Biblioteka obsługi algorytmów genetycznych (evol)

Zaawansowane Programowanie

Autorzy: Andrzej Fiedukowicz i Maciej Grzybek

Dokumentacja końcowa

Opis zagadnienia

Treść zadania:

Zdefiniować zestaw klas służący do implementacji algorytmów ewolucyjnych. Powinna istnieć łatwa możliwość zmiany funkcji oceny, funkcji odpowiadającej za mutację itd. W oparciu o zaproponowane klasy stworzyć dwie wersje algorytmu ewolucyjnego - jedną standardową, natomiast w drugiej wersji osobniki powinny być oceniane parami (osobnik i łączy się z osobnikiem j na n generacji. Jakość każdego z osobników (składników pary) jest funkcją jakości całej pary (jakości osobnika j i osobnika i).

Opis biblioteki – Quick start

Aby zacząć pracować z bliblioteką należy przede wszystkim załączyć do programu bibliotekę, lub przynajmniej jej części używane w projekcie. Istnieją wewnętrzne zależności między klasami w związku z tym istnieje możliwość że załączenie jednego nagłówka spowoduje dołączenie innego. Nie należy jednak polegać na tym rozwiązaniu.

Lista plików nagłówkowych:

- *Evol.hpp* – załącza wszystkie nagłówki biblioteki evol (typowe rozwiąznie)
- *Population.hpp* – zawiera kluczowe dla algorytmu elementy, główną pętlę, standardowe implementacje mutacji, krzyżowania, losowania elementów itd.
- *Subject.hpp* – zawiera klasę bazową dla wszystkich klas osobników, które może powołać użytkownik. Zawiera standardowe scenariusze krzyżowania osobników i ich mutowania.
- *Chromosome.hpp* – zawiera opis klasy abstrakcyjnej chromosomu, której klasy pochodne tworzy użytkownik bilbioteki.
- *EvolException.hpp* – zawiera opis wyjątków dla biblioteki evol.
- *EvolFunctions.hpp* – zawiera statyczne funkcje dla biblioteki evol pozwalające na szybkie generowanie liczb losowych w określony sposób a także rzutowanie w dół hierarchii klas z typów inteligentnych wskaźników.
- *Observer.hpp* – zawiera opis klas bazowych dla obserwatorów.

Standardowe scnariusze biblioteki

Definiowane przez użytkownika parametry:

- Referencja na oczekiwaną wartość funkcji celu (jeśli główna pętla nie zostanie w inny sposób przerwana, to będzie wykonywana do momentu gdy najlepszy obiekt nie będzie równy lub lepszy w porządku wynikającym z funkcji celu od tej wartości). Referencja ta stanowi również prototyp dla kolejnych obiektów tego rodzaju potrzebnych do obliczenia wartości funkcji celu dla pozostałych obiektów.
- Sprytny wskaźnik na obiekt typu odziedziczonego po Subject, stanowiący prototyp dla klasy (korzystając z prototypu i metody clone() biblioteka produkuje dodatkowe obiekty konkretnej klasy jeśli są potrzebne – wzorzec prototypu).
- Rozmiar populacji – maksymalny rozmiar populacji po fazie selekcji. Rzeczywista maksymalna liczba osobników w każdym punkcie głównej pętli wynosi (współczynnik krzyżowania+1)*(rozmiar populacji).
- Szansa mutacji – prawdopodobieństwo (od 0.0 do 1.0) z jakim w czasie każdego z obiegów głównej pętli może zostać zmutowany każdy z obiektów.
- Współczynnik krzyżowania – ile procent rozmiaru populacji ma zostać odtworzonych w

wyniku krzyżowania. np. gdy Współczynnik krzyżowania wynosi 2.00 zostanie wybranych 2*rozmiar populacji par, które zostaną skrzyżowane ze sobą.

