Uczenie agenta w środowisku *Robocode* poprzez ewolucje rozmytego sterownika

Karol Bonenberg

11 lutego 2009

1 Opis zadania

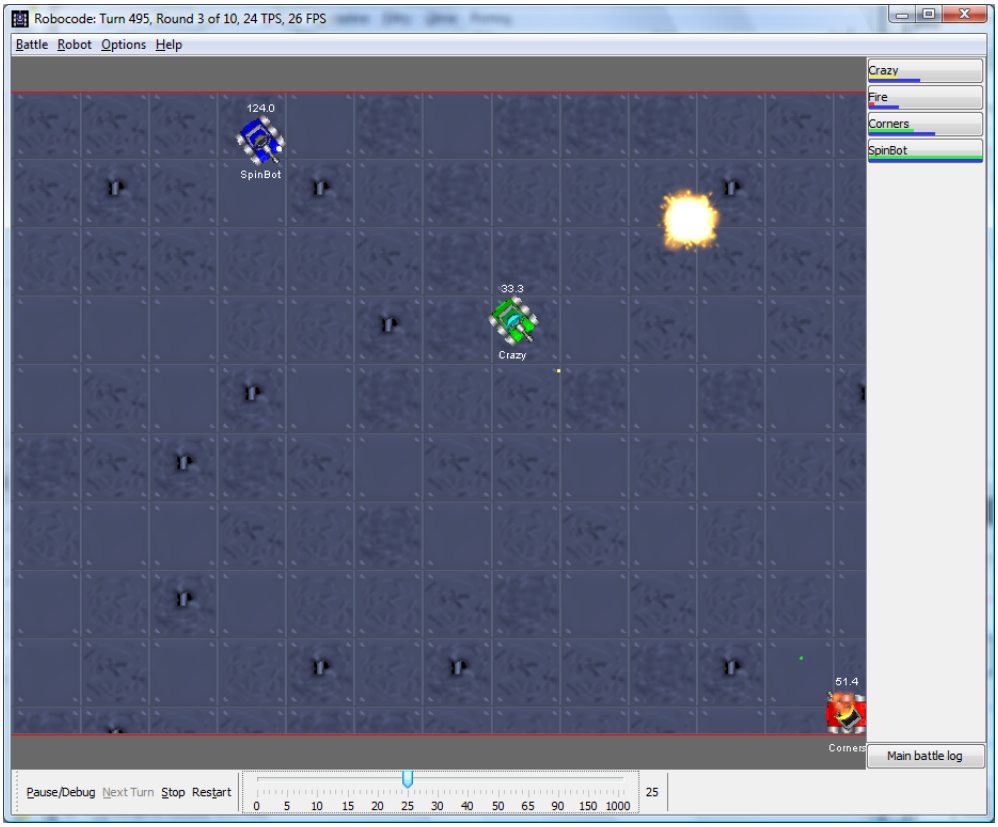
Zadanie polegało na implementacji systemu tworzącego sterownik do robota w środowisku *Robocode*. Sterownik miał mieć postać reguł rozmytych sterujących działaniem robota. Jako format zapisu sterownika przyjęto pliki *fcl*. Celem robota było wygranie walki z arbitralnie wybranym robotem.

1.1 Środowisko *Robocode*

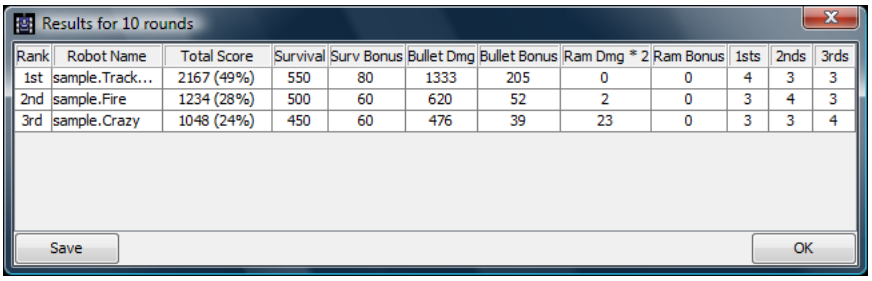
Robocode jest grą programistyczną, w której zadaniem gracza jest napisanie sztucznej inteligencji do robota. Następnie zaprogramowane roboty bez ingerencji użytkownika walczą na arenie. Środowisko działa na platformie *Java*. Roboty użytkowników są dostarczane jako klasy w języku *Java*. Więcej informacji na temat środowiska można znaleźć pod adresem `http://robocode.sourceforge.net`.

