Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

EWOLUCJA STEROWNIKA POJAZDU Z WYKORZYSTANIEM KARTEZJAŃSKIEGO PROGRAMOWANIA GENETYCZNEGO

Witold Szymaniak

Promotor
dr hab. inż. Krzysztof Krawiec

Poznań, 2009 r.

Tutaj karta pracy dyplomowej;
w oryginale w wersji do archiwum PP, w kopiach ksero.

Spis treści

1 Wstęp **1**

2 Cel i zakres pracy **3**

3 Dostępne technologie **5**
3.1 Środowiska symulacyjne 5
3.2 TORCS 7
3.3 Trasy i kontrolery 8
3.4 Aplikacja kliencka i sterowanie 8
3.5 Regulamin zawodów SCRC 10

4 Środowisko obliczeń ewolucyjnych **13**
4.1 Architektura ECJ 16
4.2 Konfiguracja eksperymentu 18

5 Kartezjańskie Programowanie Genetyczne **20**
5.1 Definicja 20
5.2 Reprezentacja 20
5.3 Cechy CGP 21
5.4 Rozszerzenia i modyfikacje CGP 23
5.5 Operatory genetyczne w CGP 23

6 Przegląd podejść do ewolucji sterowników pojazdów **25**
6.1 Zgłoszenia WCCI 2008 SCRC 25
6.2 Koewolucja sterowników pojazdów 27

7 Zagadnienia dynamiki pojazdów **29**
7.1 Podstawowe zagadnienia dynamiki 29
7.2 Ruch po prostej 29
7.3 Ruch krzywoliniowy 31
7.4 Techniki kierowania pojazdem 32

8 Opis przeprowadzony eksperymentów **33**
8.1 Programy i rozszerzenia CGP 33
8.2 Przetwarzanie danych z sensorów 34
8.3 Ewaluacja osobników 36
8.4 Inicjalizacja 37
8.5 Zastosowane operatory genetyczne 37
8.6 Fenotypowanie 37

8.7 Funkcja oceny . . . 38
8.8 Koewolucja . . . 39
8.9 Wybór torów . . . 40
8.10 Przeprowadzone eksperymenty . . . 40
8.11 Obliczenia na wielu maszynach . . . 40

9 Analiza wyników 42
9.1 Rozmiar programu CGP . . . 42
9.2 Wprowadzenie stałych w programach CGP . . . 42
9.3 Zastosowanie reprezentacji z podprogramami . . . 44
9.4 Metody krzyżowania . . . 46
9.5 Rozmiar populacji . . . 48
9.6 Porównanie różnych modeli sensorów . . . 48
9.7 Koewolucja . . . 49
9.8 Analiza najlepszych osobników . . . 51

10 Podsumowanie 55
10.1 Napotkane problemy . . . 55
10.2 Propozycje dalszego rozwoju . . . 56

Literatura 57

Rozdział 1

Wstęp

Roboty odgrywają coraz większą rolę w naszym życiu. Stanowią podstawę wyposażenia nowoczesnych fabryk (roboty przemysłowe), przejmują domowe obowiązki oraz są źródłem rozrywki. Wspomagają ludzką pracę w warunkach narażenia życia i w zadaniach, w których wymagana jest wyjątkowa precyzja. Szczególną klasą robotów są autonomiczne roboty mobilne. Wykorzystując sieć sensorów są one w stanie eksplorować otoczenie, przemieszczać się w nim i przetwarzając zebrane informacje podejmować decyzje niezbędne do wykonania zadania. Zadaniem ich twórców jest nie tylko zaprojektowanie elektroniki i mechaniki ale również zaprogramowanie sztucznej inteligencji *AI* (ang. *Artificial Inteligence*). Stworzenie dobrego sterownika jest problemem nietrywialnym i bardzo pracochłonnym. Jest szczególnie trudne gdy brak dobrego środowiska symulacyjnego do testowania rozwiązań. Wzrost wydajności komputerów powoduje, że coraz popularniejsze stają się rozwiązania, w których sterownik poddawany jest procesowi uczenia.

Dawno dostrzeżono możliwości drzemiące w autonomicznych robotach. Amerykańska agencja rządowa zajmując się rozwojem technologii wojskowej *DARPA* (Defense Advanced Research Projects Agency) w ramach projektu Systemów Bojowych Przyszłości *FCS* (ang. *Future Combat Systems*) przeprowadziła trzy edycję konkursu DARPA Challenge [dar]. Zadaniem uczestników było przygotowanie bezzałogowego samochodu, który samodzielnie porusza się w nieznanym środowisku starając się dotrzeć do punktu docelowego. Pierwsze dwie edycję odbyły się na pustynnych terenach stanu Nevada. Trzecia edycja nosząca nazwę *DARPA* Urban Challenge odbyła się na specjalnie przygotowanych terenach miejskich w Kalifornii i obejmowała takie zadania jak poruszanie się w ruchu ulicznym przy udziale innych uczestników ruchu. Podnoszenie wymagań stawianych uczestnikom zawodów pokazuje jak dynamicznie rozwija się robotyka. Zastosowania militarne nie są jedyną dziedziną, w której wykorzystywane są autonomiczne roboty. Na popularności zyskują sprzęty domowe samodzielnie wykonujące drobne prace. Powstają zautomatyzowane kosiarki do trawy, która samodzielnie dbają o wygląd trawnika. Jednym z wielu urządzeń tego typu jest Mow-Bot [mow]. Urządzenie wyposażone w czujniki zbliżeniowe i czujniki zderzakowe wykrywa i omija pojawiające się na jego drodze przeszkody. Ciekawą koncepcją rozwoju sztucznej inteligencji jest projekt robota Sony Aibo [aib]. Robot ten jest interaktywną zabawką wyglądem przypominającą psa. Mimo niepozornego wyglądu Aibo posiada spore możliwości. Urządzenie wyposażono w 64-bitowy procesor *RISC*, czujnik obrazu *CMOS*, bogaty zestaw czujników: odległości, dotyku, przyspieszenia, drgań. Cała ta aparatura pozwala zaadoptować się do otoczenia oraz nabywać nowe, wygenerowane w interakcji ze środowiskiem, psie zachowania. Komercjalizacja produktu zapewnia pozyskanie odpowiednich środków na rozwój projektu.

Jedną z metodologii rozwijania sterowników (ang. *controller*) odpowiadających za pracę robota jest Robotyka Ewolucyjna *ER* (ang. *Evolutionary Robotics*).Polega ona na osadzeniu bazowej, lo-

sowo wygenerowanej populacji osobników w środowisku testowym. Genotyp każdego osobnika reprezentuje konkretną implementację sterownika robota. Osobniki w środowisku testowym mają pełną swobodę poruszania się i interakcji ze środowiskiem. Ich działania są determinowane przez aktualnie wykonywany algorytm sterowania. Ocena osobników wynika ze sprawności wykonywanych zadań. Najlepiej przystosowane osobniki poddawane są replikacji i modyfikacją zgodnie z wykorzystywanymi operatorami genetycznymi. Proces ten jest powtarzany do czasu uzyskania osobnika odpowiednio realizującego założone zadanie lub braku znaczącej poprawy w określonej liczbie kolejnych pokoleń [NF00].

Ważną rolę w ewolucji sterowników pełnią środowiska symulacyjne. Pozwalają one w szybki oraz dokładny sposób oceniać uzyskanie rozwiązania, nie narażając sprzętu na uszkodzenie. Z obecnie rozwijanych projektów na wyróżnienie zasługuje środowisko Webots [web] oraz Microsoft Robotics Developer Studio [mic]. Są to kompleksowe narzędzia służące do modelowania i symulowania robotów mobilnych. Użytkownik ma możliwość definiowania wszystkich parametrów obiektów znajdujących się na scenie. Zachowanie robota można przetestować w pełni trójwymiarowym świecie z realistyczną fizyką z wykorzystaniem odpowiednio *ODE* [ode] i PhysiX [phy]. Użytkownik może zdecydować się oprogramować jeden z predefiniowanych robotów dostępnych na rynku lub zbudować własnego robota używając gotowych komponentów. Dopracowany kontroler można skopiować na urządzenie i testować w świecie rzeczywistym.

Współczesne gry komputerowe stanowią wyzwanie dla metod sztucznej inteligencji. Liczne gatunki gier, poczynając od gier strategicznych, poprzez symulatory a na strzelankach kończąc *FPS* (ang. *First Person Shooter*) wymagają zupełnie innych podejść niż gry planszowe. Wykorzystanie algorytmów przeszukujących przestrzeń rozwiązań jest z góry skazane na niepowodzenie ze względu na niemalże nieskończoną przestrzeń stanów. Także inne rozwiązania należące do rodziny systemów symbolicznych nie są skuteczne ponieważ rozwiązania, które są wstanie wykształcić przy rozwiązywaniu prostych problemów nie prowadzą do dobrych strategii w pełnej rozgrywce [Mil06]. Z drugiej strony gracze oczekują coraz inteligentniejszych zachowań od ich komputerowych przeciwników. Ponadto powinni oni prezentować zachowania zbliżone do ludzkich, aby satysfakcja z konkurencji była jak największa. Agent ang. *agent* mając do dyspozycji ogromne, realistyczne środowisko z możliwością pełnej interakcji ma dużą swobodę działań. Do zdefiniowania jego zachowania często wykorzystuje się języki skryptowe takie jak LUA [lua]. Narzucają one jednak dość sztywne zachowania i pozbawiają rozgrywkę elementu zaskoczenia. Innym podejściem jest uczenie agentów odpowiednich zachowań przy użyciu metod sztucznej inteligencji. Ta droga jest sukcesywnie rozwijana od wielu lat. Gdy pojawiły się gry wideo zaczęto wykorzystywać sieci neuronowe (ang. *Neural Network*) by uzyskać NPC (ang. *Non Player Character*) sprawnie poruszające się w środowisku 3D. W grach strategicznych pojawiają się coraz śmielsze próby wykorzystywania (ang. *Emergent Behavior*) do uzyskania ciekawych zachowań grupy jednostek [HC04].

Rozdział 2

Cel i zakres pracy

Celem pracy jest próba wyewoluowania sterownika pojazdu w środowisku *TORCS* (ang. *The Open Racing Car Simulator*). Kontroler powinien być zdolny do samodzielnej jazdy po dowolnej trasie bazując na informacji z sensorów. Drugim wariantem testowym będzie konkurencja z człowiekiem lub zaprogramowanym sterownikiem. Jako główne kryteria oceny ewoluowanych osobników przyjęta zostanie średnia prędkość na trasie i uszkodzenia pojazdu wynikające z nieostrożnej jazdy. Logika sterowania zostanie w całości wyewoluowana. Dopuszczalne jest wykorzystanie funkcji pomocniczych stworzonych przez programistę w sytuacjach wyjątkowych takich jak powracanie na tor. Wyewoluowany sterownik zostanie zgłoszony na zawody w celu porównania osiągniętego poziomu z innymi uczestnikami.

W roku 2009 po raz drugi organizowane będą mistrzostwa w symulacjach wyścigów samochodowych (*SCRC*, ang. *2009 Simulated Car Racing Championship*). Zawody przeprowadzone zostaną na trzech konferencjach:

- *IEEE CEC-2009*, Trondheim (Norwegia), 18 - 21 maj.
- *ACM GECCO-2009*, Montreal (Kanada), 8 - 12 czerwiec.
- *IEEE CIG-2009*, Mediolan (Włochy), 7 - 11 wrzesień.

Zadanie uczestników polega na zgłaszaniu sterowników do pojazdów w środowisku *TORCS* o interfejsie zgodnym z przygotowanym przez organizatorów. Następnie na trzech losowo wybranych torach rozgrywane są wyścigi. Każdy wyścig składa się z eliminacji i właściwej rozgrywki. W eliminacjach kontroler uczestnika musi pokonać jak największy dystans w zadanej liczbie dyskretnych kroków symulacji nie przekraczając dopuszczalnego poziomu uszkodzeń pojazdu. W drugiej turze najlepsze sterowniki z eliminacji współzawodniczą ze sobą w wyścigu, startując z miejsc odpowiadających ich pozycją w eliminacjach. Punkty przydzielane są na podstawie zajmowanych miejsc w wyścigu ze wspólnego startu. Regulamin zawodów nie nakłada żadnych ograniczeń co do sposobu wytworzenia sterownika. Może on być w całości napisany przez programistę lub uzyskany na drodze eksperymentu ewolucyjnego.

W celu przeprowadzenia odpowiednich eksperymentów ewolucyjnych należy wykonać szereg zadań. Po pierwsze dokonany zostanie przegląd istniejących narzędzi obliczeń ewolucyjnych w celu wyboru najbardziej odpowiedniego do nadzorowanie i przeprowadzania obliczeń. Uwzględniając możliwości tychże narzędzi wybrany zostanie najbardziej właściwe i spełniające wymagania opisywanego problemu.

Przed zdefiniowaniem eksperymentu przeprowadzona zostanie analiza rozwiązań zgłaszanych na poprzednią edycję konkursu. Kontrolery zostaną ocenione pod kątem skuteczności rozwiązania i stopnia ingerencji programisty w kod sterownika. Pominięte zostaną rozwiązania w całości

zaprogramowane. Opierając się na istniejąc publikacjach dotyczących ewolucji sterowników robotów mobilnych i pojazdów w środowiskach symulacyjnych sformułowane zostaną wnioski dotyczące obecnie stosowanych podejść i ich skuteczności. Analizując literaturę na temat dynamiki pojazdów przedstawiona zostanie charakterystyka suboptymalnego sterownika i parametrów jakie powinien uwzględniać.

Sterowniki pojazdów ewoluowane będą jako programy *Kartezjańskiego Programowania Genetycznego* (*CGP*, ang. *Cartesian Genetic Programming*) [MT00]. Metoda ta posiada szereg własności potencjalnie korzystnych dla wybranego zastosowania, między innymi generowanie programów o kilku wyjściach. Algorytm zostanie dostosowany do problemu ewolucji kontrolera. Wybrana zostanie forma reprezentacji genotypu. Wykonany zostanie przegląd stosowanych operacji genetycznych dla *CGP* i przeprowadzona seria eksperymentów z ich zastosowaniem. Wybrany zostanie zestaw operacji arytmetyczno-logicznych, z których budowany będzie program kontrolera. Wprowadzonych zostanie też kilka autorskich modyfikacji klasycznego podejścia.

W środowisku *TORCS* ewaluacja kontrolerów odbywać się będzie na zadanym torze. Tory różną się długością, szerokością jezdni, rodzajem nawierzchni, liczbą zakrętów, Możliwy jest wybór jednego z gotowych torów dostarczanych z grą lub stworzenie własnych przy użyciu gotowego edytora. Na potrzeby eksperymentów może zajść potrzeba opracowania generatora torów wyścigowych o zadanych parametrach.

Wykonany zostanie szereg eksperymentów przy doborze różnych wartości parametrów, różnych funkcji oceny. Uzyskane wyniki zostaną przeanalizowane na wielu płaszczyznach: zachowanie wyewoluowanych kontrolerów na różnych torach, zbieżność uzyskanych programów ze sterownikami napisanymi przez programistów. Najlepsze uzyskane sterowniki zostaną ocenione pod kątem współzawodnictwa z człowiekiem i gotowymi kontrolerami.

W kolejnych rozdziałach pracy opisane zostaną następujące zagadnienia:

- Rozdział 3 - opis środowisk symulacyjnych ze szczególnym naciskiem na środowisko *TORCS*
- Rozdział 4 – porównanie narzędzi obliczeń ewolucyjnych
- Rozdział 5 – opis *Kartezjańskiego Programowania Genetycznego* i zaproponowanych rozszerzeń
- Rozdział 6 – przegląd podejść wykorzystanych do ewolucji kontrolerów pojazdów
- Rozdział 7 – analiza zagadnień dynamiki pojazdów
- Rozdział 8 – definicje eksperymentów ewolucyjny
- Rozdział 9 – analiza wyników
- Rozdział 10 – podsumowanie

Rozdział 3

Dostępne technologie

3.1 Środowiska symulacyjne

Przed wyborem środowiska *TORCS* wykonano rozpoznanie dostępnych środowisk symulacyjnych. Rozważano wiele różnych narzędzi przeznaczonych do szerokiej gamy zastosowań: od środowisk do testowania i symulacji robotów mobilnych, poprzez symulatory wyścigów a na uproszczonych modelach służących wyłącznie rozrywce kończąc. Każde z narzędzi zostało zweryfikowane pod kątem przydatności do obliczeń ewolucyjnych. Podstawowe kryteria związane z funkcjonalnością, na które postawiono nacisk podczas testów to:

- elastyczny interfejs (*API*) – architektura klient-serwer, prosty protokół, wsparcie wielu języków programowania, podmiana kontrolera bez konieczności restartowania środowiska.
- stabilna praca – możliwość ciągłej pracy przy dużym obciążeniu środowiska przez wiele godzin: brak wycieków pamięci, tworzenia zbyt dużej liczby wątków, nieoczekiwanego przerywania działania.
- działanie w trybie wsadowym (ang. *batch mode*) – przeprowadzenie symulacji w trybie przyspieszonym z pominięciem wszystkich elementów graficznych przystosowanych do interakcji z użytkownikiem. Działanie w systemie bez środowiska graficznego było dodatkową zaletą umożliwiającą uruchamianie na systemach operacyjnych z rodziny *UNIX* z poziomu konta konsolowego (ang. *shell*).

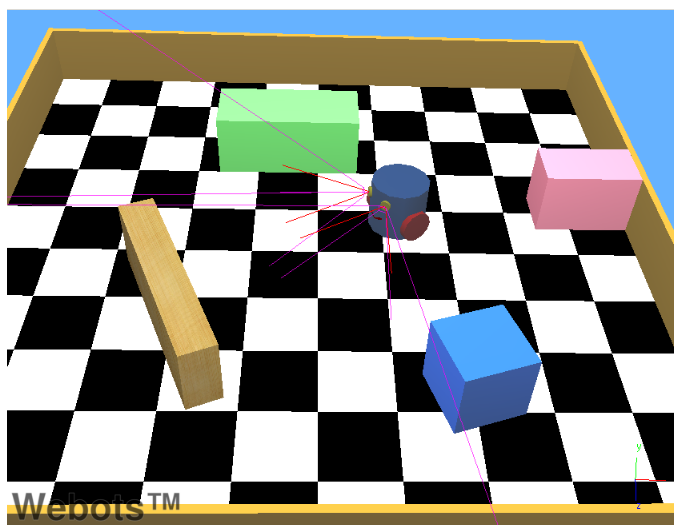
Istniało także wiele dodatkowych funkcjonalności wpływających na korzyść środowisk symulacyjnych. Narzędzia otwarte (ang. *Open Source*) z dużą społecznością użytkowników i programistów były rozpatrywane w pierwszej kolejności. Jeśli dane środowisko dostępne było na kilku platformach sprzętowych dodatkowo wpływało to na jego korzyść. Kolejną cechą rozpatrywaną przy wyborze była popularność danego środowiska: duża liczba dostępnych kontrolerów stworzonych przez użytkowników, wykorzystanie do testowania różnych form sztucznej inteligencji, liczba publikacji opisująca wyniki badań przeprowadzone w środowisku.

Pierwsze testy przeprowadzono przy użyciu środowiska *Webots* [web]. Webots jest trójwymiarowym środowiskiem symulacji robotów. Początkowo rozwijane było jako projekt naukowy mający umożliwić testowanie różnych algorytmów sterowania robotów. Umożliwia ono użytkownikowi definiowanie środowisk 3D z rozbudowanym modelem fizyki zachowującym takie własności ciał jak masa i współczynniki tarcia. W obecnej wersji 6.1.5 możliwe jest tworzenie własnych wtyczek (ang. *plugins*) służących modelowaniu fizyki. Przykładowo domyślny model można zastąpić uproszczonym, który analizuje interakcję ciał w dwóch wymiarach co umożliwia szybszą symulację. Roboty charakteryzują się różnorodnością wyposażenia: sensory odległości, czujniki

nacisku, kamery, nadajniki radiowe, odbiorniki. Użytkownik może definiować obiekty statyczne - przeszkody będące elementami środowiska oraz aktywne czyli roboty. Definiowanie środowiska odbywa się za pomocą *VRML* (ang. *Video Reality Modeling Language*). Środowisko wyposażone jest w wbudowany edytor, który umożliwia szybką edycję kodu kontrolera. Środowisko wspiera programowanie kontrolerów w językach: C, C++, JAVA, Python i MATLAB. W wersji przeznaczonej na platformę LINUX możliwe jest uruchamianie środowiska w trybie wsadowym wystarczy w linii poleceń podać odpowiedni przełącznik i ścieżkę do skryptu. Środowisko *Webots* może być wykorzystywane do następujących zadań:

- modelowanie i prototypowanie robotów.
- algorytmy inteligencji grupowej (ang. *swarm intelligance*) [BM08]
- ewolucja zachowań adaptacyjnych

Niekorzystnie na ocenę symulatora wpływa wysoki koszt zakupu licencji oraz brak możliwości uruchomienia środowiska w trybie serwera, w którym roboty są sterowane przy wykorzystaniu komunikacji sieciowej przez zdalnych klientów.



Rysunek 3.1: Symulacja robota w środowisku Webots [web].

Kolejnym testowanym narzędziem było środowisko *Robocode*. Jest to dynamicznie rozwijany projekt typu Open Source. Został on zaprojektowany jako gra edukacyjna służąca popularyzacji języka Java. Rozgrywka toczy się na planszy będącej prostokątem o ograniczonych rozmiarach. Zadanie programisty polega na napisaniu kontrolera sterującego miniaturowym robotem - czołgiem. Agent gracza wyposażony jest w jednostkę napędową, ruchomą wieżyczkę z działem i radarem. Radar stanowi sensor umożliwiający wykrywanie wrogich jednostek. Możliwe jest sterownie radarem niezależnie od ruchu wieżyczki. W jeden grze mogą brać udział dwa lub więcej robotów. Rozgrywka może się toczyć w trybie indywidualnym oraz drużynowym. W trybie indywidualnym zadana liczba robotów walczy przeciwko sobie traktując każdy napotkany czołg jako wrogi. W trybie drużynowym w starciu biorą udział dwie drużyny robotów liczące 2 lub 5 czołgów. Celem gry jest zdobycie jak największej liczby punktów podczas rozgrywki. Punkty przyznawane są za zadanie obrażeń przeciwnikowi, odbierane za niecelne strzały i uszkodzenia własnego pojazdu. Proste *API* gry powoduje, że cieszy się ona popularnością. Regularnie rozgrywane są zawody dla fanów *Robocode*. Autorzy przygotowali specjalny system nazywany *RoboRumble@Home* służący

do przeprowadzenia rozgrywek turniejowych. Wystarczy, że użytkownik pobierze odpowiednie narzędzie a następnie uruchomi system. Automatycznie pobierane zostają kody wszystkich robotów i definicje potyczek. Zawody rozgrywane są w kilku kategoriach:

- *Megabots* – brak ograniczeń na długość kodu sterownika
- *Minibots* – kod sterownika poniżej 1500 bajtów
- *Microbots* – kod sterownika poniżej 750 bajtów
- *Nanobots* – kod sterownika poniżej 250 bajtów

Prosta implementacja robota w środowisku *Robocode* wygląda następująco:

```
1  package man;
2   import robocode.*;
3
4   public class MyFirstRobot extends Robot {
5       public void run() {
6           while (true) {
7               ahead(100);
8               turnGunRight(360);
9               back(100);
10              turnGunRight(360);
11          }
12      }
13
14      public void onScannedRobot(ScannedRobotEvent e) {
15          fire(1);
16      }
17  }
```

Istnieje szereg czynników wpływających na korzystną ocenę środowiska *Robocode*. Jest ono w całości napisane w języku Java. W internecie można znaleźć setki gotowych sterowników przeciwko którym można testować własne rozwiązania. Możliwość uruchamiania na systemach operacyjnych bez środowiska graficznego. Rozgrywka w środowisku jest dynamiczna i stosunkowo krótka co umożliwia wykonanie wielokrotnych powtórzeń tego samego starcia z uwzględnieniem różnych pozycji startowych. *Robocode* umożliwia symultaniczne przeprowadzenie symulacji na wielordzeniowych procesorach. Głównym minusem jest bardzo uproszczone modelowanie fizyki ograniczające wykorzystanie uzyskanego rozwiązania wyłącznie do zastosowań rozrywkowych.

