

Samouczący się automat sterujący postacią gracza w prostej grze zręcznościowej.

Krzysztof Nowak

21 grudnia 2011

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.	3
1.4	Podstawy algorytmów genetycznych.	5
2	Analiza wymagań	9
2.1	Opis problemu	9
2.2	Wstępna analiza problemu	11
2.2.1	Projekt chromosomu	11
2.3	Diagram przypadków użycia	14
2.4	Diagram klas	14
3	Realizacja poszczególnych elementów systemu	14
3.1	Realizacja warstwy genetycznej	14
3.1.1	Struktura danych chromosomu	14
3.1.2	Dane konfiguracyjne	16
3.1.3	Widok Populacji i Chromosomu	17
3.2	Realizacja warstwy symulacyjnej	17
3.2.1	Logika gry	17
3.2.2	Edytor Map	20
3.3	Użyte narzędzia i technologie	22
3.4	Język aplikacji	22
3.5	Java	23
3.6	Swing	24

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 Marvinę Minską 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 Prinzę. 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 spowrotem na półki science-fiction. W roku 1973, znaczna część funduszy przeznaczonych na rozwój sztucznej inteligencji została wstrzy-

mana 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.

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. Dotyczy on gier o sumie stałej – proporcja zysku jednego z graczy do straty drugiego jest stała. Najprostszym przykładem jest gra “Kółko i krzyżyk” gdzie przeszukiwane jest rekurencyjne drzewo wywołań kolejno sprawdzające wszystkie możliwe stany gry. W dowolnym momencie gry możemy przeanalizować wszystkie możliwe posunięcia każdego z graczy. 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 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. Zatem na początku gry podjęciu pierwszej decyzji musimy przeanalizować $9 * 8 * 7 * \dots * 1 = 9!$ ruchów. Łatwo zauważyć iż algorytm taki ma ogromną złożoność i jedynie dla małych gier takich jak kółko i krzyżyk daje wynik w realnym czasie. Bez optymalizacji, obraz 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 nielada wyzwanie nawet dla współczesnych superkomputerów. Do dziś nie stworzono programu który wygrywałby z profesjonalnymi zawodnikami GO.

Większość gier daje się sklasyfikować ze względu na czas:

- Gry turowe. Gracze naprzemiennie wykonują ruchy, przy czym czas na podjęcie decyzji jest relatywnie duży - nawet do 10 sekund. Wiele nieskomplikowanych gier 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 - wyżej wspomniana gra GO.
- Gry czasu rzeczywistego. Gra toczy się w dynamicznym środowisku gry, często z wieloma obiektami/graczami na raz. Często czas na podjęcie optymalnej decyzji przez algorytm jest mocno ograniczony - często należy podejmować nawet do 30

razy w ciągu sekundy. Oprócz tego próba dyskretyzacji środowiska i znalezienia najlepszego rozwiązania z przestrzeni stanów gry jest w praktyce niewykonalne. Przykładem mogą być różnego rodzaju trójwymiarowe gry akcji które często posiadają złożone środowiska gry, oraz wiele dynamicznych obiektów oraz graczy uczestniczących w rozgrywce.

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 niedeterministyczny, 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 zasymulowania idealnej partii gry, jednak często wystarcza to dla osiągnięcia celu końcowego.

1.4 Podstawy algorytmów genetycznych.

Często wykorzystywanym sposobem realizacji celu są algorytmy genetyczne. Opierają się one na prawach 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 poprzedniego podejścia, iż jest niedeterministyczny. Początkowy chaos z czasem jest zastępowana przez odpowiednio przystosowaną populację rozwiązań, o ile zasymulujemy jej ewolucję dostateczną ilość razy. 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, długość życia. Ponieważ potomstwo dziedziczy geny po swoich rodzicach, “zwycięskie” cechy w kolejnym pokoleniu są bardziej powszechne. Odpowiednikiem funkcji przystosowania jest właśnie wynikowa cech danego osobnika która określa prawdopodobieństwo przekazania jego genów w kolejnym pokoleniu. Funkcja przystosowania jest dość prosta w realizacji, o ile dane dotyczące osobnika są łatwe do zmierzenia – wówczas może być to jedynie kwestia policzenia wartości funkcji liniowej z odpowiednimi wagami, gdzie argumentami są wyniki osobnika podczas symulacji w środowisku. Mimo to w większości algorytmów genetycznych dobranie odpowied-

nich wag w funkcji przystosowania jest kluczowym czynnikiem nad którym później można długo pracować przy optymalizacji algorytmu.