Populacja (główna pętla algorytmu):

- Stwórz populacje losową – klonuje prototyp podany przy tworzeniu populacji i woła na nim metodę setInitialValue() - do zaimplementowania przez użytkownika.
- Powiadom obserwatora nowej generacji (funkcje obserwatora opisano niżej).
- Skrzyżuj obiekty – wybiera losowo dwa obiekty z obecnej populacji i wywołuje na ich odpowiadających sobie chromosomach metodę crossWith(…) - do zaimplementowania przez użytkownika
- Zmutuj obiekty – z określonym prawdopodobieństwem wywołuje na wszystkich chromosomach obiektu metodę mutate() - do zaimplementowania przez użytkownika
- Dokonaj selekcji – sortuje osobniki wg warunku porównania funkcji celu. Następnie odrzuca tyle najgorszych by pozostało tyle ile zadano jako parametr.
- Jeśli pętla nie została przerwana (lub nie został osiągnięty cel) wróć na jej początek.

Do zaimplementowania przez użytkownika

Klasy biblioteki evol pozwalają na swobodne dziedziczenie i modyfikowanie dowolnych elementów kodu, jednak podstawowa funkcjonalość (i cel tworzenia biblioteki) stanowi tworzenie algorytmów rozwiązujących konkretne problemy w taki sposób by wymagane było jedyne ścisłe opisanie problemu. W przypadku gdyby zmiana klas bazowych okazała się konieczna, pomocny może okazać się załącznik nr 1 do niniejszej dokumentacji opisujący szczegółowo kod i strukturę klas biblioteki.

Programista chcący skorzystać z biblioteki musi zaimplementować przynajmniej klasy:

- Funckję celu (dziedzicząc po FitnessFunction).
  Obowiązkowe metody:
  - clone() - powinien zwracać kopię funkcji celu na której został zawołany
  - calculate(...) - wylicza wartość funkcji celu dla zadanego osobnika
  - operator >(...) - sprawdza czy funkcja celu na której został wywołany ma LEPSZĄ (cokolwiek to znaczy w danym problemie) wartość niż ta podana w argumencie
  - operator ==(...) - sprawdza czy funkcja celu na której został wywołany ma TAK SAMO DOBRĄ (cokolwiek znaczy to w danym problemie) wartość niż ta podana w argumencie.
  - //print() - dostępne tylko w wersji developerskiej
- Osobnika (dziedzicząc po Subject)
  Obowiązkowe metody
  - clone() - powinien zwracać kopię osobnika na którym został zawołany
  - setInitialValue() - ustawia wartość początkową dla obiektów w startowej populacji – zwykle funkcje zwraca losową wartość, ale może też np. wybrać konkretne dane początkowe np. wczytane z pliku.
  - //print() - dostępne tylko w wersji developerskiej
- Chromosmów (co najmniej jednego, opisującego osobnika, dziedziczą po Chromosomes)
  - crossWith(...) - opisuje sposób krzyżowania danego chromosomu z innym tego samego typu
  - mutate() - opisuje sposób mutacji chromosomu (zwykle jest to drobna losowa zmiana).

Konieczne do zaimplementowania klasy (a w szczególności zakres odpowiedzialności obowiązkowych metod) opisane są szczegółowo w załączniku nr 1.
W załączniku nr 2 opisano przykład tworzenia nowej aplikacji opartej o bibliotekę evol.

Możliwości modyfikacji algorytmu

Algorytm stworzony przy użyciu biblioteki evol można modyfikować na dwa sposoby z czego pierwszy jest zalecany, lecz daje mniejsze możliwości (zaleca się korzystać z niego jeśli jest on w stanie zrealizować daną modyfikację)

