Sprawozdanie z ReCORE 1.0

# Wprowadzenie

Wykonałem pierwszą iterację programu, który miałby działać na wzór **CORE** opisanego w (Tan, Yi i Ang 2006). Nie zaimplementowałem wszystkiego, a jedynie minimalną – wystarczającą do działania koewolucji – część.

Nie korzystałem z wcześniejszej bazy kodu. Zacząłem projekt od zera korzystając z niedawno nauczonych technik programowania obiektowego oraz TDD. Najważniejsze części programu wyizolowałem i przetestowałem. Stworzyłem proste mechanizmy wizualizacji takich pojedynczych części. Implementację oceny osobnika sprawdzono z implementacją w pakiecie WEKA. Kod źródłowy znajduje się pod adresem: <https://github.com/eurekin/reCORE>.

Ta część uwzględnia dwie populacje: reguł i zbiorów reguł. Populacja reguł poddana jest pełnej ewolucji, natomiast populacja zbiorów reguł przeszukiwaniu nieukierunkowanemu – losowemu. Reguły poddane są: mutacji, ocenie oraz selekcji. Zbiory reguł poddawane są wyłącznie mutacji i ocenie. W obecnej postaci, selekcja zbiorów reguł wydaje się niepotrzebna.

Gdyby dodać selekcję zbiorów reguł do algorytmu w obecnej postaci, to nic by to nie dało. Dzieje się tak dlatego, że populacja reguł zmieniana jest diametralnie z pokolenia na pokolenie. Osobnik zbioru reguł kodowany jest jako lista indeksów. Pojedynczy krok ewolucji reguł niszczy powiązania – jest wysoce nieprawdopodobne, aby reguła po selekcji zachowywała swój indeks (pozycję na liście w populacji). Dlatego osobnik kodujący zbiór reguł po pojedynczym kroku ewolucji reguł zmienia całkowicie swoją postać.

## Propozycja rozszerzenia

Proponuję dodać do systemu mechanizm śledzący reguły. Umożliwiałoby to identyfikowanie reguł po pojedynczym kroku ewolucji. Oczywiście nie wszystkie reguły przejdą do następnego pokolenia, a te które się znajdą, mogą zostać zmienione. Jednak mechanizm śledzący znacznie wspomoże ewolucję na poziomie zbiorów reguł – umożliwi promowanie dobrych osobników w populacji zbioru reguł.

# Rozbieżności z oryginalnym algorytmem CORE

Na potrzeby pierwszej iteracji pominąłem kilka istotnych aspektów oryginalnej implementacji. Podziału na podpopulacje nie zamierzam wprowadzać, gdyż wydają się koncepcyjnie zbyt zawiłe w stosunku do oferowanych korzyści. Krzyżowanie jest koncepcyjnie nietrudne, więc można sprawdzić jego działanie w późniejszych etapach. Wydaje się, że największe korzyści można uzyskać przez stosowanie mechanizmu niszowania.

## Podpopulacje

Nie zaimplementowałem podziału każdej z populacji na mniejsze podpopulacje. Algorytm, na którym się wzorowałem, stosował podział na podpopulacje zarówno dla reguł jak i dla zbiorów reguł. W przypadku reguł, każda z podpopulacji zawierała reguły o określonej liczbie selektorów. W przypadku zbiorów reguł, podpopulacje zawierały osobniki o jednakowej liczbie reguł. Wykorzystane w tym programie sposoby kodowania wykluczają konieczność stosowania tego typu podejścia. Każda reguła posiada genotyp o określonej i stałej długości.

Inaczej jest w przypadku zbiorów reguł. Tutaj osobnik może w trakcie ewolucji zmieniać długość genotypu. Nie istnieje jednak konieczność stosowania różnych podpopulacji, aby uzyskać taki sam efekt końcowy. W oryginalnej publikacji podpopulacje służą wprowadzeniu do optymalizacji kolejnego kryterium optymalizacyjnego: długości reguły. Sprowadza się to do preferowania krótszej reguły w przypadku gdy istnieją dwie reguły o takiej samej długości. Ten mechanizm bardzo łatwo wdrożyć bez podziału na podpopulacje.

