

Samouczący się automat sterujący postacią gracza w prostej grze zręcznościowej.

Krzysztof Nowak

6 stycznia 2012

Spis treści

1	Wstęp teoretyczny	2
1.1	Historia sztucznej inteligencji.	2
1.2	Sztuczna w dzisiejszych zastosowaniach.	3
1.3	Podstawy algorytmów w grach.	5
1.4	Podstawy algorytmów genetycznych.	8
2	Analiza wymagań	13
2.1	Opis problemu	13
2.2	Wykorzystane narzędzia i języki programowania	14
2.2.1	Język Java	14
2.2.2	Biblioteka Swing	16
2.2.3	Narzędzia	16
2.3	Wstępna analiza problemu	17
2.3.1	Sterowanie	17
2.3.2	Projekt chromosomu	17
2.4	Moduł silnika i symulatora gry	21
2.4.1	Wymagania funkcjonalne	21
2.4.2	Wymagania нефункционалне	23
2.4.3	Diagram przypadków użycia	24
2.4.4	Opis tekstowy przypadków użycia	25
2.4.5	Diagramy czynności	31
2.4.6	Diagram stanów	34
2.5	Moduł edytora map	34
2.5.1	Wymagania funkcjonalne	34
2.5.2	Wymagania нефункционалне	35
2.5.3	Diagram przypadków użycia	36
2.5.4	Opis tekstowy przypadków użycia	36
2.5.5	Diagramy czynności	38
2.6	Diagram klas	39

3	Opis interfejsu systemu	48
3.1	Moduł silnika i symulatora gry	48
3.1.1	Główne okno aplikacji	48
3.1.2	Wygląd środowiska gry	49
3.1.3	Dane konfiguracyjne	50
3.1.4	Widok Populacji i Chromosomu	51
3.2	Moduł edytora map	53
3.3	Realizacja warstwy symulacyjnej	53
3.3.1	Edytor Map	53

1 Wstęp teoretyczny

1.1 Historia sztucznej inteligencji.

Na początku lat 40 matematycy i inżynierowie z ośrodków badawczych zaczęli zastanawiać się nad możliwością stworzenia sztucznego mózgu. Pierwsze formalne centrum badawcze pracujące nad zagadnieniem sztucznej inteligencji zostało powołane do życia w 1956 roku w Dartmouth College, 16 lat po wynalezieniu pierwszego programowalnego komputera. Początkowo nazywane przedsięwzięciem stworzenia pierwszego “Elektronicznego mózgu” zostało poważnie potraktowane przez ówczesnych naukowców, i tworzyło dobre perspektywy dla ekonomistów i bankierów. Wielu badaczy zapowiedziało stworzenie maszyn dorównujących inteligencją ludziom w niespełna kilka dekad. Specjalnie na ten cel rząd amerykański oraz brytyjski przeznaczyły budżet rzędu milionów dolarów.

Pierwsze prace nad sztuczną inteligencją skupiały się na odwzorowaniu realnej pracy ludzkiego mózgu - sieci neuronów. Naukowcy tacy jak Norbert Wiener, Claude Shannon oraz Alan Turing opracowali pierwsze pomysły stworzenia elektronicznego mózgu. Powstało pojęcie sieci neuronowych, mocno później rozwijane m.in. przez Marviną Minskiego w jego pracach przez następne 50 lat. Pierwsze programy skupiające się na sztucznej inteligencji w grach powstały na początku lat 50. Christopher Strachey był autorem pierwszego programu grającego w Warcaby. Pierwszy program szachowy został napisany przez Dietricha Prinza. W tym samym okresie Alan Turing opublikował pierwsze prace dotyczące możliwości utworzenia maszyny dysponującej ludzką inteligencją. Zdefiniował

test pozwalający to zmierzyć, nazywany potem Testem Turinga.

W końcu po wielu latach stało się oczywiste iż symulacja nawet najprostszych mechanizmów myślowych jest niezwykle trudna w realizacji, a nawet najszybsze ówczesne komputery nie były w stanie wygrać z człowiekiem w partii szachów. Ostatecznie dziedzina sztucznej inteligencji odebrano nieco wiarygodności, a wcześniej zapowiadane maszyny przerastające inteligencją ludzi, trafiły z powrotem na półki science-fiction. W roku 1973, znaczna część funduszy przeznaczonych na rozwój sztucznej inteligencji została wstrzymana przez amerykański i brytyjski rząd. Niemniej prace nad sztuczną inteligencją trwają do dziś, aczkolwiek są bardziej uszczegółowione w naturze problemów których dotyczą.

1.2 Sztuczna w dzisiejszych zastosowaniach.

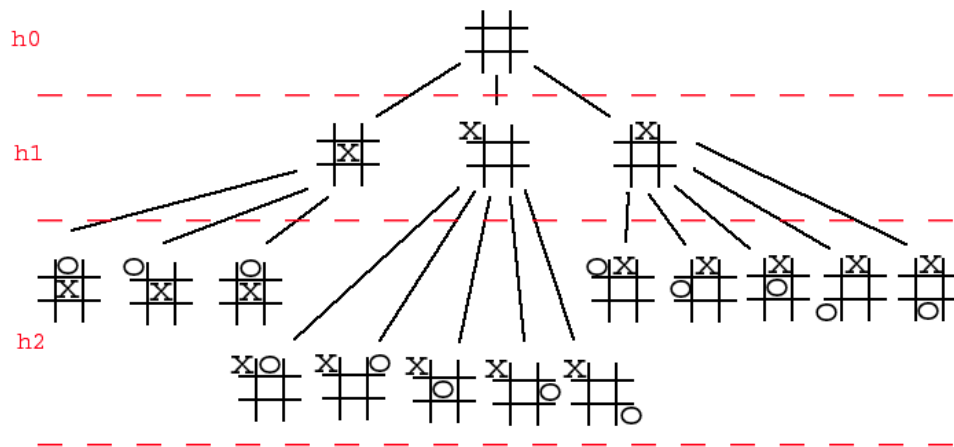
Oprócz realizacji zadań w dziedzinie kategoryzacji danych, oraz rozpoznawaniu mowy lub obrazu, spora część badań skupia się na realizacji systemów podejmujących decyzje w ściśle określonym środowisku gry. Pozornie służą one jedynie dostarczaniu rozrywki w szeroko popularnych grach komputerowych, szybko można się przekonać iż wiele takich projektów jest później podstawą do stworzenia bardziej praktycznych systemów. Zmieniając jedynie definicję środowiska okazuje się iż można te same strategie zastosować np. w grze na giełdzie. Pojedynek Garriego Kasparowa z programem szachowym Deep Blue przeszedł już na stałe do historii jako pierwsze starcie człowieka z maszyną w dziedzinie intelektu. Innym, dość nowym przykładem może być klaster komputerów Watson, który przez kilka tygodni konkurował z czołówką graczy teleturnieju Jeopardy, popularnym w USA od 1964 roku. Obydwa projekty pokonały swoich ludzkich przeciwników, podnosząc tym samym nieco nadszarpnięty wizerunek sztucznej inteligencji. Wiele współczesnych zastosowań sztucznej inteligencji zyskało dużą popularność dzięki globalnej sieci internet. Prym wiedzie tutaj firma Google, początkowo znana jedynie jako twórca najpopularniejszej i najdokładniejszej wyszukiwarki, rozbudowuje bazę swoich aplikacji o translatory, aplikacje do nawigacji map, czy program graficzny Picasa, potrafiący np. zidentyfikować na zdjęciu ludzką twarz. Większość wyszukiwarek internetowych wykorzystuje różnego

rodzaju systemu wspomagające w wyszukiwaniu informacji na temat konkretnej frazy. Aktywna lista podpowiedzi wyświetlająca się po wpisaniu początku popularnej frazy stała się wręcz standardem w projektowaniu wyszukiwarki. Wyszukiwarka coraz częściej poprawia błędy ortograficzne lub niepoprawnie przeliterowane wyrazy związane np. z bliskim sąsiedztwem liter na klawiaturze. Innym ciekawym zastosowaniem jest tłumaczenie tekstu z dowolnego języka na inny. Nie działa to na zasadzie bazy danych przechowującej odpowiedniki słów w każdym z języków, lecz wykorzystuje metodę “uczenia się” całych wyrażen bądź zwrotów które są równorzędne w obu językach, dzięki czemu zyskujemy dość precyzyjne tłumaczenie nawet jeśli chodzi o nazwy własne, akronimy, odmianę oraz składnię, np.: Przy tłumaczeniu z angielskiego na polski frazy “United States of America” uzyskujemy “Stany Zjednoczone”, natomiast “in UK” tłumaczone na język polski poprawnie daje zwrot “w Wielkiej Brytanii”. W pierwszym przykładzie naiwna metoda nie ominęłaby słowa “America” oraz prawdopodobnie nie zachowałaby szyku słów. Wynikiem mogłoby być wówczas “Zjednoczone Stany Ameryki”. Inne ciekawe zastosowaniem może być aplikacja Wolfram Alpha stworzona przez Stephena Wolframa. Podobnie jak wyszukiwarka Google, opiera ona swoje wyniki na frazach wpisywanych do okna wyszukiwarki przez użytkownika, jednak jest ona skupiona głównie na przetwarzaniu danych matematycznych. Potrafi interpretować wzory wpisane przez użytkownika, wyświetlić wykres w przestrzeni oraz dostarczyć wielu dodatkowych informacji np. znaleźć pierwiastki rzeczywiste. O ile sztuczna inteligencja w świetle dzisiejszego dostępu ogromu danych do przetwarzania stwarza duży potencjał na tworzenie skomplikowanych systemów przetwarzających wciąż spotykamy się z jej rosnącym zastosowaniem w grach. Historia informatyki od jej wczesnych początków związana była z grami komputerowymi. W roku 1952 Alexander Shafto Douglas opisał temat komunikacji człowiek-komputer w swojej pracy doktoranckiej, oraz stworzył program grający w popularne “Kółko i Krzyżyk”. Do dziś uznawany jest on za pierwszą graficzną grę komputerową.

1.3 Podstawy algorytmów w grach.

Jednym z podstawowych przykładów zastosowania sztucznej inteligencji w grach komputerowych są gry logiczne. Podstawowym algorytmem stosowanym w projektowaniu sztucznej inteligencji jest algorytm minmax. Najprostszym przykładem jest gra "Kółko i krzyżyk" gdzie przeszukiwane jest tzw. drzewo gry, kolejno sprawdzając wszystkie możliwe stany gry. W dowolnym momencie gry możemy przeanalizować wszystkie możliwe posunięcia każdego z graczy, i dążyć do sytuacji gdzie mamy gwarantowany sukces. Gra może zakończyć się remisem, zwycięstwem gracza A, bądź zwycięstwem gracza B. Optymalizując decyzję gracza A, musimy kolejno sprawdzać możliwe posunięcia na planszy i reakcje gracza B. Dla każdego z możliwych ruchów nich możemy wówczas przeanalizować optymalną strategię dla gracza B (ponieważ zakładamy że do takiej będzie on dążył) i starać się znaleźć najlepszą ścieżkę która prowadzi do zwycięstwa gracza A, bądź remisu. Łatwo zauważyć iż algorytm taki ma ogromną złożoność dla rozbudowanych drzew, i jedynie dla małych gier takich jak kółko i krzyżyk daje wynik w realnym czasie. Bez optymalizacji, oraz brania pod uwagę symetrii planszy, daje to drzewo wywołań składające się $9!$ węzłów (łącznie z liśćmi). Oczywiście algorytm można zoptymalizować chociażby poprzez programowanie dynamiczne, lecz dla gier bardziej złożonych nie będziemy w stanie przeanalizować wszystkich możliwych sytuacji w grze w realnym czasie. Przez lata szachy były jedną z gier niemożliwych do rozwiązania za pomocą powyższego podejścia. Nawet dziś najlepsze programy szachowe nie grają idealnie - nie analizują wszystkich możliwych ruchów a jedynie kilkadziesiąt ruchów w przód. Przy zastosowaniu optymalizacji oraz bazy danych zawierającej wiele strategii szachowych, współczesne programy szachowe wygrywają z najlepszymi graczami. Nieco inna sytuacja jest w grze GO, gdzie plansza rozmiaru 19×19 stanowi duże wyzwanie nawet dla współczesnych superkomputerów. Do dziś nie stworzono programu który wygrywałby z profesjonalnymi zawodnikami GO. Jak widać na Rys. 1, pierwsze 2 poziomy drzewa gry Kółko i Krzyżyk nie są zbyt skomplikowane jeśli weźmiemy pod uwagę symetrię. Początkowo złożoność drzewa rośnie wykładniczo, jednak możliwości na planszy kończą się zanim zaczyna to być problemem.

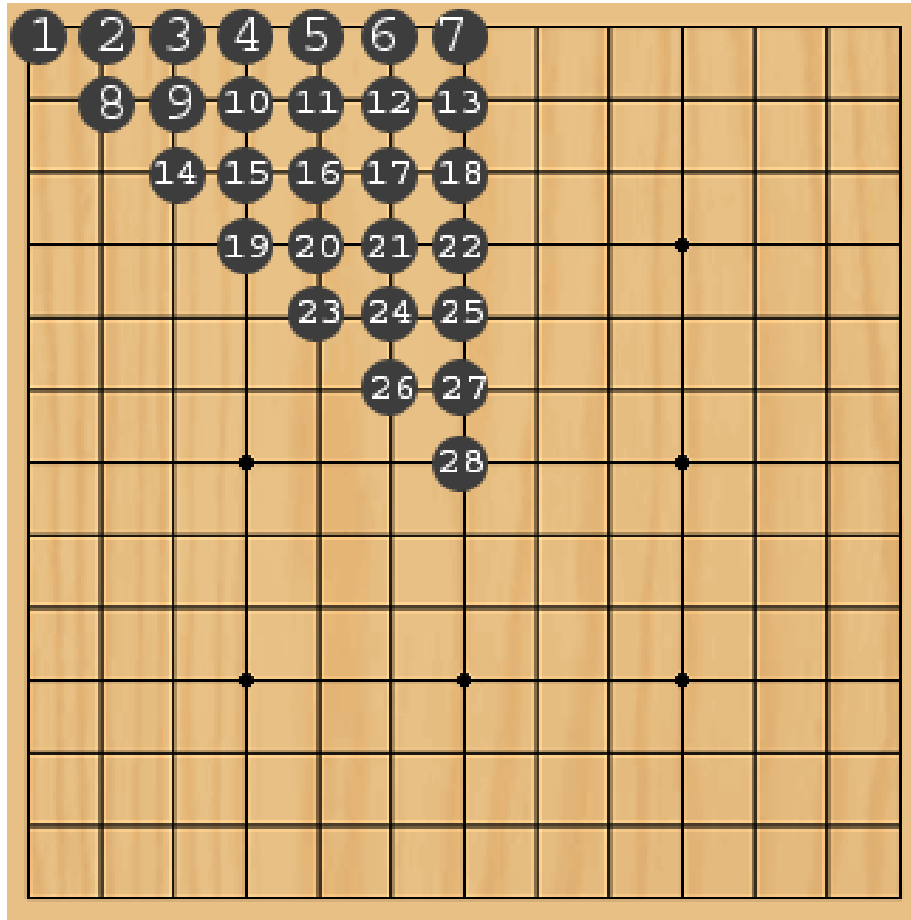
Inaczej wyglądałby przypadek drzewa gry GO (plansza rozmiaru 13x13), gdzie pierwszy ruch można wykonać na 28 unikalnych sposobów: Rys. 2 przedstawia przykładowe pierwsze 28 węzłów drzewa gry GO (13x13) biorąc pod uwagę symetrię planszy. Sprawa komplikuje się gdy rozpatrujemy większe plansze. Dla plansz o wymiarach 19x19 - standardowym rozmiarze obowiązującym na wszystkich turniejach gry GO - pierwszy ruch można wykonać na 55 sposobów (analogicznie do Rys.2). Plansza GO ma 4 osie symetrii, może się wówczas okazać że odpowiedź przeciwnika będzie na tyle “niesymetryczna”, że już na drugim poziomie drzewa musimy rozpatrzyć wszystkie możliwe pola na planszy, czyli 359 ($19 \times 19 - 2$).