Ostatecznie po przeanalizowaniu omawianych środowisk dokonano wyboru środowiska *TORCS*.

3.2 TORCS

TORCS (ang. *The Open Car Racing Simulator*) jest środowiskiem symulacyjnym wyścigów samochodowych. Dostępny jest na wielu platformach: *GNU/Linux*, *FreeBSD*, *Mac OS X* i *Microsoft Windows*. Kod źródłowy udostępniony jest na licencji *GNU GPL*. Całe środowisko zostało napisane w języku C i C++. Środowisko *TORCS* jest rozwijane od 1997 roku. Początkowo był to prosty symulator 2D, w którym samochody nie posiadały silników a wyścig odbywał się na pochyłym torze.

Symulator został napisany z wykorzystaniem biblioteki *GLUT* (ang. *OpenGL Utility Toolkit*) służącej do zarządzania oknami aplikacji i przetwarzania operacji wejścia wyjścia. Minimalne

wymagania sprzętowe, które należy spełnić, żeby uruchomić środowisko *TORCS* to: 600 Mhz CPU, 256MB RAM, karta graficzna kompatybilna z OpenGL 1.3 z 64MB RAM.

Środowisko może pełnić funkcję gry komputerowej. Gracz wybiera własny pojazd, poziom trudności, liczbę przeciwników, tor i rozpoczyna jazdę. Sterując klawiaturą, joystickiem lub dedykowanym urządzeniem pokonuje zadaną liczbę okrążeń. Możliwe jest też napisanie własnego robota, sterującego pojazdem na podstawie aktualnego stanu gry. W jednej rozgrywce może brać udział do 10 pojazdów. Symulacja odbywa się w dyskretnych krokach, które odpowiadają 10ms czasu rzeczywistego.

Fizyka zamodelowana w symulatorze uwzględnia opory powietrza, pracę silnika, model kolizji oraz różne współczynniki przyczepności kół.

Co krok symulacji stan pojazdu jest przetwarzany przez kod odpowiedzialny za sterowanie. Obliczana jest wartość zmiennych sterujących na podstawie których zmienia się pozycja i stan pojazdu. W przypadku gdy sterownie pojazdem trwa dłużej niż 10ms wartości wyjść są ignorowane - wykorzystane zostają poprzednie obliczone wartości.

Argumentem decydującym o wybraniu *TORCS* jako środowiska symulacyjnego była informacja o przeprowadzeniu serii zawodów w oparciu o ten symulator w roku 2009 pod nazwą *SCRC*. Przeprowadzenie w roku 2008 pierwszej edycji zawodów umożliwiało wykorzystanie poprzednio uzyskanych wyników jako punktu wyjścia badań. Środowisko posiada też portal ang. *The TORCS Racing Board* gdzie pojawiają się informacje o organizowanych zawodach i możliwościach uczestnictwa.

3.3 Trasy i kontrolery

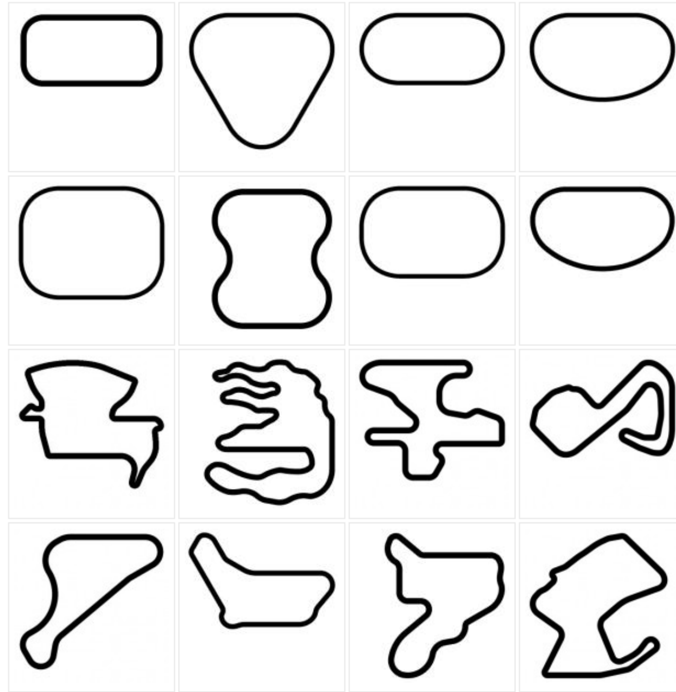
Środowisko zawiera bogaty zestaw predefiniowanych tras i pojazdów. Każdy tor charakteryzuje się następującym zestawem parametrów:

- typ nawierzchni
- długość
- szerokość toru
- liczba punktów kontrolnych

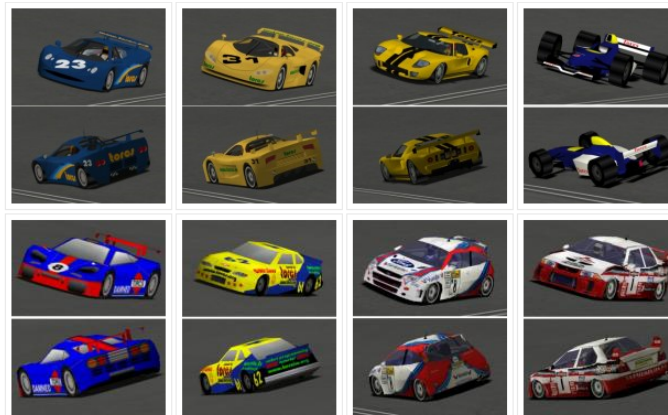
Wszystkie trasy *TORCS* zapisane są w postaci modeli ac3. Wykorzystując narzędzie *TrackEditor* użytkownik może samodzielnie zdefiniować kształt trasy w postaci pliku *XML*. Definiowanie trasy polega na uzupełnieniu atrybutów poszczególnych odcinków toru. Tor budowany jest z fragmentów, które mogą być prostą lub łukiem. Każdy odcinek może mieć równą powierzchnię lub krawędzi osadzone na różnych wysokościach (ang. *banking*). Dodatkowo należy zdefiniować globalne parametry trasy do których należą: szerokość pobocza, rodzaj nawierzchni. Użytkownik dodaje tekstury do trasy i dźwięki odtwarzane w trakcie wyścigu. Następnie przy użyciu programu dostarczonego razem ze środowiskiem o nazwie *trackgen* można skonwertować plik *XML* do postaci modelu *AC3*.

3.4 Aplikacja kliencka i sterowanie

W środowisku *TORCS* sterowniki-roboty są zaimplementowane jako niezależne moduły ładowane do pamięci podczas uruchamiania wyścigu. Takie podejście przyjęte przez autorów środowiska ma kilka wad. Odpytywanie kontrolerów o sygnały sterujące następuje w jednym wątku, więc



Rysunek 3.2: TORCS Wybrane trasy

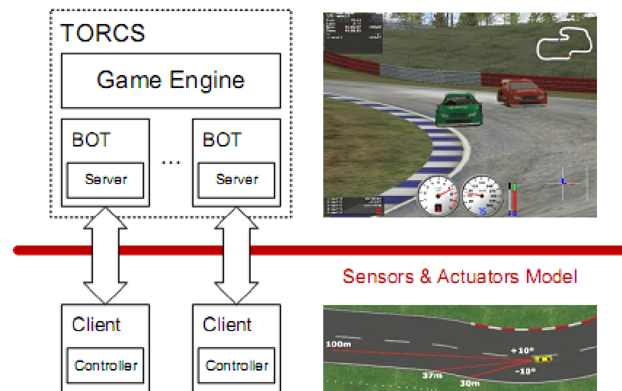


Rysunek 3.3: TORCS Wybrane pojazdy

wolno działający kontroler jest w stanie zablokować całe przetwarzanie - środowisko będzie czekać z dalszym wykonywaniem instrukcji aż do uzyskania odpowiedzi. Kolejną wadą jest możliwość dostęp do globalnego stanu symulatora z poziomu kodu kontrolera. Oprócz informacji pochodzących z sensorów pojazdu, można pobrać informację o kształcie toru, położeniu innych pojazdów. Ponadto tworząc sterowniki w klasyczny sposób programista jest ograniczony do języków C/C++, w których gra została napisana.

Dla potrzeb zawodów *2009 Simulated Car Racing Championship* stworzony został sterownik, który komunikuje się z aplikacją użytkownika, kompensując opisane wady. Po pierwsze wprowadza wzorzec architektury klient-serwer. Klient działa jako osobny proces komunikując się z serwerem przy użyciu połączenia UDP. Poprzez zdefiniowanie limitu czasu oczekiwania na odpowiedź aplikacja może działać w czasie rzeczywistym. W przypadku gdy klient nie odpowie w zadanym czasie do sterowania pojazdem używana jest ostatnio zadana wartość. Wykorzystanie komunikacji po-

przez gniazda sieciowe (ang. *Sockets*) i tekstowy protokół wymiany danych nie nakłada żadnych ograniczeń na język aplikacji klienckiej.



Rysunek 3.4: Architektura TORCS z kontrolerem zawodów

Architektura oprogramowania zawodów została przedstawiona na rysunku 3.4. *Bot* będący modułem środowiska *TORCS* posiada wydzieloną część serwerową odpowiedzialną za komunikację pomiędzy aplikacją kliencką a środowiskiem gry. Każdy klient łączy się z osobnym serwerem na odpowiadającym mu porcie poprzez wysyłanie pakietu zawierającego jego identyfikator. Kiedy wszystkie serwery uczestniczące w wyścigu otrzymają wiadomość od klientów rozpoczyna się wyścig. Po rozpoczęciu wyścigu serwery cyklicznie wysyłają do klientów stan sensorów sterowanych pojazdów a następnie oczekują na odpowiedź zawierającą pakiet sterujący przez $10ms$. Po otrzymaniu odpowiedzi serwer wykonuje instrukcje sterujące zlecone przez klienta. Na podstawie sygnałów sterujących i interakcji z otoczeniem uaktualniany jest stan klienta [DL09]. Zestawienie zmiennych reprezentujących stan pojazdu i parametrów sterujących przedstawiono w tabeli 3.2.

3.5 Regulamin zawodów SCRC

Zawody składają się z dziewięciu wyścigów podzielonych w grupy po trzy odbywające się na kolejnych konferencjach. Uczestnik ma możliwość zgłaszania poprawek w swoim sterowniku przed każdymi kolejnymi zawodami. Każdy wyścig składa się z dwóch etapów rozgrywanych na jednym torze. W pierwszym sterownik porusza się samodzielnie przez 10000 dyskretnych kroków symulacji. Wyniki uzyskane w pierwszym etapie mierzone pokonanym dystansem są podstawą do kwalifikacji do drugiego etapu. W drugim etapie osiem najlepszych sterowników z pierwszego etapu bierze udział we wspólnym wyścigu. Wyścig ten jest powtarzany dziesięciokrotnie w różnych konfiguracjach startowych wybieranych drogą losową. Zadaniem sterownika w drugim etapie jest pokonanie pięciu okrążeń i dojechanie do mety na możliwie najlepszej pozycji. Przydzielanie punktów poszczególnym uczestnikom odbywa się na zasadach podobnych jak w Formule 1. Szczegółowy wykaz liczby punktów za poszczególne miejsca w wyścigu został przedstawiony w tabeli 3.3. Zwycięzcą zostaje zawodnik, którego sterownik zdobędzie sumarycznie najwięcej punktów [LTL].

Nazwa	**Zasięg (jednostka)**	**Opis**
angle	$[-\pi, +\pi]$ (rad)	Kąt pomiędzy osią podłużną pojazdu a osią toru.
curLapTime	$[0, +\infty)$ (s)	Czas obecnego okrążenia.
damage	$[0, +\infty)$ (point)	Wartość uszkodzeń samochodu, większa wartość oznacza więcej uszkodzeń.
distFromStart	$[0, +\infty)$ (m)	Odległość od linii startu mierzona wzdłuż osi toru.
distRaced	$[0, +\infty)$ (m)	Całkowita dystans pokonany przez pojazd od rozpoczęcia wyścigu.
fuel	$[0, +\infty)$ (l)	Zapas paliwa.
gear	$1, 0, 1, \ldots, 7$	Aktualny bieg: -1 wsteczny, 0 luz i biegi od 1 do 7.
lastLapTime	$[0, \infty)$	Czas poprzedniego okrążenia.
opponents	$[0, 100]$ (m)	Wektor 36 wartości z sensorów wykrywających przeciwników. Odczyt z każdego sensora pokrywa wycinek koła o rozpiętości $\pi/18$ i promieniu 100 metrów i zwraca wartość odpowiadającą odległości do najbliższego przeciwnika w danym obszarze. 36 detektorów tworzy pełny stan dookoła pojazdu: zgodnie z ruchem wskazówek zegara od $+\pi$ do $-\pi$ względem osi pojazdu.
racePos	$1, 2, \ldots, N$	Obecna pozycja w wyścigu.
rpm	$[2000, 7000]$ (rpm)	Liczba obrotów silnika na minutę.
speedX	$(-\infty, +\infty)$	Prędkość względem wzdłużnej osi pojazdu.
speedY	$(-\infty, +\infty)$	Prędkość względem poprzecznej osi pojazdu.
track	$[0, 100]$ (m)	Wektor 19 sensorów odległości zwracających wartość pomiędzy pojazdem a krawędzią toru. W przypadku gdy odległość od krawędzi toru jest większa niż 100m zwracana jest wartość 100. Każdy detektor pokrywa zasięg $10^\circ = \pi/36$ w zakresie od $+\pi/2$ do $-\pi/2$ zgodnie z ruchem wskazówek zegara. Gdy pojazd jest poza torem zwracane wartości mogą być niepoprawne.
trackPos	$(-\infty, +\infty)$	Odległość pomiędzy pojazdem a osią toru z uwzględnieniem szerokości toru. Wartość 0 oznacza, że pojazd znajduje się na środku toru, -1 oznacza, że pojazd znajduje się na prawej krawędzi toru, $+1$ oznacza, że pojazd znajduje się na lewej krawędzi toru. Wartości spoza zakresu $[-1; 1]$ oznacza, że pojazd jest poza torem.
wheelSpinVel	$[0, +\infty]$ (rad/s)	Wektor 4 wartości opisujących prędkość poszczególnych kół.

Tablica 3.1: Struktura stanu pojazdu

Nazwa	**Zakres wartości**	**Opis**
accel	$[0, 1]$	Sterowanie pedałem gazu.
brake	$[0, 1]$	Sterowanie pedałem hamulca.
gear	$-1, 0, 1, \ldots, 7$	Ustawienie biegu.
steering	$[-1, 1]$	Sterowanie pojazdem: -1 maksymalny skręt w lewo, $+1$ maksymalny skręt w prawo. Maksymalny kąt skrętu to $\pi/4$.
meta	0,1	Polecenie sterujące symulacją: 0 bez zmian, 1 żądanie zrestartowania aktualnego wyścigu.

Tablica 3.2: Struktura wiadomości sterującej

Miejsce	**Liczba punktów**
1	10
2	8
3	6
4	5
5	4
6	3
7	2
8	1

Tablica 3.3: Punktacja poszczególnych miejsc w zawodach *SCRC*

Rozdział 4

Środowisko obliczeń ewolucyjnych

Do wykonania zaplanowanych eksperymentów należało wybrać odpowiednie narzędzie. Istniały dwie alternatywy kształtujące różne podejścia do problemu. Pierwsze podejście to samodzielna implementacja odpowiedniego narzędzia. Zamiast wybierać złożone środowisko stworzono by małe narzędzie jak na przykład *TinyGP* - minimalistyczne podejście do *Programowania Genetycznego* w języku C i *Java* [PLM08]. Było by one dostosowane do potrzeb danego eksperymentu i zoptymalizowane pod jego wykonywanie. Druga możliwość to skorzystanie z gotowego rozwiązania open source. Wyjście to pozwalało na wykorzystanie zaawansowanych możliwości dostępnych narzędzi: zapisywania stanu eksperymentu, generowania bogatych statystyk w postaci wykresów lub logów i przetwarzania rozproszonego opartego o komunikację sieciową. Zdecydowano o wyborze drugiego podejścia. Z dostępnych środowisk o różnych architekturach i funkcjonalności wyselekcjonowano grupę najodpowiedniejszych do planowanego zadania.

Nazwa	OpenSource	Licencja	Język	Obecna wersja (data wydania)
JGAP	tak	GNU LGPL v2.1	Java	3.4.3 (27.02.2009)
ECJ	tak	Academic Free License v3.0	Java	19 (1.09.2009)
JAGA	tak	GNU GPL v2.0	Java	1.0 Beta (13.04.04)
OpenBeagle	tak	GNU LGPL v2.1	C++	3.0.3 (29.10.2007)
EO	tak	GNU LGPL v3.0	C++	1.0.1 (23.01.2007)

Tablica 4.1: Zestawienie narzędzi obliczeń ewolucyjnych

Wybrane narzędzia zostały przedstawione w tabeli 4.1. Każde z wybranych narzędzi zostało zaimplementowane w jednym z dwóch języków: C++ lub Java. Dokonując wyboru środowiska miano na względzie jego kompatybilność z zestawem klientów przeznaczonych do tworzenia kontrolerów na zawody *WCCI2009*. Takie aplikacje klienckie istniały w językach C++ i Java. Wybór jednego z tych języków umożliwiał wykonywanie obliczeń na kilku maszynach uruchomionych na różnych systemach operacyjnych. W przypadku C++ zadanie takie wymagałoby rekompilacji źródeł na danej maszynie. Dla Javy wystarczyłoby zadbać o odpowiednią wersję maszyny wirtualnej Javy (ang. *Java Virtual Machine*). Ze względu na udogodnienia języka Java takie jak automatyczne usuwanie nieużywanych obiektów (ang. *Garbage Collection*), lepsze dopasowanie do paradygmatów programowania obiektowego (ang. *object-oriented programming*) i środowiska programistyczne (*IDE*) o dużych możliwościach podjęto decyzję aby dalsze poszukiwania zawęzić do narzędzi opartych właśnie na tym języku. Ograniczenie to zmniejszyło listę potencjalnych wyborów do trzech narzędzi: *JGAP*, *ECJ*, *JAGA*.

JGAP (ang. *Java Genetic Algorithms Package*) to narzędzie zawierające implementację *Algorytmów Genetycznych* i ang. *Programowania Genetycznego* [KM]. Środowisko umożliwia wykony-

wanie obliczeń na wielu maszynach. Klient wysyła zadanie do serwera, który rozprasza przetwarzanie na maszyny obliczeniowe. Stan populacji w danym etapie obliczeń może zostać zserializowany do pliku w formacie *XML* co umożliwia rozpoczęcie obliczeń od zadanego momentu lub dokładną analizę ewolucji. Cały kod biblioteki opatrzony jest dokumentacją *JavaDoc*. Dobrym przykładem wykorzystania środowiska jest projekt *RobocodeJGAP*. Jest to integracja narzędzia z środowiskiem symulacyjnym *Robocode*, która służy do ewolucji metodą Programowania Genetycznego kontrolerów robotów.

JAGA ang. *Java Api for Genetic Algorithms* jest darmowym narzędziem open source rozwijanym na University College London [Pap]. Opracowane przez grupę specjalistów od obliczeń ewolucyjnych i projektowania aplikacji. Wtyczkowa budowa umożliwia łatwe rozszerzanie narzędzia. Narzędzie zawiera dużą liczbę predefiniowanych algorytmów i operatorów co umożliwia łatwe zaadoptowanie do własnych potrzeb. JAGA jest wykorzystywane w wielu projektach akademickich. Minusem środowiska jest brak dokumentacji. Ponadto projekt od dłuższego czasu nie jest rozwijany.

Ostatecznie zdecydowano o wyborze biblioteki ECJ ponieważ jest to najbardziej dojrzały, dynamicznie rozwijane środowisko wykorzystywane w wielu projektach akademickich i z dużą liczbą rozszerzeń.

Środowisko obliczeń ewolucyjnych *ECJ* (ang. *Evolutionary Computation in Java*) jest rozwijane od 1997 roku [ea]. Obecnie oficjalne wydanie nosi numer 19. Projekt jest rozwijany na George Mason University pod kierownictwem Sean'a Luke'a. Aktualny kod źródłowy całego narzędzia jest dostępny poprzez repozytorium CVS. Biblioteka udostępniona jest na licencji ang. *Academic Free License*. Za wyborem narzędzia przemawia przejrzysta architektura, duża wydajność, łatwość konfiguracji. Środowisko posiada szereg funkcjonalności ułatwiający programiście pracę. Do najważniejszych należą:

- Konfiguracja eksperymentów – do ustawiania parametrów klas programu służą hierarchiczne pliki konfiguracyjne. Każda klasa, której instancje są wykorzystywane podczas obliczeń udostępnia obsługiwane parametry wraz z ich wartościami domyślnymi. Parametry mogą zostać zdefiniowane dla konkretnej instancji klasy lub dla wszystkich instancji danej klasy. Konfiguracja może się składać z wielu plików z parametrami poprzez zdefiniowanie relacji ojciec-potomek w pliku przekazanym do środowiska.

- Logowanie wiadomości – zestaw klas umożliwiający zarządzanie informacjami dotyczącymi eksperymentu przekazywanymi użytkownikowi. Użytkownik ma do dyspozycji kilka poziomów logowania takich jak: *INFO*, *WARN*. Tworzone logery posiadają konfigurowalne wyjście. Wiadomości mogą być prezentowane na konsoli, logowane do pliku, zapisywane do strumienia połączenia sieciowego.

- Punkty kontrolne (ang. *checkpoints*) – użytkownik ustawiając odpowiednie wartości parametrów wymusza aby co daną liczbę generacji populacja była archiwizowana. Następnie po przerwaniu obliczeń eksperyment może zostać wznowiony od zadanej iteracji.

- Graficzny interfejs użytkownika – wykorzystując graficzne środowisko napisane w bibliotece *Swing* możliwe jest interaktywne zarządzanie eksperymentem. Użytkownik może zatrzymywać obliczenia, przeglądać populację osobników. Z wykorzystaniem biblioteki *JFreeChart* i *iText* tworzone są wykresy zmian dowolnych estymat i zapis ich do pliku .pdf lub .eps.

- Obliczenia wielowątkowe – na jednym komputerze obliczenia mogą być wykonywane w kilku wątkach co umożliwia lepsze wykorzystanie zasobów jednostek wielordzeniowych . Parametry

obliczeń można uzależnić od wartości *thread_num* identyfikatora wątku. Przykładowo każdy wątek może wykorzystywać inny generator liczb losowych.

- Klasy bazowe eksperymentów – przy definiowaniu własnych problemów, operatorów lub statystyk wygodne jest wykorzystanie predefiniowanych klas zwanych *Forms*. Powoduje to że programista musi zaimplementować wyłącznie metody, których ciało będzie się różniło od domyślnego przetwarzania. Reszta funkcjonalności jest dziedziczona z klasy bazowej.
- Generator liczb losowych – domyślnym generatorem liczb losowych jest *Mersenne Twister*, który jest szybkim generatorem liczb losowych dostarczającym wysokiej jakości liczby losowe.