## Krzyżowanie

W żadnej populacji nie stosuję krzyżowania. Poprzednie doświadczenia wykazały znikomą skuteczność tej operacji.

## Niszowanie

W oryginalnej wersji populacja reguł jest poddawana mechanizmowi niszowania. Wydaje się, że jest to bardzo istotny element wspomagający działanie całej koewolucji. Jakość ostatecznie zbudowanego klasyfikatora zależy silnie od tego z jakich reguł go utworzono. Istotne zatem jest, aby utrzymywać jednocześnie dobre i różnorodne reguły w populacji reguł.

# Opis koncepcyjny

Populacja reguł

Osobnik kodowany jest w postaci łańcucha binarnego o stałej długości. Dla problemu uczącego o następujących parametrach:

Ilość atrybutów do zakodowania: 3.  
Dziedziny atrybutów: 1 – {0,1,2,3}, 2 – {0,1,…, 15}, 3 – {0,1,…6}  
Wymagana długość kodu do zapisania wartości: 1 – 2 bity, 2 – 4 bity, 3 – 3 bity

Przykładowy osobnik kodujący regułę mógłby wyglądać nastepująco:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numer selektora | 1 | | | 2 | | | 3 | | | Klasa |
| Kod binarny | 1 | 0 | 11 | 0 | 0 | 1101 | 1 | 1 | 000 | 1 |
| Znaczenie składowych | ON | == | 3 | OFF | == | 13 | ON | != | 0 | B |
| Znaczenie całościowe | IF Attr1 == 1 | | | - | | | Attr3 != 0 | | | Class==B |

Ogólna postać reguły:

IF ATTR1 op1 val1 AND ATTR2 op2 val2 AND … AND ATTR\_N opN valN THEN CLASS = class

Indeks atrybutu nie jest kodowany, gdyż każdemu atrybutowi odpowiada określone miejsce w łańcuchu.

## Populacja zbiorów reguł

Osobnik taki składa się z listy indeksów wskazujących wybrane reguły z populacji reguł. Oprócz tego osobnik koduję ‘domyślną klasę’, która jest odpowiedzią osobnika gdy żadna z wskazywanych reguł nie została uruchomiona. Ze względów optymalizacyjnych, osobnik posiada jednocześnie listę i zbiór indeksów. Jest to wykorzystywane przy mutacji oraz ocenianiu osobnika. Kolejna linijka przedstawia kolejno: zbiór indeksów, listę indeksów, klasę domyślną.

Następna linijka przedstawia osobnika z populacji zbioru reguł:

[628, 299] [628, 299] 0

Oznacza to, że w skład osobnika wchodzą reguły 299 oraz 299, a domyślna klasa to 0.

Naprawianie

Atrybutów o zbiorze wartości, którego liczebność nie jest potęgą dwójki, nie można jednoznacznie odzwierciedlić w takim sposobie kodowania. Jeśli przyjmiemy, że osobnikiem może być dowolny ciąg bitów to może się zdarzyć, że nie będzie się dało go odkodować. Problem można rozwiązać stosując odpowiednie procedury naprawcze, lub ostrożnie prowadzić ewolucję w taki sposób, alby nie dopuścić do zakodowania wartości spoza dziedziny atrybutu. W tej implementacji stosuje się prostą procedurę naprawczą. Po rekombinacji osobniki sprawdzane są celem wykrycia wartości spoza dziedziny. Wartości te są zamieniane na należące do dziedziny – na drodze losowania.