Rysunek 1: Schemat drzewa gry Kółko i Krzyżyk.

Jeśli chcemy wprowadzić podział gier przydatny przy projektowaniu systemu pierwszym czynnikiem będzie typ rozgrywki ze względu na czas. Większość gier można podzielić wówczas w następujący sposób:

- Gry turowe - Gracze naprzemiennie wykonują ruchy, przy czym czas na podjęcie decyzji jest relatywnie duży - od 1 sekundy nawet do 1-2 minut. Wiele nieskomplikowanych gier turowych zostało już dawno rozwiązanych przez algorytmy typu minmax, do tego stopnia, że systemy grają w nie już niemal bezbłędnie. W wielu przypadkach przestrzeń rozwiązań jest jednak wciąż zbyt duża aby zrealizować to algorytmem dokładnym - przykładem może być wyżej wspomniana gra GO.



Rysunek 2: Przykład pierwszych możliwych unikalnych ruchów w grze GO.

- Gry czasu rzeczywistego - Gra toczy się w dynamicznym środowisku gry, często z wieloma obiektami i graczami na raz. Często czas na podjęcie optymalnej decyzji przez algorytm jest mocno ograniczony - często należy podejmować odpowiednie decyzje nawet do 30 razy w ciągu sekundy. Oprócz tego próba dyskretyzacji środowiska i znalezienia dokładnego rozwiązania z przestrzeni stanów gry jest w praktyce niewykonalne. Przykładem mogą być tutaj różnego rodzaju dwu lub trójwymiarowe gry akcji, które często posiadają złożone środowiska gry, wiele dynamicznych obiektów oraz graczy uczestniczących w rozgrywce poprzez sieć komputerową. Przy projektowaniu sztucznej inteligencji w takim środowisku w tej chwili możemy liczyć jedynie na wyniki przybliżone.

Sztuczna inteligencja w grach może dotyczyć różnych aspektów gry. W grach logicznych

(turowych) głównym, i jedynym problemem jest podjęcie najlepszej decyzji dla aktualnego stanu gry, prowadzącej do zwycięstwa. W większości gier logicznych sztuczna inteligencja ma za zadanie symulację godnego przeciwnika dla człowieka. Często jednak te same algorytmy mogą służyć do celów edukacyjnych bądź do podpowiedzi - ten sam system grający w szachy może grać przeciwko nam, jak i podpowiadać nam ruchy na podstawie naszej pozycji na planszy. W grach zręcznościowych oraz akcji (gry czasu rzeczywistego) występuje często inny rodzaj sztucznej inteligencji. Ponieważ przestrzeń rozwiązań jest bardzo duża, często podjęcie decyzji może być wspomagane przez algorytm przybliżony, bądź oparty na algorytmach genetycznych. Algorytm minmax w większości przypadków zawodzi, bądź jego czas działania jest zbyt wolny do zastosowania w dynamicznym środowisku gry. Korzystając z algorytmów "jedynie" optymalizujących rozgrywkę tracimy możliwość rozegrania idealnej partii gry, jednak często wystarcza to dla osiągnięcia celu końcowego jakim jest stworzenie godnego przeciwnika.

1.4 Podstawy algorytmów genetycznych.

Często wykorzystywanym sposobem rozwiązywania złożonego problemu algorytmicznego są algorytmy genetyczne. Opierają się one na zasadach ewolucji odkrytych przez Charlesa Darwina, i wzorują się na faktycznych rozwiązaniach doboru naturalnego występujących w przyrodzie. Algorytm ewolucyjny opiera się na wprowadzeniu losowego czynnika do całej procedury, i tym też różni się od klasycznego algorytmu, iż jest niedeterministyczny. Ogólny przebieg algorytmu genetycznego może wyglądać następująco:

1. Wygenerowanie początkowej populacji osobników (rozwiązań) w sposób losowy.
2. Przeliczenie funkcji przystosowania dla każdego z osobników.
3. Uporządkowanie populacji malejąco względem wyniku funkcji przystosowania.
4. Wybranie populacji rodzicielskiej zgodnie z przyjętą metodą selekcji.
5. Krzyżowanie osobników z populacji rodzicielskiej i otrzymanie nowej populacji -

nowe potomstwo posiada cechy rodziców którzy w poprzedniej populacji byli najlepiej przystosowani do rozwiązania danego problemu.

6. Mutacja części potomstwa - wprowadzenie czynnika losowego poprzez zmianę niektórych fragmentów chromosomu w sposób losowy.
7. Jeśli warunek końcowy nie został osiągnięty powrót do kroku 2, w przeciwnym wypadku koniec algorytmu.

Wynikiem takiego algorytmu jest nie jedno rozwiązanie problemu, a cała ich populacja. W większości algorytmów genetycznych można wydzielić kilka koniecznych do zaprojektowania klas bądź procedur.

1. Chromosom oraz Populacja

Pierwszym krokiem jest zdefiniowanie typu danych odpowiednich do przetrzymywania informacji o danym osobniku. Odpowiednio zaprojektowany format danych (zwany Chromosomem) pozwoli na łatwą implementację pozostałych elementów oraz zapewni generowanie optymalnych wyników. Informacja ta często jest reprezentowana przez tablicę wartości, bądź listę cech przypisanych do danej klasy. Chromosom odpowiada za informację o pojedynczym osobniku, natomiast Populacja traktowana jest jako wszystkie osobniki należące do danego zbioru w danej iteracji algorytmu. O ile w podstawowych algorytmach genetycznych Populacja jest jedynie kontenerem, dobrze jest pamiętać o ewentualnym rozbudowaniu Populacji do bardziej złożonej klasy, dzięki czemu będziemy mieli możliwość prostego porównywania, bądź zapamiętywania całych populacji.

2. Funkcja Przystosowania

Kolejnym istotnym krokiem jest zdefiniowanie funkcji przystosowania. W doborze naturalnym występującym w przyrodzie, osobniki danego gatunku rośliny bądź zwierzęcia różnią się pod względem genetycznym. Można wówczas wywnioskować iż część z nich jest lepiej przystosowana do danego środowiska, co z kolei wpływa

na ich szanse przeżycia w trudnych sytuacjach, licznosc potomstwa, dlugosc zycia. Poniewaz potomstwo dziedziczy geny po swoich rodzicach, “zwycieskie” cechy w kolejnym pokoleniu sa bardziej powszechne. Odpowiednikiem funkcji przystosowania jest wlasnie wynikowa cech danego osobnika ktora określa prawdopodobienstwo przekazania jego genow w kolejnym pokoleniu. Funkcja przystosowania jest dosc prosta w realizacji, o ile dane dotyczace osobnika sa latwe do zmierzzenia – w owczas moze byc to jedynie kwestia policzenia wartosci funkcji liniowej z odpowiednimi wagami, gdzie argumentami sa wyniki osobnika podczas symulacji w srodowisku. Mimo to w wiekszosci algorytmow genetycznych dobrane odpowiednich wag w funkcji przystosowania jest kluczowym czynnikiem nad ktorym pozniej mozna dlugo pracowac przy optymalizacji algorytmu.

3. Krzyzowanie

Po kazdym kroku algorytmu zazwyczaj mozemy uporządkowac osobniki nalezace do biezacej populacji i wylosowac z niej pewien zbior osobnikow najlepiej przystosowanych (wpływ na to wynik funkcji przystosowania). W owczas dokonujemy krzyzowania pomiedzy nimi, dzieki czemu otrzymujemy osobniki nowe, jednak posiadajace pewne cechy swoich “rodzicow”. Krok ten jest kluczowy jesli chcemy osiagac coraz lepsze wyniki w kolejnych populacjach, poniewaz od dobrej metody krzyzowania zalezy czy kolejne populacje beda lepiej przystosowane do rozwiazania problemu. Zle zaprojektowanie krzyzowania jest jednym z czestszych powodow osiagania przez populacje zlých wyników, zwlaszcza gdy Chromosom ma złożoną strukturę. Samo krzyzowanie czesto rowniez posiada czynniki losowe (w klasycznych przykladach dotyczacych krzyzowania sie dwuch ciagow bitowych, losowany jest punkt laczenia sie dwuch ciagow).

4. Mutacja

O ile poczatkowa losowosc algorytmu polegajaca na wylosowaniu pierwszej populacji jest szybko zastepowana przez populacje osiagajaca lepsze wyniki, warto w

trakcie całego procesu próbować modyfikować kilka osobników, nawet jeśli mogłoby to spowodować chwilowe pogorszenie populacji. W innym przypadku zbyt uporządkowana procedura selekcji i krzyżowania osobników spowoduje stagnację populacji. Często można to zauważyć gdy po kilku iteracjach większość, bądź cała populacja jest identyczna. Najczęstszą realizacją mutacji jest zmiana jakiegoś parametru (bądź grupy parametrów) danego osobnika na wartość nieco inną lecz podobną, bądź zupełnie losową. Ponieważ w dużej mierze zależy to od budowy Chromosomu, nie ma uniwersalnej metody na zaimplementowanie mutacji. Najczęściej mutacja występuje z niskim prawdopodobieństwem:

$$p_m < 0.1$$

Tak aby nie ingerować zbyt mocno w algorytm. Ostatecznie należy dążyć do pewnej systematycznej optymalizacji, a nie tylko polegać na czynniku losowym.

5. Metoda Selekcji

Sama metoda wyboru populacji rodzicielskiej również ma znaczenie, ponieważ jednak jest ona oparta na wartości funkcji przystosowania, to już sama metoda wyboru ma mniej krytyczne znaczenie. Najbardziej popularne metody selekcji to:

(a) Metoda koła ruletki.

Sama nazwa bierze się od popularnej gry w ruletkę, w której pole powierzchni każdego wycinka koła jest proporcjonalne do prawdopodobieństwa wylosowania danej liczby. Oczywiście w klasycznej ruletce pola wycinków koła są równe, zatem szansa wylosowania każdej liczby jest taka sama. W samym algorytmie wirtualne “wycinki koła” nie muszą oczywiście być równe. Osobnik który osiąga lepsze wyniki w funkcji przystosowania otrzymuje większe prawdopodobieństwo włączenia do populacji rodzicielskiej niż osobniki słabsze. Aby to zrealizować losowana jest pewna wartość która potem jednoznacznie określa

który osobnik został wylosowany. Praktycznie realizowane jest to w następujący sposób:

$$p(k) = \frac{f(k)}{\sum_{i=0}^n f(i)}$$

gdzie $p(k)$ oznacza prawdopodobieństwo wylosowania k -tego osobnika z populacji, a $f(i)$ wartość funkcji przystosowania i -tego osobnika.

(b) Metoda rankingowa.

W tej metodzie sortujemy osobniki malejąco względem funkcji przystosowania i wybieramy populację rodziców jako m pierwszych osobników. Ma to pewną wadę, gdyż powoduje po pewnym czasie stagnację (brak czynnika losowego). Innym wariantem jest selekcja turniejowa w której najpierw dzielimy grupę na G podgrup spośród których wybieramy najlepsze osobniki do populacji rodzicielskiej. Otrzymujemy w ten sposób G rodziców, wśród których niekoniecznie są najlepsze osobniki globalnie (nawet z bardzo silnej grupy przechodzi tylko jeden osobnik). Daje nam to już pewną losowość w wyborze populacji rodzicielskiej.

(c) Połączenie kilku metod.

Dodatkowym elementem może być połączenie kilku metod selekcji celem otrzymania najbardziej optymalnej selekcji dla danego problemu genetycznego. W zasadzie bardziej złożone problemy ewolucyjne wręcz wymagają własnej inwencji do zaprojektowania dobrego systemu.

Dużą częścią dobrego systemu genetycznego jest odpowiednia możliwość konfiguracji danych odpowiadających za każdy z kroków. Mamy dzięki temu możliwość przetestowania różnych podejść do danego problemu bez bezpośrednich i często uciążliwych zmian w kodzie programu. Oprócz tego cały proces można zautomatyzować, dzięki czemu możemy w prosty sposób przetestować algorytm dla różnych danych konfiguracyjnych.

2 Analiza wymagań

Przed implementacją systemu należy przeprowadzić analizę problemu, określić wymagania systemu, zdecydować o narzędziach i technologiach niezbędnych w implementacji systemu. W tym rozdziale zostanie najpierw opisany problem oraz koniecznie do zaimplementowania moduły, następnie postawione zostaną wymagania jakie system musi spełniać. Ostatnim krokiem będzie omówienie narzędzi i technologii koniecznych do implementacji systemu.

2.1 Opis problemu

Tematem pracy jest zaprojektowanie i implementacja systemu podejmującego decyzje w prostej grze platformowej czasu rzeczywistego. Moduł odpowiedzialny za optymalizację przejścia gry, oraz podejmowanie akcji powinien być oparty o zagadnienie algorytmów genetycznych. Celem samej gry jest dotarcie do zdefiniowanego wcześniej celu na dwuwymiarowej mapie. Wynik końcowy przejścia może zależeć od wielu parametrów - mapa może zawierać elementy dające punkty, jak i elementy prowadzące do natychmiastowego zakończenia gry (z wynikiem pozytywnym bądź negatywnym). Efektem pracy powinien być system pozwalający na rozwiązanie tego typu problemu bazujący na algorytmie genetycznym.