3. Krzyżowanie

Po każdym kroku algorytmu zazwyczaj możemy uporządkować osobniki należące do bieżącej populacji i wylosować z niej pewien zbiór osobników najlepiej przystosowanych (wpływ na to wynik funkcji przystosowania). Wówczas dokonujemy krzyżowania pomiędzy nimi, dzięki czemu otrzymujemy osobniki nowe, jednak posiadające pewne cechy swoich “rodziców”. Krok ten jest kluczowy jeśli chcemy osiągać coraz lepsze wyniki w kolejnych populacjach, ponieważ od dobrej metody krzyżowania zależy czy kolejne populacje będą lepiej przystosowane do rozwiązania problemu. Złe zaprojektowanie krzyżowania jest jednym z częstszych powodów osiągania przez populację złych wyników, zwłaszcza gdy Chromosom jest złożony. Samo krzyżowanie również zazwyczaj posiada czynnik losowy (w klasycznych przykładach dotyczących krzyżowania się dwóch ciągów bitowych, losowany jest punkt łączenia się dwóch ciągów).

4. Mutacja

O ile początkowa losowość algorytmu polegająca na wylosowaniu pierwszej populacji jest szybko zastępowana przez populację osiągającą 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

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 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 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 mapy. Funkcja przystosowania będzie zależała od różnych czynników, a ustawienie odpowiednich wag może nakierować algorytm na określoną ścieżkę rozwoju.
- Praca 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 Wstępna analiza problemu

Aby dobrze zrealizować część odpowiedzialną za sterowanie postacią, należy użyć klasy pośredniej pomiędzy warstwą logiki silnika gry, a warstwą komunikacji z graczem. Wówczas możemy łatwo zmienić źródło sygnałów wysyłanych do postaci z bezpośrednich zdarzeń z klawiatury na akcje przechowywane przez chromosom.