1. Stosowanie obserwatorów
   Obserwatory działają na zasadzie znanej jako wzorzec projektowy obserwatora. Są to specjalne obiekty powiadamiane o określonych zdarzeniach w obiekcie (w tym wypadku w Populacji – zawierającą głowną pętlę algorytmu).
   Biblioteka standardowo dostarcza cztery typy obserwatorów
   1. Obserwator nowej generacji – powiadamiany zawsze gdy rozpoczynany jest kolejny przebieg pętli – zaczyna się tworzenie nowego pokolenia. Aby zaimplementować obserwatora tego rodzaju należy dziedziczyć po *NewGenerationObserver*
   2. Obserwator krzyżowania – powiadamiany na początku fazy krzyżowania, pozwala na dostosowanie elementów do tejże. Aby zaimplementować obserwatora tego rodzaju należy dziedziczyć po *CrossoverObserver* (np. zmianę współczynnika krzyżowania w zależności od obecnej populacji).
   3. Obserwator mutacji – powiadamiany na początku fazy mutacji, pozwala na przygotowanie obiektów do mutacji (np. zmianę prawdopodobnieństwa mutacji w zależności od obecnej populacji).
   4. Obserwator selekcji – powiadamiany na początku fazy selekcji, pozwala na przygotowanie obiektów do selekcji (np. podmianę w locie funkcji celu lub wartości referencyjnej wg której segregowane będą obiekty w zależności od obecnej populacji)
2. Dziedziczenie po dostarczonych klasach
   Większość metod w klasach bazowych zostało zaimplementowanych jako wirtualne z kwalifikatorem dostępu protected co pozwala na niemal dowolną modyfikację standardowych scenariuszy.
   W przypadku wyboru tej metody należy rozwarzyć czy korzystanie z zestawu predefiniowanych klas pozostaje nadal sensowne, może się zdarzyć tak, że szybciej i wydajniej jest stworzyć od podstaw swój własny algorytm (w przypadku gdy następuje znacząca zmiana standardowego scenariusza).
   Ponadto, należy mieć na względzie, że nadpisywanie metod klas bazowych wymaga dogłębnego przestudiowania załącznika nr 1 do niniejszej specyfikacji który opisuje szczegółowo jak zachowują się kolejne klasy i za co odpowiedzialne są konkretne metody.
   Oprócz tego wsparcie w przypadku wprowadzania zmian może stanowić udostępniony na licencji GPL kod źródłowy biblioteki dostępny w repozytorium http://github.com/fiedukow/evol.git.

Metodyka testowania

Ze względu na występowanie licznych klas abstrakcyjnych i niemożność sprawdzenia ich działania bez powoływania klas pochodnych, testy jednostkowe wykonywanne są w ramach prostego zadania dla algorytmu genetycznego (wyszukiwanie minimum funkcji x^2 w zakresie -100, 100 (liczby całkowite)).

Dla określonego ciągu liczb losowych (stałe ziarno) zasymulowano przewidywane zachowanie algorytmu i sprawdzono poprawność działania programu wykonującego automatyczne testy względem symulacji.

Sprawdzanie objęło w szczególności:

- Poprawność wartości parametrów głównej pętli.
- Poprawność operatorów porównia dla funckji celu.
- Poprawna ilość wywołań w klasach pochodnych.
- Poprawność pośrednich rozwiań (najlepszy element po każdej próbie).
- Liczebność populacji w różnych fazach pętli głównej
- Ilość generacji prowadzących do dobrego rozwiązania.
- Poprawność rozwiązania końcowego.

Zgodnośc z dokumentacją wstępną

W trakcie pracy nad projektem nie napotkaliśmy na szczególne trudności implementacyjne. Całość została zrealizowana w ramach przewidzianego we wstępnej dokumentacji konceptu i hierarchii klas w niej opisanych.

Czas pracy (przybliżona wartość)

*30 – 40h * 2 osoby (wliczony czas projektowania itp.)*

ZAŁĄCZNIKI

1. Dokumentacja kodu (doxy.tar.gz)

Opis przykładowych implementacji dostarczonych z klasami

Wraz z implementacją biblioteki dostarczono opis przy użyciu algorytmu genetycznego dwóch problemów, z czego jeden z nich opisany jest w dwóch wariantach.