# Optymalizacje

Najbardziej zaawansowaną optymalizacją w stosunku do poprzedniej implementacji jest operacja mutacji. Poprzednio mutacja wykonywana była dla każdego bitu osobnika z osobna. Korzystając z faktu, że każdy bit posiadał identyczne prawdopodobieństwo mutacji można zastosować rozkład dwumianowy do wyznaczenia ilości bitów w ciągu, które należy poddać mutacji. Następnie należy wylosować unikalne indeksy i zamienić wskazywane przez nie wartości (zera na jedynki i odwrotnie). Efekt jest taki sam jak dla mutowania każdego bitu z osobna. Dla typowego przypadku (p.mut. mniejsze niż kilka procent) oznacza to znaczne zmniejszenie ilości wywołań funkcji losowania liczb. Należy zauważyć, że często stosowane przypadki graniczne nie mają tutaj zastosowania. Mowa o przypadkach gdy rozkład dwumianowy upodabnia się do rozkładu Poissona oraz rozkładu normalnego. Dzieje się tak dlatego, że prawdopodobieństwo mutacji jest dalekie od 0,5%. W programie wykorzystano implementację rozkładu Bernoulliego, udostępnianą przez bibliotekę Apache Commons Math. Wewnętrznie korzysta ona z liczb ułamkowych, które posiadają bardzo dobre właściwości numeryczne i unikają katastrofalnego (wykładniczego) zwiększania się błędu przy wykonywaniu kolejnych przybliżeń. Prawidłowość działania potwierdzono w środowisku statystycznym R.

Równoległość

Podczas implementacji algorytmu brano pod uwagę możliwość uruchamiania współbieżnego. Z natury ten algorytm daje się łatwo zrównoleglić na poziomie danych. Każdy z etapów ewolucji operuje na zbiorze osobników. Każdemu wątkowi można przydzielić wydzieloną część do przerobienia. Oczywiście są etapy, które wymagają całościowego spojrzenia na populacje. W Tym przypadku z pomocą przychodzą mechanizmy z kategorii MapReduce. Przyspieszenie tych algorytmów nie będzie jednak liniowe w stosunku do liczby uruchomionych procesów.

Ogólnie jednak algorytm ewolucyjny w tej postaci nadaje się do rozpisania na maszyny wieloprocesorowe. Co innego w przypadku klastrów obliczeniowych(wielu jednostek obliczeniowych komunikujących się poprzez sieć opartą o TCP/IP). Stosunkowo częstość etapów wymagających komunikacji pomiędzy procesami mogłaby zniwelować korzyści płynące ze stosowania klastrów obliczeniowych.

Profiling

Z ciekawości uruchomiłem profiler na aplikacji i usunąłem oczywiste błędy implementacji, które znacząco spowalniały proces ewolucji. Po tych zabiegach gorące punkty aplikacji gromadziły się w funkcjach, które wymagały intensywnego dostępu do pamięci. Kolejne przyspieszenie można uzyskać stosując nietrywialnie w implementacji techniki cache’ingu oraz indeksy.

# Doświadczenia

W ramach weryfikacji poprawności działania wykonałem proste testy na zbiorze MONK – **M3** z publikacji (Thrun 1991). Do uczenia i oceny wybrałem zbiór uczący. Nie porównywałem ocen na zbiorze testowym, jednak porównywałem graficzną reprezentację. W publikacji wynikowy klasyfikator przedstawiany jest w postaci graficznej, gdzie rzędy i kolumny odpowiadają wartościom atrybutów, a symbole określają przynależność do klasy.

Zadanie klasyfikacji MONK zaprojektowane zostało w możliwie najprostszy sposób, aby umożliwić porównywanie szerokiej gamy algorytmów uczących się. Każdy obiekt opisywany jest za pomocą 7 atrybutów i może należeć do jednej z dwóch klas (klasyfikacja binarna). Zbiór wartości atrybutów nie jest rozbudowany – największy ma 4 dozwolone wartości. W sumie można zakodować 432 unikalne przykłady.