Ogólne wymagania dotyczące systemu:

- System powinien składać się z bazowego silnika gry, wzorowanego na rozwiązaniach w klasycznych grach platformowych. Ma to być jednocześnie warstwa prezentacyjna algorytmu. System nie powinien sztywno zakładać realizacji algorytmu na konkretnej grze platformowej, lecz być ogólnym systemem rozwiązującym gry platformowe czasu rzeczywistego.
- System powinien pozwalać zarówno na poruszanie się po mapie przez użytkownika jak i przejście w tryb treningu populacji, który na podstawie zadanych parametrów optymalizuje przejście po mapie algorytmem genetycznym.

- Do wyniku końcowego mogą być brane pod uwagę również inne zdarzenia takie jak ilość zebranych obiektów na planszy, czy czas przejścia gry. Funkcja przystosowania zależeć będzie od różnych czynników, a ich modyfikacja powinna być łatwo dostępna.
- Projekt ma mieć charakter edukacyjno-badawczy. Przydatnymi narzędziami w systemie będzie prosty w obsłudze edytor map, panel konfiguracyjny w którym możemy edytować większość parametrów związanych z samym działaniem algorytmu oraz aktywny podgląd populacji i osobników.

Sam pomysł stworzenia sztucznej inteligencji do gry platformowej w czasie rzeczywistym został już wcześniej powoływany do życia, m.in. jako projekt MarioAI. W chwili obecnej funkcjonuje on jako turniej dla programistów. Uczestnicy mogą implementować własne rozwiązania do gotowego silnika generującego losowe poziomy, oraz porównywać wyniki z innymi uczestnikami. Samo zgłoszenie składa się z implementacji własnej klasy odpowiedzialnej za podejmowanie decyzji. Strona domowa projektu znajduje się pod adresem www.marioai.org.

2.2 Wykorzystane narzędzia i języki programowania

2.2.1 Język Java

Głównym celem do zrealizowania w pracy jest problem algorytmiczny i teoretyczny. Praca w mniejszym stopniu opiera się na wykorzystaniu konkretnej technologii, czy języka programowania, wobec czego zostały wykorzystane popularne narzędzia i języki programowania. System został napisany w języku Java oraz testowany na wirtualnej maszynie javy w wersji Java SE 7u2. Biblioteka Swing została użyta do stworzenia warstwy wizualnej programu. Java jest językiem programowania wysokiego poziomu zaprojektowanym i stworzonym przez Jamesa Goslinga podczas gry pracował w firmie Sun Microsystems. Pierwsza propozycja stworzenia Javy pojawiła się w roku 1991, do głównych twórców oprócz Jamesa Goslinga zalicza się również Mike’a Sheridana oraz Patrick’a Naughtona.

Początkowo język Java był bezpośrednio związany z firmą Sun Microsystems, która kontrolowała jego rozwój do roku 2010. Od tego czasu firma Sun stała się częścią korporacji Oracle, wobec czego wszelkie prawa związane z językiem Java posiada Oracle. Java swoją składnią przypomina język C, aczkolwiek wśród pierwowzorów wymienia się również język Smalltalk. Aplikacje napisane w języku Java są kompilowane do kodu bajtowego Javy (ang. java bytecode), i uruchamiane na maszynie wirtualnej, co zapewnia pewnego rodzaju bezpieczeństwo w stosunku do języka C lub C++. Inną dużą zaletą języka Java jest jego przenośność. Jedynym wymogiem uruchomienia aplikacji jadowej na dowolnym systemie jest obecność wirtualnej maszyny javy (ang. JVM - Java Virtual Machine). Dziś większość urządzeń mobilnych posiada wirtualną maszynę javy pozwalającą na uruchamianie aplikacji napisanych w tym języku. ::BIBLIOGRAFIA TIJ-Bruce Eckel:: Jak pisze Bruce Eckel w swojej książce poświęconej językowi Java: "To, co wywarło na mnie największe wrażenie, kiedy poznawałem javę, to fakt, że wśród innych celów projektantów z firmy Sun znalazła się także redukcja złożoności dla programisty. To tak, jakby powiedzieć: "Nie dbamy o nic poza zmniejszeniem czasu i trudności tworzenia porządnego kodu."(...)". Ponieważ część algorytmiczna aplikacji wymaga przeprowadzania częstych symulacji zachowania środowiska oraz przeprowadzania całego przebiegu gry, istotnym czynnikiem był czas działania krytycznych miejsc w aplikacji - głównie pętli gry, wyświetlania oraz generowania nowej populacji na podstawie poprzedniej. Java jako język polegający na maszynie wirtualnej posiada warstwę pośrednią która spowalnia cały proces jest wolniejsza od języka C lub C++, jednak prostota realizacji części wizualnej aplikacji oraz bardzo dobra przenośność przeważały w wyborze języka. Kolejnym trafnym wyborem jeśli chodzi o język i narzędzia byłby prawdopodobnie C++ wraz z biblioteką Qt do generowania grafiki i tworzenia okienek oraz kontrolek. Tą samą aplikację można by zrealizować za pomocą języka Python i biblioteki PyGame/Qt do warstwy wizualizacyjnej, jednak wówczas koszt czasowy realizacji algorytmu genetycznego oraz logiki gry mógłby okazać się znacząco duży, ze względu na fakt iż Python jest językiem interpretowanym, i generalnie nie jest przeznaczony do dużych obliczeń. Przy konieczności

realizacji prostej wersji demonstracyjnej programu opisywanego w pracy, język Python wraz z biblioteką PyGame może okazać się najszybszym do implementacji ze względu na krótki kod, dynamiczne typy zmiennych oraz bogatą kolekcję struktur takich jak listy czy mapy incydencji konieczne do realizacji algorytmu. Sama aplikacja korzysta z kolekcji i podstawowych struktur danych występujących w języku Java.

2.2.2 Biblioteka Swing

Pierwszą próbą stworzenia biblioteki graficznej javy pozwalającej na tworzenie graficznego interfejsu użytkownika była biblioteka AWT (ang. Abstract Window Toolkit). Fala niezadowolenia wśród programistów sprawiła iż ograniczone i mało efektowne elementy graficzne biblioteki AWT zostały zastąpione przez zbiór komponentów dziś znanych jako Swing. W bibliotece Swing znajdziemy większość podstawowych komponentów występujących we współczesnych systemach operacyjnych włącznie z obsługą okien, natomiast w bibliotece AWT znajdziemy obsługę zdarzeń, co w połączeniu daje nam wystarczające narzędzia do stworzenia graficznego interfejsu. Warto zauważyć iż Swing nie korzysta z domyślnych ustawień systemu jeśli chodzi o wygląd komponentów, dzięki czemu ten sam program wygląda identycznie na każdym systemie operacyjnym. Alternatywą dla biblioteki Swing może być biblioteka SWT związana z projektem Eclipse. W odróżnieniu do Swing, komponenty korzystają z wyglądu komponentów systemu operacyjnego.

2.2.3 Narzędzia

Aplikacja w całości została zrealizowana w programie Netbeans 6.9.1 wspierającym testowanie, kompilację budowanie aplikacji napisanych w języku Java. Innym popularnym narzędziem może być Eclipse, jednak lepsze narzędzia testujące (ang. profiler) przeważały o wyborze pierwszej aplikacji.

2.3 Wstępna analiza problemu

2.3.1 Sterowanie

Aby zrealizować część odpowiedzialną za sterowanie postacią, należy użyć klasy pośredniej pomiędzy warstwą logiki silnika gry, a warstwą komunikacji z graczem. Przy takim rozwiązaniu sygnały z klawiatury bądź innych urządzeń peryferyjnych są przekazywane do warstwy pośredniczącej. Dzięki takiemu wyjściu możemy łatwo zmienić źródło sygnałów trafiających do postaci z bezpośrednich wciśnień klawiszy na akcje przechowywane w chromosomie. Warstwa pośrednia odpowiada wówczas za sterowanie postacią gracza w taki sam usystematyzowany sposób, niezależnie od źródła sygnałów. Z punktu widzenia logiki gry nie ma różnicy pomiędzy sygnałami z klawiatury, a wygenerowanymi akcjami.

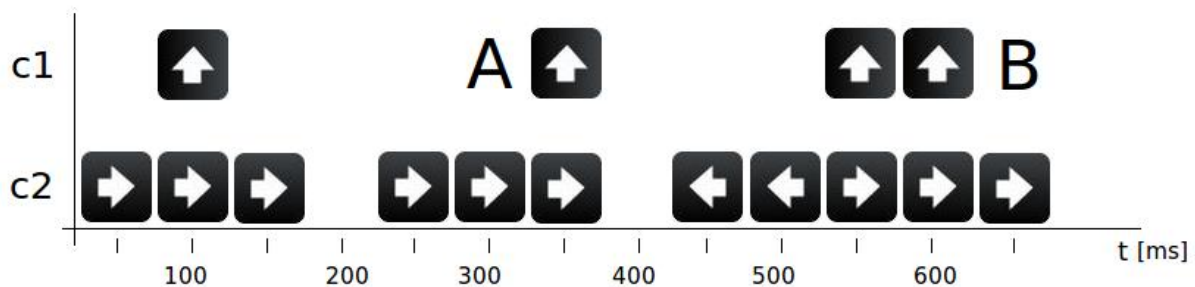
2.3.2 Projekt chromosomu

Kolejnym ważnym elementem jest odpowiednie zaprojektowanie struktury chromosomu. Dwa najbardziej trafne rozwiązania opierają się na dwóch zmiennych występujących w środowisku gry: czasie oraz pozycji gracza.

1. Czas który upłynął od rozpoczęcia danej instancji przejścia.

To rozwiązanie zakłada podejmowanie akcji w grze w zależności od czasu który upłynął od jej rozpoczęcia. Warto zauważyć iż nie jesteśmy ograniczeni położeniem postaci na planszy, dzięki czemu możliwe są takie operacje jak brak akcji, czy powrót do początku planszy jeśli to korzystne. Istotną wadą tego rozwiązania byłaby duża podatność algorytmu na zapętlanie się, lub wykonywanie dużej ilości mało przydatnych ruchów. Można łatwo zauważyć że przy równym prawdopodobieństwie ruchu w lewo i prawo, postać tylko nieznacznie będzie oddalać się punktu startowego. Prostym rozwiązaniem tego problemu jest przyporządkowanie pewnego prawdopodobieństwa każdej akcji, dzięki czemu możemy założyć że preferowanym kierunkiem jest np. ruch postaci w prawo, nie tracąc możliwości minimalnego ruchu w lewo jeśli to korzystne. Pewnym utrudnieniem może być krzyżowanie tego

typu chromosomów. Ponieważ akcje postaci w większości przypadków mają sens w kontekście jej aktualnego położenia, o tyle klasyczne krzyżowanie poprzez "cięcie" chromosomu na dwie części może okazać się kosztowne. Wybranie losowego punktu przecięcia i złączenie ze sobą dwóch chromosomów nie jest dobrym rozwiązaniem. Po połączeniu otrzymamy niespójny ciąg ruchów, które będą miały niewiele wspólnego z aktualną pozycją gracza na mapie, wobec czego będą nieużyteczne. Można temu zapobiec zapewniając łączenie się chromosomów jedynie w punktach w których postać w obu momentach znajduje się w tym samym lub zbliżonym miejscu. Wyznaczenie takich punktów może okazać się kosztowne. Przeszukiwanie punktów wspólnych można zrealizować w czasie $O(n * \log_2 n)$ najpierw sortując tablice obu osobników odpowiadające za ruch w chromosomie. Tablice sortujemy względem współrzędnej X aktualnego położenia gracza dla każdej z akcji, a następnie liniowo przechodząc po obu tablicach osobników, szukając punktów wspólnych. Wówczas widać iż trzeba przechowywać dane na temat położenia w chromosomie, co jest nieco niespójne z ideą poruszania się względem czasu. Wstępny schemat takiego rozwiązania mógłby wówczas wyglądać tak jak na rysunku 3.



Rysunek 3: Sterowanie względem czasu.

Tablice c1, c2 oznaczają odpowiednio tablicę odpowiadającą za akcje specjalne (np. skok), oraz tablicę odpowiadającą jedynie za ruch kierunkowy.

Lepszym rozwiązaniem jest realizacja krzyżowania nie poprzez klasyczne podejście, lecz modelowane statystycznie: Potomstwo nie otrzymuje bezpośrednich fragmentów chromosomu, lecz losuje za każdym razem nowe ruchy, natomiast chro-

mosomy populacji rodzicielskiej zwiększają prawdopodobieństwo wylosowania najczęściej występujących ruchów. Schemat takiego rozwiązania zaprezentowany jest na Rys. 4, gdzie przedstawione zostało krzyżowanie populacji składającej się z 3 osobników (p_1, p_2, p_3) oraz sposób liczenia nowych prawdopodobieństw w danym punkcie chromosomu. Do tego rozwiązania włączamy stałą decydującą o wadze jakie ma krzyżowanie: Jeśli rodzic posiada pewną akcję A w pewnym miejscu swojego chromosomu, wówczas potomkowi losowana jest nowa akcja, z tą różnicą iż akcja A ma teraz prawdopodobieństwo wylosowania $p_A = p_A + p_A * k$. Jeśli populacja rodzicielska składa się z n rodziców, oraz m z nich posiada w danym momencie taką samą akcję wówczas nowe prawdopodobieństwo wylosowania akcji (tylko dla tego rozpatrywanego miejsca) wynosi




$$p_A = p_A + \sum_{i=1}^m (p_A * k) = p_A + (p_A * k) * m.$$

Dzięki takiemu podejściu zachowujemy ideę krzyżowania oraz znacznie upraszczamy cały proces.



















2. Aktualna pozycja gracza.

O ile poprzednie rozwiązanie dawało większą swobodę ruchu po mapie, to było jednak mało optymalne pod względem osiągania szybko dobrych wyników. Jeśli założymy iż akcje przechowywane w chromosomie mają być aktywowane w momencie osiągnięcia przez gracza danej pozycji na osi X mapy, wówczas uprościmy cały mechanizm krzyżowania (już nie musimy szukać punktów wspólnych, gdyż dwa dowolne indeksy w obu tablicach i, j gwarantują nam takie samo położenie gracza na mapie gdy $i = j$). Oprócz tego przy założeniu że planszę da się rozwiązać poruszając się tylko w prawo upraszcza to większość operacji w algorytmie. Innym udogodnieniem będzie uproszczenie samego typu przechowywanych danych. Ponieważ rezygnujemy z postojów i ruchu w lewo, równie dobrze możemy zrezygnować z tablicy przechowującej te informacje.