BMI

- Problem: Wyszukiwanie człowieka o idealnym BMI
- Rozwiązanie genetyczne:
  - Niech współczynnik krzyżowania wynosi 1.0.
  - Niech prawdopodobieństwo mutacji wynosi 0.2
  - Niech rozmir populacji wynosi 5 (mała populacja, gdyż dla większej istnieje zbyt duża szansa „przypadkowego" rozwiązania problemu przy inicjalizacji osobników.
  - Niech wzrost (w cm) i waga (w kg) będą chromosomami.
  - Niech pola opisujące stan chromosomów przyjmują wartości całkowite.
  - Niech mutacja wzrostu dodaje do obecnego wzrostu losową liczbę z zarkesu <-1, 1>
  - Niech mutacja wagi dodaje do obecnej wagi losową liczbę z zakresu <-1, 1>
  - Niech krzyżowanie wzrostu i wagi będzie zdefiniowane w taki sposób, że potomek dwóch chromosomów ma wartość danej cechy równą średniej ważonej z wartości cechy obu rodziców o wagach (pierwsza waga = losowo od 0 do 1, druga waga = 1 – pierwsza waga)
  - Niech chromosmy mają ograniczenia na wartości
    - Wzrost <100, 200>cm
    - Waga <30, 150>kg
  - Niech człowiek będzie osobnikiem zawierającym chromosomy wzrost i waga
  - Niech człowiek będzie inicjalizowany przez ustawienie losowej wartości wzrostu z zakrkesu < 100, 200 > i wagi z zakresu < 30, 150 >
  - Niech funkcja celu odpowiada zbliżeniu się osobnika do idealnej wartości współczynnka BMI (przyjęto 21.0).
  - Niech algorytm działa póki nie osiągnie wyniku (wartości BMI) z zakresu < 20, 22 >

Plecak

- Problem: problem plecakowy (http://pl.wikipedia.org/wiki/Problem_plecakowy)
- Rozwiązanie genetyczne:
  - Niech współczynnik krzyżowania wynosi 1.5
  - Niech prawdopodobieństwo mutacji wynosi 0.1
  - Niech rozmiar populacji wynosi 1000 osobników.
  - Niech istnieje przedmiot opisany wagą i wartością.
  - Niech istnieje pojęcie sejfu jako obiektu zawierającego przedmioty określonej wagi i wartości, pozwający na wyjmowanie z siebie elementów (ale tylko jeden raz tego samego) posiadający operację wyjmowania losowego elementu spośród elementów o wadze nie więszej niż zadana.
  - Niech sejf będzie w stanie stworzyć się na podstwie pliku dane.txt (i niech będzie to sejf zwany sejfem głównym) zawierającym dane wejściowe w formacie:
    - (double) waga (int) wartosc \n np.

      2.0 23
      2.1 20
      3.41 22
      23.0 17
      11.2 20
  - Niech istnieje chromosom opisujący zawartośc plecaka w taki sposób, że zawiera

zestaw przedmiotów o określonej wartości i sumarycznej wadze nie przekraczającej określonego udźwigu wynoszącego 431 (wartość z testu do którego istnieją wyniki referencyjne).

- Niech chromosom będzie w stanie skrzyżować się z innym w taki sposób, że losowy procent przedmiotów brany jest z plecaka pierwszego a pozostałe z drugiego.
- Niech chromosom będzie w stanie dokonać mutacji w taki sposób, że z plecaka usuwany jest jeden przedmiot a w jego miejsce wstawiany jest zero lub więcej innych wybranych spośród możliwych (wystarczająco lekkich) elementów z sejfu głównego.
- Niech istnieje funkcja celu taka, że jest ona tym lepsza im większa jest wartość wszystkich przedmiotów w plecaku.
- Niech algortym poszukuje rozwiązania w którym wartość przedmiotów wynosi co najmniej 7150 (test referencyjny podał jako najlepszą uzyskaną wartość 7145) – udało nam się uzyskać nieco lepsze wyniki.
- Test referencyjny: http://homepage.ntlworld.com/walter.barker2/Knapsack%20Problem.htm wykorzystano dane oznaczone jako Large (10,000 wartości, udźwig 431).

Wariant Duo:

- Niech dodatkowo osobniki (plecaki) łączą się w pary na 10 pokoleń i co dziesięć pokoleń będą losowo przydzielane na nowo.
- Niech osobniki będą oceniane razem w taki sposób że ocena pary wynosi połowę sumy ich wartości (średnią).
- Niech sposób oceny i wartość docelowa a także inne parametry zostaną niezmienione w stosunku do wariantu podstawowego.