej części ekranu. Portrety jednostki w kolejce są sprawdzane za pomocą sieci neuronowej QNN (podpunkt 3.1.3.). Rozpoznanie jednostki również odbywa się przez rozpoznanie jej w kolejce. Przeszkody znajdujące się na polu bitwy są sprawdzane za pomocą sieci neuronowej CNN (podpunkt 3.1.2.).

Rys. 7. Kolejka

3.4.1. Deep-Q-Learning

W celu stworzenia agenta, który mógłby grać na poziomie “ludzkim”, wykorzystaliśmy algorytmy Deep-Q-Learningowe. Jest to bardzo popularne rozwiązanie w dziedzinie tworzenia botów do gier. W naszym rozwiązaniu wykorzystujemy dwie sieci neuronowe. Pierwsza, służy do wyznaczania następnej

akcji na podstawie obecnego stanu bitwy. Druga sieć odpowiada za akcje przeciwnego agenta, co pozwala nam na szybsze uczenie sieci neuronowych. Sieć neuronowa przyjmuje na wejście komórki z pola bitwy oraz atrybuty dla danego pola (np. czy stoi na nim jednostka). Natomiast każdy neuron w warstwie wyjściowej odpowiada akcji do wykonania w ramach bitwy. Strategia optymalna polega na wyborze akcji odpowiadającej neuronowi z największym wynikiem. Sieć trenowana jest za pomocą propagacji wstecznej i algorytmu Adam z użyciem punktacji, jaka została przyznana przez grę po wykonaniu określonej akcji. Użyliśmy również parametru “epsilon”, który sprawia, że na początku uczenia agent wykonuje więcej losowych funkcji, a później wykonuje ich coraz mniej. Nazywamy to parametrem eksploracji.

3.4.2. System cząstkowych nagród

Zdecydowaliśmy się na zastosowanie “cząstkowego” systemu nagród dla naszego agenta. Oznacza to, że nagroda nie była przyznawana za wygranie całej bitwy, ale za poszczególne akcje, które dawałyby poszczególny rezultat, taki jak zadane obrażenia, czy “czy jednostka przeciwnika jest w zasięgu jednostki sprzymierzonej?”. Stąd określenie “cząstkowego” systemu nagród. Pełna lista początkowych nagród dla agenta była taka sama jak w artykule wymienionym w punkcie 2.2.

Table 1: Estimated action results

% of player units in queue	% of opponent units reaching	damage inflicted*
% of player units reaching	% of opponent units in range	damage received*
% of player units in range	% of protected shooters	enemy retaliation taken
% of opponent units killed	% of released shooters	is waiting
% of opponent units in queue	% of blocked shooters	is dead

* computed using special formula

Rys. 8. Lista nagród na początku rozwoju projektu.

Niestety, taki system nie przyniósł oczekiwanych rezultatów po określonym czasie uczenia. Mianowicie, agent nie atakował jednostek przeciwnika, tylko je unikał za wszelką cenę. Zmusiło to nas do zmiany systemu nagród, która polegała na

wykorzystaniu tylko nagród ofensywnych oraz zmusiliśmy agenta do atakowania, kiedy zdarzyła się okazja. To dało *lepsze* wyniki, aczkolwiek po testach, nie jest to poziom rozgrywki, którą prowadzi człowiek.

3.5. Niewykorzystane rozwiązania

Stworzenie projektu od podstaw wymagało badania różnych metod do rozwiązywania konkretnych problemów. Niektóre z różnych powodów opisanych niżej nie zostały wykorzystane.

3.5.1. Rozpoznawanie obiektów na mapie za pomocą sieci neuronowych

Na początku projektu, podczas szukania dostępnych rozwiązań do przetwarzania obrazu, testowaliśmy i uczyliśmy sieci neuronowe do rozpoznawania obiektów na mapie. Ostatecznie, po tygodniach prób, zdecydowaliśmy się porzucić to rozwiązanie, ponieważ dataset używany do trenowania sieci był zbyt mały. Z tego powodu sieć okazywała się zawodna, ponadto przypadki, w których była widoczna tylko część wykrywanego obiektu jeszcze bardziej zmniejszała skuteczność sieci.