1.2 Zasady gry

Rozpoczynając walkę każdy robot ma zadaną ilość energii MAX_ENERGY. W przypadku gdy gracz otrzyma obrażenia jego poziom energii spada. W przypadku gdy gracz zada obrażenia przeciwnikowi jego poziom energii rośnie. W przypadku gdy gracz nie trafi w p rzeciwnika j ego p oziom energii spada o wartość równą mocy pocisku.



Rysunek 1: Przykładowy ekran gry



Rysunek 2: Okno z wynikami gry

Podstawy systemu punktowania:

- zadanie obrażeń - $+3 fire\_power$
- niecelny strzał - $-fire\_power$
- taranowanie - $+1.2$
- kolizja - -0.6

- zniszczenie robota - $+0.20 damage\_inflicted$
- wygranie tury - $+10$

2 Rozwiązanie

2.1 Podejście

Problem postanowiono rozwiązać w następujący sposób:

1. Ustalenie parametrów określających stan robota - w klasie kontrolera robota dodano metodę, która co zadany okres czasu (w grze czas jest dyskretny liczony w turach), pobierze dane na temat stanu robota.
2. Zdefiniowanie termów lingwistycznych dla parametrów wejściowych systemu. Aby możliwe było zastosowanie chromosomu o stałej długości (dla takiego chromosomu łatwiej przeprowadzić operacje krzyżowania), liczba termów nie może ulegać zmianie.
3. Stworzenie operatorów krzyżowania i mutacji odpowiednich dla problemu.
4. Implementacja narzędzi umożliwiających konwersje z kontrolera *fcl* do chromosomu i odwrotnie (embriogeneza - mapowanie).
5. Ewolucja systemu z funkcją *fitness* określającą przystosowanie danego osobnika na podstawie jego zdolności bojowych i złożoności sterownika.
6. Zapisanie najlepszego sterownika w celu późniejszej prezentacji.

2.2 Założenia

W celu uproszczenia modelu sterownika przyjęto, że każdą funkcję przynależności dla termu określającego zmienną wejściową można przedstawić w postaci funkcji trójkątnej charakteryzującej się trzema parametrami:

$$TERM \quad t1 := TRIAN \quad 0.0 \quad 50.0 \quad 100;$$

Ponadto przyjęto, że liczba termów dla danej zmiennej wejściowej jak i wyjściowej jest niezmienna w całym procesie ewolucji. Dla sterujących reguł rozmytych przyjęto stały operator łączący przesłanki (konfigurowalny): *AND* lub *OR*. Jako stałą przyjęto też metodę defazyfikacji i łączenia reguł.

2.3 Zastosowane narzędzia

Implementując system zdecydowane się na zastosowanie dwóch zewnętrznych pakietów:

- *jFuzzyLogic* – pakiet umożliwiający zapis, odczyt i ewaluację rozmytych sterowników napisanych w języku *fcl*. Sterownik można wczytać z pliku lub stworzyć w języku *Java* korzystając z *API* narzędzia. Pakiet wspiera podstawowe ciągłe funkcje przynależności: *Sigmoidal*, *Trapetzoidal*, *Gaussian*, *PieceWiseLinear*, *Triangular*, oraz metody defazyfikacji: *CenterOfGravity*, *RightMostMax*, *CenterOfArea*, *LeftMostMax*, *MeanMax*. Ponadto różne metody agregracji reguł i łączenia przesłanek.