ECJ oferuje też szeroki zakres funkcjonalności dotyczących obliczeń ewolucyjnych, programowania genetycznego i strategii ewolucyjnych:

- Wsparcie dla obliczeń rozproszonych - obliczenia mogą być wykonywane na wielu maszynach przy wykorzystaniu komunikacji sieciowej. Dla ewolucji wielopopulacyjnej zaimplementowany jest asynchroniczny model wyspy ang. *Asynchronous island model* z wykorzystaniem protokołu TCP/IP. W przypadku gdy ewolucji poddawana jest jedna populacja użytkownik może skorzystać zdecydować się na obliczenia w architekturze Master/Slaves, aplikacje klienckie (Slaves) mogą być dynamicznie podłączane i odłączane od Mastera.
- Algorytmy genetyczne i programowanie genetyczne – wsparcie dla ewolucji z różnymi wariantami zmian: Steady State, Generational Evolution. Eksperymenty mogą być wykonywane z zachowanie elitaryzmu lub bez.
- Strategie ewolucyjne – wsparcie dla strategii ewolucyjnych z operatorami (μ, λ) i $(\mu + \lambda)$.
- Elastyczna architektura tworzenia populacji – proces wykonywanie operacji genetycznych na populacji został zaprojektowany jako zestaw bloków, które mogą być łączone w dowolny sposób. Każdy blok, który reprezentuje operator posiada wyznaczoną liczbę rodziców. Umożliwia to tworzenie dowolnych struktur bazując na gotowych elementach.
- Operatory genetyczne i reprezentacja – duża liczba operatorów genetycznych. Różne typy selekcji: ruletka, turniejowa, losowy. Operatory krzyżowania i mutacji. Genotypy mogą być reprezentowane jako wektory stałej lub zmiennej długości o polach różnych typów.
- Koewolucja – eksperymenty koewolucyjne dla jednej populacji z wykorzystaniem kompetytywnej funkcji oceny lub wielu populacji.
- Podpopulacje i gatunki – w obrębie jednej populacji możliwe jest zdefiniowane podpopulacji i gatunków.
- Wielokryterialna optymalizacja – wykorzystanie algorytmu SPEA2 do poszukiwania rozwiązań Pareto-Optymalnych.
- Ewolucja różnicowa.
- Programowanie Genetyczne – ewolucja drzew silnie typowanego programowania genetycznego. Dodanie do genotypu stałych ulotnych (ang. *Ephemeral Constants*. Rozbudowanie standardowej reprezentacji o Automatycznie Definiowane Funkcję i Autmatycznie Definiowane Makra. Ewolucja drzew i lasów. Rózne algorytmy tworzenia drzew.
- Inne algorytmy optymalizacyjne – ang. *Particle Swarm Optimization*

4.1 Architektura ECJ

Rysunek 4.1: Diagram klas pakietu ECJ

Klasy środowiska ECJ zostały podzielone w logiczne grupy odpowiadające za wykonywanie i zarządzanie różnymi etapami eksperymentu. W pierwszej fazie inicjalizowana jest populacja osobników, następnie wielokrotnie powtarzane są kroki algorytmu genetycznego: selekcja, mutacja, krzyżowanie. W trakcie tych obliczeń wielokrotnie następuje ocena osobników. Przetwarzane są informacje o eksperymencie, które w formie statystyk umieszczane są w pliku logu lub na konsoli. Cyklicznie globalny stan eksperymentu może być zapisywany na dysku w postaci binarnego pliku. Po spełnieniu warunku zakończenia następuje zwolnienie zasobów i usunięcie z pamięci osobników i obiektów pomocniczych. Wykorzystując semantykę języka Java logiczne grupy zostały odseparowane za pomocą pakietów. Główne pakiety dzielące między sobą najważniejsze funkcjonalności to:

- **ec.util** – pomocnicze klasy nie związane bezpośrednio z obliczeniami ewolucyjnymi.
- **ec** – zbiór uogólnionych klas odpowiadających za ewolucję. Zawiera podstawowe klasy tworzące populację, implementację operatorów selekcji, ewolucję pokoleniową lub typu *Steady state*, optymalizację wielokryterialną.
- **ec.gp** – klasy związane z Programowanie Genetycznym.
- **ec.vector** – zawiera wektorową reprezentację osobników i odpowiadających im gatunków. W pakiecie znajduje się kilka domyślnych reprezentacji różniących się typami alleli. Do

dyspozycji użytkownika są wektory następujących typów: *Bit, Byte, Double, Float, Integer, Short, Long.*

- **ec.app** – przykładowe eksperymenty dla różnych algorytmów. Zawiera zestaw znanych problemów dla poszczególnych klas obliczeń ewolucyjnych.
- **ec.experiment** – tymczasowy niestabilny kod eksperymentów.

Implementacja klas w ECJ jest zgodna z przyjętymi konwencjami. Spójność ta ułatwia pisanie nowych klasy i analizowanie działania już istniejących. Nadrzędną abstrakcją jest interface *Setup*. Posiada on jedną metodę o nazwie *setup()*. Jest ona wywoływana podczas inicjalizacji eksperymentu. Obiekty implementujące ten interfejs mogą być konfigurowane poprzez parametry eksperymentu (*Parameter* i *ParameterDatabase*). Istotnym z punktu widzenia wydajności aspektem jest również zarządzanie cyklem życia obiektów. ECJ wykorzystuje kilka wzorców projektowych tworzenia obiektów i ograniczania liczby ich instancji. Klasy typu *Singleton* posiadają jedną globalną instancję. Modyfikacją wzorca *Singleton* jest *Klika* czyli kilka klas posiadających wspólny typ bazowy z instancją dla każdego wyspecjalizowanego podtypu. Dla klas, których obiekty często są tworzone i usuwana stosowany jest wzorzec *Prototyp*, kolejne instancje tworzone są poprzez kopiowanie prototypowego obiektu. Przykładem użycia tej strategii jest zarządzenia obiektami klas *Individual.* Szczególnym przypadkiem wzorca *Prototyp* jest wzorzec *Group* stosowany na przykład dla klas *Population.* Obiekty tego typu są zarządzane jak obiekty prototypowe z wyszczególnieniem, że obiekt prototypowy jest wykorzystywany podczas obliczeń a nie stanowi wyłącznie szablonu do tworzenia nowych obiektów.

Główną klasą zarządzającą przebiegiem eksperymentu jest klasa *Evolve.* Zawiera ona metodę *main*, która jest punktem startowym aplikacji. Inicjalizuje ona dodatkowe narzędzia wykorzystywane w ewolucji odpowiedzialne za: zapis stanu ewolucji, logowanie, bazę parametrów, obiekt stanu ewolucji *EvolutionState.* Klasa ta reprezentuje stan całej populacji co umożliwia przekazanie go jako parametru metody lub zapis stanu ewolucji poprzez serializację tejże klasy. Dostęp do bazy parametrów konfigurujących eksperyment również odbywa się z pośrednictwem klasy *EvolutionState.* Dodatkowo zawiera ona kilka podstawowych klas typu *Singleton*:

- **Initializer** – tworzy początkową populację. Do tego celu może zostać wykorzystany generator liczb losowych lub populacja może zostać wczytana z pliku.
- **Evaluator** – służy do oceny osobników i ustawiania ich wartości przystosowania.
- **Breeder** – implementuje mechanizm tworzenie kolejnych generacji populacji bazując na poprzednim stanie.
- **Exchanger** – odpowiada za wymianę danych/osobników pomiędzy podpulacjami oraz pomiędzy procesami uczestniczącymi w obliczeniach.
- **Finisher** – wykonuje wszystkie operacje związane ze zwalnianiem zasobów i właściwym usuwaniem obiektów z pamięci.

Główną składową stanu ewolucji jest populacja. Populacja składa się z jednej lub wielu podpopulacji osobników. Obiekt populacji jest przekazywany do metod odpowiedzialnych za inicjalizowanie i modyfikowanie populacji pomiędzy kolejnymi generacjami. Każda podpopulacja składa się z osobników i gatunków, do których należą osobniki. Genotyp osobnika stanowi potencjalne rozwiązanie problemu ewolucyjnego. Osobnik poddawany jest ewaluacji, która ustawia jego wartość fitness. Wszystkie operacje genetyczne przyjmują jako argumenty obiekty klasy *Individual.*

Narzędzie *ECJ* definiuje specyficzny mechanizm reprodukcji osobników. Mechanizm tworzenia populacji dla kolejnych generacji na podstawie poprzednich pokoleń nosi nazwę ang. *Breeding Pipeline*. Każdy potok definiuje określoną liczbę źródeł, którą należy go zasilić. Dla przykładu potok odpowiadający operatorowi krzyżowania potrzebuje dwóch poprzedników dostarczających osobników poddawanych krzyżowaniu. Źródłem danego potoku może być inny potok lub operator selekcji ang. *SelectionMethod*. Są to specyficzne odmiany potoków nietworzące nowych osobników a jedynie odpowiedzialne za wybór odpowiednich osobników i przekazanie ich dalej. Tworzone osobniki bazują na przekazanych do metody reprodukcji. Klasa *Evaluator* na podstawie charakterystyki zdefiniowanego problemu ewolucyjnego dokonuje oceny osobników.

4.2 Konfiguracja eksperymentu

Środowisko *ECJ* jest bardzo elastyczne. Cecha ta jest następstwem zaprojektowania architektury, która w łatwy sposób pozwala na konfigurowanie parametrów klas. ECJ wykorzystuje hierarchiczne pliki konfiguracyjne. Na podstawie pliku lub zbioru plików tworzona jest instancja klasy *ParameterDatabase*. Będąc składową stanu ewolucji jest ona przekazywana do metod odpowiedzialny za ustawienie pożądanego stanu obiektu. Parametry pozwalają dobrać stałe eksperymentu, wybrać klasy rozszerzające bazowe abstrakcje, zdefiniować strukturę procesu reprodukcji populacji. Ścieżka do głównego pliku konfiguracyjnego jest przekazywana z linii poleceń do aplikacji ECJ. Opis parametrów przekazywanych do danej klasy jest opisany w dokumentacji *javadoc* klasy. Wartości jakie mogą przyjmować parametry definiowane w pliku konfiguracyjnym:

- ciągi znaków
- ścieżki względne i bezwzględne
- wartości stało i zmiennoprzecinkowe
- wartości logiczne
- nazwy klas

Struktura nazwy określającej obiekt do, którego zostanie zastosowany dany parametr jest hierarchicznej struktury. Kolejne elementy ścieżki do obiektu odseparowane są przy użyciu kropki. W przypadku klas typu *Group* lub *Prototype* w literale może pojawić się liczba dziesiętna określająca indeks obiektu. Parametry dla danej klasy mogą zostać zdefiniowane globalnie czyli dla wszystkich instancji lub dla konkretnej wskazanej instancji.

Ze względu na strukturę konfiguracji dopuszczalne są wielokrotne wpisy definiujące wartość danego parametru. W takim przypadku stosowane są następujące reguły określające priorytet danego wpisu, obowiązuje zasada pierwszego dopasowania:

- Użyj wartości, która została zdefiniowana przez programistę w kodzie programu.
- Użyj wartości, która została przekazana jako argument z linii poleceń.
- Użyj wartości, która została zdefiniowana w pliku głównym. W trakcie sekwencyjnego przetwarzania pliku nowo odczytane wartości nadpisują wszystkie poprzednie.
- Użyj wartości, które zostały zdefiniowane lub wywiedzione poprzez dziedziczenie z pliku rodzica o większym indeksie

Przykładowy plik konfiguracyjny dla problemu wyszukiwania genotypu o maksymalnej liczbie jedynek:

```
verbosity        = 0

breedthreads        = 1
evalthreads        = 1
seed.0                = 4357

state                = ec.simple.SimpleEvolutionState

pop                = ec.Population
init                = ec.simple.SimpleInitializer
finish                = ec.simple.SimpleFinisher
breed                = ec.simple.SimpleBreeder
eval                = ec.simple.SimpleEvaluator
stat                = ec.simple.SimpleStatistics
exch                = ec.simple.SimpleExchanger

generations                = 200
quit-on-run-complete        = true
checkpoint                = false
prefix                        = ec
checkpoint-modulo        = 1

stat.file                = \$out.stat

pop.subpops                = 1
pop.subpop.0                = ec.Subpopulation

pop.subpop.0.size                = 10
pop.subpop.0.duplicate-retries        = 0
pop.subpop.0.species                = ec.vector.VectorSpecies

pop.subpop.0.species.fitness        = ec.simple.SimpleFitness
pop.subpop.0.species.ind        = ec.vector.BitVectorIndividual

pop.subpop.0.species.genome-size        = 20
pop.subpop.0.species.crossover-type        = one
pop.subpop.0.species.crossover-prob        = 1.0
pop.subpop.0.species.mutation-prob        = 0.01

pop.subpop.0.species.pipe                        = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0                = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0        = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1        = ec.select.TournamentSelection

select.tournament.size                = 2

eval.problem                = ec.app.tutorial1.MaxOnes

breed.elites.0 = 1
```

Rozdział 5

Kartezjańskie Programowanie Genetyczne

5.1 Definicja

Algorytmy genetyczne są to inspirowane biologią metody przeszukiwania przestrzeni rozwiązań oparte na mechanizmach doboru naturalnego i dziedziczności. W każdym pokoleniu powstaje zbiór osobników reprezentowanych przez ciągi bitowe utworzony z fragmentów najlepiej przystosowanych osobników poprzedniego pokolenia. Dodatkowo mogą występować samoistne zmiany w strukturze osobników co wprowadza element losowości do procesu poszukiwania rozwiązania. Jedną z form algorytmów genetycznych jest programowanie genetyczne. Jego celem jest zautomatyzowane tworzenie programów komputerowych w oparciu o definicję rozwiązywanego problemu oraz zbiór funkcji podstawowych oraz terminali.

Kartezjańskie Programowanie Genetyczne CGP (ang. *Cartesian Genetic Programming*) jest odmianą programowania genetycznego, w którym program jest reprezentowane jako skierowany graf. Podejście to wykazuje się sporym podobieństwem reprezentacji genotypu co *Rozproszonego Równoległego Programowania Genetycznego PDGP* (ang. *Parallel Distributed Genetic Programming*) [Pol97]. Metoda *CGP* została opracowana przez Juliana F. Millera i Petera Thompsona [MT00]. Początkowo miała ona służyć automatycznemu projektowaniu układów cyfrowych składających się z określonego zbioru bramek logicznych rozmieszczonych na dwuwymiarowej kracie. Od tego czasu *CGP* stosowano z powodzeniem w takich dziedzinach jak projektowanie filtrów cyfrowych, przetwarzanie obrazów, sztuczne życie (ang. *artificial life*), ewolucyjna sztuka.

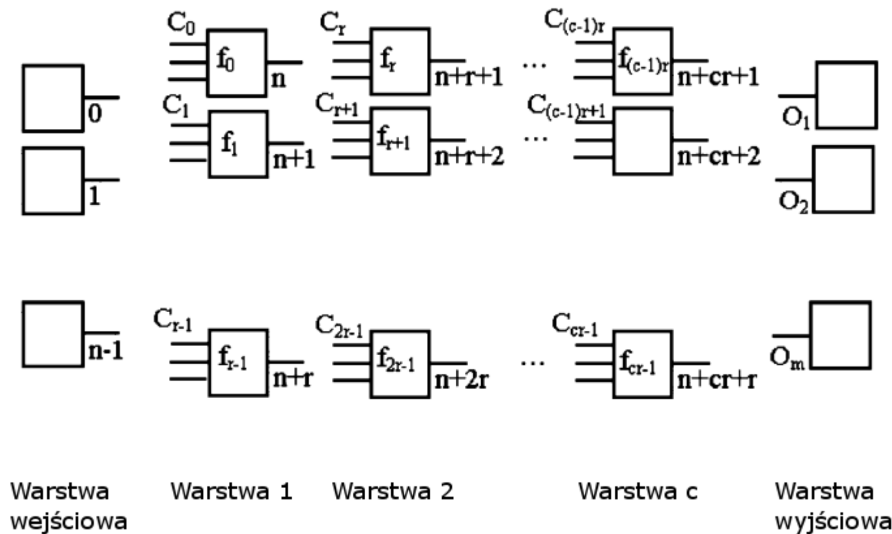
5.2 Reprezentacja

Genotyp dla problemu *CGP* jest wektorem liczb całkowitych lub zmiennoprzecinkowych. Podczas fenotypowania wartości alleli mapowane są na bloki funkcyjne. Przykładowe mapowanie genotypu na fenotyp przedstawiono na rysunku 5.2. Każdy blok funkcyjny jest definiowany poprzez numer porządkowy realizowanej funkcji oraz indeksy wierzchołków, które są jej argumentami. W pierwotnej postaci *CGP* było reprezentowane jako kartezjański układ współrzędnych, na którym wierzchołki były zorganizowane w warstwy i konieczne było określenie liczby wierzchołków w każdym wierszu (r) i kolumnie (c). Zabronione było tworzenie połączeń między wierzchołkami znajdującymi się w jednej kolumnie podobnie jak w wielowarstwowej sieci neuronowej. Dodatkowe ograniczenia na topologię sieci połączeń nakłada wprowadzony parametr $level - back$ określający ile warstw wstecz mogą sięgać połączenia. Specjalną warstwę tworzą wejścia programu, które nie

podlegają modyfikacjom. W wielu implementacjach *CGP* przyjmuje się jako liczbę wierszy wartość jeden a liczbę kolumn jako maksymalną, określaną przez użytkownika, liczbę wierzchołków programu. Ostatnie m pozycji genotypu koduje indeksy wierzchołków wyjściowych. Formalnie genotyp można przedstawić jako ciąg:

$$C_0, f_0; C_1, f_1; \ldots; C_{cr-1}, f_{cr-1}; O_1, O_2, \ldots, O_m$$

gdzie C_i oznacza połączenia wierzchołka i z wierzchołkami, które stanowią wejście dla operacji f_i. Wartość f_i określa jedną z zdefiniowanego przez użytkownika zbioru funkcji. Wierzchołki pośrednie, znajdujące się w warstwach od 1 do c, są numerowane kolejno. Indeks pierwszego wierzchołka W_0 ma wartość n, dzięki czemu przy użyciu jednej wartości możliwe jest adresowanie zarówno wierzchołka z warstw pośrednich oraz wejścia programu. Wartości O_i określają indeksy wierzchołków, z których pobierane są wyjścia programu. Powszechna jest sytuacja gdy arność funkcji ze zdefiniowanego zbioru jest różna. W takich sytuacjach przyjmuje się jako liczbę argumentów każdej z funkcji w zbiorze maksimum po arności wszystkich funkcji. Dzięki takiemu zapisowi długości genotypu pozostają stałe. W celu uniknięcia cykli w grafie reprezentującym program dla danego wierzchołka W_i z jego wejściami połączone mogą być wyjścia wierzchołków z warstw o indeksach mniejszych niż warstwa tego wierzchołka [MS06].



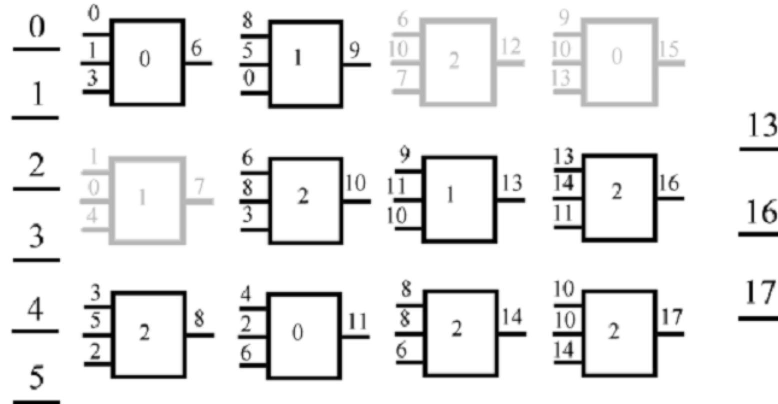
Rysunek 5.1: Uogólniona reprezentacja Programu Kartezjańskiego dla n wejść i m wyjść. Parametry zdefiniowane przez użytkownika: liczba wierszy (r), liczba kolumn (c), liczba warstw wstecz, do których można tworzyć połączenia ($level - back$) oraz zbiór funkcji. Każdy wierzchołek posiada C_i alleli opisujący wejścia i pojedynczy allel opisujący realizowaną funkcję.

5.3 Cechy CGP

Metoda kartezjańskiego programowania genetycznego posiada wiele potencjalnie korzystnych cech, do których należą:

- ograniczony rozmiar programu –ponieważ genotyp programu kartezjańskiego nie zmienia długości w wyniku stosowania operatorów mutacji i krzyżowania, górne ograniczenie na długość programu jest znane od początku eksperymentu. W przypadku gdy w fenotypie pojawiają się wierzchołki niepołączone, długość programu ulega skróceniu. Odróżnia to metodę *CGP* od

0 1 3 *0* 1 0 4 *1* 3 5 2 *2* 8 5 0 *1* 6 8 3 *2* 4 2 6 *0*
6 10 7 *2* 9 11 10 *1* 8 8 6 *2* 9 10 13 *15* 13 14 11 *16* 10 10 14 *2* 13 16 17



Rysunek 5.2: Przykładowe fenotypowanie genotypu dla CGP.

tradycyjnego programowania genetycznego, którego problemem jest wzrost rozmiaru kodu w kolejnych etapach ewolucji. Zjawisko to określa się mianem ang. *bloat*. Przejawia się ono wzrostem średniego rozmiaru programu po początkowym okresie eksperymentu kiedy średni rozmiar programu jest stały. Wzrost długości programu nie powoduje jednak wzrostu średniej wartości przystosowania. Długie programy genetyczne wymagają większej mocy obliczeniowej, są trudniejsze w analizie i są bardziej podatne na zjawisko przeuczenia [PLM08].

- prostota reprezentacji – ze względu na przyjętą reprezentację dla wariantu zmiennoprzecinkowego, każda kombinacja wartości kodujących jest poprawna. Nie zachodzi więc potrzeba naprawiania genotypu po wykonaniu operacji genetycznych, co mogłoby być zadaniem nietrywialnym i złożonym obliczeniowo. Ponadto w ewolucyjnych eksperymentach przy wykorzystaniu *CGP* możliwe jest stosowanie dowolnych operatorów genetycznych, wykorzystywanych dla algorytmów genetycznych o stałej długości genotypu.

- struktura *DAG* – reprezentacja grafowa jest ogólniejsza od właściwej programom genetycznym reprezentacji drzewa. reprezentacja programu powoduje, że zmiana liczby wyjść programu wymaga wyłącznie dodania odpowiedniej liczby alleli do genotypu. Wykorzystanie *CGP* dla problemów o wielu wyjściach jest wskazane w przypadkach gdy na podstawie wiedzy dziedzinowej problemu zachodzi podejrzenie silnych zależności między zmiennymi wyjściowymi układu. Współczesne podejścia do programowania genetycznego umożliwiają wielokrotne wykorzystanie bloku kodu poprzez definiowanie pomocniczych struktur takich jak *Automatycznie Definiowane Funkcje* (ang. *Automatically Defined Functions*) i Automatycznie Definiowane Metody (ang. *Automatically Defined Methods*). W programach *CGP* jedną z zalet przyjętej reprezentacji jest możliwość wielokrotnego wykorzystywania wyjść wierzchołków (ang. *Automatic Re-used Outputs*), co zapewnia podobną funkcjonalność jak *ADF* i *ADM*. Dodatkowo użycie *ARO* nie wymaga żadnych zmian w kodowaniu.

- nadmiarowość genotypu i neutralność – ze względu na wprowadzenie w *CGP* fazy fenotypowania potencjalnie istnieje wiele genotypów o takim samym fenotypie co wpływa na zjawisko

neutralności. Istotność neutralności jest powszechnie uznawana w naturalnej ewolucji na poziome molekularnym. Jak dowiedziono zjawisko neutralności w zbieżnej populacji może przyczynić się do poprawy uzyskiwanych wyników. Zjawisko neutralności jest ściśle związane z pojęciem *nadmiarowości genotypu*. W kartezjańskim programowaniu genetycznym istnieje kilka źródeł nadmiarowości. Pierwszym typem jest *nadmiarowość wierzchołkowa* (ang. *node redundancy*), będąca następstwem pojawiania się w fenotypie niepołączonych wierzchołków. Kolejnym typem nadmiarowości jest *nadmiarowość wejść* (ang. *input redundancy*), wynika ona ze stałej liczby, równej maksymalnej arności funkcji ze zbioru funkcji, pozycji przypadających na allele definiujące wejścia danego wierzchołka. Ostatnim typem nadmiarowości zachodzącym również w *GP* jest *nadmiarowość funkcyjna* (ang. *functional redundancy*) dotyczy ona przypadków gdy obliczanie pewnej wartości pośredniej angażuje więcej operacji niż jest to wymagane np. poprzez dodanie nigdy nie spełnionego warunku logicznego [MS06].