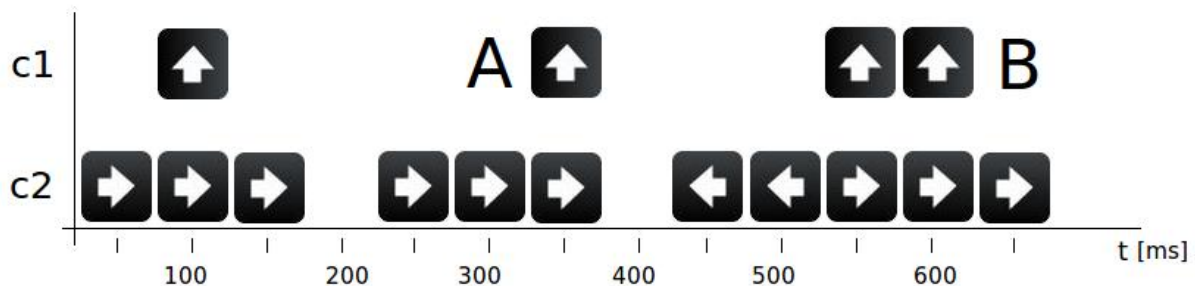
2.2.1 Projekt chromosomu

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

1. Czas który upłynął od rozpoczęcia danej instancji przejścia.

To rozwiązanie zakłada podejmowanie akcji w oparciu o aktualny czas w grze, a sama tablica akcji przechowuje akcje których wykonanie następowałoby po kolei z pewnym interwałem, np. 30 ms (co daje ok 33 akcje w ciągu sekundy). Warto zauważyć że dzięki temu iż przebieg symulacji nie zależy od pozycji gracza, mamy swobodę ruchu, a jeśli to korzystne możemy założyć iż dobrym rozwiązaniem w niektórych przypadkach będzie np. odczekanie określonego czasu, bądź powrót do miejsca w którym już byliśmy. Istotną wadą tego rozwiązania była duża podatność algorytmu na zapętlanie się, lub wykonywanie dużej ilości mało przydatnych ruchów. Jeśli chcielibyśmy dobrze zaprojektować taki algorytm musielibyśmy brać pod uwagę fakt, iż średnio przy równym prawdopodobieństwie ruchu w lewo jak i prawo, postać będzie przesuwana się bardzo powoli, bądź na dłuższą metę stała w miejscu. Prostim rozwiązaniem tego problemu jest przyporządkowanie pewnego prawdopodobieństwa każdej akcji (dzięki czemu możemy założyć że preferowanym kierunkiem jest np. ruch postaci w prawo). 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 po-

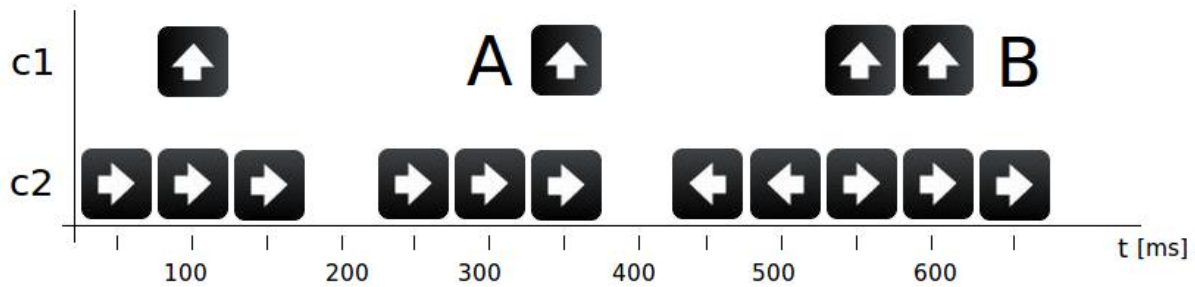
przez "cięcia" chromosomu na dwie części jest kosztowne. Po sklejeniu otrzymamy wówczas niespójny ciąg ruchów, które będą miały niewiele wspólnego z aktualną pozycją gracza na mapie. Można temu zapobiec zapewniając łączenie się chromosomów jedynie w punktach w których postać w obu momentach znajduje 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 1.



Rysunek 1: Sterowanie względem czasu.

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

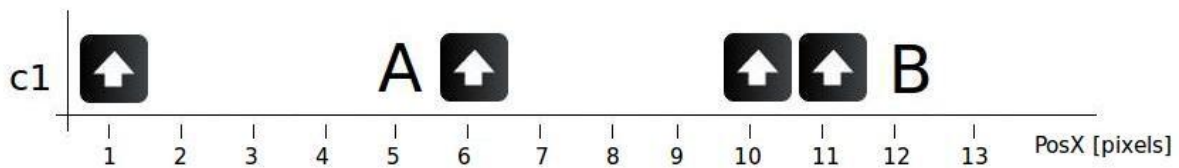
Lepszym rozwiązaniem jest realizacja krzyżowania nie poprzez klasyczne podejście, lecz modelowane statystycznie: Potomstwo nie otrzymuje bezpośrednich fragmentów chromosomu, lecz losuje za każdym razem nowe ruchy. Przy czym chromosomy populacji rodzicielskiej zwiększają prawdopodobieństwo wylosowania podobnych sobie ruchów. Schemat takiego rozwiązania zaprezentowany jest na rysunku 2. Więcej szczegółów na ten temat zostanie opisanych w dalszej części pracy.



Rysunek 2: Krzyżowanie statystyczne.

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.