Prawdopodobieństwa wystąpienia:

	0.05
	0.6
	0.3
brak akcji	0.5

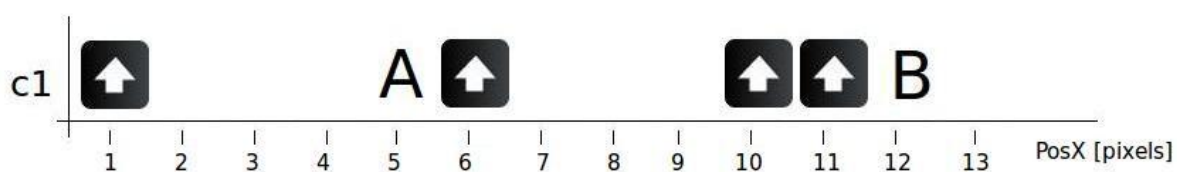
$k = 1.5$

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
p1											
p2											
p3											

Prawdopodobieństwa wystąpienia w czasie t1 ze stałą $k = 1.5$:

	$0.05 + 0.05 * k = 0.125$
	0.6
	$0.3 + 0.3 * k + 0.3 * k = 1.2$
brak akcji	0.5

Rysunek 4: Krzyżowanie statystyczne.



Rysunek 5: Sterowanie względem pozycji gracza.

To podejście posiada jednak kilka poważnych wad i wymaga pewnych ograniczających założeń. Plansza musi być ukierunkowana, i być rozwiązywalna przy ciągłym ruchu w określonym kierunku. Jest to rozwiązanie działające jedynie dla bardzo

wąskiej grupy gier platformowych (np. wspomniane wcześniej Super Mario Brothers). Przeniesienie systemu do zastosowania w grze platformowej o nieco innym schemacie ruchu (np. rozwiązywania labiryntu) może okazać się trudne i wymagające dużych zmian w samym algorytmie. Jeśli chcemy tego uniknąć i traktować system bardziej ogólnie, lepiej jest skorzystać z pierwszego podejścia.

2.4 Moduł silnika i symulatora gry

2.4.1 Wymagania funkcjonalne

System nie wymaga różnicowania użytkowników ze względu na rolę. Wymagania funkcjonalne wyglądają następująco:

- **Poruszanie się w środowisku gry za pomocą klawiatury.**

Podstawowym wymogiem systemu jest implementacja silnika prostej gry platformowej pozwalającego na poruszanie się postacią po mapie. Sterowanie zrealizowane powinno być intuicyjne i analogiczne do przyjętych rozwiązań w grach platformowych. Ponieważ oprawa graficzna nie jest priorytetem w projekcie, wystarczy prosta reprezentacja obiektów za pomocą prostokątów.

- **Wczytanie mapy do środowiska gry.**

System powinien pozwalać na wczytywanie map z plików tekstowych do bieżącego środowiska symulującego przebieg gry. Wówczas bieżąca gra zostaje przerwana i inicjowany jest nowy przebieg gry na nowowczytanej mapie.

- **Wczytanie nowej logiki do środowiska gry.**

Ponieważ system powinien być ogólnym systemem rozwiązującym gry platformowe, dostępnych powinno być kilka przykładowych gier, różniących się między sobą pod względem logiki. Podobnie jak w powyższym przypadku bieżąca gra powinna zostać przerwana i powinien zostać zainicjowany nowy przebieg gry z nową logiką.

- **Przejsięcie w tryb treningu populacji.**

Po przejściu w tryb treningu populacji użytkownikowi odbierana jest możliwość poruszania się postacią po ekranie. Jeśli nie istnieje jeszcze zainicjowana żadna populacja początkowa, następuje wylosowanie pierwszej populacji, po czym system rozpoczyna trening populacji w danym środowisku gry, zgodnie z danymi ustawieniami w systemie. Przechodzenie pomiędzy treningiem populacji a grą użytkownika powinno być możliwe w obie strony. Wówczas jeśli użytkownik zainicjuje trening populacji, następnie przejdzie w tryb swobodnej gry, a ostatecznie znowu rozpocznie trening populacji, domyślną populacją jest ta która została zapisana podczas ostatniego treningu.

- **Zainicjowanie nowej populacji.**

Może okazać się konieczne zainicjowanie nowej populacji podczas działania systemu, np. gdy populacja wpadła w stagnację. Inicjowanie nowej populacji podczas zmiany logiki, ustawień bądź wczytywania mapy jest realizowane automatycznie.

- **Otworzenie okna ustawień.**

System ze względu na warstwę genetyczną powinien być w łatwo konfigurowalny. Po wyświetleniu okna ustawień użytkownik ma możliwość zmiany poszczególnych parametrów algorytmu bądź symulacji gry.

- **Zastosowanie nowych ustawień.**

Ponieważ niektóre ustawienia wymagają porzucenia aktualnego wyniku populacji i rozpoczęcia symulacji od początku (np. rozmiar chromosomu). Użytkownik jest o tym fakcie informowany i pytany o ponowne uruchomienie algorytmu. W przypadku gdy nie jest to konieczne, algorytm nie zostaje przerwany, a użytkownik wedle woli może to uczynić własnoręcznie.

- **Otworzenie okna populacji.**

Podczas działania algorytmu powinien być możliwy podgląd aktualnej populacji. Użytkownikowi widoczna jest wówczas lista osobników, oraz krótki opis każdego

z nich: Wynik końcowy, ilość zebranych punktów, czas działania, wartość funkcji przystosowania. Dodatkowym elementem jest możliwość przerywania aktualnego przebiegu gry i uruchomienie tymczasowo nowego przebiegu dla dowolnego osobnika z listy wybranego przez użytkownika. Po zakończeniu przebiegu algorytm powraca do treningu populacji.

- **Otworzenie okna osobnika.**

Bezpośrednio z okna populacji użytkownik ma możliwość podejrzenia szczegółów dotyczących osobników. Wówczas otwierane jest nowe okno zawierające wszystkie informacje na temat danego osobnika takie jak długość tablicy chromosomu bądź tekstowa reprezentacja całego chromosomu.

- **Przyspieszenie pracy algorytmu.**

Podczas treningu populacji często niepotrzebne jest użytkownikowi śledzenie ruchów algorytmu szczególnie w początkowym stadium. Aby szybciej osiągnąć wyniki działania algorytmu rozgrywka jest przyspieszana przez wstrzymanie wyświetlania grafiki na mapie. Oprócz tego opóźnienie konieczne do osiągnięcia 60 klatek na sekundę podczas wyświetlania zostaje wyłączone - algorytm w tle przeprowadza symulacje.

2.4.2 Wymagania нефunkcjonalne

- **Prędkość działania aplikacji.**

Ponieważ aplikacja wymaga przeprowadzenia wielu symulacji gry do osiągnięcia wyniku, konieczne jest optymalne zaprojektowanie funkcji biorących udział w działaniu algorytmu genetycznego.

- **Elastyczność.**

System powinien być napisany w taki sposób aby możliwe było go późniejsze rozwijanie, szczególnie jeśli chodzi o rozszerzanie systemu o nowe logiki gier. System

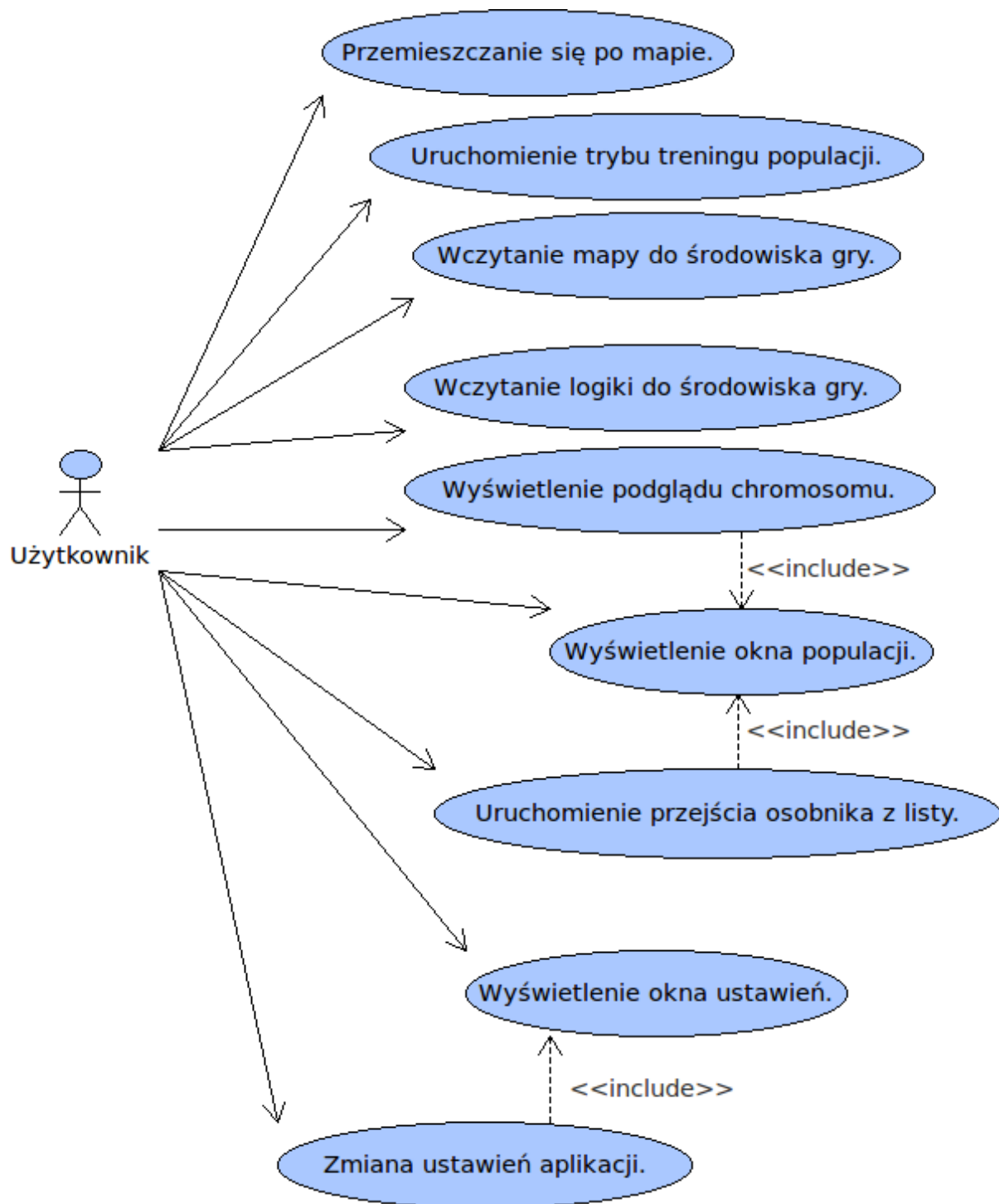
musi zostać rozpatrzony jako generalny system rozwiązujący gry czasu rzeczywistego opierające się na 4 klawiszach kierunkowych i do 4 klawiszy specjalnych.

- **Oprawa Graficzna.**

Oprawa graficzna samej gry nie jest istotna w systemie. Jeśli system ma być elastyczny jeśli chodzi o mechanikę gier, wprowadzenie bogatej grafiki niepotrzebnie skomplikuje proces dodawania nowego typu gry do systemu. Innym powodem zachowania prostej grafiki jest swoboda rozmiarów obiektów. W większości gier platformowych stosowana jest grafika rastrowa która przy skalowaniu obiektów bez zachowania proporcji wygląda źle. Rozwiązanie tego problemu wiązałoby się z generowaniem grafiki proceduralnie bądź używaniem grafiki wektorowej, co jednak odbiega od istoty pracy.

2.4.3 Diagram przypadków użycia

Diagram przypadków użycia został przedstawiony na Rys. 6.



Rysunek 6: Diagram przypadków użycia.

2.4.4 Opis tekstowy przypadków użycia

We wszystkich przypadkach użycia aktorem jest użytkownik aplikacji.

- Opis przypadku użycia **Przemieszczanie się po mapie** .

1. Podstawowy ciąg zdarzeń:

- (a) Po uruchomieniu instancji gry w głównym oknie aplikacji wyświetlona zostaje mapa wraz ze znajdującymi się na niej obiektami.
- (b) Użytkownik za pomocą klawiszy na klawiaturze przesyła dane dotyczące ruchu i akcji specjalnych do środowiska gry.
- (c) Postać gracza znajdująca się w grze reaguje na żądane akcje, logika gry decyduje o reakcjach obiektów w środowisku na postępy gracza.
- (d) Gracz wchodzi w kolizję z jednym z obiektów kończących grę z danym wynikiem pozytywnym bądź negatywnym.
- (e) Po zakończeniu gry mapa jest ponownie wczytywana i automatycznie rozpoczynana jest nowa instancja gry.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik w trakcie gry uruchamia tryb treningu populacji.
- (b) Użytkownik w trakcie gry wczytuje nową mapę, bądź nową logikę gry, przez co gra jest przerywana i rozpoczynana od nowa.

3. Zależności czasowe:

- (a) Częstotliwość wykonania: Samo przemieszczanie się po mapie jest akcją o charakterze ciągłym. Rozpoczynanie akcji poruszania się po mapie ze stanu treningu populacji jest bliżej nieokreślone, lecz może być wykonywane średnio 1-2 razy na minutę działania aplikacji.
- (b) Typowy czas realizacji: 1 minuta.
- (c) Maksymalny czas realizacji: nieokreślony.

4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:

- (a) Jeśli użytkownik zakończy tryb poruszania się po mapie przez przełączenie na tryb treningu populacji, wówczas traci kontrolę nad postacią i nie ma wpływu na akcje rozgrywane w środowisku gry.

- Opis przypadku użycia **Uruchomienie trybu treningu populacji.** .

1. Podstawowy ciąg zdarzeń:

- (a) Podczas działania aplikacji gracz uruchamia tryb treningu populacji.
- (b) Jeśli jest to pierwsze uruchomienie trybu treningu wówczas nie istnieje wcześniejsza populacja, wówczas losowana jest populacja początkowa. W przeciwnym wypadku, symulacja zaczyna się od ostatniej populacji wygenerowanej przez system.
- (c) System kolejno przeprowadza wszystkie kroki algorytmu genetycznego na bieżącej populacji.
- (d) Jeśli wyświetlone jest okno widoku populacji jest ono aktualizowane po każdym przebiegu gry.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik wybiera w oknie populacji osobnika i przeprowadza jego tymczasową symulację w środowisku, przez co trening jest chwilowo wstrzymany. Po symulacji gry danego osobnika trening jest wznowiany od ostatniego osobnika.
- (b) Użytkownik wczytuje nową logikę bądź mapę, przez co aktualny przebieg gry jest przerywany. Zostaje uruchomiona nowa gra i trening populacji jest automatycznie uruchamiany od początku.
- (c) Użytkownik zmienia ustawienia algorytmu genetycznego. Jeśli są to zmiany wymagające ponownego uruchomienia gry jest wyświetlony komunikat potwierdzający operację, w innym wypadku zmiany następują bez zaburzania aktualnego przebiegu treningu.
- (d) Użytkownik uruchamia tryb przyspieszonego treningu, przez co wyłączona zostaje aktualizacja stanu gry w oknie aplikacji.