5.4 Rozszerzenia i modyfikacje CGP

W standardowym podejściu genotyp dla kartezjańskiego programu genetycznego jest reprezentowany przez wartości całkowitoliczbowe, które to wartości są w odpowiedni sposób dekodowane na program kartezjański. W celu poszerzenia możliwości stosowania różnych operatorów genetycznych opracowano reprezentację zmiennoprzecinkową. Zmodyfikowana reprezentacja *CGP* wykorzystuje do kodowania grafu wektor wartości zmiennoprzecinkowych. Każdy wartość odpowiada pojedynczemu genowi z reprezentacji całkowitoliczbowej i przyjmuje wartości z zakresu $[0; 1]$. Podobnie jak w reprezentacji całkowitoliczbowej genotyp podzielony jest na fragmenty, z których pierwsza wartość określa funkcję realizowaną przez dany wierzchołek a kolejne definiują argumenty przyjmowane przez tą funkcję. Podczas mapowania genotypu na fenotyp, przy założeniu ogólnego przypadku, w którym $r = 1$ i $c = gene_{total}$, postać zmiennoprzecinkowa jest zamieniana na całkowitoliczbową zgodnie z następującymi przekształceniami:

$$gene_i = floor(genef_i * func_{total}) \tag{5.1}$$

$$gene_j = floor(genef_j * nodeterm_j) \tag{5.2}$$

gdzie w (5.1) $gene_i$ koduje funkcję, a w (5.2) $gene_j$ koduje wejście, dla wartości i z zakresu $0 \leq i < gene_{total}$, gdzie $gene_{total}$ to całkowita liczba genów w genotypie, $func_{total}$ to liczność zbioru funkcji, $nodeterm_j$ jest indeksem obecnie przetwarzanego wierzchołka. Mapowanie to ma postać wiele do jeden - istnieje wiele wartości zmiennoprzecinkowych odpowiadających jednej wartości dyskretnej. Zakresy wartości zdefiniowane są odpowiednio:

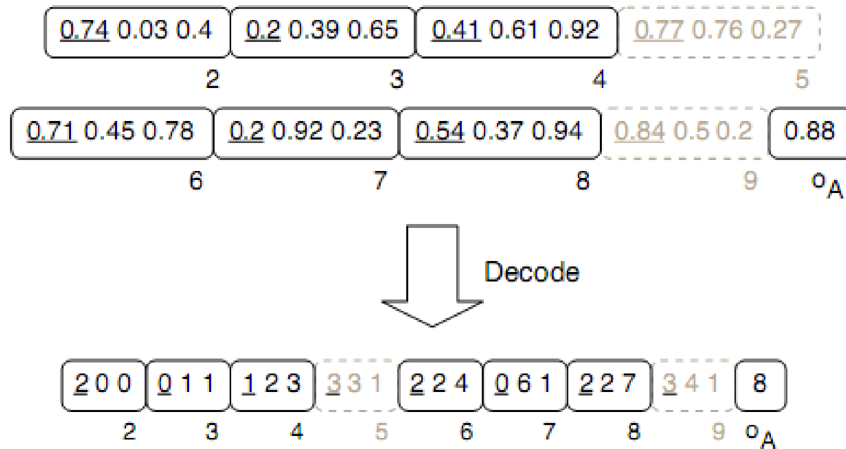
$$func_k \in [\frac{func_k}{func_{total}}, \frac{func_k + 1}{func_{total}}] \tag{5.3}$$

$$input_j \in [\frac{nodeterm_j}{nodeterm_{total}}, \frac{nodeterm_j + 1}{nodeterm_{total}}] \tag{5.4}$$

gdzie w (5.3) $func_k$ to k-ta funkcja ze zbioru funkcji, $func_{total}$ to liczność tego zbioru. W równaniu (5.4) $input_j$ to wejście podłączone do i-tego terminala.

5.5 Operatory genetyczne w CGP

Podstawowym operatorem genetycznym wykorzystywanym dla *CGP* jest *mutacja punktowa*. W przypadku całkowitoliczbowej reprezentacji genotypu ważne jest aby nowa wartość została wybrana



Rysunek 5.3: Faza dekodowania zmiennoprzecinkowej reprezentacji genotypu.

z odpowiedniego zakresu na przykład nie powodowała powstawania cykli w grafie. Dla reprezentacji zmiennoprzecinkowej możliwe jest dowolne dobranie wartości z zakresu $[0; 1]$, ponieważ w fazie fenotypowania jest ona przekształcana na odpowiednią funkcję lub odpowiedni, poprawny, adres wierzchołka. Często dla *CGP* wykorzystuje się podejście *Strategii Ewolucyjnej* (ang. *Evolutionary Strategy*) z wykorzystaniem mutacji punktowej.

Istnieją udane próby adaptacji operatorów krzyżowania w kartezjańskim programowaniu genetycznym, dla niektórych klas problemów. Przyjmując opisaną wcześniej reprezentację zmiennoprzecinkową 5.4 możliwe jest zastosowanie operatora krzyżowania geometrycznego wykorzystywanego w algorytmach genetycznych w przypadku kodowania liczbami zmiennoprzecinkowymi. W pierwszej fazie następuje wybór dwóch rodziców p_1 i p_2, następnie tworzeni są dwaj potomkowie o_1 i o_2 zgodnie ze wzorem (5.5), gdzie $0 < r_i < 1$ dla $i = 1, 2$

$$o_i = (1 - r_i) * p_1 + r_i * p2 \tag{5.5}$$

Badania wykazały, że zastosowanie wyżej opisanego operatora krzyżowania ma duży wpływ na zbieżność algorytmu dla problemów regresji [CWM07].

Rozdział 6

Przegląd podejść do ewolucji sterowników pojazdów

Ze względu na rosnącą popularność zawodów takich jak *SCRC*, pojawia się coraz więcej prac naukowych związanych z uczeniem sterowników pojazdów. Zawody takie stanowią kompromis pomiędzy teoretycznymi badaniami a wdrażaniem gotowych rozwiązań nastawionych na osiąganie konkretnych celów. W roku 2008, podczas konferencji *IEEE WCCI 2008* odbyły się zawody, których celem była konfrontacja sterowników zaprojektowanych lub nauczonych przez uczestników. Architektura środowiska testującego oraz interfejs komunikacyjny tych zawodów zostały praktycznie w niezmienionej formie wykorzystane w *SCRC*. W przeglądzie istniejących podejść do ewolucji sterowników uwzględnione zostaną zgłoszenia z opisanych zawodów oraz prace dotyczące koewolucji sterowników pojazdów.

6.1 Zgłoszenia WCCI 2008 SCRC

Na zawody w 2008 roku zgłoszono pięć sterowników. Prezentowały one różne podejścia do problemu sterowania poczynając od rozwiązań całkowicie napisanych przez programistów, poprzez ewolucję zbioru reguł decyzyjnych a na sztucznych sieciach neuronowych kończąc [LTL+08].

Idea rozwiązania zgłoszonego przez Leonarda Kinnaird-Heether i Roberta Reynoldsa polegała na dostosowaniu limitów prędkości dla pojazdu przy pokonywaniu zakrętów o różnych kątach. Aby uzyskać wspomniane ograniczenia prędkości skorzystano z *Algorytmu Kulturowego* (ang. *Cultural Algorithm*). Pomysł na zastosowanie tego podejścia wywodzi się z obserwacji, iż kierowanie pojazdem jest czynnością społeczną. Celem eksperymentu było nauczenie sterownika zasad jazdy spójnych z zachowaniem agenta poruszającego się w grupie: zachowanie orientacji zgodnej z grupą i odpowiedniej odległości od sąsiednich agentów. Kryterium optymalizacyjnym było zachowanie jak największej prędkości podczas pokonywania zakrętów. Wydzielono 10 kategorii zakrętów od specyficznego przypadku czyli jazdy po prostym odcinku aż do zakrętu o kącie 90°. Samo sterownie odbywało się w oparciu o zdefiniowane przez programistów reguły sterujące, które realizowały dwa zadania. Sterownik naprowadzał pojazd na środek toru i utrzymywał jego kierunek możliwie zgodny z osią toru oraz dostosowywał prędkość do aktualnie obowiązującego limitu.

Inne podejście zostało zaproponowane przez Matta Simmersona, który podjął próbę rozwiązania problemu sterowania przy pomocy *Sztucznych Sieci Neuronowych* (ang. *Artificial Neural Network*) wykorzystując algorytm *NEAT* i jego implementację *NEAT4j*. Algorytm ten utrzymuje populację osobników, z których każdy reprezentuje konfigurację sieci neuronowej. Sieci konstruowano w ten sposób, że spośród 29 dostępnych wejść ustalonych przez programistę, wybierane

były 3 wartości stanowiące warstwę wejściową. Wybrano następujące sensory stanowiące możliwe wejścia układu:

- aktualna prędkość – 2 wartości
- kąt pomiędzy osią toru a pojazdem
- sensory wykrywające krawędzie toru – 19 wartości
- horyzontalna pozycja na torze
- wybrany bieg
- prędkości obrotowe poszczególnych kół – 4 wartości
- liczba obrotów silnika na minutę

Układ posiadał trzy wyjścia odpowiadające efektorom: sterowania, zmiany prędkości i zmiany biegu. Wszystkie wartości zostały przeskalowane do zakresu $[-1; 1]$ dla wejść, które przyjmują wartości ujemne oraz $[0; 1]$ dla wejść, które przyjmują tylko wartości dodatnie. Faza ewolucji dla algorytmu *NEAT* polega na wykorzystaniu *Algorytmu Genetycznego* do ewolucji topologii sieci neuronowej. Oceny osobników dokonywano w oparciu o symulację przez 4000 dyskretnych kroków na torze o dużej liczbie zakrętów. Formuła obliczania wartości przystosowania miała następującą postać:

$$f_c = (2 * Dr)do + speedmax + C$$

gdzie,
Dr – odległość pokonana przez sterownik
d – odniesione obrażenia
o – miara tego jak bardzo pojazd pozostawał na torze
$speedmax$ – maksymalna osiągnięta prędkość
C – stała zapewniająca dodatnią wartość całej formuły, arbitralnie ustawiona na 10000
Sterownik uzyskany przy pomocy wyewoluowany w wyniku eksperymentu zajął pierwsze miejsce w zawodach *SCRC 2007*.

Kolejne podejście wykorzystane przez Diego Pereza i Yago Saeza polegało na ewoluowaniu zbioru reguł decyzyjnych sterujących pojazdem [DP08]. Ze zbioru wszystkich wejść wybrano, następujące wartości, które w kolejnym kroku zostały zdyskretyzowane:

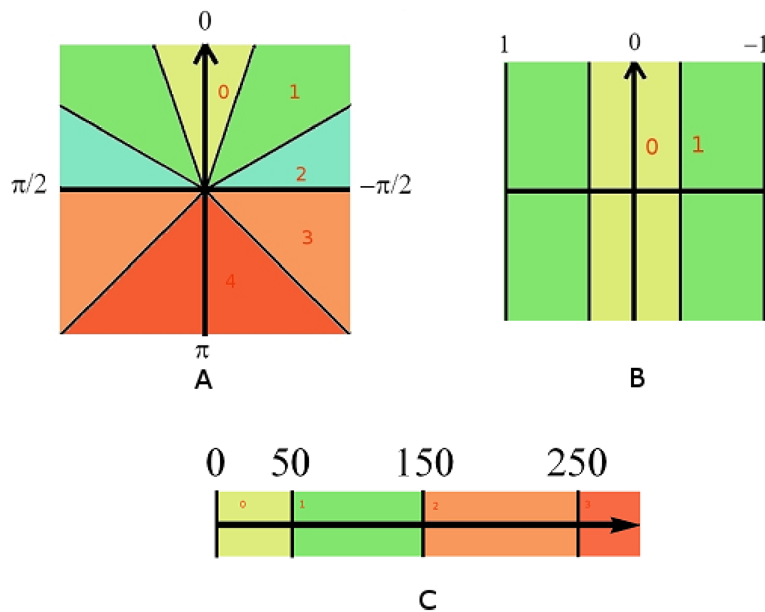
- angle – kąt pomiędzy osią toru a pojazdem,
- trackPos – pozycja pojazdu względem środka toru,
- speedX – prędkość pojazdu,
- track – wektor trzech kolejnych sensorów, z których środkowy wskazuje na wprost.

Metoda dyskretyzacji poszczególnych wartości została przedstawiona na rysunku 6.1. Ograniczono przestrzeń przeszukiwań poprzez wykorzystanie symetrii – na wejścia sterownika podawano wartości bezwzględne i w zależności od znaku poszczególnych wejść wartości wyjść były negowane lub nie. Efektorami sterującymi pojazdem były dwie wartości: *throttle_and_brake*, czyli sterowanie zmianą prędkości zakodowane na jednej zmiennej, *steer* zmienna sterująca zmianą kierunku pojazdu. Populacja składała się ze 120 osobników, z których każdy reprezentował regułę decyzyjną o innej przesłance. Zamiast losowego generowania bazowej populacji zdecydowano o odpowiednim

doborze reguł, które pozwalały pojazdowi na ukończenie okrążenia. Funkcja oceny osobnika miała następującą postać:

$$f_{ind} = 0.4 * lap_time + 0.6 * damage$$

gdzie, *lap_time* to pomierzony czas przejazdu okrążenia a *damage* to suma uszkodzeń podczas wyścigu. W eksperymencie zastosowano operator krzyżowania z losowym wyborem rodziców. Zastosowanie operatora mutacji dla wybranej reguły polegało na dodaniu ±1 do warunku z przesłanki i zmiany decyzji sterującej o wartość ±0.3. W przypadku pozytywnej oceny danej reguły, dodawaną ją do zbioru a usuwaną regułę o takiej samej przesłance. W wyniku eksperymentu ewolucyjnego uzyskaną znaczą poprawę czasu pokonywania okrążeń oraz zmniejszono obrażenia odniesione przez pojazd.



Rysunek 6.1: Dyskretyzacja wejść dla *angle* (A), *trackPos* (B), *speedX* (C) [DP08]

6.2 Koewolucja sterowników pojazdów

Opisane poniżej podejście do koewolucji pojazdów zostało opracowane w oparciu o publikację dotyczącą *Ewolucyjnych Wyścigów Pojazdów ECR* (ang. *Evolutionary Car Racing*) opracowaną przez Juliana Togeliusa i Simona M. Lucasa [TL06]. W przeciwieństwie do przypadku gdy eksperyment ewolucyjny dotyczy jazdy indywidualnej, w wariancie koewolucji ocena osobnika zależy również od sterownika przeciwko któremu jest testowany. Podczas porównywania dwóch osobników można zdefiniować bezwzględną miarę przystosowania, na przykład jako całkowity pokonany dystans jak i względną poprzez odległość od konkurencyjnego pojazdu po zadanej liczbie kroków symulacji. Możliwe jest też wprowadzenie funkcji przystosowania jako sumy ważonej powyższych.

Wszystkie eksperymenty przeprowadzono w specjalnie przygotowanym środowisku symulacyjnym naśladującym zachowanie samochodów-zabawek sterowanych radiem. Na torze rozmieszczono

przeszkody i punkty kontrolne ułatwiające ocenę osobników. W zamodelowanej fizyce uwzględniono poślizgi, kolizje, słabą przyczepność pojazdów. Informacjami wejściowymi systemu sterującego były sieci sensorów odległościowych dwojakiego typu: wykrywających ściany oraz wykrywające przeciwników. Do reprezentacji sterownika wykorzystano wielowarstwową sieć neuronową. Podczas ewolucji zmianie ulegały nie tylko wagi połączeń sieci neuronowej ale również rozmieszczenie i typ sensorów pojazdu.

W ewolucji i koewolucji wykorzystano strategię ewolucyjną $(\mu+\lambda)$. Ocena osobników odbywała się na trzech różnych torach w celu uniknięcia nadmiernego przystosowania sterownika do danej trasy. Funkcję oceny łączącą informację o bezwzględnym i względnym przystosowania zdefiniowano następująco:

$$fitness = p * absfit + (1 - p) * relfit$$

gdzie p to wartość parametru ustalona dla danego eksperymentu.

Przeprowadzono szereg eksperymentów, których celem było porównanie sterowników uzyskanych drogą ewolucji i koewolucji. W pierwszej serii eksperymentów ewoluowano sterownik do jazdy indywidualnej na czas, określany dalej jako *solo-controller*. Następnie sprawdzono jak radzi sobie on, w przypadku wyścigu, w którym udział bierze więcej niż jeden pojazd. Wyniki pokazały, że bliski kontakt lub kolizja z przeciwnikiem powodują duże zaburzenia systemu sterującego. Na tej podstawie uznano, że przygotowanie sterownika do wyścigu z przeciwnikami zdecydowanie różni się od problemu jazdy samodzielnej. W kolejnych eksperymentach porównano wyniki jazdy indywidualnej sterowników uzyskanych metodami koewolucji z *solo-controller*. Żaden z uzyskanych sterowników nie był w stanie nawiązać podobnych czasów co sterownik dedykowany do jazdy na czas. Na koniec wykonano eksperyment, w którym ewoluowany sterownik konkurował na torze z *solo-controller* starając się nauczyć skutecznej strategii w walce z konkretnym przeciwnikiem. Nie udało się jednak uzyskać sterowników zdolnych do pokonania pojazdów referencyjnych.

Rozdział 7

Zagadnienia dynamiki pojazdów

7.1 Podstawowe zagadnienia dynamiki

Środowisko symulacyjne *TORCS* zapewnia realizm rozgrywki poprzez wierne odwzorowanie sił fizycznych działających na obiekty. Pojazd jest wyposażony w jednostkę napędową, której przełożenie może ulegać zmianie przy użyciu skrzyni biegów. W przypadku jazdy po łuku ze zbyt duża prędkością gracz doświadcza poślizgu pojazdu. Następstwem niefortunnego sterowania może być zderzenie z bandą ograniczającą tor lub pojazdem przeciwnika, które skutkują uszkodzeniami pojazdu. Modele pojazdów różnią się między sobą profilem aerodynamicznym, parametrami silnika, typem zawieszenia.

Analizując wpływ poszczególnych sił na zachowanie pojazdu można określić charakterystykę optymalnego sterownika, zdefiniować jego zachowanie w sytuacjach drogowych zachodzących podczas wyścigu. Przypadki te dotyczą ruchu prostoliniowego: przyspieszanie i zwalnianie oraz jazdy po łuku. Sytuacje te zostaną opisane w kontekście sił działających na samochód w rzeczywistości. Pewne obliczenia będą operować na uproszczonym modelu co pozwoli na zachować zwięzłość bez większych strat dokładności. Ocena stopnia złożoności poszczególnych manewrów będzie przydatna podczas definiowania eksperymentu ewolucyjnego.

7.2 Ruch po prostej

Ruch po prostej może być analizowany w następujących przypadkach: przyspieszanie, ruch jednostajny, hamowanie. Samochód poruszający się po prostym odcinku toru przyspiesza dzięki działaniu siły napędowej. Parametry ruchu pojazdu można wyznaczyć posługując się równaniem równowagi sił działających na pojazd. Pojazd jest wprawiany w ruch siłą napędową przenoszoną od silnika do kół poprzez układ napędowy. Siła ta musi być na tyle duża, żeby nie tylko poruszać pojazd, ale również przeciwdziałać oporom ruchu, którymi są: opór toczenia, opór powietrza, opór bezwładności masy pojazdu. W dalszych obliczeniach uwzględniona zostanie siła oporu toczenia i siła oporu powietrza jako główne siły hamujące pojazd. Pominięte zostaną też przypadki gdy pojazd porusza się pokonując wzniesienie oraz gdy na danym odcinku drogi krawędzie jezdni znajdują się na różnych wysokościach (ang. *banking*). Wypadkowa sił działających na pojazd w ruchu prostoliniowym ma postać:

$$F_{podluzna} = F_{napedowa} + F_{op_powietrza} + F_{op_toczenia}$$

Opór toczenia, nazywany inaczej oporem podstawowym, wywoływany jest głównie ciągłym uginaniem i rozprężaniem fragmentu opony stykającej się w danym momencie z jezdnią. W przypadku jazdy po nieutwardzonej nawierzchni, wzrasta jego wartość ze względu na efekt uginania podłoża

i tarcia o boki opony. Decydujący wpływ na wartość oporu toczenia mają deformacje opony, które zależą od ciśnienia powietrza w kole, promienia koła, składu mieszanki gumowej opony oraz prędkość jazdy. Opór ruchu powodowany tarciem w łożyskach kół stanowi niewielką część oporu toczenia i jest w jego obliczeniach pomijany [GST09]. Opór toczenia można wyznaczyć ze wzoru:

$$F_{opr_toczenia} = Q_s g f_r$$

Q_s – aktualna masa pojazdu $[kg]$
g – przyspieszenie ziemskie $[m/s^2]$
f_r – współczynnik oporu toczenia, który dla przeciętnych wielkości ogumionych kół i w zależności od rodzaju nawierzchni przyjmuje wartości $[0.01; 0.30]$ dla podłoża nawierzchni asfaltowej do luźnego piasku.

Opór powietrza wywoływany jest niesymetrycznym i turbulentnym opływem powietrza wokół nieregularnego kształtu nadwozia. Na opór powietrza wpływ ma wielkość powierzchni czołowej pojazdu, kształt zewnętrzny karoserii, gęstość i ciśnienie powietrza i prędkość względna powietrza. Samochody konstruowane są w taki sposób, że opór dolnej i górnej części nadwozia jest różny. Jest on przyczyną wytwarzania się w czasie jazdy różnicy ciśnień pomiędzy górą i spodem samochodu, która powoduje jego odciążenie. Siła odciążająca jest proporcjonalna do prędkości pojazdu i odgrywa znaczącą rolę przy prędkościach większych od $150[km/h]$. Przy uproszczonych obliczeniach może ona zostać pominięta. Pomijając efekt odciążenia aerodynamicznego pojazdu opór powietrza oblicza się ze wzoru:

$$F_{opr_powietrza} = c_x A v_w^2 \frac{\rho}{2}$$

gdzie:
c_x – współczynnik oporu powietrza. Wartość ta jest określana przez przeprowadzanie testu dla samochodu w tunelu aerodynamicznym. Dla samochodów sportowych wartość ta nie powinna być większa niż 0.3.
A – pole powierzchni czołowej pojazdu wyrażone w $[m^2]$. Przykładowe wartości zostały przedstawione w tabeli.
v_x – względna prędkość naporu powietrza (względna prędkość pomiędzy pojazdem i powietrzem z uwzględnieniem siły i kierunku wiatru). Wartość wyrażona w $[m^2/s]$

Klasa pojazdu	c_x	A
Samochody sportowe	$[0.18; 0.25]$	$[1.20; 1.50]$
Małe samochody osobowe	$[0.30; 0.36]$	$[1.70; 1.90]$
Inne samochody osobowe	$[0.28; 0.36]$	$[1.90; 2.20]$
Samochody ciężarowe	$[0.80; 1.00]$	$[4.00; 7.00]$

Tablica 7.1: Przykładowe wartości współczynnika oporu powietrza i pola powierzchni czołowej dla wybranych klas pojazdów [GST09].