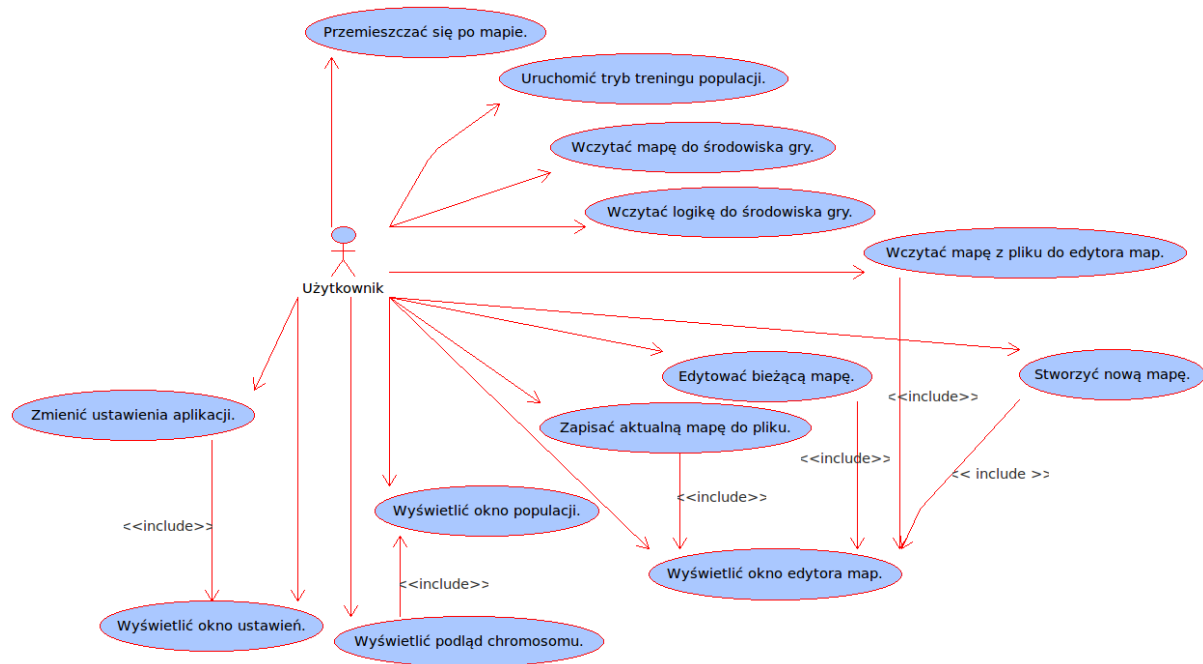
Rysunek 3: Sterowanie względem pozycji gracza.

To podejście posiada jednak kilka poważnych wad i wymaga pewnych ograniczających założeń. Plansza musi być ukierunkowana, i być rozwiązywalną przy ciągłym ruchu w określonym kierunku. Przeniesienie systemu do zastosowania w grze platformowej o nieco innym schemacie ruchu może okazać się trudne i wymagające

zmian w założeniach początkowych.

2.3 Diagram przypadków użycia

Diagram przypadków użycia został przedstawiony na Rys. 4.



Rysunek 4: Diagram przypadków użycia.

