MED 13Z
Implementacja algorytmu Fun
dokumentacja końcowa

Michał Iwanowski, Mateusz Modrzejewski, Łukasz Jasiński

1 Cel projektu

Celem projektu była programowa implementacja algorytmu *Fun* opisanego w [1], przeprowadzenie testów na stworzonym rozwiązaniu oraz dostarczenie dokumentacji. Zaimplementowane zostały różne warianty metod wchodzących w skład algorytmu - zarówno dostosowane do zwykłych partycji, jak i uproszczonych. Na potrzeby projektu została przez nas również zaproponowana własna implementacja drzewa mieszającego.

2 Działanie programu

Program działa w trybie tekstowym i przetwarza dane wsadowo. Dane wejściowe wczytywane są z pliku `.csv`, a wynik wypisywany jest na standardowe wyjście.

Postać wywołania programu wygląda następująco:

```
fun <filename> <target_col> "<col1>,...,<coln>" [options]
```

Parametry wywołania oznaczają odpowiednio:

- `<filename>` - nazwa pliku z danymi wejściowymi programu,
- `<target_col>` - kolumna zawierająca atrybut decyzyjny,
- `"<col1>,...,<coln>"` - wektor numerów kolumn, które będą atrybutami badanymi,
- `[options]` - dodatkowe opcje.

Kolumny liczone są kolejno od lewej strony pliku `.csv`. Aplikacja umożliwia zatem elastyczne przydzielenie ról kolumn w pliku.

Dostępne opcje to:

- `--holds` - wykorzystanie standardowej metody `holds()`[1] (ustawienie domyślne),
- `--sholds` - wykorzystanie metody `strippedHolds()`,
- `--rsholds` - wykorzystanie metody `reducedStrippedHolds()`,
- `--prsholds` - wykorzystanie metody `partReducedStrippedHolds()`,
- `--product` - wykorzystanie metody `product()` (ustawienie domyślne),
- `--sproduct` - wykorzystanie metody `strippedProduct()`,
- `--optprune` - wykonywanie przycinania (*pruning*),
- `--nooptprune` - niewykonywanie przycinania (ustawienie domyślne),

3 Architektura rozwiązania

3.1 Wykorzystane narzędzia

Program został napisany zgodnie z paradygmatem programowania obiektowego w języku C++. Nie zostały użyte żadne zewnętrzne biblioteki ani rozszerzenia. Wykorzystywane są jedynie kontenery z biblioteki standardowej (`map`, `set`, `vector`).

3.2 Struktura programu

Zgodnie z praktyką programowania w C++, podzielony jest na pliki nagłówkowe i źródłowe, przy czym każda odpowiadająca sobie para plików zawiera implementację jednej klasy (z wyjątkiem hierarchii węzłów drzewa mieszającego, które ze względu na niewielką objętość kodu i powiązane działanie czytelniej było umieścić w jednej parze plików).

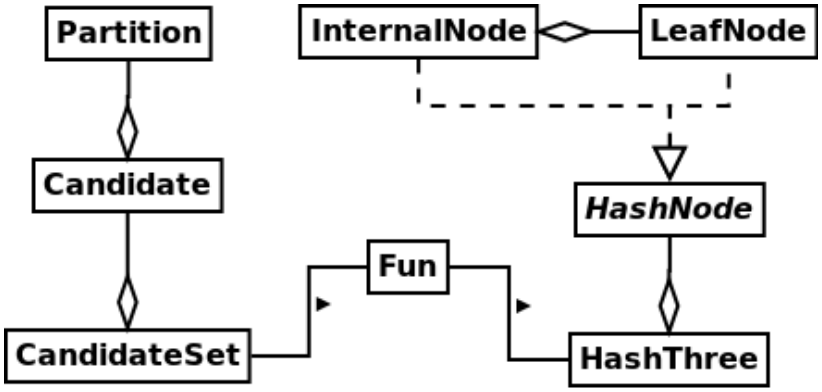
Kod i zasoby programu podzielone są na katalogi. Pliki nagłówkowe znajdują się w katalogu *include/*, zaś źródłowe w *lib/*. W katalogu *tests/* zawarte są przykładowe pliki testowe:

- *przyklad.csv* - będący krótkim przykładem, łatwym do prześledzenia na standardowym wyjściu,
- *letter-recognition.csv*
- *nursery.csv*

[1] Na potrzeby czytelności niniejszej dokumentacji, tam, gdzie to możliwe, przy nazwach funkcji pominięte zostały argumenty wywołania.

Dokładniejszy opis tych plików znajduje się w sekcji niniejszego dokumentu poświęconej testom aplikacji.

W poniższych podrozdziałach znajdują się krótkie opisy najważniejszych klas aplikacji. Schematyczny diagram klas programu przedstawiony jest na rysunku 1.