Kolejnym ważnym zjawiskiem występującym przy przyspieszaniu i hamowaniu jest dynamiczne przeniesienie ciężaru. W trakcie jazdy można ten efekt zaobserwować poprzez pochylanie obrysu samochodu do przodu lub do tyłu. W dalszych obliczeniach przyjęto, że samochód jest napędzany na tylną oś. W trakcie przyspieszania i hamowania zmienia się ciężar przypadający na poszczególne osie. Zmiana ciężaru powoduje zmianę oporu toczenia przypadającą na daną oś. Gdy samochód stoi w miejscu rozkład ciężaru na osie jest proporcjonalny do odległości poszczególnych osi względem środka ciężkości:

$$l = c + b$$

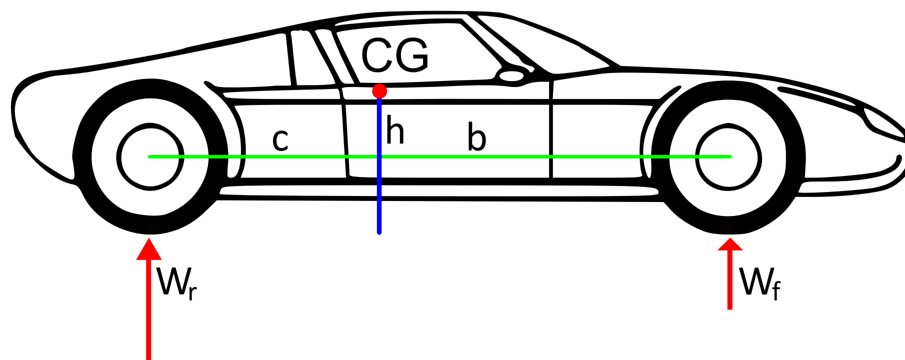
$$W_r = \frac{b}{l}W$$
$$W_f = \frac{c}{l}W$$

W przypadku gdy samochód porusza się ruchem jednostajnie przyspieszonym z wartością przyspieszenia równą a wartości rozkładu ciężaru pojazdu ulega następującym zmianom:

$$W_r = \frac{c}{l}W - \frac{h}{l}ma$$
$$W_f = \frac{b}{l}W + \frac{h}{l}ma$$

gdzie:
W_r – ciężar tylnej osi pojazdu
W_f – ciężar przedniej osi pojazdu
a – wartość przyspieszenia
m – całkowita masa pojazdu
h – wysokość środka ciężkości

Konsekwencją doświadczaną przez kierowców w przypadku zbyt dużego przeniesienia ciężaru na tylną oś jest podsterowność. Podsterowność można zdefiniować jako sytuację gdy podczas skręcania w wyniku utraty przyczepności przez przednie koła ruch kierownicy nie powoduje właściwego skrętu kół - następuje poślizg przednich kół. Odwrotną sytuacją jest nadsterowność, która występuje gdy tylne koła samochodu tracą przyczepność w zakręcie i tył samochodu jest wynoszony na zewnątrz zakrętu. Ze względu na zależność pomiędzy wartością przeniesienia ciężaru a wysokością środka ciężkości samochody sportowe projektowane są w ten sposób aby środek ten znajdował się jak najniżej.



Rysunek 7.1: Siły działające na poszczególne osie

7.3 Ruch krzywoliniowy

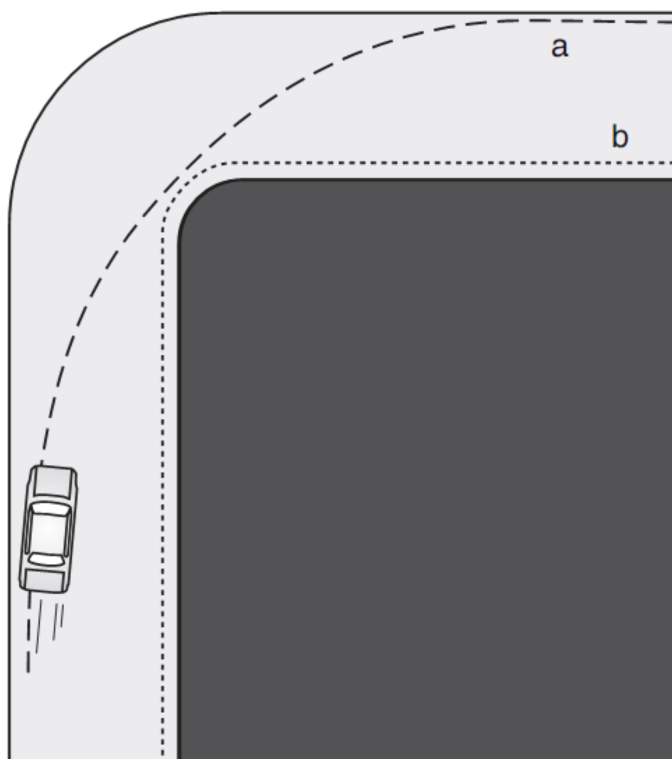
Najbardziej newralgicznym manewrem wykonywanym podczas wyścigu jest pokonywanie zakrętów z dużą prędkością. Ze względu na działanie siły odśrodkowej pojazd jest znoszony w stronę zewnętrznej osi toru. Przy pokonywaniu zakrętów z dużą prędkością następuje znoszenie pojazdu, kierunek ruchu pojazdu jest inny niż wynikałoby to z ustawienia kół względem pojazdu. Dla kół na przedniej i tylnej osi pojazdu można zdefiniować kąty poślizgu α jako kąt pomiędzy kierunkiem kół a kierunkiem pojazdu. Ich wartość zależy od poślizgu bocznego pojazdu, odchylenia pojazdu względem osi pionowej. Dodatkowo dla przedniej osi wpływ na kąt poślizgu ma ustawienie kół względem pojazdu.

7.4 Techniki kierowania pojazdem

Istnieje szereg technik prowadzenia pojazdem pozwalających na zachowanie lepszej sterowności pojazdu lub szybsze pokonywanie zakrętów. Pierwszą dobrą praktyką zwiększającą skuteczność hamowania jest hamowanie pulsacyjne. W odróżnieniu od sytuacji gdy hamowanie jest wykonywane poprzez maksymalne wciśnięcie pedału hamulca, hamowanie pulsacyjne polega na odpowiednim wciskaniu i puszczaniu pedału hamulca. Metoda ta zmniejsza ryzyko zablokowaniu kół, hamowanie odbywa się przed zerwaniem przyczepności kół. Skutkuje to zwiększeniem efektywności hamowania i skróceniem drogi hamowania.

Kolejną techniką umożliwiającą zyskanie kilku sekund podczas wyścigu jest pokonywanie zakrętów po odpowiednim torze. Celem tej techniki jest uzyskanie jak największej prędkości wyjścia z zakrętu. Przedstawiony na rysunku 7.2 tor ruchu *a* przedstawia jazdę po maksymalnym promieniu krzywizny zakrętu. Dzięki szerokiemu wejściu w zakręt i ścięciu w stronę bandy przy jego szczycie możliwe jest wczesne rozpoczęcie przyspieszania.

Efektywne ruszanie pozwala zyskać przewagę nad rywalami na starcie wyścigu. Dodatkowo w przypadku wyścigu na skomplikowanym torze o dużej liczbie zakrętów pojazd użytkownika będzie zmuszony kilkukrotnie powtarzać ten manewr po wypadnięciu z trasy. Klasyczna technika ruszania polega na maksymalnym wciśnięciu pedału gazu. Nie jest to technika optymalna, ponieważ może spowodować utratę przyczepności kół co obniża efektywność ruszania. Podstawą szybkiego ruszania jest skierowanie kół zgodnie z kierunkiem jazdy. Wyprostowane koła mają większą przyczepność i pozwalają na mocniejsze dodanie gazu przy ruszaniu. Nie bez znaczenia jest także sterowanie pedałem gazu - przyciśnięcie go do poziomu, dla którego uzyskuje się maksymalne przyspieszenie bez utraty przyczepności kół.



Rysunek 7.2: Przykładowe tory jazdy podczas pokonywania zakrętów [Par03]

Rozdział 8

Opis przeprowadzony eksperymentów

8.1 Programy i rozszerzenia CGP

We wszystkich eksperymentach przyjęto, że program CGP ma dwa wyjścia – efektory sterujące zmianą prędkości pojazdu i kierunkiem jazdy. Liczba wejść była zależna od przyjętego modelu. W zdecydowanej większości eksperymentów wykorzystano następujący zbiór funkcji:

- **F_ADD** – dwuargumentowa suma
- **F_SUB** – dwuargumentowa różnica
- **F_MUL** – dwuargumentowy iloczyn
- **F_DIV** – dwuargumentowe bezpieczny iloraz
- **F_MAX** – maksimum z dwóch argumentów
- **F_MIN** – minimum z dwóch argumentów
- **F_COS** – jednoargumentowa funkcja trygonometryczna cosinus
- **F_TANH** – jednoargumentowa funkcja tangensa hiperbolicznego
- **F_ABS** – wartość bezwzględna z argumentu
- **F_IF** – czteroargumentowa warunek logiczny postaci $if \quad (a > b) \quad then \quad c \quad else \quad d$

W celu uzyskania lepszych wyników w eksperymentach wprowadzono kilka rozszerzeń do implementacji *CGP* z biblioteki *ECJ*. Po pierwsze dodano znane z programowania genetycznego stałe ulotne (ang. *ephemeral constants*), kodowane w genotypie i interpretowane przez program jako wejścia. Przyjmowały one wartości z zakresu $[0; 1]$ i ulegały modyfikacjom w czasie ewolucji. Ich liczba była określana parametrem *subindividuals* w pliku konfiguracyjnym. Jako kolejne rozszerzenie wprowadzono reprezentację przez jeden genotyp kilku podprogramów CGP. Dla takiej struktury ostateczną wartość efektorów stanowiła średnia arytmetyczna wyjść podprogramów. Rozwiązanie to mogło ograniczyć problem nieciągłości sterowania, występujący dla brzegowych wartości wejść wykorzystanych w węzłach reprezentujących testy logiczne. Podejście to umożliwiło zastosowanie nowego operatora krzyżowania dla genotypów, a mianowicie wymiany podprogramów. Krzyżowanie to miało postać analogiczną do krzyżowania jednopunktowego z tą różnicą że zamiast wymiany poszczególnych alleli wymieniane były całe fragmenty reprezentujące podprogramy. We wszystkich przeprowadzonych eksperymentach wykorzystano najogólniejszy wariant kartezjańskiego programowania genetycznego, w którym każdy węzeł leży w innej warstwie. Zgodnie z definicją oznacza

```
1 public interface ISensorModelPreprocessor {
2
3         public EvoSensorModel process(SensorModel sensorModel);
4
5         public boolean isInverseSteering();
6
7         public boolean isSymmetric();
8
9 }
```

Listing 8.1: Interface przetwarzania danych z sensorów

to, że parametry wymiarów kraty przyjmują wartości: $r = 1$ $c = nodes_total$. Ze względu na eksperymenty z wykorzystaniem operatorów krzyżowania wybrano zmiennoprzecinkową reprezentację genotypu.

8.2 Przetwarzanie danych z sensorów

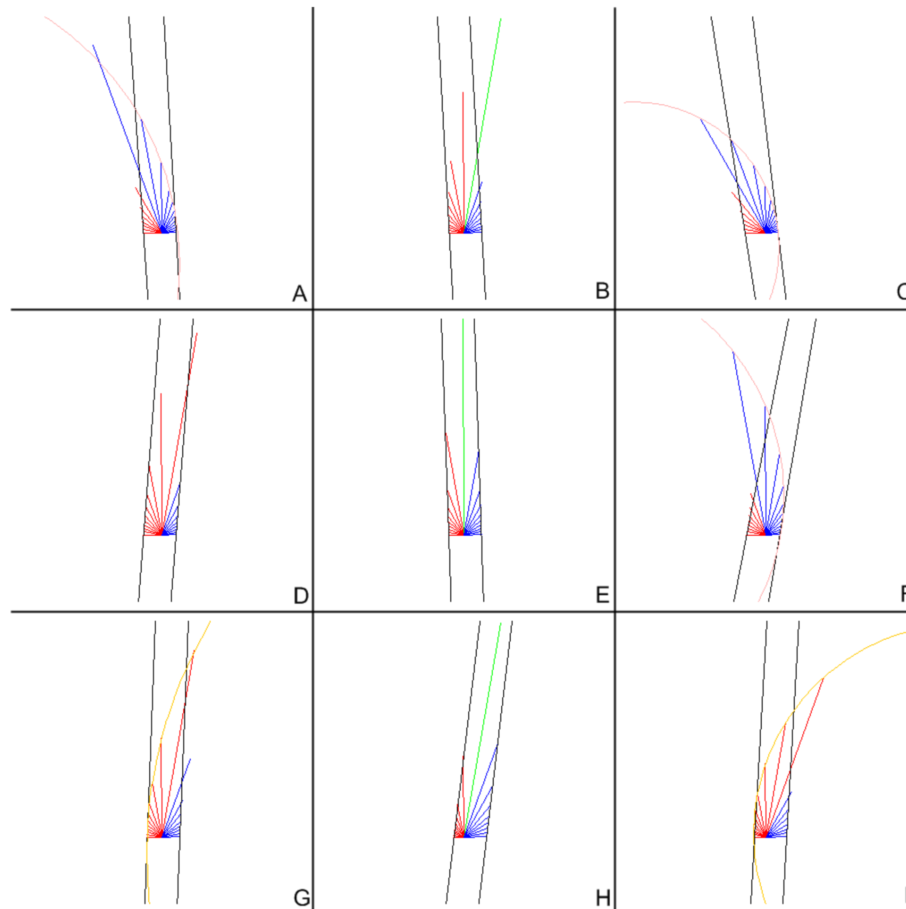
Podstawowy zestaw wejść reprezentujący stan obiektu zawiera około 60 wartości skalarnych. Tak duża przestrzeń możliwych wejść powoduje, że nauka sterownika byłaby bardzo trudna. Dokonując pobieżnej analizy danych odbieranych od środowiska dotyczących pojazdu wydzielono grupę, która ma największy wpływ na efektywne sterowania pojazdem. Do grupy tej można zaklasyfikować informacje o aktualnej prędkości, pozycji względem toru i charakterystyce pokonywanego fragmentu toru. Dlatego zadecydowano o wprowadzeniu pewnych uproszczonych modeli informacji o stanie sterownika. Zredukowanie o rząd wielkości liczby wejść stanowi duże uproszczenie w procesie ewolucji. Wymagania funkcjonale dla klas odpowiedzialnych za wstępne przetwarzanie wejść zdefiniowano poprzez interface *ISensorModelPreprocessor* przedstawiony na listingu 8.1. Oprócz metody mapującej oryginalny model opisu stanu obiektu zawiera on dodatkowo predykaty określające czy dany system przetwarzania wykorzystuje symetrię i czy ostatnio przekształcony model wymaga symetrycznego odbicia wartości sterujących. Wyodrębnienie kontraktu funkcjonalnego umożliwiło wybór metody wstępnego przetwarzania z poziomu pliku konfiguracyjnego eksperymenty bez konieczności ponownej re-kompilacji całego programu. Większość eksperymentów wykorzystywała jeden z dwóch zaproponowanych metod przedstawienia stanu pojazdu.

Pierwszy z użytych modeli przetwarzania wstępnego o nazwie *SymTurnPreprocessor* analizuje informację z sensorów wykrywających krawędzie toru i dokonuje detekcji zakrętów. Zakręt opisany jest poprzez kąt oraz promień krzywizny. Za wykrywanie parametrów zakrętów odpowiada klasa *TurnDetector*. Dokonuje ona klasyfikacji wszystkich odczytów z detektorów typu *track* do jednej z klas: prosty odcinek toru po lewej stronie, prosty odcinek toru po prawej stronie, zewnętrzna krawędź zakrętu. Przykładowe działanie klasyfikacji zostało przedstawione na rysunku 8.1. Detektor działa kilkuetapowo. W pierwszym kroku rozpoczynając od skrajnych punktów po obydwu stronach wyznacza równania prostych opisujących prosty fragmenty toru. Tak długo jak błąd dopasowania kolejnych punktów do prostych jest mniejsze od zadanego progu traktowane są one jako pomiary opisujące prosty fragment. Pozostałe niedopasowane odczyty dzielone są względem największej różnicy pomiędzy parą kolejnych wartości na przypisane do lewej i prawej strony. Dla każdej z grup następuje próba wyznaczenia równania okręgu opisanego na tych punktach. Może się ona zakończyć sukcesem tylko dla jednej z grup, ponieważ możliwe jest otrzymanie z sensorów informacji opisującej tylko zewnętrzną krawędź zakrętu. W przypadku gdy nie ma wystarczającej liczby punktów tworzących zakręt detektor uznaje, że w obecnym stanie pojazd znajduje się na prostym odcinku drogi. Ostrość zakrętu wyznaczana jest na podstawie długości łuku pomiędzy

skrajnymi punktami pomiarów sensorów leżących na okręgu. Jako parametr do modelu podawana jest odwrotność promienia opisanego okręgu czyli $\frac{1}{r}$. Wprowadzone do modelu zmienne związane z wykrywaniem zakrętów to: odwrotność krzywizny zakrętu (oznaczenie *curv_4*) i kąt skrętu (oznaczenie *curva_5*). Dodatkowo w modelu wykorzystano następujące wejścia:

- *ahead_sensor* – odczyt sensora wskazującego na wprost (oznaczenie *as_1*).
- *angle_to_track_axis* – kierunek względem osi toru (oznaczenie *atta_2*).
- *track_pos* – położenie względem osi toru (oznaczenie *tp_3*).
- *speed* – składowa prędkości równoległa do osi pojazdu (oznaczenie *sp_6*).
- *l_speed* – składowa prędkości prostopadła do osi pojazdu (oznaczenie *lsp_7*).

Wszystkie wartości wejściowe przeskalowano do przedziału $[-\pi;\pi]$ dla zmiennych, których dziedzina obejmuje wartości ujemne lub do przedziału $[0;\pi]$ w przeciwnym wypadku. Przetwarzanie symetryczne oznacza, że każdy przypadek kombinacji wejść jest rozpatrywany jako skręt w prawo, lecz w przypadku gdy w rzeczywistości jest to zakręt w drugą stronę wyjście systemu sterującego jest negowane.



Rysunek 8.1: Przykładowe działanie klasy wykrywającej zakręty. Na czerwono oznaczono odczyty z sensorów zaklasyfikowane jako wykrywające lewą krawędź toru, na niebiesko prawą krawędź toru. W przypadku gdy detektor uznał, że przed pojazdem znajduje się zakręt oznaczono dopasowanie zewnętrznej krawędzi do okręgu.

Drugi testowany model był asymetryczny – na wejście sterownika przekazywano wartości w niezmienionej postaci. W przeciwieństwie do poprzedniej reprezentacji model ten ograniczał się

wyłącznie do wyboru spośród wartości oryginalnego wektora wejściowego. Podstawę informacji o torze stanowiły odczyty z trzech sensorów odległościowych wskazujących na wprost, 20° na lewo (oznaczenie *l20_4*) i 20° na prawo (oznaczenie *r20_5*).

Dla potrzeb koewolucji zastosowano rozszerzony model obejmujący dodatkowo wartości wybrane z wektora wartości z sensorów detektora przeciwników. Ze względu na brak jednoznaczności co do przyjętego odniesienia postanowiono zrezygnować z wykorzystania modelu symetrycznego w przypadku jazdy z przeciwnikiem.

8.3 Ewaluacja osobników

Cały eksperyment wykonywany jest na jednej instancji środowiska *TORCS*. W trakcie eksperymentu trasa wyścigu nie ulega zmianie a sam wyścig nie jest powtarzany od nowa. Przełączniki uruchamiania symulatora z linii poleceń umożliwiają wyłączenie limitu paliwa i limitu uszkodzeń co w połączeniu z ustawieniem liczby okrążeń na maksymalnie duża wartość zapewnia, że wyścig nie zakończy się przedwcześnie. Protokół serwera *torcs2009server* wymaga aby cały czas, w trakcie trwania wyścigu, utrzymywane było połączenie pomiędzy klientem a serwerem. Z tego względu zaniechano podejścia, w którym każdy wyewoluowany sterownik jest samodzielną jednostką nawiązującą połączenie z serwerem na czas ewaluacji. Zastąpiono je wprowadzeniem sterownika *EvoMetaController* będącego jednostką odpowiedzialną za komunikację. Zapewnia on ciągłość sterowania pojazdem w środowisku *TORCS* niezależnie od aktualnie realizowanego zadania. Sterownik *EvoMetaController* może znajdować się w jednym z trzech stanów:

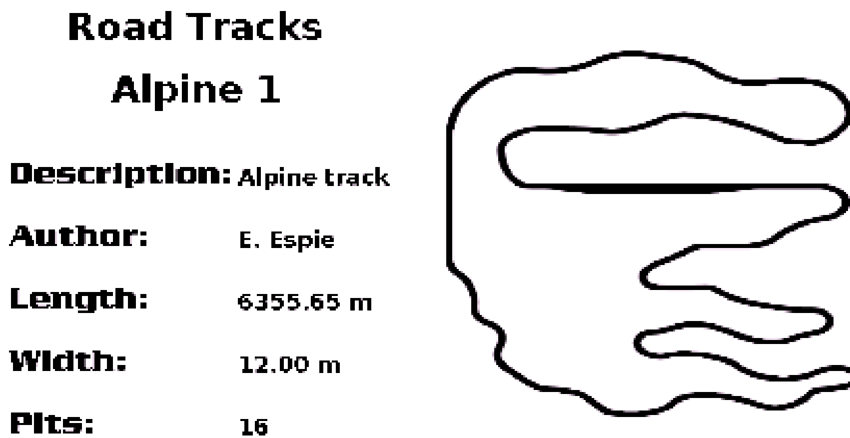
- **EVALUATE** – w trakcie oceny sterownika
- **RESCUE** – po wykonaniu oceny osobnika, pojazd jest doprowadzany do stanu, w którym możliwe jest rozpoczęcie oceny kolejnych osobników
- **IDLE** – pojazd znajduje się w stanie startowym i oczekuje na kolejne zgłoszenia

Sterownik *EvoMetaController* działa w osobnym wątku a ocena osobników odbywa się asynchronicznie. Kolejno zgłaszane sterowniki do oceny są dodawane do kolejki oczekujących. Po zakończeniu ewaluacji osobnika wątek, który zgłosił zadanie jest informowany poprzez mechanizm synchronizacji *wait()* i *notify()*. Rolą sterownika aktywnego w trybie RESCUE jest ustawienie pojazdu w określonej odległości od osi toru i skierowanego zgodnie z kierunkiem toru. W przypadku gdy ostatni oceniany sterownik spowoduje, że pojazd znajdzie się poza torem jego zadaniem w pierwszym etapie jest zapewnienie aby wrócił on na tor. Przy małych populacjach, szczególnie w początkowych generacjach mogłaby nastąpić sytuacja, że kolejne pokolenia są oceniane na różnych fragmentach toru o różnym stopniu trudności. Aby tego uniknąć pojazd w trybie sterowania RESCUE musi pokonać pewną ustaloną odległość zanim zgłosi gotowość oceny kolejnego osobnika.

Istnieją dwa przypadki kiedy ewaluacja sterownika zostanie przerwana przed upływem liczby kroków przewidzianej na jego ocenę. Po pierwsze w sytuacji gdy pojazd wyjedzie poza krawędź toru a kąt pomiędzy kierunkiem jazdy a osią toru będzie większy niż zadany próg $\alpha_{threshold}$. Drugi przypadek to gdy prędkość pojazdu będzie w danym oknie czasowym mniejsza niż przyjęty próg v_{min}. Warunki te w znaczny sposób przyspieszają proces ewaluacji w początkowych generacjach.

Istotnym problemem pojawiającym się przy ewaluacji jest zaszumienie (ang. *noise*) oceny osobników – ewaluacja sterowników na różnych odcinkach toru powoduje przypisanie im różnej wartości funkcji przystosowania. Jednym ze sposobów zmniejszenia tych wahań jest ocena na podstawie trudności przejechanego odcinka a nie bezpośrednio jego długości. Wprowadzono też możliwość oceny osobnika jako średniej z kilku ewaluacji. Konsekwencją ograniczenia szumu w taki sposób

jest wzrost narzutu obliczeniowego dla jednego osobnika, co niekorzystnie odbija się na wynikach eksperymentów.