2.4 Diagram klas

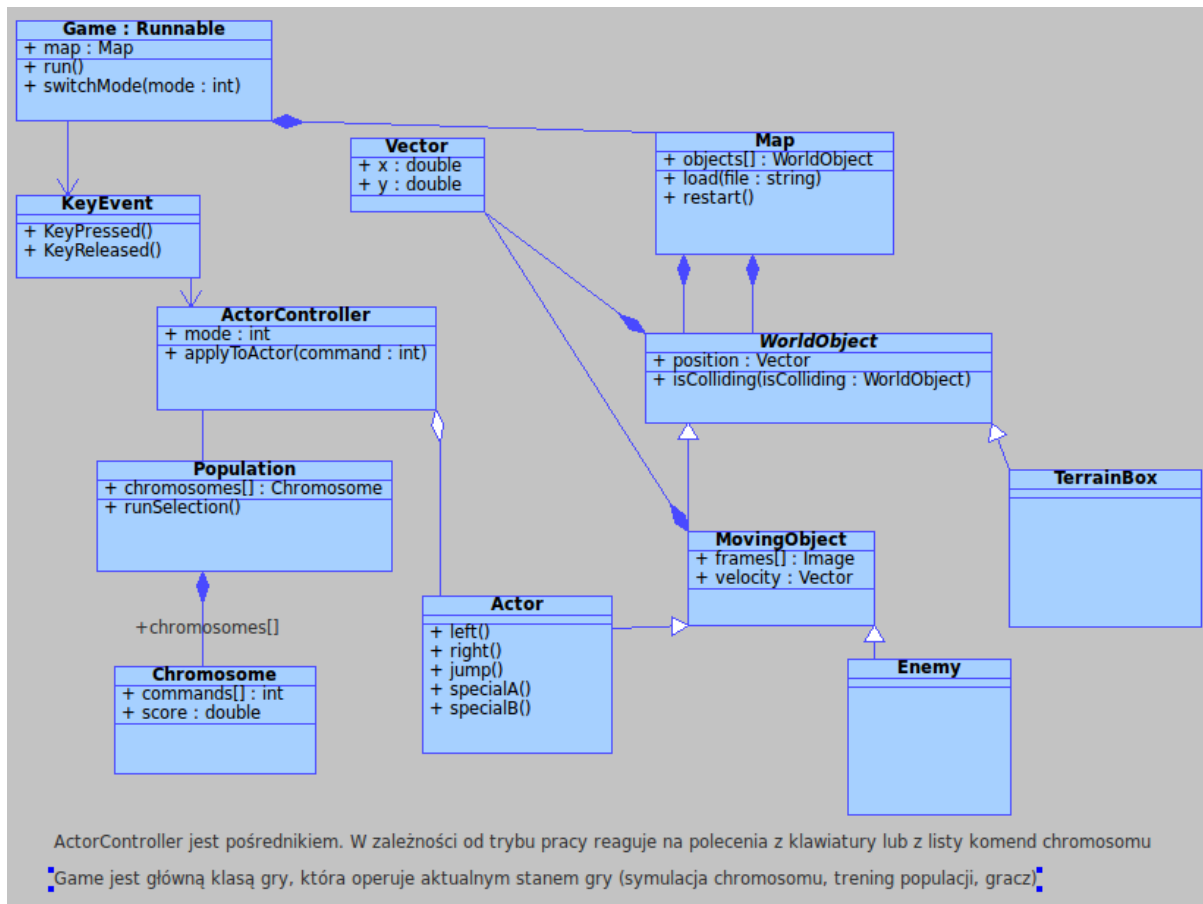
Diagram klas projektu przedstawiony został na rysunku 5.

3 Realizacja poszczególnych elementów systemu

3.1 Realizacja warstwy genetycznej

3.1.1 Struktura danych chromosomu

Jeśli chodzi o typ struktury danych przyjęto pierwszy schemat (rys.1), wraz z krzyżowaniem statystycznym. O ile ukierunkowanie systemu na gry jednego typu może lepiej



Rysunek 5: Diagram Klas.

się sprawdzić jeśli zależy nam jedynie na szybkim i optymalnym wyniku, to generalizacja systemu i elastyczna implementacja całego algorytmu da nam lepsze narzędzie nie tylko dydaktyczne, ale też badawcze. Przykładem może być popularna gra Galaxian. Sama definicja klawiszy w Galaxian i Super Mario Brothers jest podobna (klawisze kierunkowe + klawisze specjalne), jednak schemat poruszania się jest już inny: O ile gry typu Mario Brothers pasują do powyższych założeń o kierunkowości ruchu, to w grze Galaxian już tak nie jest. Korzystniej zatem jest traktowanie projektu jako systemu rozwiązującego gry platformowe czasu rzeczywistego na podstawie przyciśnień klawiszy w czasie.

Sam chromosom przechowuje następujące dane:

- Tablicę dotyczącą akcji ruchu przechowującą wartości typu wyliczeniowego: UP, DOWN, LEFT, RIGHT, NONE

- Tablicę dotyczącą akcji specjalnych: A, B, C, D, NONE
- Instancję obiektu *ResultData* przechowującego dane dotyczące wyniku funkcji przystosowania, oraz wartości ustalane po przetestowaniu danego chromosomu takie jak czas który upłynął, rodzaj wyniku, ilość zebranych punktów.

Wartość NONE w obu tablicach odpowiada za brak akcji. Wartości A,B,C,D odpowiadają za kolejne klawisze specjalne które mogą być traktowane przez środowisko gry jako skok lub strzał. Ograniczamy się do 4 klawiszy specjalnych, które wystarczają w zupełności do zrealizowania większości gier platformowych.