3. Zależności czasowe:

- (a) Częstotliwość wykonania: Przełączanie na trening populacji może być średnio uruchamiane 1-2 razy w ciągu minuty działania aplikacji.
 - (b) Typowy czas realizacji: 10 sekund - 5 minut. Samo działanie treningu populacji stanowić może około 90% czasu działania aplikacji.
 - (c) Maksymalny czas realizacji: nieokreślony.
4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:
- (a) Jeśli użytkownik uruchomi tryb poruszania się po mapie wówczas przywrócone zostaje wyświetlanie stanu gry (jeśli był włączony tryb przyspieszonego treningu). Użytkownik uzyskuje kontrolę nad postacią gracza.
- Opis przypadku użycia **Zmiana ustawień aplikacji.** .
1. Podstawowy ciąg zdarzeń:
- (a) Użytkownik wyświetla okno ustawień aplikacji.
 - (b) W polach tekstowych użytkownik dokonuje zmian na poszczególnych parametrach aplikacji
 - (c) Użytkownik zatwierdza zmiany przyciskiem.
 - (d) System sprawdza czy zmiany nie wymagają przerwania bieżącej gry i ponownego uruchomienia algorytmu.
 - i. Jeśli zmiany nie wymagają ponownego uruchomienia algorytmu, zmiany zostają wprowadzone bez przerywania bieżącej instancji gry.
 - ii. W wypadku gdy ponowne uruchomienie jest konieczne wyświetlone zostaje okno dialogowe. Użytkownik jest pytany o ponowne uruchomienie algorytmu, ma wówczas on do dyspozycji zgodę na operację, bądź cofnięcie zmian wymagających ponownego uruchomienia.
 - iii. Po podjęciu decyzji okno dalej pozostaje otwarte, a zgodnie z wyborem zmiany zostają dokonane bądź nie.
 - (e) Użytkownik dalej może dokonywać zmian ustawieniach aplikacji.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik dokonuje zmian, lecz zamiast zatwierdzić je przyciskiem - zamyka okno ustawień. Wówczas po ponownym otwarciu ładowane są aktualne ustawienia a zmiany wprowadzone przez użytkownika nie są zapamiętywane.

3. Zależności czasowe:

- (a) Częstotliwość wykonania: 0-5 razy w ciągu działania aplikacji.
- (b) Typowy czas realizacji: 1 minuta.
- (c) Maksymalny czas realizacji: nieokreślony.

4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:

- (a) brak

• Opis przypadku użycia **Wyświetlenie podglądu populacji.** .

1. Podstawowy ciąg zdarzeń:

- (a) Podczas treningu populacji użytkownik z menu w oknie głównym gry wybiera opcję "Population".
- (b) Okno zostaje wyświetlone, i utworzony zostaje komponent JList wypełniony aktualnymi osobnikami z populacji.
- (c) Jeśli dany osobnik na liście już przeprowadził symulację, widoczne są wartości punktowe zdobyte przez danego osobnika, czas przejścia oraz wynik końcowy.
- (d) Podczas działania algorytmu lista jest automatycznie aktualizowana.
- (e) Użytkownik zamyka okno populacji.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik zaznacza osobnika i wyświetla o nim szczegóły.
 - i. Uruchamiany jest przypadek użycia "Wyświetlenie podglądu chromosomu".

- (b) Użytkownik zaznacza osobnika i uruchamia jego przejście gry.
 - i. Uruchamiany jest przypadek użycia “Uruchomienie przejścia osobnika z listy.”.
- 3. Zależności czasowe:
 - (a) Częstotliwość wykonania: 0-5 razy w ciągu działania aplikacji.
 - (b) Typowy czas realizacji: 1 minuta.
 - (c) Maksymalny czas realizacji: nieokreślony.
- 4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:
 - (a) brak
- Opis przypadku użycia **Wyświetlenie podglądu chromosomu.** .
 - 1. Podstawowy ciąg zdarzeń:
 - (a) Użytkownik zaznacza osobnika z listy w oknie populacji.
 - (b) Przyciskiem “Show details” wyświetla okno szczegółów na temat danego osobnika.
 - (c) Dane na temat osobnika są w postaci tekstowej - użytkownik może je skopiować do schowka.
 - (d) Użytkownik zamyka okno szczegółów.
 - 2. Alternatywne ciąg zdarzeń:
 - (a) Użytkownik otwiera okna szczegółów dla kilku osobników po kolei - Otwiera się kilka osobnych okien.
 - 3. Zależności czasowe:
 - (a) Częstotliwość wykonania: 0-5 razy w ciągu działania aplikacji.
 - (b) Typowy czas realizacji: 20 sekund.
 - (c) Maksymalny czas realizacji: nieokreślony.
 - 4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:

(a) brak.

- Opis przypadku użycia **Uruchomienie przejścia osobnika z listy.** .

1. Podstawowy ciąg zdarzeń:

- (a) Użytkownik zaznacza osobnika z listy w oknie populacji.
- (b) Przyciskiem **Żun selected** zatwierdza wybór osobnika.
- (c) Jeśli jest aktualnie przeprowadzana symulacja wówczas zostaje ona przerwana i rozpoczyna się przejście gry wybranego wcześniej osobnika.
- (d) Po zakończeniu przejścia system ponownie wraca do treningu populacji i zaczyna od ostatniego osobnika.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik uruchamia przejście osobnika w trybie przemieszczania się po mapie. Symulacja nie zostaje wówczas przeprowadzona.

3. Zależności czasowe:

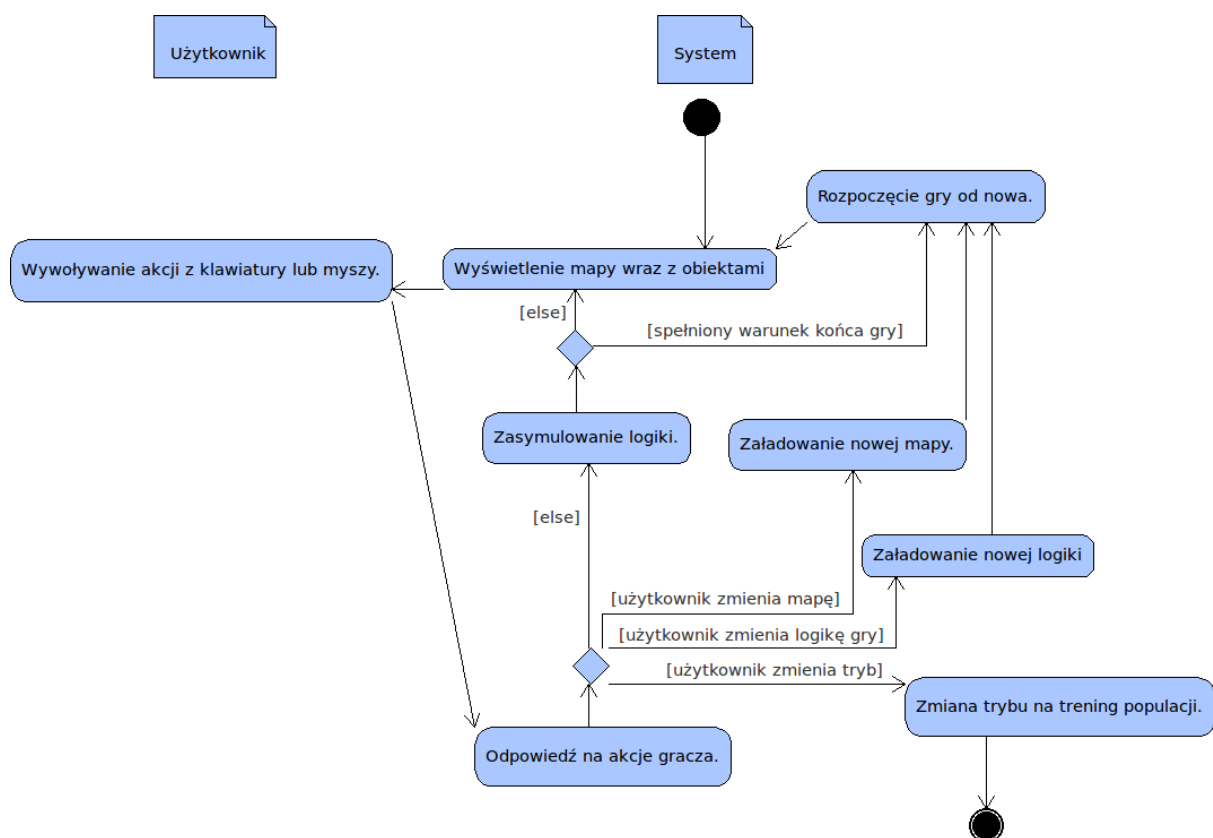
- (a) Częstotliwość wykonania: 0-10 razy w ciągu działania aplikacji.
- (b) Typowy czas realizacji: zależny od długości przejścia osobnika 1 - 40 sekund.
- (c) Maksymalny czas realizacji: nieokreślony.

4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:

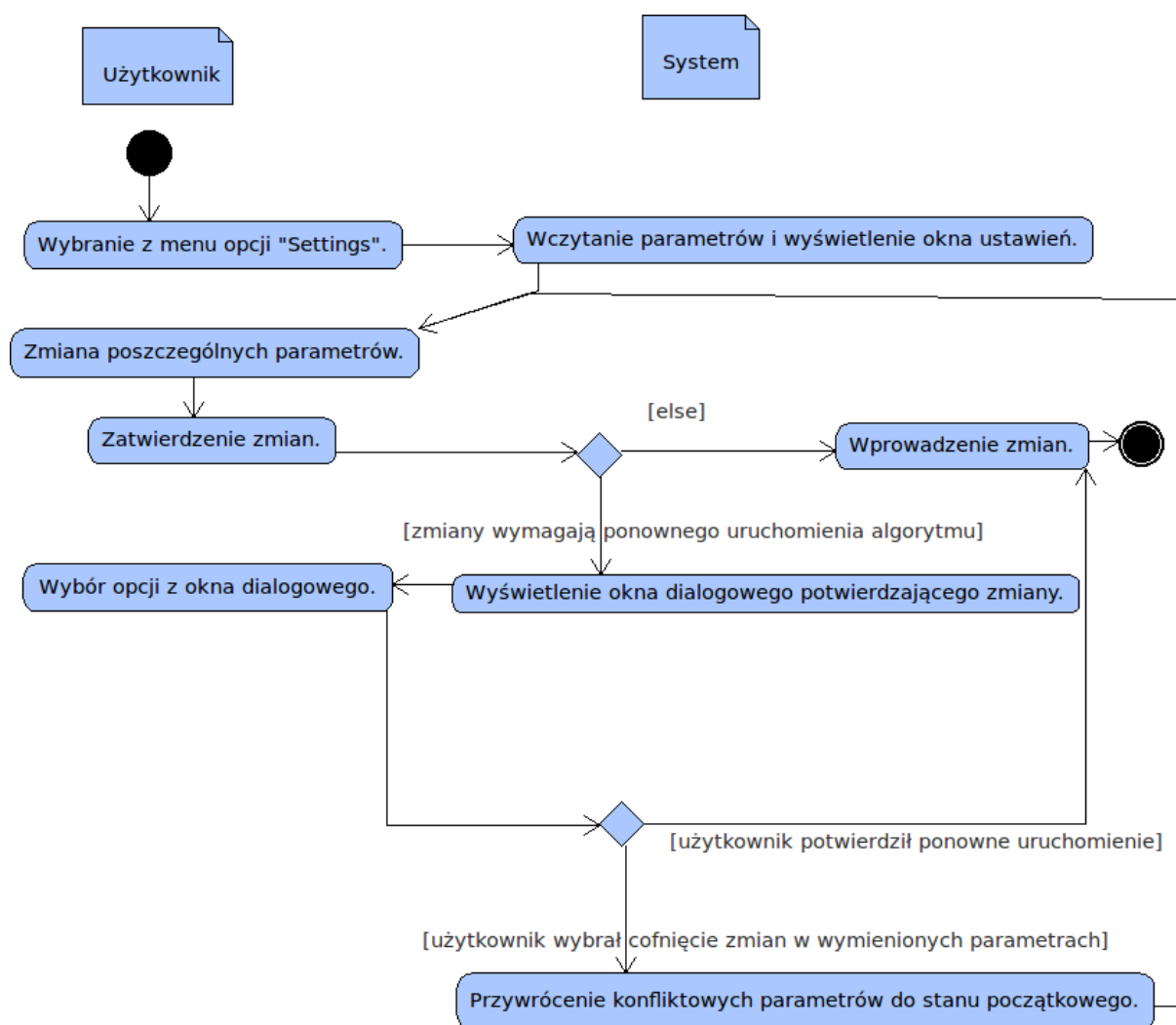
(a) brak.

2.4.5 Diagramy czynności

Poniżej zostaną przedstawione diagramy najistotniejszych czynności w systemie.



Rysunek 7: Diagram czynności “Przemieszczanie się po mapie”.



Rysunek 8: Diagram czynności “Zmiana ustawień aplikacji”.

2.4.6 Diagram stanów

System sam w sobie nie posiada wielu stanów jakie może przyjąć. Dwa główne stany to gra użytkownika w której akcje z klawiatury są interpretowane jako ruchy gracza, oraz tryb treningu populacji.



Rysunek 9: Diagram czynności “Diagram stanów systemu”.

Stany dla Systemu (Rys. 9): Użytkownik po uruchomieniu aplikacji może poruszać się postacią po ekranie (stan “Tryb poruszania się przez użytkownika”). Jeśli zmieni tryb działania systemu automatycznie przechodzi do trybu treningu (stan “Tryb treningu populacji”). Będąc w stanie treningu populacji użytkownik może ponownie przejść w tryb swobodnego poruszania się. Użytkownik może wyłączyć wyświetlanie grafiki i tym samym przyspieszyć proces treningu (stan “Tryb przyspieszonego treningu”). Ze stanu treningu przyspieszonego użytkownik może przejść do trybu poruszania się postacią po ekranie poprzez zmianę trybu, bądź do trybu treningu populacji poprzez włączenie wyświetlania grafiki.

2.5 Moduł edytora map

2.5.1 Wymagania funkcjonalne

System nie wymaga różnicowania użytkowników ze względu na role. Wymagania funkcjonalne wyglądają następująco:

- **Otworzenie okna edytora map.**

Ważnym elementem systemu jest narzędzie pozwalające tworzyć nowe mapy oraz modyfikować istniejące. Powinno być ono dostępne jako osobne okno edytora map.