## Wywołanie

Aby uzyskać kod źródłowy należy najpierw ściągnąć kod źródłowy za pomocą narzędzia git:

git clone git://github.com/eurekin/reCORE.git

cd reCORE

Następnie zbudować program komendą

ant

Oraz uruchomić doświadczenie

java -cp build/classes core.examples.RuleSetPopulationExample

## Parametry ewolucji

Podczas doświadczenia obrano następujące parametry algorytmu:

Ogólne:

* funkcja oceny: miara F

Reguły:

* prawdopodobieństwo mutacji: 0,002
* typ selekcji: ruletka

Zbiór reguł:

* prawdopodobieństwo mutacji: 0,01
* maksymalna liczba reguł: 5

## Wizualizacja

Oryginalny problem M3 przedstawiony został na poniższych rysunkach.

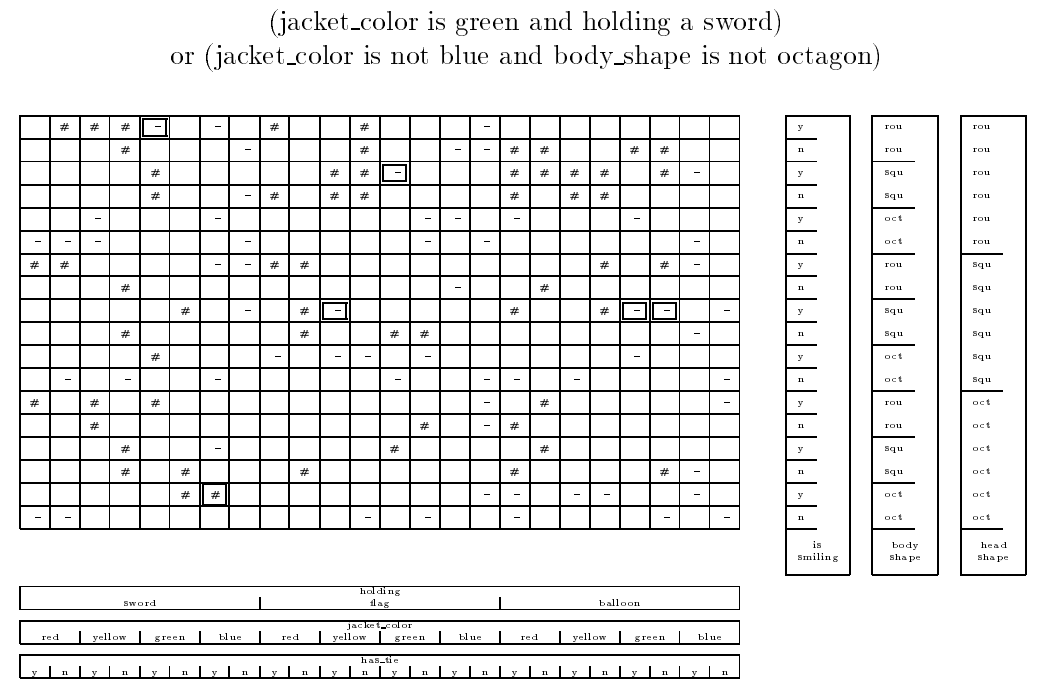


Figure 1 Monk M3 Train set

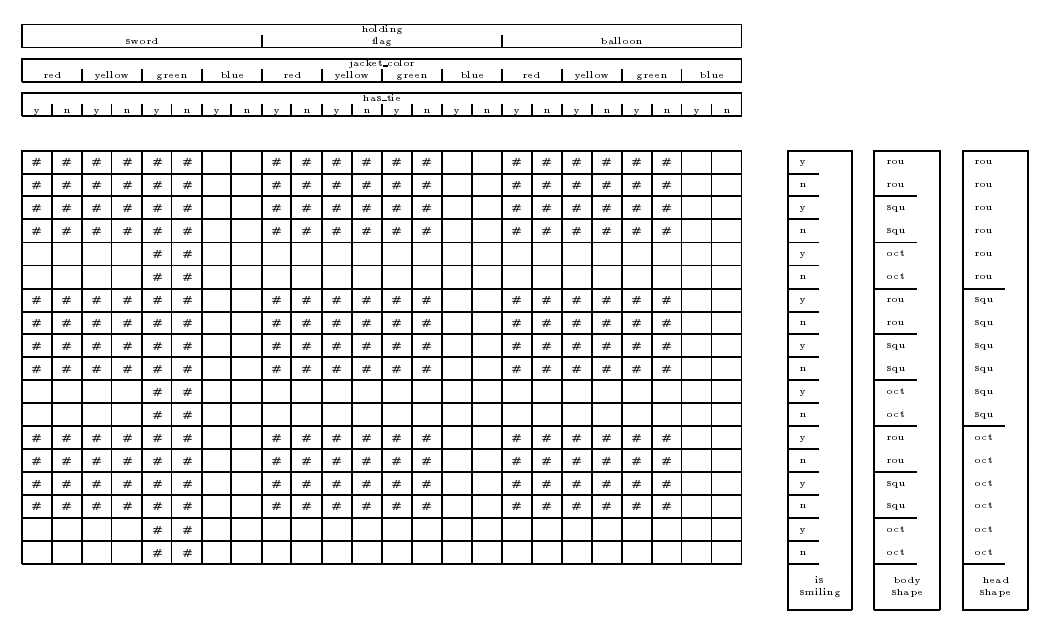


Figure 2 Monk M3 Test set

Wynik działania programu zawiera wizualizacje zbioru reguł, która nie posiada opisu rzędów i kolumn. Odpowiadają one tym z powyższych rysunków.

## Opis przykładowego wyniku

Opis znalezionego rozwiązania rozpoczyna się wartością początkową generatora liczb pseudolosowych:

Seed = 1292969085196

Podanie tej wartości pozwala powtórzyć kompletny przebieg ewolucji.

Następnie przedstawione są miary oceny osobnika: True Positive Rate, False Positive Rate, True Positives, False Negatives, True Negatives, False Positives, Accuracy (dokładność), F Score (miarę F), Recall oraz Precision.

TPR: 95,1% FPR: 4,8% TP: 116 FN: 6 TN: 0 FP: 12