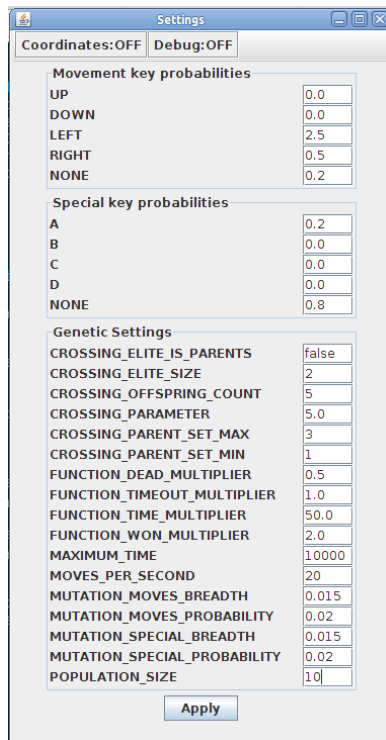
Oprócz tego każdy Chromosom uzupełnia interfejs Comparable, dzięki czemu obiekty mogą być porównywane ze sobą. Porównanie składa się jedynie z porównania wyniku wartości funkcji przystosowania. Dzięki temu możemy łatwo posortować całe populacje. Obiekt ten przechowuje dane na temat wyniku danego chromosomu i danych pomocniczych, które są uzupełniane po przetestowaniu chromosomu. Oprócz tego chromosom posiada metody pozwalające na mutację zarówno tablicy ruchu jak i tablicy akcji specjalnych.

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 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 6. 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 na



Rysunek 6: Panel Konfiguracyjny.

zwrócenie losowej akcji w zależności od prawdopodobieństwa jej wystąpienia - Rys. 6.