Rysunek 8.2: Trasa *Alpine1* wykorzystana w części eksperymentów.

8.4 Inicjalizacja

W eksperymencie ewolucyjnym zmiennoprzecinkowy genotyp reprezentujący sterownik był inicjalizowany losowym ciągiem liczb z zakresu $[0; 1]$. Przy koewolucji eksperyment rozpoczęto wykorzystując genotypy uzyskane w eksperymencie ewolucyjnym. Ponieważ informacja o wejściach nie jest kodowana w genotypie możliwe było wykorzystanie programów uzyskanych dla mniejszej liczby wejść czyli w eksperymencie ewolucyjnym gdzie nie uwzględniono informacji o przeciwnikach.

8.5 Zastosowane operatory genetyczne

W przeprowadzonych eksperymentach testowano różne operatory genetycznych. Zmianie ulegały typy operatorów i prawdopodobieństwo ich wystąpienia. Podstawowym stosowanym operatorem była mutacja punktowa powszechnie stosowana w CGP. Wykonano również szereg eksperymentów oceniających wpływ zastosowania krzyżowania na zbieżność algorytmu. Zastosowano krzyżowanie opisane w rozdziale 5 dla zmiennoprzecinkowej reprezentacji genotypu. W oparciu o zdefiniowane rozszerzenia CGP o reprezentację w genotypie kilku podprogramów zaproponowano nowy operator krzyżowania podobny do krzyżowania jednopunktowego z tą różnicą, że wymianie podlegały całe podprogramy.

8.6 Fenotypowanie

W CGP genotyp jest zamieniany na acykliczny graf skierowany, w którego wierzchołkach znajdują się funkcje a krawędzie oznaczają przepisanie wyjścia na wejście. W implementacji rozszerzenia do biblioteki *ECJ* jako postać wyjściowa działania klasy *ec.cgp.Evaluator* odpowiedzialnej za fenotypowanie otrzymywane są wartości wyjść dla zadanych wejść oraz kod programu dla każdej zmiennej wyjściowej w odwrotnej notacji polskiej *ONP*. Aby powstały kod można było osadzić w klasie sterownika bez konieczności przy każdym odpytywaniu o wartości sterujące wykonywa-

```
1  public Action control(EvoSensorModel sensorModel) {
2                  double as_1 = (double) sensorModel.getAheadSensorNormalized();
3                  double atta_2 = (double) sensorModel.getAngleToTrackAxisNormalized();
4                  double tp_3 = (double) sensorModel.getTrackPosNormalized();
5                  double l20_4 = (double) sensorModel.getL20SensorNormalized();
6                  double r20_5 = (double) sensorModel.getR20SensorNormalized();
7                  double sp_6 = (double) sensorModel.getSpeedNormalized();
8                  double lsp_7 = (double) sensorModel.getLateralSpeedNormalized();
9
10                 Action a = new Action();
11
12                 double[] _cvals0 = { 0.50767624, 0.39011067 };
13
14                 /* subtree0 */
15                 double steering0 = (Math.min(atta_2, tp_3) * tp_3);
16                 double accel_brake0 = (((((atta_2 / Math.min(Math.min(atta_2, tp_3),
17                                 _cvals0[1])) - atta_2) - Math.max(_cvals0[1], _cvals0[1])) + atta_2) +
18                                 Math.min(as_1, tp_3));
19
20                 double steering = (steering0) / 1;
21                 double accel_brake = (accel_brake0) / 1;
22
23                 steering = normalize(steering, -1.0, 1.0, true);
24                 accel_brake = normalize(steering, -1.0, 1.0, true);
25
26                 if (_inverse)
27                         a.steering = -steering;
28                 else
29                         a.steering = steering;
30
31                 if (accel_brake > 0.0)
32                         a.accelerate = accel_brake;
33                 else
34                         a.brake = -accel_brake;
35
36                 return a;
37         }
```

Listing 8.2: Kod metody sterującej uzyskany na podstawie genotypu osobnika.

nia żmudnych obliczeń należało zamienić tą postać na instrukcję języka *Java*. Poprzez modyfikację sposobu przetwarzania grafu uzyskano reprezentację, którą łatwo można osadzić w kodzie klasy sterownika. Przykładową funkcję sterującą utworzoną automatycznie na podstawie genotypu przedstawia listing 8.2. Kod najlepszych osobników w danym eksperymencie zamieszczono w przygotowanych szablonach i zapisano jako kompletne klasy języka *Java*. Po skompilowaniu w łatwy sposób można wizualnie ocenić ich działanie w środowisku uruchomionym w trybie podglądu.

8.7 Funkcja oceny

We wszystkich eksperymentach ewolucyjnych funkcja oceny stanowiła sumę ważoną trzech składników:

- *damage* – uszkodzenia pojazdu
- *dist* – ocena przejechanego odcinka
- *ticks* – liczba kroków symulacji wykorzystana przez sterownik

Waga każdej miary mogła być ustawiona z poziomu pliku konfiguracyjnego. Wagi odpowiadały trzem podstawowym miarom oceny przystosowania sterownika: długości przejechanej trasy, liczbie kroków symulacji wykorzystanej przez sterownik, sumarycznym odniesionym obrażeniom. Dwa

pierwsze kryteria są typu zysk, odniesione obrażenia to kryterium typu starta. Parametr *ticks* wprowadzono ponieważ w wielu sytuacjach możliwe jest zakończenie ewaluacji osobnika przed upływem zadanej liczby kroków symulacji co jest przeważnie wynikiem wykonywania niebezpiecznych manewrów na drodze. Składowa oceny pochodząca z przejechanej odległości jest wartością zagregowaną. Reprezentuje ona przejechany dystans mierzony nie w jednostkach długości lecz jako ustaloną funkcję trudności poszczególnych odcinków trasy. Trudność fragmentów danej toru oceniana była na podstawie statystyk wygenerowanych z narzędzia *SafetyCarStatsGen*. Dla wybranego toru przeprowadzało ono symulację przejazdu ustalonej liczby okrążeń przez samochód bezpieczeństwa czyli jeden z predefiniowanych sterowników preferujących zachowawczą jazdę. Na podstawie pomiarów prędkości tego sterownika wyznaczono średnią liczbę kroków symulacji wymaganą do pokonania 10 metrowych odcinków toru. Na tej podstawie dla ocenianego osobnika wyznaczana była wartość funkcji oceniającej trudność pokonanego odcinka.

8.8 Koewolucja

Ważnym aspektem wyścigów jest interakcja z przeciwnikami. W zawodach *SCRC* drugi etap każdego wyścigu polega na współzawodnictwie ze sterownikami innych uczestników. Dlatego ważne było wykształcenie w ewoluowanych sterownikach umiejętności radzenia sobie z napotkanymi pojazdami. W szczególnym przypadku oznacza to nauczenie sterownika wyprzedzania. Wykonano szereg eksperymentów koewolucyjnych, w których ocena osobnika zależała od wyniku porównania z innymi sterownikami.

Zastosowane podejście bazuje na *Competitive Fitness Function*. Utrzymywano jedną populację, której osobniki oceniona na podstawie umiejętności współzawodnictwa z innymi pojazdami. Sterowniki referencyjne do eksperymentów wybierano z wprowadzonego *HoF Hall Of Fame*. Proces oceny wybranego sterownika z populacji polegał na testowaniu go przeciwko kilku losowo wybranym osobnikom z *HoF*. Po każdej generacji do *HoF* trafiał osobnik o najwyższej wartości funkcji przystosowania. W przypadku gdy dodanie nowego osobnika powodowałoby przekroczenie rozmiaru *HoF* usuwano najstarszego osobnika. Dodatkowo zaletą wprowadzenia *HOF* było zapobieganie utraty informacji genetycznej nabytej we wcześniejszych pokoleniach.

Eksperymenty koewolucyjne przeprowadzono w oparciu o metasterownik *CoevoMetaController*. Jego zadaniem było równoległe zarządzanie dwoma sterownikami. Przełączenie w stan początkowy, w którym możliwe było przyjmowanie żądań oceny osobnika, następowało kiedy obydwa sterowniki znajdowały się na trasie a dzieląca je odległość była w przedziale $[dist_{min}; dist_{max}]$. Wtedy sterowanie pojazdem z przodu przekazywane było osobnikowi z *HoF*, natomiast pozycję z tyłu obejmował testowany osobnik. Aby uprościć naukę wyprzedzania wprowadzono globalny parametr określający stosunek liczby wyprzedzeń zakończonych sukcesem do nieudanych prób. W zależności od jego wartości nakładano ograniczenie na maksymalny dopuszczalny bieg, na którym mógł jechać pojazd referencyjny. Analogicznie zdefiniowano parametr określający współczynnik udanych wyprzedzeń poniżej, którego powracano do poprzedniego ograniczenia.

Ocena osobnika w eksperymencie koewolucyjnym składała się z czynnika względnego i bezwzględnego. Poprzez czynnik względny rozumie się zmianę odległości $\delta dist$ pomiędzy sterownikiem testowanym a jego przeciwnikiem z dodatkową premią za udane wyprzedzenie. Czynnik bezwzględny to składowe funkcji oceny zdefiniowane dla eksperymentu ewolucyjnego.

W eksperymentach koewolucyjnych wykorzystano asymetryczny model informacji z sensorów poszerzony o odczyty z trzech wybranych detektorów *opponents* umożliwiających określenie położenia przeciwnika.

```
1 # cgp features
2 pop.subpop.0.species.num-functions = 10
3 pop.subpop.0.species.nodes = 25
4 pop.subpop.0.species.inputs = 7
5 pop.subpop.0.species.constants = 5
6 pop.subpop.0.species.subindividuals = 2
7 pop.subpop.0.species.outputs = 2
8 pop.subpop.0.species.maxArity = 4
```

Listing 8.3: Sekcja konfiguracji odpowiedzialna za ustalenie parametrów CGP.

8.9 Wybór torów

Eksperymenty ewolucyjne przeprowadzono na kilku torach o różnym stopniu trudności. Zdecydowano o wyborze tras o nawierzchni asfaltowej określanych w środowisku *TORCS* jako ang. *RoadTracks*. Preferowano trasy o niezbyt szerokiej jezdni i występującym na poboczu pasie zieleni. Umożliwiło to wykrywanie przypadków kiedy pojazd znalazł się poza torem. Dla części tras bandy ograniczające tor znajdują się dokładnie na krawędzi jezdni co powoduje, że odczyt detektora określającego pozycję względem toru zawsze przyjmuje wartości z prawidłowego zakresu. Przy długich trasach istotne było aby subiektywnie postrzegana trudność na dłuższych fragmentach była podobna. Unikano tras, które składały się z długiej prostej i następującej po niej serii ostrych zakrętów. Przykładową trasę o dużej liczbie ostrych zakrętów prezentuje rysunek 8.2.

8.10 Przeprowadzone eksperymenty

Wykonano serie eksperymentów dla różnych konfiguracji parametrów eksperymentu. Zasadniczo testowane parametry podzielić można na trzy kategorie. Pierwsza to ogólne parametry związane z ewolucją takie jak rozmiar populacji, prawdopodobieństwa występowania poszczególnych operacji genetycznych. Drugi kategoria to parametry związane z CGP określające liczbę wejść, węzłów programu, stałych zapisanych w genotypie, maksymalną arność funkcji. Przykładowe ustawienie tych parametrów przedstawiono na listingu 8.3

Trzeci typ parametrów to te ściśle związane z ewaluacją i działaniem środowiska *TORCS*. Wpływają one na czas potrzebny do wykonania pełnego cyklu ewolucyjnego dla danej populacji poprzez zdefiniowanie liczby kroków symulacji przypadających na jeden sterownik, wagi poszczególnych składowych funkcji oceny, klasę odpowiedzialną za wstępne przetwarzanie informacji z sensorów. Przykładowa konfiguracja tej sekcji została zamieszczona na listing 8.4. W trakcie badań przeprowadzono wiele eksperymentów dla różnych ustawień parametrów. Testowano wpływ rozmiaru populacji na uzyskiwane wyniki. Sprawdzono jaki wpływ na ocenę osobników ma zastosowanie różnych modeli informacji z sensorów. Eksperymentowano z różnymi zbiorami funkcji i rozmiarami grafu dla CGP. W eksperymentach zastosowano różne konfiguracje potoków tworzenia nowych pokoleń w oparciu o selekcje turniejową. Przeprowadzono też eksperymenty, w których postawiono na strategie ewolucyjne $(\mu + \lambda)$ i zastosowanie wyłącznie operatora mutacji.

8.11 Obliczenia na wielu maszynach

Testowanie pojazdów w rozbudowanym symulatorze 3D o dokładnym modelu fizyki pochłania dużo czasu procesora. Środowisko *TORCS* umożliwia przeprowadzenie symulacji w trybie przyspieszonym, pomijając wyświetlanie elementów graficznych interfejsu użytkownika i renderowania obiektów podczas samego wyścigu. Pozwala to przyspieszyć ocenę osobników około 10

```
1 # specific torcs params
2 eval.problem.ticks-per-controller = 500
3 eval.problem.time-init-state = 50
4 eval.problem.model-preprocessor = pl.put.evotorcs.client.extensions.TrackSensorsPreprocessor
5 eval.problem.damage-weight = 0.2
6 eval.problem.dist-weight = 0.5
7 eval.problem.ticks-weight = 0.3
```

Listing 8.4: Sekcja konfiguracji odpowiedzialna za ustalenie parametrów symulacji środowiska TORCS.

```
1 #master configuration
2 eval.masterproblem = ec.eval.MasterProblem
3 eval.masterproblem.debug-info = false
4 eval.masterproblem.max-jobs-per-slave = 3
5 eval.compression=false
6 eval.master.port = 6667
```

Listing 8.5: Sekcja konfiguracyjna dla instancji nadrzędnej

krotnie. W kluczowych momentach przeprowadzania eksperymentów, przed zgłaszaniem sterowników na kolejne etapy zawodów *SCRC*, zdecydowano o rozproszeniu wykonywanych obliczeń na kilka maszyn. Zastosowano podejście ang. *Master-Slaves* wspierane przez bibliotekę *ECJ*. Żeby obliczenia na wielu komputerach były możliwie, na każdym z nich należy odpowiednio skonfigurować środowisko testowe a następnie wgrać odpowiedni plik konfiguracyjny eksperymentu. Instancja środowiska obliczeń wskazana do pełnienia roli nadrzędnej odpowiada za rozsyłanie genotypów do jednostek wykonawczych. W *MasterProblem* gdzie realizowana jest główna pętla obliczeń ewolucyjnych, gdy zostanie zgłoszone żądanie oceny osobnika to zadanie to jest delegowane na maszyny na których działają instancje *Slave*. Cała komunikacja odbywa się w oparciu o protokół TCP/IP. Konfigurowalne parametry dla instancji *Master* przedstawiono na listingu 8.5. Problemem natury technicznej jaki pojawił się przy okazji zdalnego uruchamiania środowiska *TORCS* był brak możliwości uruchomienia programu z poziomu powłoki. Ograniczenie to spowodowało, że obliczenia można było wykonywać wyłącznie na urządzeniach, które umożliwiały połączenia za pomocą zdalnego pulpitu. W obliczeniach wykorzystano komputery z rodziny Microsoft Windows, a uruchamianie środowiska odbywało się poprzez narzędzie Remote Desktop. Na każdej z maszyn typu *Slave* uruchomiono kilka instancji środowiska w zależności liczby rdzeni procesora. W obliczeniach wykorzystano następujące komputery:

- Intel Core 2 Duo Mobile T7250 @ 2,0 Ghz, 4GB DDR2-SDRAM, NVidia GeForce 9300M GS
- Intel Core 2 Duo E7300 @ 2,66 Ghz, 2GB DDR2-SDRAM, NVidia GeForce 9600GT
- Intel CoreQuad U9400 @ 2,66 GHz, 2GB DDR2-SDRAM, 2x NVidia GeForce 9800GTX

Skrócenie czasu obliczeń było szczególnie zauważalne w eksperymentach, dla których liczba kroków symulacji przeznaczona dla jednego osobnika była rzędu kilku tysięcy. Dla tych przypadków zysk wynikający ze wzrostu mocy obliczeniowej przewyższał straty wynikające z opóźnień przesyłania wiadomości w sieci.

Rozdział 9

Analiza wyników

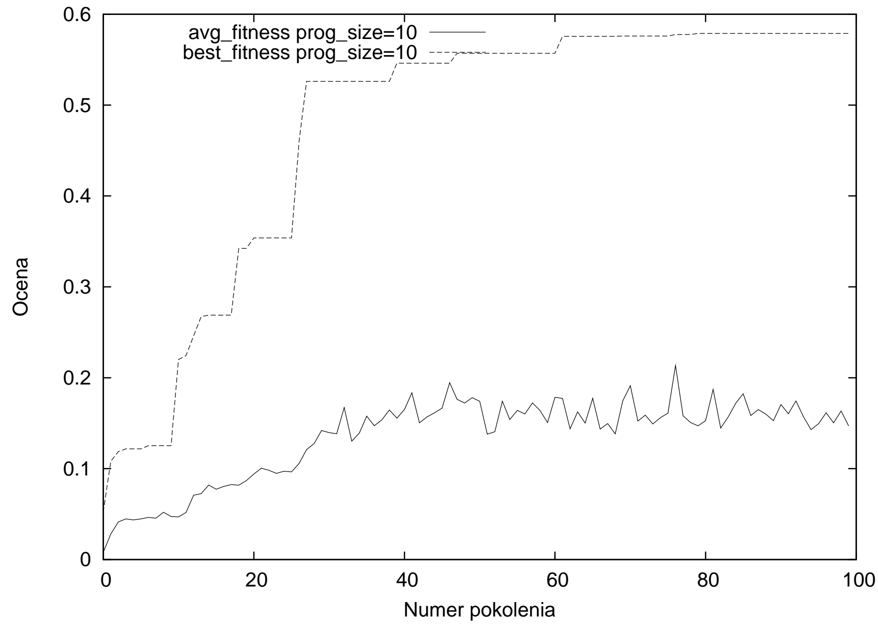
W większości eksperymentów badających wpływ poszczególnych parametrów na jakość uzyskiwanych rozwiązań zastosowano strategie ewolucyjną $(\mu+\lambda)$. W podejściu tym dana populacja jest ograniczana do μ najlepszych osobników, którzy stanowią rodziców λ osobników. Zakłada się, że λ jest wielokrotnością μ wtedy każdy rodzic generuje taką samą liczbę potomków. W przeprowadzonych eksperymentach, za wyjątkiem tych gdzie zostało wprost zaznaczone że jest inaczej, wybrano wartości $\mu = 10 \quad \lambda = 40$. Zastosowano tylko operator mutacji punktowej. Prawdopodobieństwo mutacji ustalono na poziomie 3%. Każdy eksperyment został powtórzony trzykrotnie a zaprezentowane wyniki stanowią średnią arytmetyczną z wyników przeprowadzonych eksperymentów.

9.1 Rozmiar programu CGP

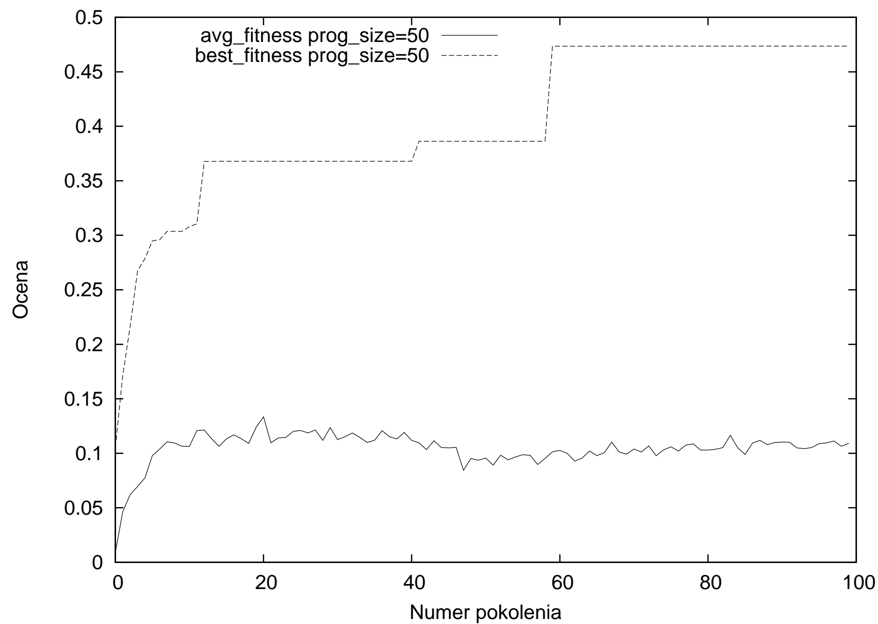
Pierwsze eksperyment dotyczył wpływu górnego ograniczenia na liczbę wierzchołków programu CGP na jakość uzyskiwanego rozwiązania. Dowiedziono, że dla trudnych problemów przy wykorzystaniu dużych programów maleje wartość miary *Wysiłku Obliczeniowego* (ang. *Computational Effort*) potrzebna do odkrycia suboptymalnego rozwiązania. Wyniki pokazują, że najlepiej dobrany rozmiar programu dla problemu sterownia leży w przedziale $[200; 1000]$ węzłów. Duży rozmiar programu CGP wpływa też korzystnie na zjawisko neutralności opisane w rozdziale 5.3. Przy skrajnie małych programach uzyskuje się szybko średniej jakości rozwiązania ale ich poprawa jest bardzo trudna.

9.2 Wprowadzenie stałych w programach CGP

Stałe do programu CGP można wprowadzić na kilka sposób. Pierwszy, niejawny polega na samodzielnym wykształceniu przez osobnika węzłów reprezentujących stałe. Na podstawie istniejących wejść i zastosowania określonych funkcji arytmetycznych możliwe jest uzyskanie węzłów, których wartości pozostają niezmienne względem przestrzeni wejść. Drugi sposób polega na zwiększeniu warstwy wejściowej o sztuczne węzły reprezentujące stałe wartości np. 0.0 lub 1.0. Trzeci sposób to dodanie zmiennych ulotnych znanych z programowania genetycznego interpretowanych jako wejścia lecz kodowanych w genotypie i podlegających modyfikacją w procesie wariacji. W eksperymentach zweryfikowano wpływ ostatniego typu stałych na zbieżność algorytmu. W strategii ewolucyjnej z górnym ograniczeniem na rozmiar programu równym 200 węzłów badano osiągane wyniki przy różnej liczbie stałych. Wyniki porównania prezentuje wykres. Jak widać na wykresie 9.5 wprowadzenie odpowiedniej liczby stałych znacznie wpływa na średnie przystosowanie osobników w populacji oraz znajdywanie lepszych rozwiązań. Z jednej strony wprowadzenie dużej liczby stałych może mieć negatywny wpływ na algorytm ponieważ zwiększa wymiar przestrzeni do

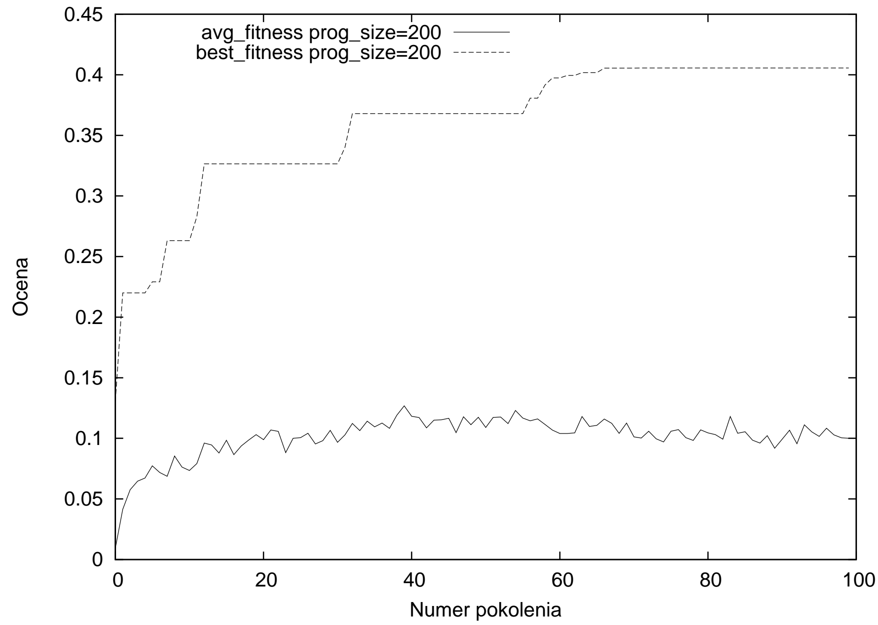


RYSUNEK 9.1: Wartość średnia i najlepsza przystosowania dla strategii ewolucyjnej $(\mu + \lambda)$ o maksymalnym rozmiarze programu 10 węzłów.