ACC: 95,1% F: 95,1% RCL: 95,1% PRC: 95,3%

Następnie przedstawiony jest osobnik – zbiór reguł klasyfikujących – w postaci tekstowej:

IF body\_shape!=octagon AND jacket\_color!=blue THEN class=B

IF body\_shape==octagon AND holding==sword

AND jacket\_color==green THEN class=B

ELSE class=A

Następnie przedstawiony jest osobnik w postaci wykresu. Opisy atrybutów pominięto, jednak są one takie same jak w publikacji Thrun et. al.

Visualization:

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][]

[][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][]

[][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][][][][][] [][][][][][] [][][][][][]

[][]

[][]

Przedstawiony osobnik odpowiada regułom, które użyto do wygenerowania zbiorów uczącego problemu M3. Na zbiorze testowym posiada 100% dokładność.

# Ogólne wnioski z tego prototypu

Program posiada potencjał. Zaimplementowano jedynie część przedstawioną w publikacji, a mimo to potrafi znaleźć rozwiązanie prostego, aczkolwiek nietrywialnego problemu. Jednak, daleko do stosowalności w problemach ogólnych, trudnych.

# Works Cited

Tan, K. C., Q. Yi, and J. H. Ang. "A coevolutionary algorithm for rules discovery in data mining." *International journal of systems science* 37, no. 12 (2006): 835-864.

Thrun, S. B., et al. *The MONK's Problems A Performance Comparison of Different Learning Algorithms.* Carnegie Mellon University, 1991.