- **Wczytywanie mapy do edytora.**

Użytkownik powinien mieć możliwość edycji dowolnej wcześniej stworzonej mapy, w tym celu powinien móc z menu wybrać odpowiednią opcję pozwalającą na wczytanie pliku mapy z dysku. Wczytywanie może być zrealizowane analogicznie do wczytywania mapy do środowiska gry.

- **Zapis mapy do pliku.**

Podobnie jak odczyt, zapis mapy powinien być dostępny dla użytkownika z menu. Dzięki zapisowi mapy na dysk twardy użytkownik ma możliwość przechowywania wcześniej tworzonych map, a co ważniejsze otworzenie ich bezpośrednio w symulatorze gry.

- **Edycja mapy za pomocą narzędzi graficznych.**

Edycja mapy powinna być intuicyjna i prosta nawet dla osoby nie znającej szczegółów formatu zapisu mapy. Ponieważ każdy obiekt w grze posiada prostokątny obszar kolizji, edytor powinien pozwalać na łatwe tworzenie prostokątnych obiektów. Zrealizowane może być to przez wyznaczanie dwóch rogów prostokąta za pomocą metody “przeciągnij i upuść”. Narzędzia powinny być dostępne po wybraniu odpowiedniej ikony w panelu narzędzi edytora map. Przydatne w edycji map mogą okazać się często używane akcje takie jak cofnięcie ostatniej zmiany, zaznaczenie elementów w edytorze i usuwanie ich, bądź czyszczenie całej mapy.

2.5.2 Wymagania niefunkcjonalne

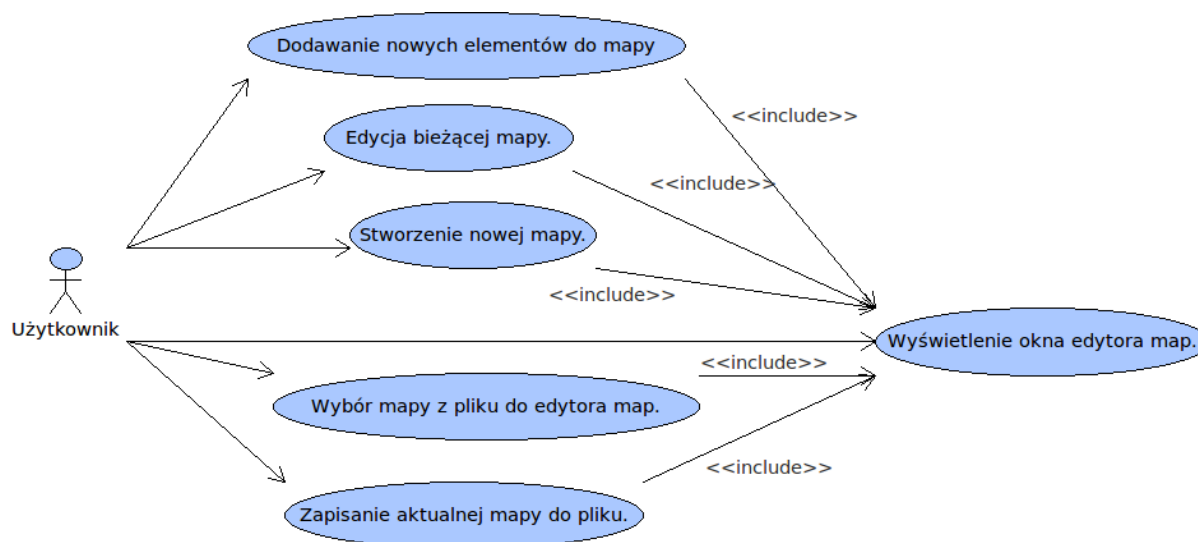
- **Otwarty format mapy.**

Istotną rzeczą może być tutaj przechowywanie mapy w pliku tekstowym. O ile zapis binarny może okazać się szybszy i bardziej kompaktowy, to ważniejsze jest jednak

umożliwienie użytkownikom edycji mapy ręcznie, bądź poprzez programy trzecie.
Dzięki temu możliwe będzie generowanie np. bardzo długich losowych map.

2.5.3 Diagram przypadków użycia

Diagram przypadków użycia został przedstawiony na Rys. 10.



Rysunek 10: Diagram przypadków użycia modułu edytora map.

2.5.4 Opis tekstowy przypadków użycia

We wszystkich przypadkach użycia aktorem jest użytkownik aplikacji.

- Opis przypadku użycia **Edycja bieżącej mapy.** .

1. Podstawowy ciąg zdarzeń:

- Użytkownik klika w przycisk menu dotyczący edytora map.
- Wyświetlone zostaje nowe okno aplikacji zawierające przyciski dotyczące edycji mapy.
- Do edytora map domyślnie zostaje wczytana bieżąca mapa wczytana do środowiska symulatora.

- (d) Użytkownik wybiera obiekty gry użyciu przycisków znajdujących się w menu okna a następnie metodą przeciągnij-upuść rysuje prostokątne obiekty odpowiadające za obiekty w grze.
- (e) Użytkownik z menu wybiera opcję zapisu mapy do pliku.
- (f) Wyświetlone zostaje okno dialogowe zapisu pliku w którym użytkownik wybiera nazwę pliku.
- (g) Użytkownik zamyka okno edytora map.

2. Alternatywne ciąg zdarzeń:

- (a) Użytkownik wczytuje nową mapę do edytora map.
 - i. Wyświetlone zostaje okno dialogowe dotyczące otwarcia pliku z systemu.
 - ii. Wszystkie obiekty z planszy zostają usunięte, i wczytywane są nowe z wybranej wcześniej mapy.

3. Zależności czasowe:

- (a) Częstotliwość wykonania: 0-5 razy w ciągu działania aplikacji.
- (b) Typowy czas realizacji: 5 minut.
- (c) Maksymalny czas realizacji: nieokreślony.

4. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:

- (a) Użytkownik po zapisaniu stworzonej mapy do pliku ma możliwość wczytania mapy z dysku do symulatora gry i przeprowadzenia treningu populacji na nowej mapie.

• Opis przypadku użycia **Stworzenie nowej mapy.** .

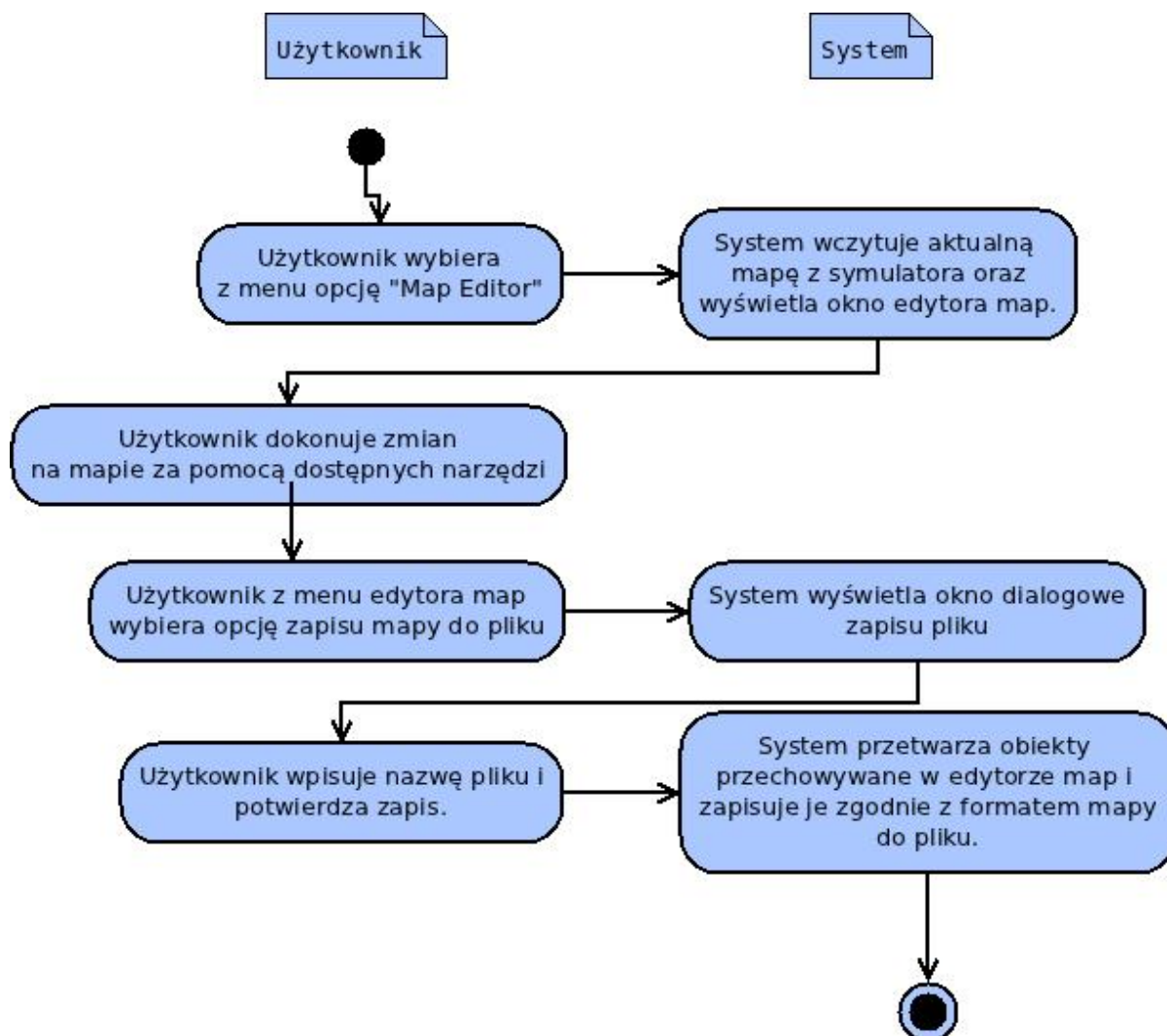
1. Podstawowy ciąg zdarzeń:

- (a) Użytkownik wybiera z menu przycisk dotyczący edytora map.
- (b) Do edytora map domyślnie zostaje wczytana bieżąca mapa wczytana do środowiska symulatora.

- (c) Użytkownik z panelu narzędzi edytora map wybiera usunięcie wszystkich obiektów z mapy.
 - (d) System usuwa obiekty z edytora map.
 - (e) Użytkownik za pomocą narzędzi buduje elementy nowej mapy.
 - (f) Użytkownik z menu wybiera opcję zapisu mapy do pliku.
 - (g) Wyświetlone zostaje okno dialogowe zapisu pliku w którym użytkownik wybiera nazwę pliku.
 - (h) Użytkownik zamyka okno edytora map.
2. Zależności czasowe:
- (a) Częstotliwość wykonania: 0-5 razy w ciągu działania aplikacji.
 - (b) Typowy czas realizacji: 5 minut.
 - (c) Maksymalny czas realizacji: nieokreślony.
3. Wartości uzyskiwane przez aktorów po zakończeniu przypadku użycia:
- (a) Użytkownik po utworzeniu nowej mapy ma możliwość wczytania jej w symulatorze.

2.5.5 Diagramy czynności

Poniżej zostaną przedstawione diagramy najistotniejszych czynności w systemie dotyczące modułu edytora map.



Rysunek 11: Diagram czynności "Edycja bieżącej mapy".

2.6 Diagram klas

Diagram klas projektu przedstawiony został na rysunku 12. Poniżej zostały opisane poszczególne klasy w kolejności alfabetycznej.

1. Actor

Actor jest klasą dziedziczącą po klasie WorldObject. Reprezentuje ona obiekt postaci poruszającej się po ekranie w środowisku gry. Ponieważ instancja obiektu Actor jest przemieszczającym się obiektem niezbędne było dodanie wektora prędkości do klasy który odpowiada za zmianę pozycji w czasie. Klasa posiada metody takie jak moveLeft, moveRight, moveUp oraz moveDown pozwalające na łatwą

zmianę kierunku ruchu aktora. Metody `tickX`, `tickY` aktualizują pozycje gracza. Aby wykryć kolizję możliwe jest wywołanie metody `previewX` (odp. `previewY`) która zwraca “przyszłą” pozycję obiektu po wykonaniu przez niego ruchu w osi X (odp. osi Y).

2. **Bonus**

Klasa `Bonus` jest abstrakcyjną klasą dziedziczącą po klasie `WorldObject`. Odpowiada ona za obiekty wpływające na przebieg gry. Kolizja aktora z obiektem powoduje wywołanie metody `evalCollision`, która decyduje o efekcie zdarzenia. Obiekty `Bonus` posiadają przypisaną wartość punktową która w zależności od typu obiektu różnie wpływa na rozgrywkę.

3. **BonusCoin**

Klasa `BonusCoin` reprezentuje jedynie punkty umieszczone na planszy. Kolizja z obiektem klasy `BonusCoin` powoduje zwiększenie licznika punktów gracza bądź osobnika populacji. Poza dodaniem punktów obiekt klasy `BonusCoin` nie wpływa bezpośrednio na przebieg gry.

4. **BonusLose**

Obiekt klasy `BonusLose` reprezentuje obiekt kończący grę ze skutkiem negatywnym. Kolizja aktora z obiektem automatycznie kończy przebieg gry ze skutkiem `RESULT_LOST`.

5. **BonusWin**

Obiekt klasy `BonusWin` stanowi cel każdej gry. Kolizja z obiektem typu `BonusWin` automatycznie kończy rozgrywkę ze skutkiem `RESULT_WON`, co jest brane pod uwagę podczas liczenia funkcji przystosowania.

6. **Camera**

Obiekt `Camera` przechowuje wektor przesunięcia obrazu i jest używany do przesuwania ekranu za pomocą myszy. Jest ona wykorzystywana podczas symulacji do

“podążania” za obiektem gracza, bądź do swobodnego przesuwania ekranu gry za pomocą myszy. Analogicznie w edytorze map użytkownik ma możliwość przesuwania ekranu za pomocą myszy, co upraszcza tworzenie dużych map. Obiekt Camera przechowuje również referencję do aktualnie śledzonego obiektu. Metoda update pozwala na aktualizowanie kamery wraz z przesuwaniem się postaci po ekranie. metody translateScreen oraz translateScreenNeg odpowiednio przesuwają ekran o żądany wektor przesunięcia.

7. Config

Klasa Config przechowuje zmienne dotyczące konfiguracji programu, wykorzystywane w celu wyszukiwania błędów. Przechowuje ona ustawienia wyświetlania informacji o pozycji każdego obiektu w grze, bądź dodatkowych zmiennych takich jak czas który upłynął od początku gry. Domyślnie informacje te nie są widoczne, lecz można je włączyć z panelu konfiguracyjnego.