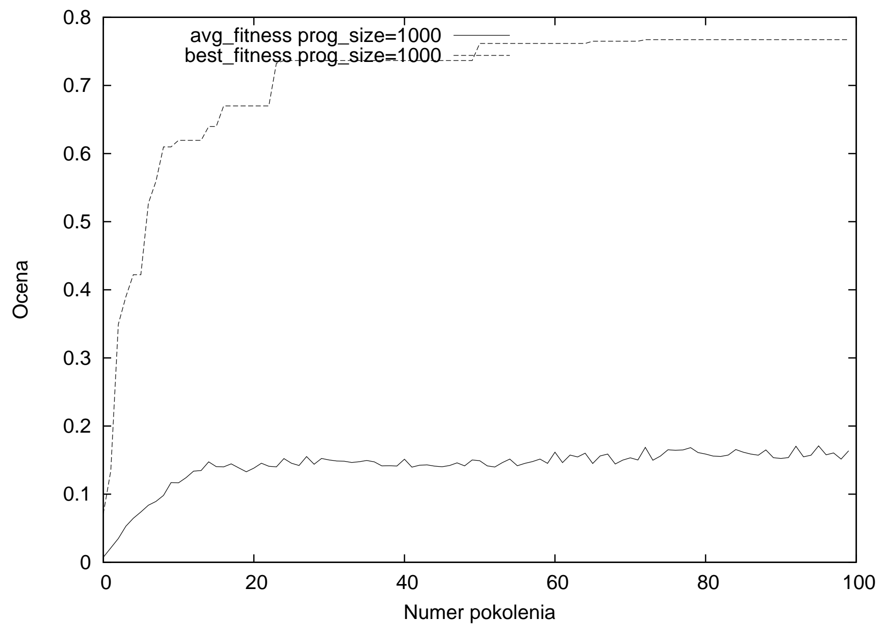


RYSUNEK 9.2: Wartość średnia i najlepsza przystosowania dla strategii ewolucyjnej $(\mu + \lambda)$ o maksymalnym rozmiarze programu 50 węzłów.

przeszukania z drugiej strony odpowiednia liczba stałych może wprowadzić wartości usprawniające proces sterowania, na przykład stały współczynnik przyspieszenia. Przy zwiększeniu liczby stałych z 4 do 6 odnotowano pogorszenie uzyskiwanych rozwiązań.



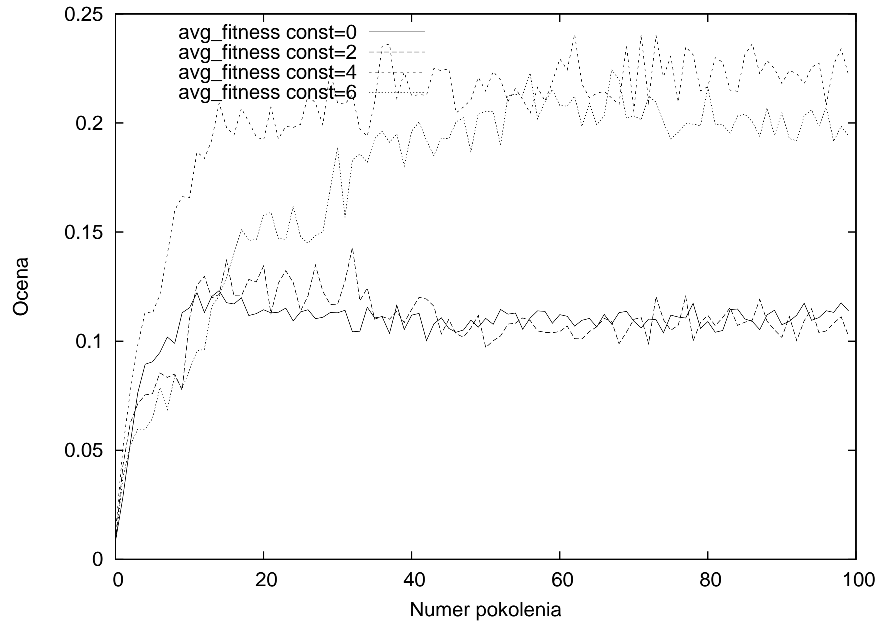
Rysunek 9.3: Wartość średnia i najlepsza przystosowania dla strategii ewolucyjnej $(\mu + \lambda)$ o maksymalnym rozmiarze programu 200 węzłów.



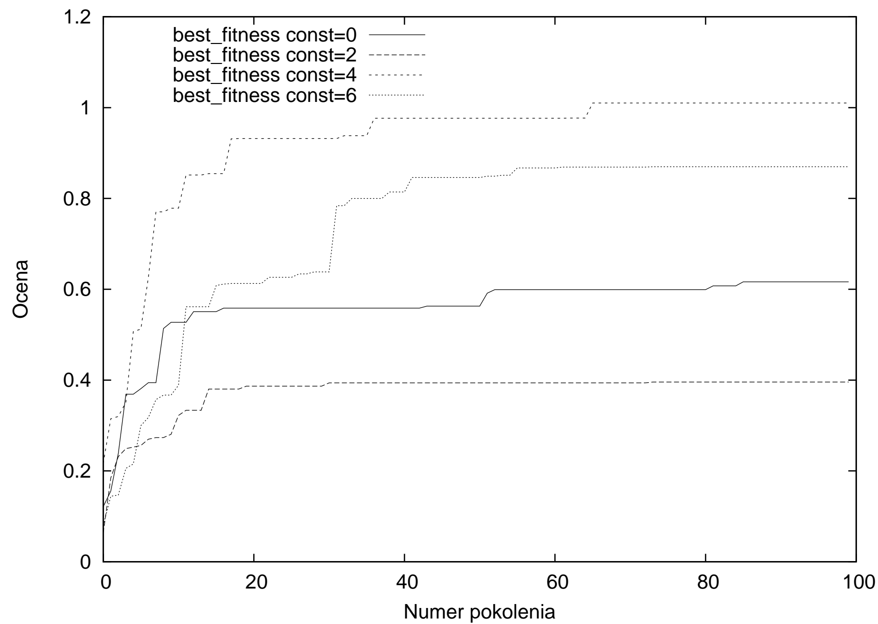
Rysunek 9.4: Wartość średnia i najlepsza przystosowania dla strategii ewolucyjnej $(\mu + \lambda)$ o maksymalnym rozmiarze programu 1000 węzłów.

9.3 Zastosowanie reprezentacji z podprogramami

W trzech seriach eksperymentów przeprowadzono porównanie działania algorytmu z odpowiednio $1, 2, 3$ podprogramami. W eksperymencie stosowano strategię ewolucyjną z operatorem mutacji jednopunktowej. Rozmiar programu ustalono na 200 wierzchołków. Dla genotypów reprezentujących więcej niż jeden podprogram ograniczenia dobrano w ten sposób aby sumaryczna liczba węzłów we wszystkich podprogramach była nie większa niż 200. Na podstawie wykresów 9.7 i 9.8 można powiedzieć, że stosowanie podprogramów ma dwojaki wpływ na jakość znajdowa-

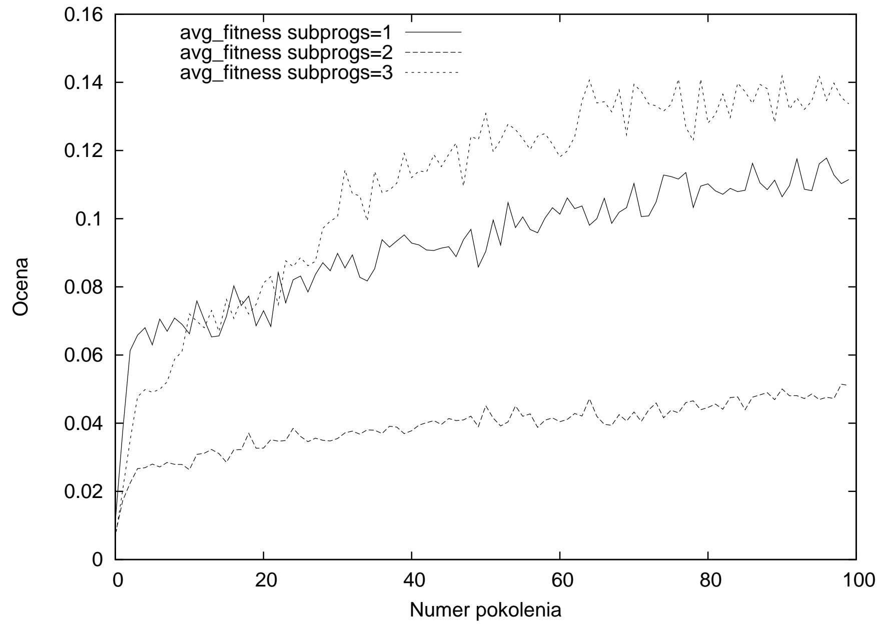


RYSUNEK 9.5: Zestawienie średniej wartości *fitness* dla różnej liczby stałych w genotypie.

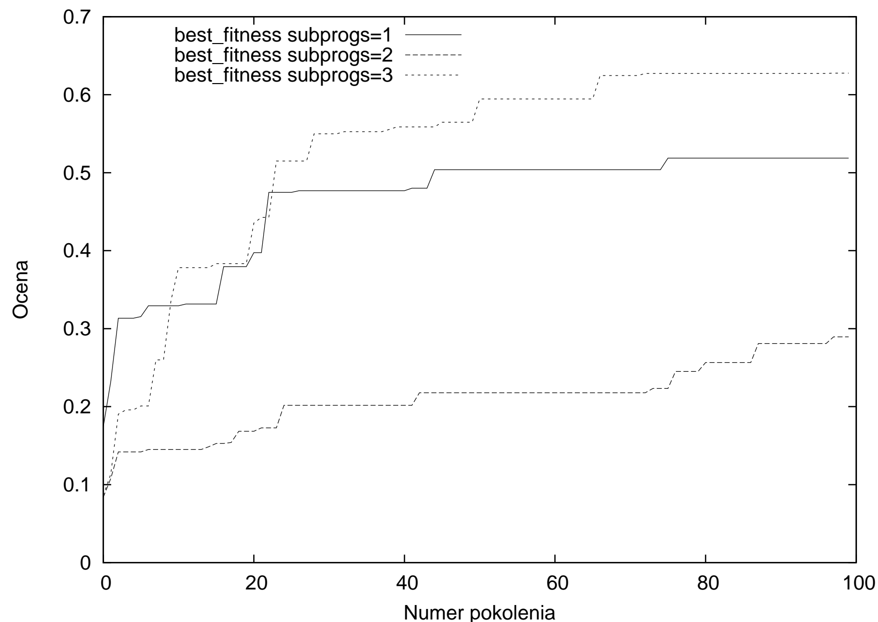


RYSUNEK 9.6: Zestawienie wartości *best_fitness* dla różnej liczby stałych w genotypie.

nych rozwiązań. Po pierwsze powoduje, że częściej zastosowanie mutacji wpływa na wprowadzenie drobnych ulepszeń do genotypu. Z drugiej strony uznając, że optymalne sterowanie jest funkcją, której wartość przybliżamy zdecydowanie trudniej estymować jej wartość przy użyciu kilku składowych programów. Uzyskane wyniki świadczą, że stosowanie więcej kilku podprogramów może poprawić wyniki eksperymentu. Wadą stosowania wielu podprogramów jest trudność analizy kodu sterownika.



RYSUNEK 9.7: Zestawienie średniej wartości $fitness$ dla różnej liczby podprogramów.

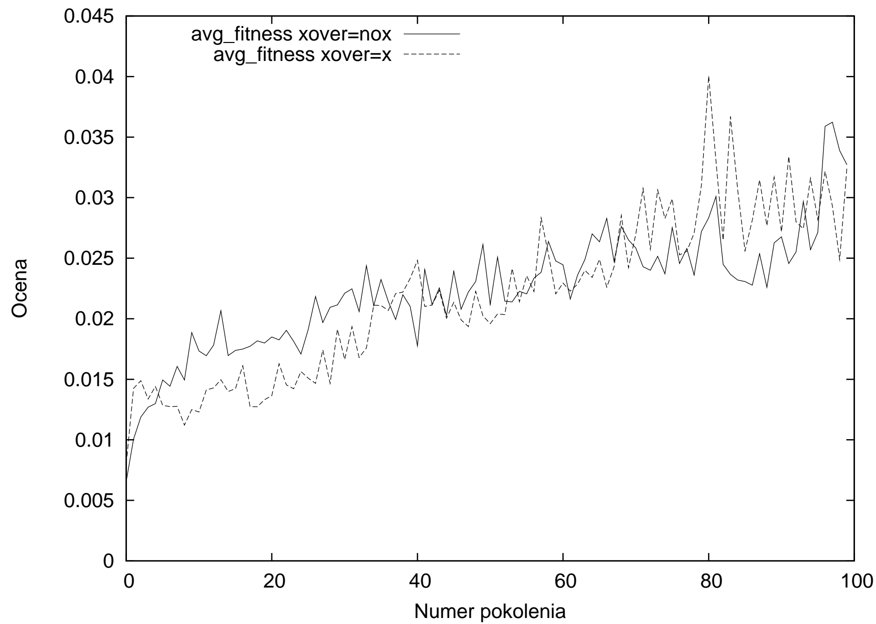


RYSUNEK 9.8: Zestawienie średniej wartości $fitness$ dla różnej liczby podprogramów.

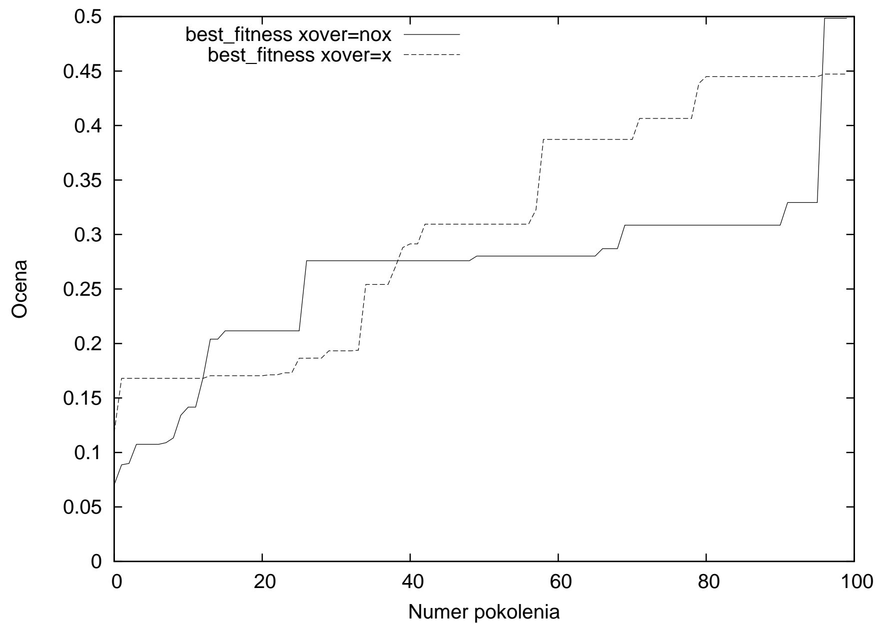
9.4 Metody krzyżowania

Operator krzyżowania arytmetycznego z powodzeniem stosowano w CGP dla zadań regresji. W przeprowadzonych badaniach starano się poprawić zbieżność uczenia stosując operator krzyżowania dla zmiennoprzecinkowej reprezentacji genotypu. Do struktury odpowiadającej za reprodukcję populacji wprowadzono potok (ang. *Pipeline*) o dwóch wejściach reprezentujący krzyżowanie. Każdy rodzic wybierany był na podstawie selekcji turniejowej najlepszy z dwóch. Prawdopodobieństwo krzyżowania w populacji wynosiło 10%. Przeprowadzone eksperymenty, których wyniki zamiesz-

czono na wykresach 9.9 i 9.10 wskazują, że zastosowanie operatora krzyżowania wzrost wariancji wartości średniej przystosowania. Wynika to z faktu, iż skrzyżowanie dwóch osobników powoduje uzyskanie programu, który ma zupełnie inna strukturę niż jego rodzice.



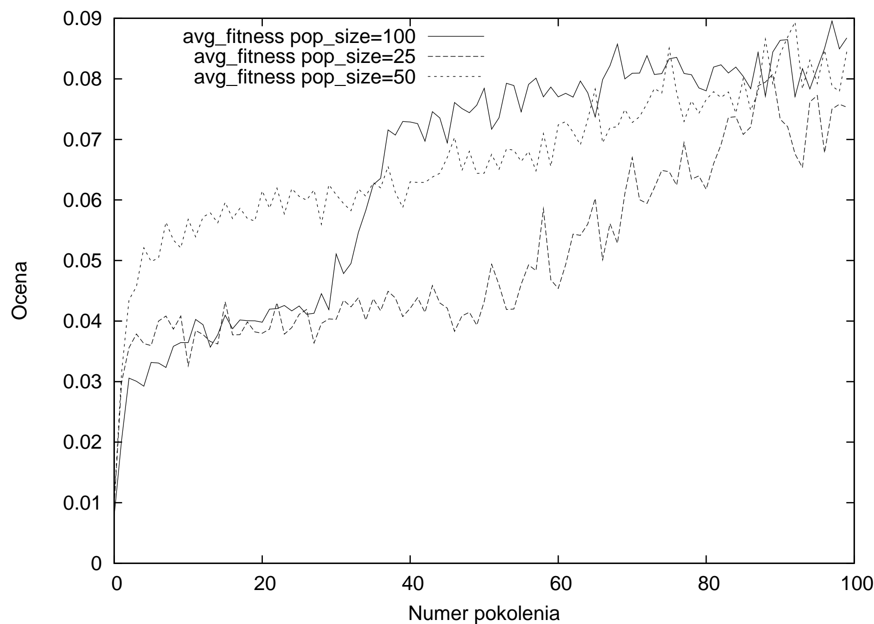
RYSUNEK 9.9: Zestawienie średniej wartości *fitness* przy zastosowaniu operatora krzyżowania oraz bez krzyżowania.



RYSUNEK 9.10: Zestawienie wartości *best_fitness* przy zastosowaniu operatora krzyżowania oraz bez krzyżowania.

9.5 Rozmiar populacji

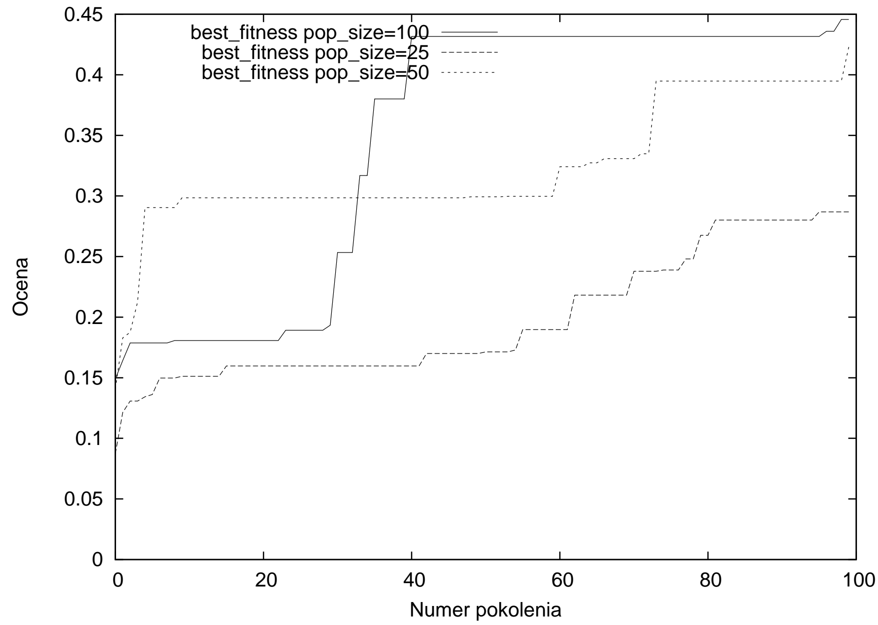
Istnieją różne podejścia do wyboru początkowego rozmiaru populacji w programowaniu genetycznym i kartezjańskim programowaniu genetyczny. Koza przeprowadzając eksperymenty dla problemu regresji symbolicznej preferował duże populacje rzędu kilku tysięcy osobników uzyskując akceptowalne wyniki po małej liczbie generacji. Dla problemu ewolucji sterownika robota mobilnego z zastosowaniem CGP, który to problem stanowi podobną złożoność jak nauka sterownika pojazdu w środowisku *TORCS* otrzymano dobre wyniki dla populacji 50 osobników. Zbadano wyniki działania algorytmu dla populacji od 25 do 100 osobników. Dla dużej populacji uzyskano nieznacznie lepsze wyniki. Należy jednak dodać, że wykonanie eksperymentu dla populacji o rozmiarze 100 osobników wymaga wykonania większej liczby ewaluacji.



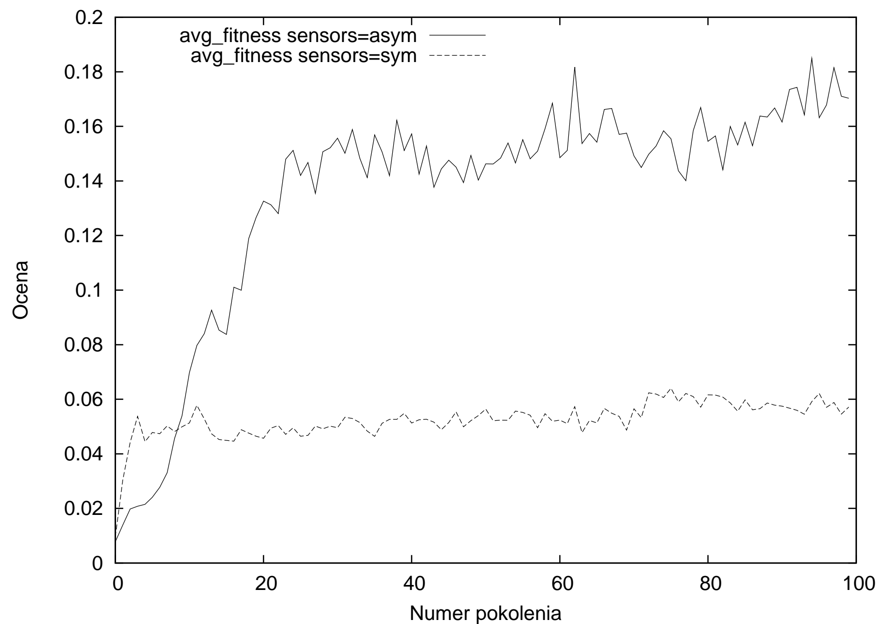
Rysunek 9.11: Zestawienie średniej wartości *fitness* dla różnych rozmiarów populacji.