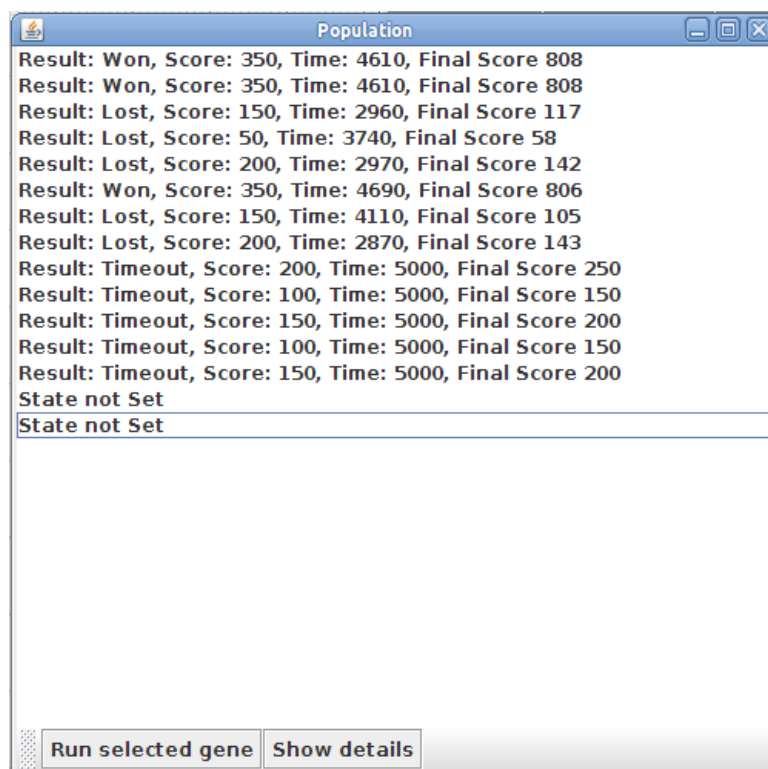
3.1.3 Widok Populacji i Chromosomu

3.2 Realizacja warstwy symulacyjnej

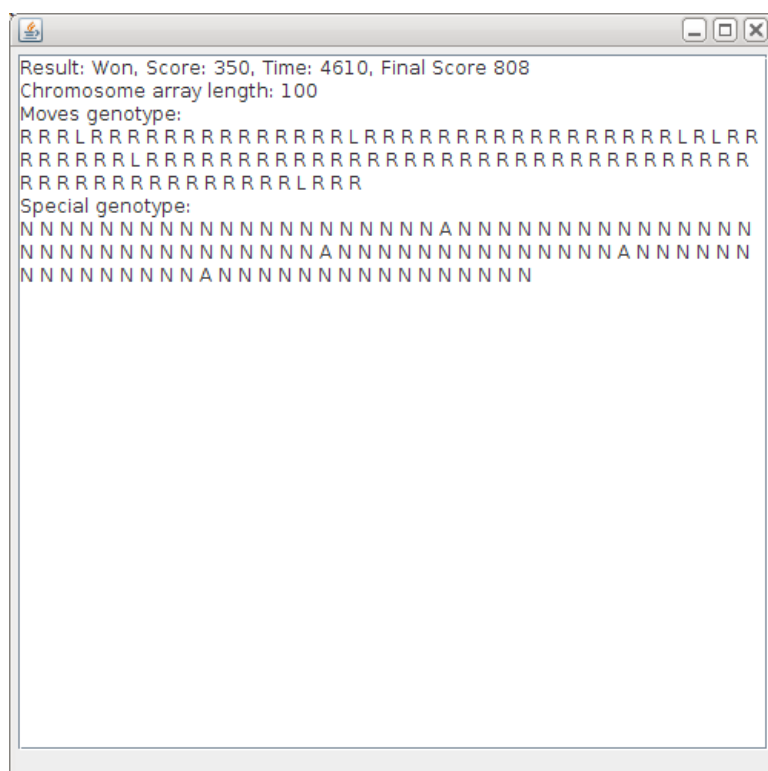
3.2.1 Logika gry

Sama warstwa logiki gry podejmuje decyzje o relacjach pomiędzy obiektami w grze. Wszystkie elementy gry dziedziczą po klasie *WorldObject*, która przechowuje podstawowe dane konieczne do wyświetlenia go w oknie takie jak położenie oraz wymiary. Kolejno można je podzielić na:

- Obiekt klasy *Actor*. Obiekt reprezentujący postać w grze. Klasa ta posiada zmienną



Rysunek 7: Okno populacji.



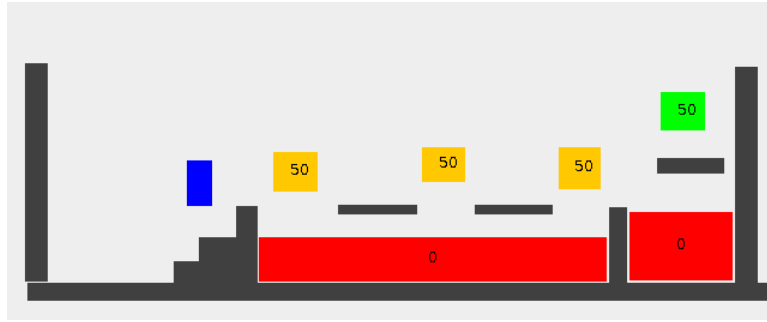
Rysunek 8: Okno szczegółów chromosomu.

velocity, oznaczającą wektor prędkości obiektu. W każdym cyklu gry zostaje on dodany do położenia aktora. Na zmianę wektora wpływa klasa *Controller* która w zależności od trybu gry przekazuje wciśnięcia klawiszy z klawiatury, lub akcje wygenerowane przez algorytm genetyczny.

- Obiekt klasy *Terrain*. Z nich zbudowana jest mapa gry. Z punktu widzenia logiki nie wpływają one w żaden sposób na wynik, a jedynie kolidują z obiektem aktora. To z nich zbudowana jest mapa gry.
- Obiekty klasy *Bonus* Są to obiekty wpływające na wynik gry. Kolizja Aktora z nimi powoduje zmianę wyniku gry. Klasa *Bonus* jest klasą po której dziedziczą podklasy:
 - BonusCoin - Obiekt reprezentujący punkty w grze. Aktorowi po kolizji z nimi zostaje dodana wartość punktowa przechowywana w obiekcie. Z punktu widzenia algorytmu zwiększają one ostateczną wartość funkcji przystosowania, która zależy od ilości punktów zebranych na mapie.
 - BonusWin - Kolizja aktora z obiektem tego typu kończy przejście chromosomu bądź gracza i zapisuje w stanie gry wynik końcowy *RESULT_WON*. W praktyce oznacza on zakończenie gry z pozytywnym skutkiem, który jest brany pod uwagę podczas wyliczenia funkcji przystosowania.
 - BonusLose - Obiekt kończący grę ze skutkiem negatywnym. Ma on reprezentować wszelkiego rodzaju pułapki w grze, które natychmiastowo kończą przejście chromosomu. W stanie gry zostaje wówczas zapisany wynik końcowy *RESULT_LOST*.