8. Controller

Klasa Controller jest klasą pośredniczącą w wymianie informacji pomiędzy sygnałami z klawiatury, bądź akcjami zapisanymi w chromosomie, a środowiskiem gry. Odseparowuje ona obie warstwy dzięki czemu możliwe jest pisanie logiki gry, bądź reakcji na poszczególne akcje, niezależnie od źródła sygnału. Klasa Controller jest klasą abstrakcyjną i uzupełnia interfejsy takie jak KeyListener, MouseListener, MouseMotionListener (tym samym implementuje wszystkie związane z nimi metody), dzięki czemu wszelkie zdarzenia wywołane w oknie gry zostają przechwycone i odpowiednio obsłużone. Klasa Controller posiada również zmienne całkowite które są kolejnymi potęgami 2, związane jest to z optymalizacją przechwytywania ruchów z klawiatury - każde wciśnięcie klawisza ustawia odpowiedni bit.

9. ControllerHuman

Klasa dziedzicząca po klasie Controller, nadpisuje powyższe metody, tak aby wciśnięcia klawiszy przez gracza odpowiednio poruszały postacią w grze. Posiada ona

metodę `down(int)` która związana jest z wyżej wymienioną optymalizacją przechwytywania ruchu. Dzięki operacjom bitowym szybko sprawdzany jest stan każdego klawisza.

10. **ControllerGenetic**

Klasa dziedzicząca po klasie `Controller`, odpowiada za odczytywanie wygenerowanych akcji z osobników znajdujących się w populacji i odpowiednie przekazywanie ich do środowiska gry (logiki). Instancja klasy zawiera w sobie obiekt klasy `Population` który przechowuje całą populację osobników.

11. **Chromosome**

`Chromosome` jest klasą reprezentującą obiekt osobnika. Uzupełnia ona interfejs `Comparable`, dzięki czemu możliwe jest wykorzystanie algorytmów sortujących ze standardowej biblioteki `java`. Każdy obiekt tej klasy zawiera w sobie instancje obiektu `ResultData` która przechowuje dane na temat wyniku gry. Klasa posiada metody takie jak `mutateSpecial` i `mutateMoves` które odpowiednio dokonują mutacji osobnika na tablicy akcji specjalnych i tablicy ruchów. Parametr `breadth` decyduje o “rozległości” mutacji, np: wartość 0.2 spowoduje mutację losowych 20% akcji w danej tablicy chromosomu. Oprócz standardowego konstruktora klasy jest także konstruktor przyjmujący tablicę innych chromosomów. Traktowane jest to jako tworzenie nowego Chromosomu na podstawie kilku innych, czyli opisane wcześniej krzyżowanie statystyczne z grupy rodzicielskiej z poprzedniej populacji.

12. **GameState**

Klasa `GameState` przechowuje aktualny stan gry, wraz ze zmiennymi takimi jak: czasy rozpoczęcia gry, aktualny oraz zakończenia (wszystkie wartości jako czasy systemowe w milisekundach), zebrane przez postać punkty bądź wartość wyliczona z funkcji przystosowania. Funkcja `updateTime(d_time)` aktualizuje czas gry na podstawie faktycznego czasu który upłynął w systemie, dzięki czemu gra z punktu widzenia logiki przebiega tak samo bez względu na wydajność komputera na którym

uruchamiany jest program.

13. **GeneticsConfig**

Klasa GeneticsConfig przechowuje wszystkie informacje dotyczące parametrów algorytmu genetycznego. Wartości prawdopodobieństw wylosowania akcji specjalnych (specialProb) bądź ruchu postaci (moveProb) przechowywane są w strukturach typu HashMap dostępnych w standardowych bibliotekach javy. Oprócz prawdopodobieństw klasa przechowuje też parametry algorytmów genetycznych takie jak rozmiar populacji, wielkość tablicy chromosomu, stałą krzyżowania, czy rozległość mutacji poszczególnych tablic. Parametry te również przechowywane są w tablicy asocjacyjnej, dzięki czemu możliwe jest odwoływanie się do nich poprzez typ wyliczeniowy "Parameter". Oprócz samych wartości przechowywany jest też typ każdego parametru. Metody registerParameter, updateParameter oraz getParameter służą do aktualizacji bądź pobierania parametrów z tablicy. Klasa oprócz przechowywania wartości posiada metody zwracające losowy ruch. Korzysta z tego głównie klasa Chromosome przy generowaniu nowych osobników.

14. **GlobalVariables**

Klasa GlobalVariables przechowuje zmienne statyczne widoczne w całym systemie takie jak aktualny tryb pracy, zmienną SLEEP_TIME decydującą o tempie gry oraz zmienną odpowiadającą za zablokowanie dostępu do zasobów - program jako aplikacja okienkowa korzysta z dwóch wątków wobec czego jest to konieczne przy współdzieleniu zasobów takich jak obiekty gry.

15. **LabeledTextBox**

Jest to klasa pomocnicza przy tworzeniu interfejsu użytkownika - dzięki niej możliwe jest automatycznie generowanie par JLabel i JTextBox widocznych w oknie ustawień aplikacji.

16. **Logic**

Klasa Logic odpowiada za symulację logiki gry - głównie dotyczy to aktora i jego

reakcji na różnego rodzaju akcje, gdyż jest on jedynym ruchomym obiektem w grze. Dzięki odseparowaniu logiki gry od pozostałych warstw możliwe jest proste rozszerzanie systemu o własne reguły gry. Klasa posiada metodę `doLogic` która jest cyklicznie wykonywana w każdej pętli gry. Oprócz tego metody `executeMoveAction` oraz `executeSpecialAction` decydują o reakcji środowiska oraz aktora na akcje generowane przez `chromosom`, bądź użytkownika.

17. **LogicLabirynth**

Jest to przykładowa implementacja gry, w której postać porusza się w 4 kierunkach. Klasa została rozszerzona o wartości ruchu postaci w obu osiach.

18. **LogicMario**

Podobnie jak klasa `LogicLabirynth` jest to prosta implementacja gry wzorowanej na grze `Super Mario Brothers`. Pomocnicza zmienna `actor_falling` pomaga w realizacji fizyki w grze (wpływie grawitacji na postać). Zmienna `velocity_x` odpowiada za wartość prędkości postaci w poziomie.

19. **Main**

Klasa `Main` jest główną klasą w grze. Zawiera ona referencje do poszczególnych komponentów systemu takich jak silnik renderujący, logika gry, mapy gry, warstwy kontrolujące wykonywanie akcji oraz bieżącego stanu gry. Podczas uruchomienia aplikacji uruchamiana jest metoda `main(String[])` która inicjuje wszystkie konieczne komponenty i wyświetla okno aplikacji. Klasa uzupełnia interfejs `Runnable`, dzięki czemu działa jako osobny wątek. Oprócz tego klasa `Main` dziedziczy po klasie `JPanel` i nadpisuje metodę `paintComponent(Graphics)`, dzięki czemu możliwe jest stworzenie warstwy wizualizacyjnej dla środowiska gry.

20. **MapEditor**

Klasa `MapEditor` podobnie jak klasa `Main` dziedziczy po klasie `JPanel`. Zawiera ona listę obiektów niezależną od obiektów istniejących w symulatorze gry. Metoda `loadMap` pozwala na wczytanie mapy z pliku na dysku, metoda `saveMap` służy do

zapisu aktualnie edytowanej mapy do pliku tekstowego. Klasa uzupełnia interfejsy `MouseListener` oraz `MouseMotionListener` przez co nadpisuje metody służące do obsługi zdarzeń.

21. **Population**

Klasa `Population` przechowuje tablicę osobników biorących udział w treningu populacji. Metoda `getNextPopulation` odpowiada za wygenerowanie nowej populacji na podstawie wyników poprzedniej - to w niej wykonywane są wszystkie kroki związane z algorytmem genetycznym. Oprócz tego klasa posiada konstruktor generujący nową Populację na podstawie populacji rodzicielskiej (wspomniane wcześniej krzyżowanie statystyczne). Metoda `getParents` zwraca populację rodzicielską na podstawie aktualnej populacji.

22. **Renderer**

`Renderer` jest klasą odpowiedzialną za wyświetlanie elementów gry w oknie. Warstwa ta została odseparowana, dzięki czemu podczas dalszej rozbudowy aplikacji łatwo można zastąpić silnik renderujący na inny, bądź wprowadzić używanie tekstur do programu.

23. **RendererSimple**

Jest to prosta implementacja silnika renderującego. Głównym założeniem jest tutaj jedynie rysowanie prostokątnych obiektów, różniących się kolorami (Każda klasa dziedzicząca po `WorldObject` posiada inny kolor).

24. **Resources**

Klasa `Resources` przechowuje wszystkie obiekty świata gry. Metoda `loadResources(String)` interpretuje plik tekstowy przekazany w parametrze jako ścieżkę na dysku i zamienia opis tekstowy mapy na obiekty w grze. Warto zauważyć iż metoda `getMainActor` zwraca jeden obiekt Aktora, natomiast w zasobach gry może być ich kilku (kolekcja `actors` jest listą). Jest to rozwiązanie mające na celu późniejsze rozwinięcie systemu np. do obsługi gry wieloosobowej.

25. **ResultData**

ResultData jest klasą pomocniczą przechowującą dane na temat przejścia gry każdego osobnika. Przechowuje ona zmienne takie jak wynik funkcji przystosowania (zmienna `final_score`), ilość zebranych punktów na mapie (`score`), czas przejścia (`time_elapsed`) oraz stan w jakim przejście się zakończyło (`final_state` przyjmujący wartości `RESULT_TIMEOUT`, `RESULT_WON` bądź `RESULT_LOST`). Oprócz tego posiada ona metodę liczącą funkcję przystosowania na podstawie parametrów w ustawieniach algorytmu genetycznego.

26. **Terrain**

Klasa Terrain jest klasą dziedziczącą po klasie WorldObject. Odpowiada za statyczny teren. Najczęściej kolizja z nim powoduje zatrzymanie ruchu, jednak zależy to od implementacji logiki gry.

27. **Vector**

Klasa pomocnicza przechowująca współrzędne punktu (typ `double`).

28. **WorldObject**

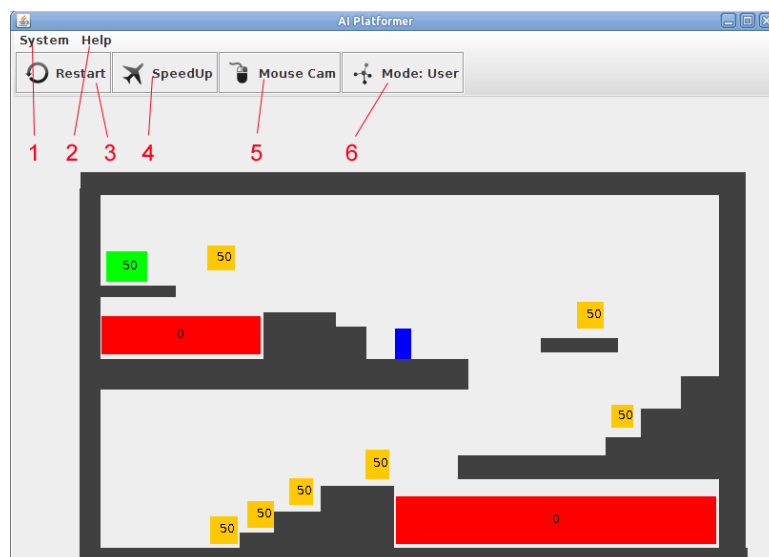
Klasa abstrakcyjna po której dziedziczą wszystkie elementy gry. Podstawową jej zmienną jest położenie na mapie oraz rozmiar (odpowiednio zmienne „`position`” oraz „`size`”). Klasa posiada metodę `getRectangle` zwracającą obiekt `Rectangle` dostępny w standardowej bibliotece, dzięki czemu realizowanie kolizji pomiędzy dwoma obiektami jest realizowanie poprzez wywołanie metody `intersects(Rectangle)`. Każdy dziedziczący po `WorldObject` obiekt ma do dyspozycji metodę `evalCollision` która decyduje o czynnościach wykonywanych podczas kolizji dwóch obiektów. Obiekty klasy `WorldObject` posiadają również metodę `paint(Graphics)` która odpowiada za rysowanie obiektu na ekranie (obiekcie `Graphics`).

3 Opis interfejsu systemu

3.1 Moduł silnika i symulatora gry

3.1.1 Główne okno aplikacji

Podstawowym oknem aplikacji widocznym tuż po uruchomieniu jest główne okno symulatora gry. Domyślnym trybem jest tryb poruszania się po mapie przez użytkownika. Sterowanie postacią przez klawiaturę zrealizowane jest za pomocą strzałek kierunkowych, oraz klawiszy *a*, *s*, *d*, *f* jako akcji specjalnych. Reakcja środowiska na poszczególne klawisze zależy od logiki gry, domyślnie jest to logika reprezentowana przez klasę LogicMario (strzałki kierunkowe odpowiadają za ruch postaci, natomiast klawisz *f* za skok). Poniżej przedstawiony został opis poszczególnych elementów menu. W oknie aplikacji widoczne



Rysunek 13: Główne okno aplikacji.

jest menu oraz pasek narzędzi zawierający podstawowe przyciski sterujące systemem:

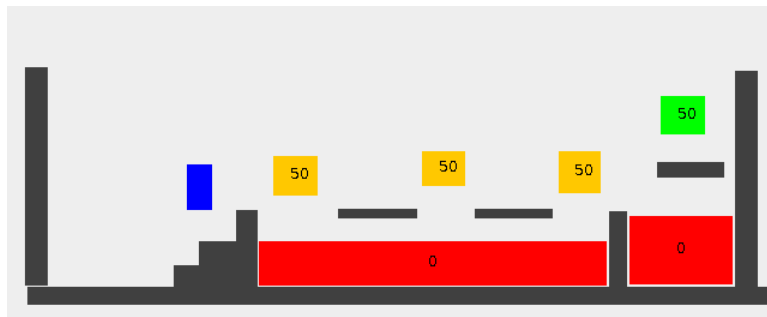
1. Z menu “System” użytkownik ma możliwość otwarcia dodatkowych okien aplikacji takich jak edytor map, panel ustawień, podgląd populacji oraz menu dialogowe dotyczące wczytania nowej mapy z pliku do środowiska gry.
2. Menu “Help” zawiera informacje dotyczące programu oraz krótki opis skrótów kła-

wiszowych.