9.6 Porównanie różnych modeli sensorów

W trakcie analizy zadania sterowania pojazdem założono, że wykorzystanie symetrii w kierowaniu powinno zredukować złożoność problemu i uprościć naukę. Do podobnych wniosków doszli autorzy zgłoszeń na zawody *SCRC 2007*. Weryfikacji tego założenia dokonano przeprowadzając dwie serie eksperymentów modyfikując wyłącznie zbiór wejść sterownika. W pierwszym eksperymencie był to model asymetryczny, w drugim przypadku zastosowano model symetryczny z informacją o kącie i krzywiźnie zakrętów. Uzyskane wyniki zaprzeczają tej tezie. Przyczyny należy się dopatrywać w pewnych wadach którymi jest obciążony system wykrywania zakrętów. Na przykład nie w każdej sytuacji możliwe jest odpowiednie oszacowanie promienia i kąta skrętu. Optymalne wydaje się połączenie informacji z dwóch typów sensorów i stworzenie modelu hybrydowego.



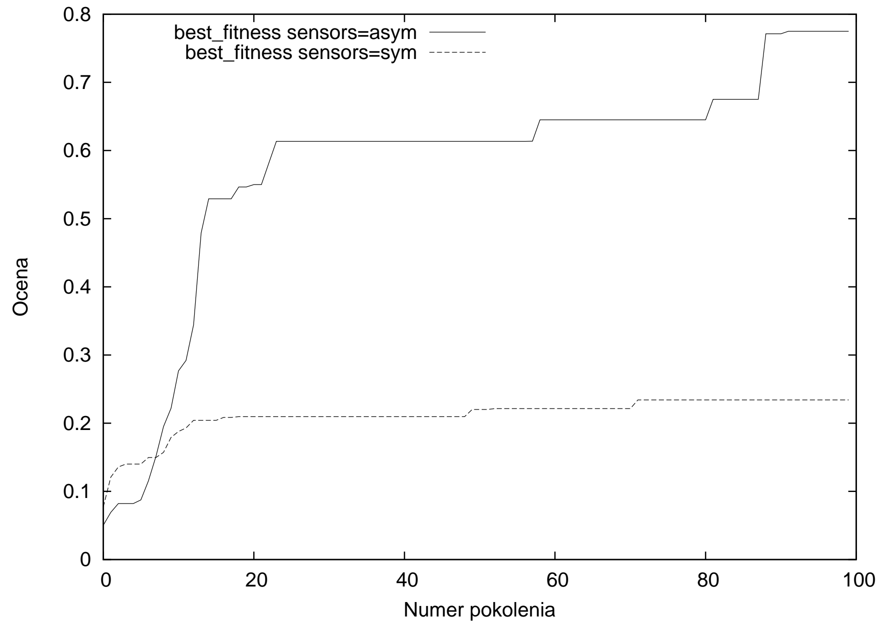
Rysunek 9.12: Zestawienie wartości $best_fitness$ dla różnych rozmiarów populacji.



Rysunek 9.13: Zestawienie średniej wartości $fitness$ dla różnych modeli sensorów.

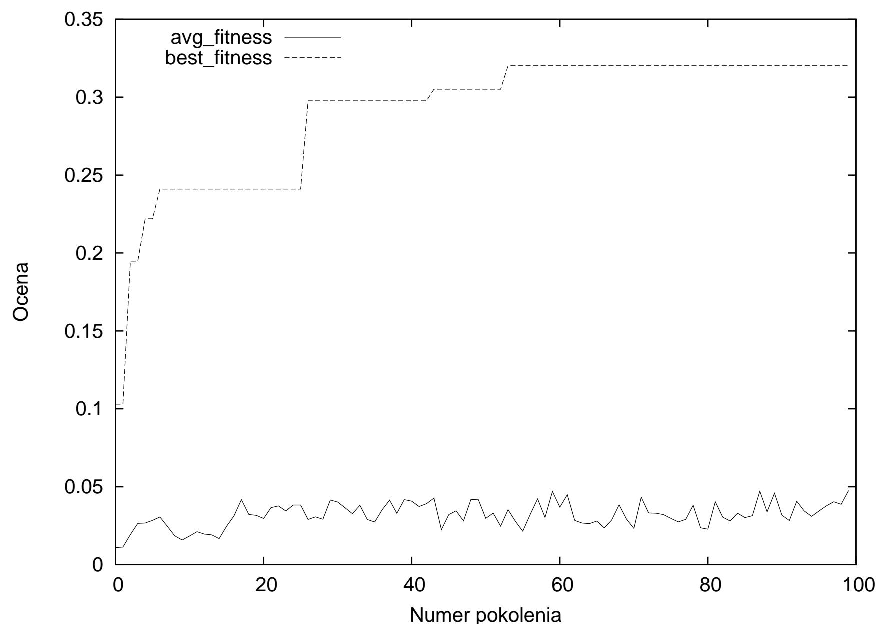
9.7 Koewolucja

W przeprowadzonym eksperymencie koewolucyjnym zastosowano HoF o rozmiarze 20 osobników. W danym pokoleniu do HoF dodawano najlepszego osobnika w populacji. Ocena osobników dokonywano w oparciu o wyścig z 3 losowo wybranymi osobnikami z HoF. Parametry określające globalne współczynniki wyprzedzeń dla redukcji i zwiększenia maksymalnego dopuszczalnego biegu dla pojazdu wyprzedzanego zdefiniowano na 5% i 10%. Uzyskane wyniki, zamieszone na wykresach 9.15 i 9.16, pokazują, że w populacji wykształciły się pewne zachowania poprawnego

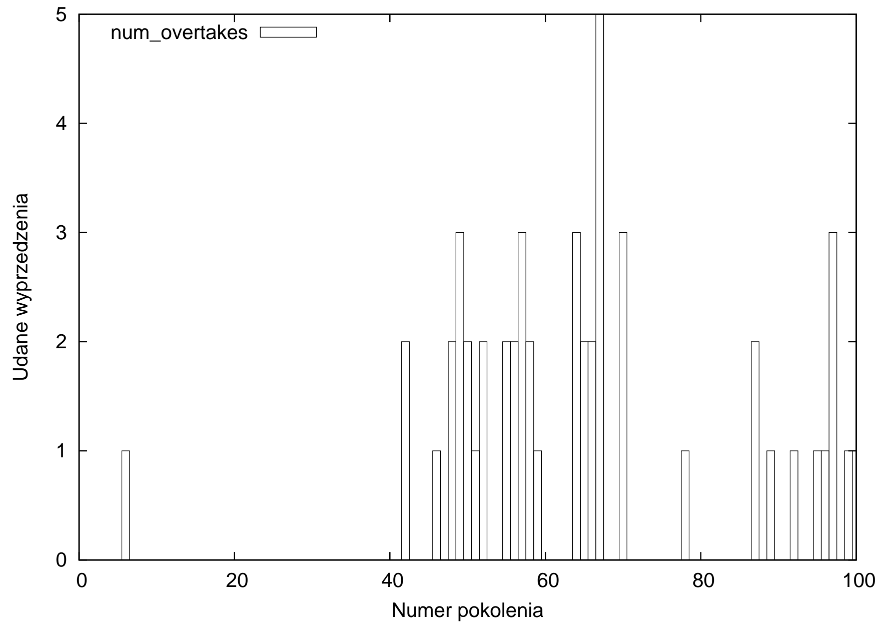


RYSUNEK 9.14: Zestawienie wartości *best_fitness* dla różnych modeli sensorów.

wyprzedzania. Jednak manewr ten jest na tyle skomplikowany, że wyprzedzanie następowało tylko w ściśle określonych warunkach na przykład tylko na określonym fragmencie toru i często wynikało z błędu sterownika znajdującego się z przodu. Wzrost narzutu związanego z obliczeniami ewaluacją osobników uniemożliwił przeprowadzenie dłuższych eksperymentów, które mogły doprowadzić do stabilniejszego zachowania.



RYSUNEK 9.15: Zestawienie średniej wartości *fitness* dla różnych modeli sensorów.



RYSUNEK 9.16: Zestawienie wartości *best_fitness* dla różnych modeli sensorów.

9.8 Analiza najlepszych osobników

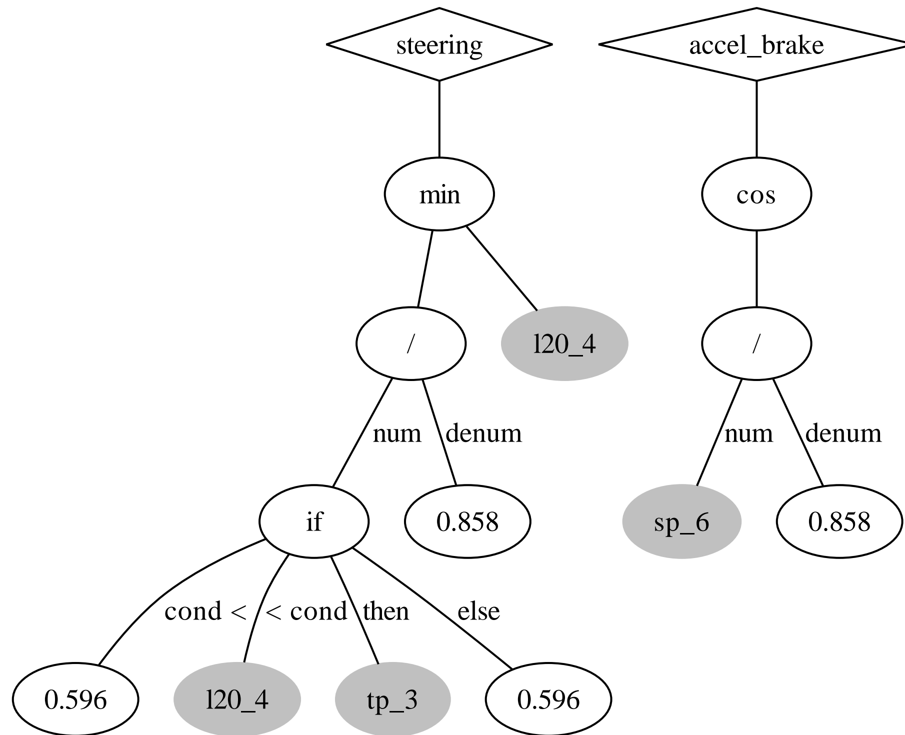
Najlepsze osobniki uzyskane w eksperymentach zapisano w postaci metod odpowiedzialnych za sterowanie. Metody te zostały osadzone w klasie sterownika *EvolvedController*. Funkcjonalność tej klasy oprócz sterowania i przetwarzania wektora wejść obejmowała monitorowanie stanu pojazdu i interwencję w przypadkach wystąpienia niepoprawnego stanu. Za stan taki uznano sytuację gdy pojazd znalazł się poza torem. Ze względu na błędne odczyty z detektorów w tym stanie wykształcenie umiejętności powracania w trakcie ewolucji było niemal niemożliwe. Gdy pojazd znalazł się poza torem na przykład w wyniku zderzenia sterownie przejmowała metoda *rescue()*. Algorytm powrotu na tor został opracowany przez programistę. Gotowe klasy po skompilowaniu zostały przetestowane na różnych torach. Porównano wykształcone techniki jazdy z dobrymi praktykami opisanymi w rozdziale 7.

Do porównania wybrano trzy sterowniki. Pierwszy sterownik *SimpleDriver* został w całości napisany przez programistę. Jego sterowanie uwzględnia estymację ostrości zakrętu, kontrolowanie poślizgu kół i utratę sterowania przy gwałtownych skrętach. Kolejne dwa sterowniki zostały uzyskane w wyniku eksperymentu ewolucyjnego. Pierwszy *EvolvedSideswipe* charakteryzuje się dobrymi wynikami oraz bezpieczną jazdą z umiarkowanie skomplikowanym kodem sterującym. Drugi *EvolvedProwl* pomimo bardzo zwięzłego kodu wykształcił umiejętność jazdy po trudnych trasach. Programy sterowników przedstawiono na rysunkach 9.17 i 9.18.

Zestawienie przeprowadzonych wyścigów zamieszczono w tabeli 9.1. Uzyskane wyniki świadczą, że trasa wybrana do eksperymentu była na tyle ogólna, iż pozwoliła wykształcić zachowania pozwalające na ukończenie większości wyścigów. Charakterystyka *EvolvedSideswipe* okazała się dość defensywna, ponieważ na normalnych trasach jego największa prędkość nie przekraczała $150[km/h]$. Wynika to z definicji eksperymentu, którego celem w pierwszej kolejności było uzyskanie sterowników potrafiących bezpiecznie przejechać daną trasę. Agresywne strategie często kończyły się wypadnięciem z trasy i niską oceną osobnika. Trasa *B-Speedway*, na której *EvolvedSideswipe* rozwinął prędkość $266[km/h]$ jest dość specyficzna – ma owalny kształt i szeroką jezdnię co

Rysunek 9.17: Graf reprezentujący program sterownika *EvolvedSideswipe*.

umożliwia przejechanie pełnego okrążenia bez wyraźnego zwalniania. Przeciętne czasy uzyskiwane przez sterownik *EvolvedSideswipe* były zdecydowanie lepsze niż *SimpleDriver* co dowodzi możliwość uzyskiwania dobrych sterowników przy wykorzystaniu *CGP*. *EvolvedSideswipe* nie poradził sobie tylko na trasie *Dirt3*. Jest to tor o nawierzchni szutrowej z dużymi wahaniami wysokości terenu. Sterownik zatrzymał się na zboczu wzniesienia. Oznacza to, że w wykształconym programie sterującym wartość przyspieszenia w głównej mierze zależała od charakterystyki toru a nie aktualnej prędkości w osi podłużnej pojazdu. Sterownik *EvolvedProwl* był w stanie pokonać połowę tras, na których odbyły się wyścigi. Pomimo bardzo krótkiego, prostego w analizie kodu



Rysunek 9.18: Graf reprezentujący program sterownika *EvolvedProwl*.

sterującego, jego zachowanie można określić jako poprawne. Zauważalną wadą tego sterownika był brak płynności jazdy.

Oceniając sterownik *EvolvedSideswipe* pod kątem zachowania w czasie jazdy można sformułować kilka ogólnych spostrzeżeń. Po pierwsze sterownik trzymał się lewej krawędzi toru co znacznie ułatwia mu pokonywanie zakrętów w prawą stronę. Dodatkowo większy promień skrętu pozwalał mu uzyskać większą prędkość wyjścia z zakrętu. Sam manewr skręcania odbywał się pulsacyjnie – sterownik wykonywał serię ostrych, krótkich skrętów. Ruch pojazdu po prostej pojazdu był płynny.

Nazwa toru	Sterownik	l. okrążeń	czas całkowity [$min : sek : msec$]	najszybsze okrążenie [$min : sek : msec$]	maks. prędkość [km/h]	obrażenia
E-Track6	Simple	2	06:37:17	03:15:08	149	0
	Sideswipe	2	05:06:48	02:26:74	154	0
	Prowl	2	06:16:98	02:51:78	100	4101
B-Speedway	Simple	2	04:12:72	02:04:38	149	6
	Sideswipe	2	02:04:01	55:15	266	6
	Prowl	2	04:47:15	02:18:51	107	1585
Dirt3	Simple	2	04:14:61	02:04:40	149	183
	Sideswipe	2	–	–	72	434
	Prowl	2	–	–	99	179
E-Road	Simple	2	05:47:60	02:51:49	149	0
	Sideswipe	2	04:07:44	01:57:88	149	641
	Prowl	2	–	–	108	10145
CG Track 2	Simple	2	03:54:94	01:54:85	149	0
	Sideswipe	2	03:28:62	01:38:65	151	2577
	Prowl	2	03:53:86	01:54:62	103	0
Street 1	Simple	2	05:47:02	02:51:44	149	0
	Sideswipe	2	05:31:02	02:25:61	136	3992
	Prowl	2	–	–	108	4018

TABLICA 9.1: Zestawienie wyników uzyskanych przez sterowniki na różnych trasach.

Rozdział 10

Podsumowanie

Celem niniejszej pracy była ewolucja sterownika pojazdu dla środowiska *TORCS* metodą kartezjańskiego programowania genetycznego. Zadaniem sterownika była autonomiczna jazda po dostępnych w środowisku torach zarówno w trybie indywidualnym jak i w wyścigu z przeciwnikami.

W ramach pracy zaimplementowano środowisko testowe do ewolucji sterowników pojazdów w środowisku *TORCS*. Rozszerzono funkcjonalność modułu CGP do biblioteki *ECJ* o stałe ulotne i reprezentacje kilku podprogramów. Zaimplementowano szereg narzędzi ułatwiających wykonywanie eksperymentów, przetwarzanie i analizę wyników.

Przetestowano różne reprezentacje informacji z sensorów. Zweryfikowano wpływ wykorzystania symetrii na działanie sterowników. Przeprowadzono serię eksperymentów koewolucyjnych uczących sterownik interakcji z przeciwnikami na torze.

Wykazano, że możliwe jest zastosowanie kartezjańskiego programowania genetycznego dla problemu ewolucji sterownika pojazdu. Stanowi to potwierdzenie badań dowodzących, że ta odmiana programowania genetycznego sprawdza się dla problemów o wielu wyjściach, między którymi istnieją silne zależności. Wyewoluowane sterowniki były w stanie samodzielnie pokonać zadane trasy uzyskując czasy porównywalne ze słabo grającym człowiekiem sterującym pojazdem przy wykorzystaniu klawiatury. Cel postawiony w rozdziale 2 został osiągnięty.

Najlepsze programy zostały zgłoszone do udziału w zawodach *SCRC* na konferencjach *ACM GECCO-2009* i *IEEE CIG-2009*. Niestety w konfrontacji ze sterownikami innych uczestników odniosły one dość przeciętne wyniki. Przyczyn tego problemu może być kilka. Po pierwsze wiele ze zgłoszonych sterowników zostało w większości napisanych przez programistów a nie uzyskanych drogą ewolucji. Kolejną przyczynę może stanowić mała moc obliczeniowa jaką dysponowano podczas eksperymentów i stosunkowo krótki czas ich przeprowadzania.

10.1 Napotkane problemy

Większość problemów napotkanych podczas pracy wynikała bezpośrednio lub pośrednio z działania środowiska *TORCS*. Znanym błędem (ang. *bug*) w środowisku są wycieki pamięci (ang. *memory leaks*) podczas restartowania wyścigu. Dlatego konieczne było zaimplementowanie własnego meta sterownika przekazującego sterownia do ocenianych osobników i naprowadzającego pojazd na właściwą pozycję po skończonym wyścigu. Kolejnym problemem stanowił graficzny interface *TORCS*. Po uruchomieniu środowiska włączenie wyścigu jest możliwe wyłącznie poprzez nawigację i klikanie myszą co skutecznie uniemożliwiło zautomatyzowanie procesu wykonywania eksperymentów.

10.2 Propozycje dalszego rozwoju

Trudny sprawdzian dla wyewoluowanych sterowników jakim niewątpliwie były zawody *SCRC* pokazał, że należy w projekt włożyć jeszcze sporo pracy aby móc się mierzyć z najlepszymi. Rozsądne byłoby powtórzenie najbardziej obiecujących konfiguracji eksperymentów dla większych populacji i większej liczby pokoleń, co w obecnej sytuacji nie było możliwe ze względu na napięte ograniczenia czasowe. Dalsze badania powinny się skupić w pierwszej kolejności na koewolucji sterowników. Wykształcenie zachowań efektywnej jazdy w grupie pojazdów znacznie podniosłoby jakość sterowników. W tym celu trzeba by zmienić definicję eksperymentu koewolucyjnego tak aby ocena osobnika wynikała z współzawodnictwa z większą liczbą przeciwników jednocześnie. Ze względu na koszt obliczeniowy symulacji wykonywanych w środowisku *TORCS* i niedogodności wynikające z pracy z nim, ciekawym pomysłem byłaby implementacja własnego środowiska wyścigowego na potrzeby testów. Uproszczona wersja symulatora zachowująca podstawowe zasady fizyki, z reprezentacją stanu pojazdu kompatybilną z *TORCS* i możliwością pełnego zarządzania z poziomu konsoli ułatwiłaby i przyspieszyłaby przeprowadzanie eksperymentów.

Uzyskane w wyniku wyżej opisanych ulepszeń sterowniki mogłyby zostać zgłoszone do kolejnej edycji konkursu *SCRC*.

Literatura

[aib] Sony AIBO. [on-line] `http://support.sony-europe.com/aibo/`; rok 2006.

[BM08] Christian Blum, Daniel Merkle, redaktorzy. *Swarm Intelligence: Introduction and Applications*. Natural Computing Series. Springer, 2008.

[CWM07] Janet Clegg, James Alfred Walker, Julian Frances Miller. A new crossover technique for cartesian genetic programming. *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, strony 1580–1587, New York, NY, USA, 2007. ACM.

[dar] Defense Advanced Research Projects Agency. [on-line] `http://www.darpa.mil/`; rok 2009.

[DL09] Pier Luca Lanzi Daniele Loiacono, Luigi Cardamone. *Simulated Car Racing Championship 2009 Competition Software Manual*, wydanie 1.0, 2009.

[DP08] Yago Saez Diego Perez. Evolving a rule system controller for automatic driving in a car racing competition. *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, 2008.

[ea] Sean Luke et al. A java-based evolutionary computation research system. [on-line] `http://www.cs.gmu.edu/~eclab/projects/ecj/`; rok 2009.

[GST09] Stanisław Gaca, Wojciech Suchorzewski, Marian Trach. *Inżynieria ruchu drogowego – teoria i praktyka*. 2009.

[HC04] Jin-Hyuk Hong, Sung-Bae Cho. Evolution of emergent behaviors for shooting game characters in robocode. *CEC2004*, 2004.

[KM] Neil Rotstan Klaus Meffert. Java Genetic Algorithms Package. [on-line] `http://jgap.sourceforge.net/`; rok 2009.

[LTL] Daniele Loiacono, Julian Togelius, Pier Luca Lanzi. Simulated car racing championship rules. [on-line] `http://cig.dei.polimi.it/`; rok 2009.

[LTL+08] Daniele Loiacono, Julian Togelius, Pier Luca Lanzi, Leonard Kinnaird-Heether, Simon M. Lucas, Matt Simmerson, Diego Perez, Robert G. Reynolds, Yago Saez. The WCCI 2008 simulated car racing competition. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.

[lua] Lua programming language. [on-line] `http://www.lua.org/`; rok 2009.

[mic] Microsoft Robotics. [on-line] `http://msdn.microsoft.com/en-us/robotics/`; rok 2009.

[Mil06] Ian Milington. *Artificial intelligence for games*. Morgan Kaufmann, CMU, Pittsburgh, Pa, 2006.

[mow] MowBot. [on-line] `http://www.mowbot.co.uk`; rok 2009.

[MS06] Julian F. Miller, Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.

[MT00] Julian F. Miller, Peter Thomson. Cartesian genetic programming. *Proceedings of the European Conference on Genetic Programming*, strony 121–132, London, UK, 2000. Springer-Verlag.

[NF00] Stefano Nolfi, Dario Floreano. *Evolutionary Robotics: The Biology,Intelligence,and Technology.* MIT Press, Cambridge, MA, USA, 2000.

[ode] Open Dynamics Engine. [on-line] `http://www.ode.org/`; rok 2007.

[Pap] Greg Paperin. Java API for Genetic Algorithms. [on-line] `http://www.jaga.org/` ; rok 2004.

[Par03] Barry Parker. *The Isaac Newton School of Driving.* The Johns Hopkins University Press, 2003.

[phy] PhysiX. [on-line] `http://www.nvidia.pl/object/nvidia_physx_pl.html`; rok 2009.

[PLM08] Riccardo Poli, William B. Langdon, Nicholas F. Mcphee. *A field guide to genetic programming.* March 2008.

[Pol97] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming, 1997.

[TL06] Julian Togelius, Simon M. Lucas. Arms races and car races, 2006.

[web] Webots 6. [on-line] `http://www.cyberbotics.com/`; rok 2009.

© 2009 Witold Szymaniak

Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Witold Szymaniak",
    title = "{Ewolucja sterownika pojazdu z wykorzystaniem kartezjańskiego programowania
genetycznego}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2009",
}
```