- *jgap* – javowy framework dla algorytmów genetycznych i programowania genetycznego. Posiada on zestaw gotowych obiektów do projektowania algorytmu genetycznego. Poprzez obiekt klasy *Configuration* można zdefiniować jakie operatory krzyżowania i mutacji mają być stosowane w eksperymencie, w jaki sposób ma przebiegać selekcja i jaki ma być rozmiar populacji. Dodatkowo choromosom można w łatwy sposób składać z predefiniowanych genów np. *IntegerGene*, *DoubleGene*. Pakiet wspiera również tworzenie ewolucyjnych algorytmów rozproszonych. Do przydatnych funkcjonalności pakietu należy również możliwość zapisu stanu ewolucji i wszystkich chromosomów do pliku.

2.4 System rozmyty

Wspomniano już wcześniej, że wszystkie funkcję przynależności w systemie mają trójkątne funkcję przynależności. Opis wszystkich wejść i wyjść systemu:

Nazwa	Typ	Opis
pos_x	IN	Pozycja x
pos_y	IN	Pozycja y
heading_robot	IN	obrót gąsienic
heading_gun	IN	obrót działa
enemy_dist	IN	odl. do przeciwnika
enemy_velocity	IN	pred. przeciwnika
enemy_heading	IN	obrót działa przeciwnika
gun_angle_to_enemy	IN	kąt działa względem przeciwnika
hit_elapsed_time	IN	czas kiedy przeciwnik został trafiony
damage_elapsed_time	IN	czas kiedy nasz robot został trafiony
scanned_elapsed_time	IN	czas od wykrycia przeciwnika
turn	OUT	skręt gąsienic
gun_turn	OUT	skręt działa
velocity	OUT	prędkość
shoot	OUT	moc strzału

2.5 Sterownie

Klasa robot walczący w środowisku *Robocode* musi rozszerzać klasę podstawową *Robot* lub *AdvancedRobot*. W implementacji systemu przyjęto, że klasa *FuzzyAdvancedRobot* będzie robotem zawansowanym. Główna różnica polega między dwiema klasami podstawowymi polega na tym, że robot zaawansowany ma możliwość ustalania zadań jakie mają być wykonane a następnie po wywołaniu metody *execute()* przetwarzane są rozkazy. W przypadku zwykłego robota wszystkie akcje są wywoływane sekwencyjnie. Serce pracy robota stanowi nieskończona pętla w metodzie *run()* gdzie wywoływane są wszystkie akcje.

```
1
2  @Override
3  public void run() {
4    readRobotData();
5
6    setAdjustGunForRobotTurn(true);
7
8    while (true) {
9      m_cycleCounter++;
10     // get inputs state
11     RobotState inputState = gatherRobotData();
12     // fire fuzzy system
13     RobotState outputState = m_controller.getSystemResponse(inputState);
14     // set actions to be done turning, moving
15     scheduleActions(outputState);
16     execute();
17   }
18 }
```

Możliwości sterowania robotem są nastepujące:

- *setAhead()*, *setBack()* - rozkaz jazdy w przód lub cofania
- *setTurnLeft()*, *setTurnRight()* - rozkaz skrętu w lewo lub prawo o zadaną liczbę stopni
- *setTurnGunLeft()*, *setTurnGunRight* - rozkaz skrętu działa w lewo lub prawo. Działo porusza się niezależnie od podstawy robota. Razem z działem przesuwa się radar.
- *setFire()* - rozkaz strzelania z zadaną mocą.

2.6 Genotyp

Zapis genotypu można przedstawić w następującej sposób:

$$d_1 d_2 \dots d_n r_1 \dots r_m$$

Pierwsza część formuły opisu funkcję przynależności termów użytych do opisu zmiennych $n = 3num\_of\_terms$. Allele tych genów są typu *double*. Pozostałe geny opisu reguły. Przyjmijmy, że chcemy wygenerować system o R

regułach. Reguły są równo rozdzielane pomiędzy zmienne wyjściowe, dlatego też ($out\_vars \mid R$). Rozważając zapis pojedynczej reguły

$$r_1 1 r_1 2 \ldots r_1 n$$

Wartości o indeksach od $r_1 1$ do $r_1(n-1)$ określają czy w danej regule występuje dana zmienna wejściowa (0 jeśli nie) i wartość którego termu przyjmuje. Allele tych genów są typu *integer*.