3. Przycisk “Restart” pozwala na ponowne uruchomienie algorytmu, czyli wygenerowanie losowej populacji.
4. Przycisk “SpeedUp” odpowiada za przejście systemu w tryb treningu przyspieszonego, jeśli populacja aktualnie znajduje się w trybie treningu. Przestaje wówczas być aktualizowana grafika, a system w tle symuluje kolejne populacje osobników w środowisku gry.
5. Domyślnym trybem kamery jest śledzenie obiektu aktora. Przycisk “Mouse Cam” pozwala na przejście w tryb swobodnej kamery - prawy przycisk myszy służy do “przesuwania” ekranu.
6. Przycisk “Mode: User”(“Mode: Genetic”) służy do zmiany trybu działania systemu. Domyślnie pierwszym trybem jest tryb poruszania się po mapie przez użytkownika. Po wciśnięciu przycisku system przechodzi do trybu treningu populacji. Kolejne wciśnięcie przycisku przywraca poprzedni tryb swobodnego poruszania się.

Środowisko gry znajduje się w centralnej części okna gry. Wszystkie elementy gry z punktu widzenia kolizji są prostokątnymi obszarami, różniącymi się reakcją na kolizję z obiektem gracza. W podstawowym silniku renderującym zostały one wyróżnione kolorem.

Na Rys. 14 widać obiekty świata gry: *Actor*, *BonusCoin*, *BonusLose*, *BonusWin*, *Terrain*.



Rysunek 14: Poszczególne obiekty świata gry.

3.1.2 Dane konfiguracyjne

Jak już zostało wspomniane, podstawą dobrego systemu, szczególnie do zastosowań badawczych jest łatwo dostępna konfiguracja. W pracy rolę tę pełni pośrednio klasa *GeneticsConfig* która zawiera wszystkie najważniejsze parametry dotyczące algorytmu (takie jak prawdopodobieństwo mutacji) są przechowywane jako wartości tablicy asocjacyjnej (klasa *JHashMap*). Oprócz tego sama tablica została opakowana w taki sposób iż rejestracja nowej wartości do tablicy wiąże się z automatycznym wygenerowaniem odpowiedniego pola w panelu konfiguracyjnym. Dzięki temu mamy pewność iż wszystkie parametry biorące udział w algorytmie będą w pełni konfigurowalne.

Każda z akcji, zarówno ruchu jak i specjalna posiada pewne prawdopodobieństwo wystąpienia. Wartości te również dostępne są w panelu konfiguracyjnym, przy czym są one normalizowane do sumy wszystkich wartości z danej kategorii. Przykładowo dla wartości podanych na rys 15. Wartość ruchu w prawo (grupa “Movement key probabilities”) wynosi $\frac{2.5}{2.5+0.5+0.2} = 0.78125$. Oprócz tego klasa *GeneticsConfig* posiada metody pozwalające

The screenshot shows a 'Settings' window with three main sections: Movement key probabilities, Special key probabilities, and Genetic Settings. Each section contains a list of parameters with corresponding input fields.

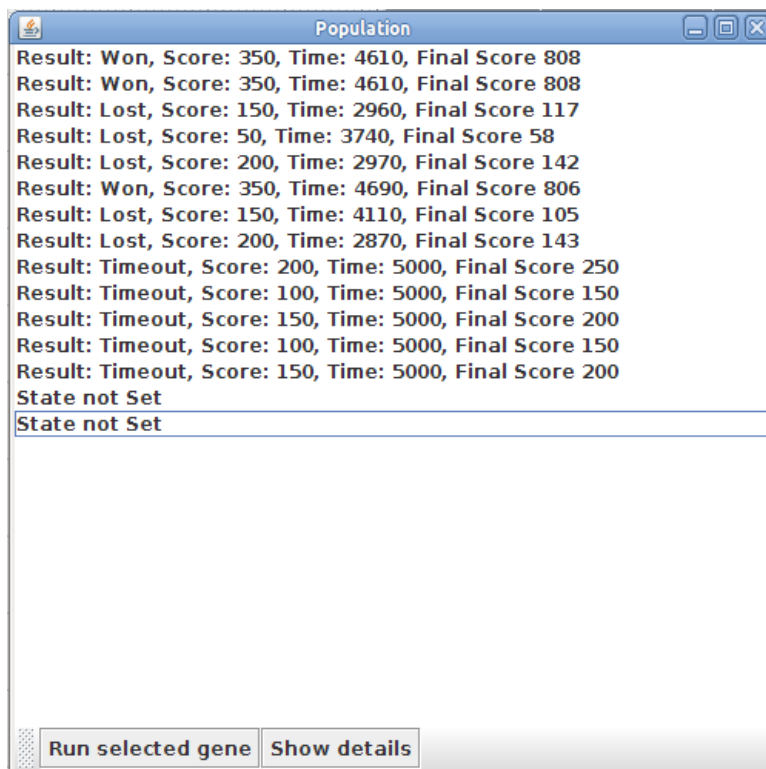
Section	Parameter	Value
Movement key probabilities	UP	0.0
	DOWN	0.0
	LEFT	2.5
	RIGHT	0.5
	NONE	0.2
Special key probabilities	A	0.2
	B	0.0
	C	0.0
	D	0.0
	NONE	0.8
Genetic Settings	CROSSING_ELITE_IS_PARENTS	false
	CROSSING_ELITE_SIZE	2
	CROSSING_OFFSPRING_COUNT	5
	CROSSING_PARAMETER	5.0
	CROSSING_PARENT_SET_MAX	3
	CROSSING_PARENT_SET_MIN	1
	FUNCTION_DEAD_MULTIPLIER	0.5
	FUNCTION_TIMEOUT_MULTIPLIER	1.0
	FUNCTION_TIME_MULTIPLIER	50.0
	FUNCTION_WON_MULTIPLIER	2.0
	MAXIMUM_TIME	10000
	MOVES_PER_SECOND	20
	MUTATION_MOVES_BREADTH	0.015
	MUTATION_MOVES_PROBABILITY	0.02
	MUTATION_SPECIAL_BREADTH	0.015
MUTATION_SPECIAL_PROBABILITY	0.02	
POPULATION_SIZE	10	

An 'Apply' button is located at the bottom of the window.

Rysunek 15: Panel Konfiguracyjny.

na zwrócenie losowej akcji w zależności od prawdopodobieństwa jej wystąpienia - Rys. 15.

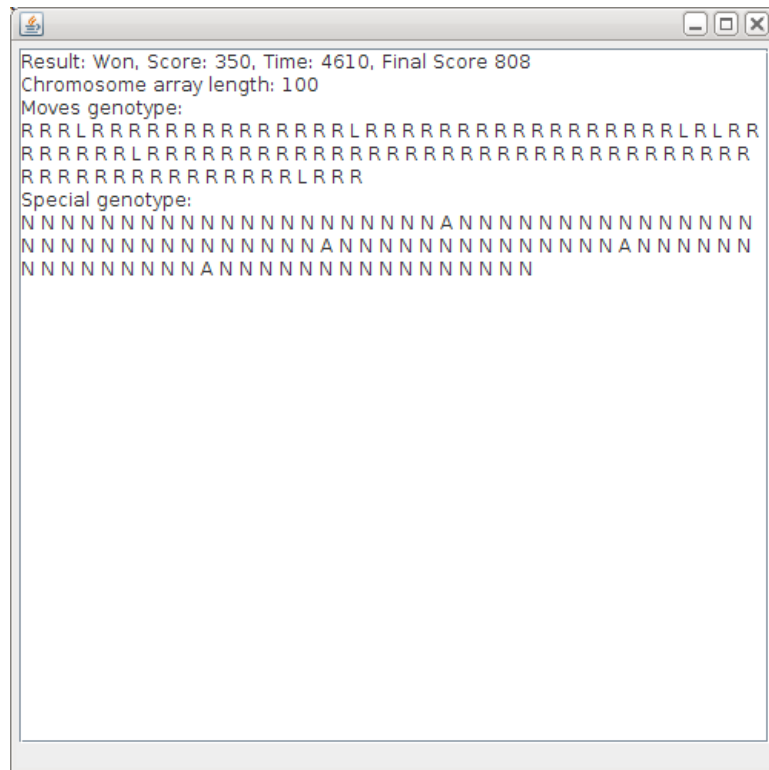
3.1.3 Widok Populacji i Chromosomu



Rysunek 16: Okno populacji.

Podstawową logiką gry i detekcją kolizji pomiędzy obiektami zarządza obiekt klasy *Logic*. użytym w pracy schematem logiki jest schemat gry "Super Mario Brothers". Wobec czego domyślną logiką gry przy uruchomieniu programu jest instancja klasy *LogicMario*. Zakłada ona ruch w 2 kierunkach, skok oraz działanie grawitacji na obiekt aktora. Ponieważ klasa *LogicMario* dziedziczy po klasie *Logic* możliwe jest napisanie własnej logiki i podmiana tej obowiązującej w systemie. Ponieważ działanie algorytmu nie jest związane z fizyką ani zasadami obowiązującymi w grze, system może działać dla wielu różnych typów gier platformowych i zręcznościowych. Jedynym wymogiem jest to iż muszą one dać się opisać za pomocą wyżej zdefiniowanych klas i cech:

1. Obiekty świata muszą należeć do którejś z klas dziedziczących po klasie *WorldO-*



Rysunek 17: Okno szczegółów chromosomu.

bject.

2. Możliwe rezultaty zakończenia algorytmu muszą być wśród zbioru rezultatów: {Koniec Czasu, Wygrana, Przegrana},
3. Rodzaje możliwych akcji do wykonania w grze muszą być przypisane do 4 akcji ruchu kierunkowego oraz 4 akcji specjalnych.

Jedyną pracą jaką należy wykonać przy implementacji własnego środowiska gry jest uzupełnienie własnej klasy dziedziczącej po klasie *Logic*. Wówczas przy poprawnej implementacji logiki gry system automatycznie symuluje zbiory chromosomów w środowisku gry.

3.2 Moduł edytora map

3.3 Realizacja warstwy symulacyjnej

3.3.1 Edytor Map

Przy testowaniu różnych ustawień algorytmu genetycznego przydatnym narzędziem może okazać się edytor map. Ustawienia algorytmu, szczególnie dotyczące prawdopodobieństwa wylosowania akcji w dużej mierze zależą od typu mapy.

- Mapa może być ukierunkowana bądź nie, wówczas ustawienie odpowiedniego stosunku akcji ruchu może znacznie przyspieszyć szukanie wyniku. Przykładowo: Jeśli problemem jest znalezienie rozwiązania na szerokiej mapie, której cel znajduje się w jej prawym krańcu, dobrą strategią będzie ustawienie ruchu w prawo na wysoką wartość np. 0.95 a ruchu w lewo na wartość 0.05.
- Podobna sytuacja ma miejsce z akcjami specjalnymi. Należy pamiętać iż wartość *NONE* może być szczególnie istotna w ustawieniach genetycznych akcji specjalnych. Powołując się znów na przykład gry “Super Mario Brothers”, można zauważyć iż ciągle skok nie koniecznie jest optymalną strategią - przez większość czasu nasza postać porusza się po podłożu.
- Logika gry może zakładać różną ilość kierunków w których można się poruszać, co w dużej mierze zależy od typu gry jaki reprezentuje. Niektóre wymagają 2 klawiszy kierunkowych do poruszania się po mapie, inne 3 lub 4. Aby nie brać pod uwagę akcji które i tak nie będą interpretowane przez logikę, najlepiej jest ustalić prawdopodobieństwo wylosowania ruchów nieaktywnych na wartość 0. Wówczas istnieje pewność iż w genotypie nie znajdują się niepotrzebne informacje.

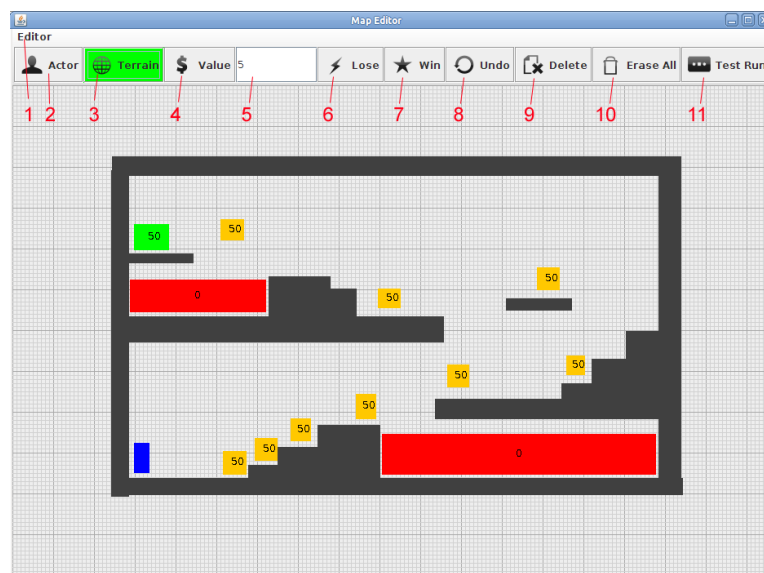
Dobrze zaprojektowany edytor map może okazać się szczególnie przydatny dla bardziej zaawansowanych użytkowników którzy mogą korzystać z systemu w celach edukacyjnych, a logiki zaimplementowane w systemie nie spełniają ich wszystkich oczekiwań. Wówczas

po napisaniu własnej klasy odpowiadającej za logikę, dobrze jest skonstruować mapy odpowiadające typowi rozgrywki jakie logika przewiduje.

Po otwarciu edytora map domyślnie wczytywana jest bieżąca mapa, dzięki czemu można dokonać szybkich edycji jeśli konieczne jest sprawdzenie działania algorytmu z pewnym wariantem, lub dodatkowym obiektem na mapie. Oprócz tego wszystkie mapy zapisywane są w prostym formacie tekstowym, dzięki czemu możliwe jest generowanie map przy pomocy programów trzecich, lub skryptów - zostawia to bardziej zainteresowanym użytkownikom na generowanie dużych, losowych poziomów, o ile wcześniej zaznajomią się z formatem zapisu mapy. Przykładowy plik z mapą może wyglądać następująco:

```
Terrain 122 361 239 27
Terrain 110 211 25 151
Terrain 355 201 28 165
Actor 140 295 20 32
BonusCoin 202 328 18 30 25
BonusCoin 248 330 16 27 25
BonusWin 327 236 18 36 25
BonusLose 313 311 33 40 0
```

Pierwszym elementem każdego wiersza jest nazwa klasy. Kolejne wartości to punkt oznaczający położenie lewego górnego rogu obiektu, oraz jego szerokość i wysokość. W wypadku obiektów klasy *Bonus* ostatnia liczba oznacza wartość punktową. Zachowanie prostego formatu mapy uprościło proces sprawdzania aplikacji, oraz otworzyło możliwość generowania dużych map poprzez skrypty. Ponieważ nie było powodu aby przechowywać mapy jako dane binarne - samo wczytywanie mapy nie spowalnia aplikacji, nie jest wykonywane często - otwarty format pozostał jako obowiązujący w systemie.



Rysunek 18: Okno edytora map.