3.5.2. Przeszukiwanie grafu

Przy wyborze celów planowaliśmy użyć grafu – zawierał wszystkie wykryte pożądane miejsca połączone ze sobą z wyznaczoną do siebie ścieżką o obliczonym koszcie. Następnie przeszukiwaliśmy wymieniony graf algorytmem BFS (Breadth-First Search), który wyznaczał nam cele, jednak te rozwiązania okazały się zbyt wolne. Czas przeszukiwania grafu wydłużałby się również wraz z odkrywaniem mapy do znacznie niezadowalających wartości.

4. Wyniki

4.1. Battle AI

Ostateczna wersja sieci wytrenowanej do walki wykazała potencjał do rozwoju, który został przedstawiony za pomocą wygranych bota w poniższej tabeli.

Battle AI				
Poziom bota	Wygrane	Przegrane	Ilość gier	Skuteczność
Easy	75	81	156	48%

Tab. 1 – Battle AI

Zmierzyliśmy również średnie czasy osiągane przez naszego bota podczas walki:

- wybór zaklęcia, wykrywanie i rzucenie go przy spellbooku z liczbą 24 zaklęć wyniosło około ~ 9.07 s

- zaklęcia przy spellbooku z liczbą 18 zaklęć ~ 7.12 s

- średni czas ruchu jednostki ~ 5.77 s

- średni czas jednej tury wykonanej przez naszego bota ~ 9.44 s

4.2. Adventure AI

Finalny algorytm odpowiadający za chodzenie agenta po mapie nie zdołał spełnić naszych wymagań. Wielokrotne błędy wynikające ze złożoności gry nie pozwoliły doprowadzić bota do zakończenia rozgrywki.

5. Podsumowanie

Udało nam się zrealizować zakładane algorytmy przedstawione w adventure AI, oprócz algorytmu estymacji zamku, ponieważ w czasie trwania prac zdecydowaliśmy, że nie jest potrzebny. Spełniliśmy również założenie, mówiące o wykorzystaniu uczenia ze wzmocnieniem w bitwie, mimo, że w finalnej wersji programu system nagród został uszczuplony o defensywne nagrody ze względu na słabe wyniki. Nie udało nam się stworzyć systemu przetwarzania obrazu, który zapewniałby nam możliwość rozgrywki na losowej mapie. Początkowy pomysł zakładający użycie CNN nie przyniósł oczekiwanych rezultatów. Podejrzewamy, że podejście zakładające wykrywanie obiektów za pomocą konwolucyjnych sieci neuronowych ostatecznie się nie udało, ponieważ posiadaliśmy zbyt mały zbiór danych wszystkich obiektów w grze.

6. Przyszły rozwój projektu

6.1. Zmiana sposobu wykorzystania uczenia ze wzmocnieniem

Jednym z pomysłów na poprawę bitwy jest zmiana systemu nagród, która zakładałaby, że agent dostaje nagrodę tylko gdy wygra bitwę. Mielibyśmy mniejszą kontrolę nad poszczególnymi akcjami bohatera, ale przy dłuższym uczeniu mogłoby to dawać lepsze wyniki niż system cząstkowy, który obecnie stosujemy.

6.2. Zastosowanie uczenia ze wzmocnieniem w adventure AI

Na obecną chwilę, nasz bot porusza się zgodnie z wartościami obliczonymi przez algorytmy hierarchiczne. Nie jest wykluczone zastosowanie elementów uczenia ze wzmocnieniem w ruchu bohaterów na mapie przygody.

7. Bibliografia

[1] Alexander Shiskin, Michał Wawrzyniec Urbańczyk, Mateusz Baran, Arseniy Shestakov, Ivan Savenko, Andrii Dalychenko, Henning Koehler, Vadim Markovstev, Mikhail Paulyshka. VCMI – open source Heroes of Might and Magic 3 engine, 2022

[2] Łukasz Wilisowski, Rafał Dreżewski. The Application of Co-Evolutionary Genetic Programming and TD(1) Reinforcement Learning in Large-Scale Strategy Game VCMI, 2015

[3] Kapteeni Ruoska. Heroes of Might and Magic 3 wiki, 2022

[4] Robert Zubek, Aaron Khoo. Applying inexpensive AI techniques to computer games, 2002

[5] Jakub Kowalski, Radoslaw Miernik. Strategic Features and Terrain Generation for Balanced Heroes of Might and Magic III Maps, 2018

[6] Xiao Cui, Hao Shi. A*-based Pathfinding in Modern Computer Games, 2010