2.7 Ewolucja

Dla opisanego chromosomu systemu rozmytego należało zdefiniować operacje krzyżowania, mutacji i selekcji:

- krzyżowanie - operacja dzielona jest na dwie części osobno krzyżowana jest część chromosomu odpowiedzialna za funkcje przynależności i odpowiedzialna za reguły. Po krzyżowaniu chromosomy są naprawiane w punkcie łączenia ciągów cześć opisującej funkcje przynależności.

- mutacja - jeśli mutacji podlega gen z części opisującej funkcje przynależności jego wartość jest modyfikowana w taki sposób aby zachowana była nierówność $lo \leqslant mid \leqslant hi$ dla danej funkcji przynależności np. jeśli modyfikacji ulega wartość mid to jest losowana z przedziału $(lo; hi)$. Jeśli mutowany jest gen opisujący regułę to dla genu przesłanki losowana jest wartość całkowita z zakres $< 0; num\_of\_terms >$, dla genu konkluzji wartość całkowita z zakresu $< 1; num\_of\_terms >$.

- selekcja - operatorem selekcji jest wybór rankingowy (n najlepszych osobników pod względem wartości miary *fitness*). Interesujące jest obliczanie wartości miary *fitness*, które przebiega etapowo. Pierwszy etap to badanie złożoności wygenerowanego systemu reguł. Jeśli łączna liczba przesłanek jest zbyt duża zwracana jest wartość $ruleComplexityFactor$ i żadne dalsze obliczenia nie są wykonywane. Jeśli złożoność systemu jest mniejsza niż wymagana następuje wstępna ocena działania systemu. Polega ona na ewaluacji systemu na określonej liczbie stanów, które pochodzą z przykładowych walk. Gdy system w zbyt wielu przypadkach na wyjściu daje 0 zwracany jest wynik i system nie podlega dalszej ocenie. Ostatnim etapem jest uruchomienie środowiska *Robocode* i ocena wyników - liczy się liczba punktów i czas trwania walki. Całkowitą formułę *fitness* można przedstawić w następujący sposób:

  $$fitness = 1 + systemResponse + battlePerformance - ruleComplexity$$

2.8 Parametry systemu

Do modyfikowalnych parametrów systemu należą:

- ilość generacji
- rozmiar populacji
- liczba reguł
- wybór przeciwników
- szkielet kontrolera
- czy widoczny ma być ekran walki

3 Wyniki

Najlepsze wyniki otrzymano dla systemu wygenerowanego dla 8 reguł z łącznikiem *OR*. Był on w stanie zdobyć 60% punktów. Jednak system taki jest w stanie opracować wyłącznie proste strategie. W tym przypadku było to jeżdżenie w kółko i prosta umiejętność celowania.

4 Wnioski

4.1 Problemy

Głównym problemem uniemożliwiającym wyewoluowanie dobrego sterownika jest stopień złożoności modelu. Przy 20 regułach długość chromosomu wynosi ponad 400 genów. Aby skutecznie przeszukać taką przestrzeń rozwiązań należy zastosować dużą populację osobników i sporą liczbę generacji. Niestety zasymulowanie dużej liczby generacji było nie możliwe ze względu na czasochłonność oceny osobnika, nawet po zastosowaniu etapowej funkcji oceny. Kolejną trudnością był element losowy w symulacji gry (roboty startują w każdej walce z różnych pozycji). Starano się wyeliminować czynnik losowy poprzez rozegranie odpowiedniej ilość rund i uśrednienie wyniku. Wraz ze wzrostem liczby rund rośnie złożoność czasowa obliczeń dlatego ostatecznie liczbę rund ustalono na 2. Przy systemach regułowych z łącznikiem *AND* trudno było uzyskać system, który reagował by w jakikolwiek sposób na większość stanów wejściowych - często na wszystkich wyjściach pojawiało się 0.