Na Rys. 9 widać obiekty świata gry: *Actor*, *BonusCoin*, *BonusLose*, *BonusWin*, *Terrain*.

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*



Rysunek 9: Poszczególne elementy mapy.

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 *WorldObject*.
2. Możliwe rezultaty zakończenia algorytmu muszą być wśród zbioru rezultatów: {Konec 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.

3.2.2 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 ciągły 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:

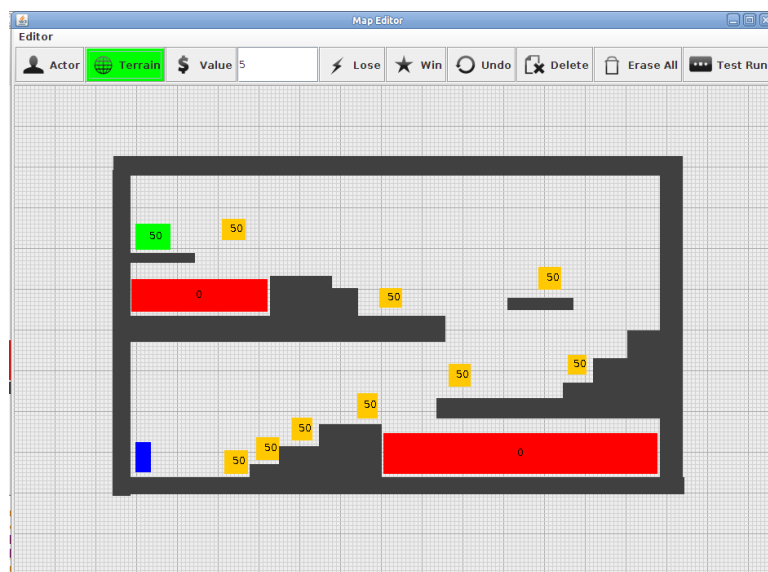
```
Terrain 122 361 239 27
Terrain 110 211 25 151
Terrain 355 201 28 165
Actor 140 295 20 32
BonusCoin 202 328 18 30 25
BonusCoin 248 330 16 27 25
BonusWin 327 236 18 36 25
BonusLose 313 311 33 40 0
```

Pierwszym elementem każdej 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.

3.3 Użyte narzędzia i technologie

3.4 Język aplikacji

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.



Rysunek 10: Okno edytora map.

3.5 Java

Java jest językiem programowania wysokiego poziomu zaprojektowanym i stworzonym przez Jamesa Groslinga podczas gry pracował w firmie Sun Microsystems. Pierwsza propozycja stworzenia Javy pojawiła się w roku 1991, do głównych twórców oprócz Jamesa Groslinga 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 javowej na dowolnym systemie jest obecność wirtualnej maszyny javy (ang. JVM - Java Virtual Machine). Dziś większość urządzeń mobilnych posiada maszynę wirtualną javy pozwalającą na uruchamianie aplikacji napisanych w tymże języku. ::BIBLIOGRAFIA TIJ-Bruce Eckel:: Jak pisze Bruce Eckel w swojej książce poświęconej językowi Java: “To, co wywarło na mnie naj-

większe 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.”(...)” Po kilku latach styczności z językiem, trudno jest się nie zgodzić z powyższym stwierdzeniem.

3.6 Swing

Pierwszą próbą stworzenia biblioteki pozwalającej na tworzenie graficznego interfejsu użytkownika była biblioteka AWT (ang. Abstract Window Toolkit). Fala niezawodzenia wśród programistów sprawiła iż szybko 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 systemu. 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.