Rysunek 1: Schematyczny diagram klas aplikacji.

Poza owymi klasami, w aplikacji znajdują się również pliki:

- *main.cpp* - będący wejściem do programu,
- *utils.h*, *utils.cpp* - zawierające kilka pomocnicznych metod (ogólne operacje na ciągach znaków, wypisywanie kontenerów na standardowe wyjście etc.).

3.3 Opis klas

3.3.1 Fun

Główną klasą programu jest klasa `Fun`, zawarta w plikach *fun.h* i *fun.cpp*. Klasa ma prywatny konstruktor, a kluczowe metody algorytmu są zaimplementowane jako statyczne metody tejże klasy. Algorytm zrealizowany jest zgodnie z opisem zawartym w [1] i implementuje wszystkie opisane tam metody.

Klasa udostępnia 4 przeciążone wersje metody `holds()`:

- `holds()`,
- `strippedHolds()`,
- `reducedStrippedHolds()`,
- `partReducedStrippedHolds()`.

Dostępne są też dwie wersje metody `product()`:

- `product()`,
- `strippedProduct()`.

Klasa `Fun` zawiera dwa statyczne wskaźniki na funkcję, które wskazują na wybrany wariant każdej ze wspomnianych metod. Aby zmienić wariant, wystarczy zmienić wskaźnik - takie rozwiązanie pozwala uniknąć zbędnych wyrażeń warunkowych w czasie działania algorytmu.

Plik z danymi wejściowymi czytany jest tylko raz i odczytywany jest z poziomu metody `generateInitialCandidates()`. Wszystkie kolejne obliczenia wykonywane są już w pamięci programu. W celach wydajnościowych aplikacja pracuje na identyfikatorach liczbowych, a nie na napisach. Wymagane zatem jest, aby pierwszy wiersz pliku wejściowego zawierał wypisane kolejno nazwy atrybutów - są one następnie mapowane na identyfikatory w postaci liczb całkowitych. Nazwy atrybutów są przechowywane w celu czytelnego wypisania wyniku dla użytkownika.

3.3.2 Partition

Klasa ta reprezentuje pojedynczy podział względem danego zestawu atrybutów. Każda z grup przechowywana jest jako zbiór, zaś sam podział reprezentowany jest wektorem wskaźników na grupy. Ta reprezentacja zbiorowa przechowywana jest jako pole klasy, zaś na życzenie obliczana jest reprezentacja tablicowa. Pamięć zaalokowana na podziały jest zwalniana w odpowiednich metodach głównej klasy, kiedy konkretne podziały przestają być potrzebne.

3.3.3 Candidate

Klasa `Candidate` reprezentuje zbiór atrybutów badany pod kątem bycia najmniejszym zbiorem, który funkcyjnie określa atrybut decyzyjny. Jako pole klasy przechowywany jest zbiór liczbowych identyfikatorów atrybutów, a także wskaźnik na podział wynikający z danych atrybutów. Klasa udostępnia metody umożliwiające dostęp do danych.

3.3.4 CandidateSet

Klasa `CandidateSet` reprezentuje zbiór wygenerowanych k-elementowych kandydatów. Jako pole klasy przechowywany jest wektor wskaźników na `Candidate`. Klasa udostępnia metody umożliwiające dostęp do wektora, w tym operator indeksowania.

3.3.5 HashTree

Klasa `HashTree` reprezentuje drzewo mieszające [2] wykorzystywane przez algorytm. Drzewo zawiera wskaźnik na korzeń abstrakcyjnego typu `HashNode`. Liście drzewa reprezentowane są klasą `LeafNode` i zawierają wektor kandydatów oraz wektor liczności podziałów odpowiadających kandydatom. Kolejne poziomy wewnętrznych węzłów drzewa reprezentowane są przez klasę `InternalNode`, która jako pole zawiera wektor wskaźników na potomków.

4 Testy

4.1 Opis testów

Testy zostały przeprowadzone na niewielkim zbiorze danych podanym w [1] w celu zweryfikowania poprawności algorytmu, a także na dużych zbiorach danych pobranych z [3]. Zbiór ten zawarty jest w pliku *przyklad.csv*.

Test na niewielkim zbiorze wykonywany jest ręcznie za pomocą uruchomienia programu z terminala, natomiast testy na dużych zbiorach zostały zautomatyzowane za pomocą skryptów powłoki. Duży plik dzielony jest na pliki zawierające arbitralnie dobraną liczbę wierszy pliku oryginalnego, a następnie dla każdego z wygenerowanych plików wykonywany jest algorytm *Fun* w pięciu wariantach, które przedstawia poniższa tabela. Użycie metody `holds()` w innym wariancie niż podstawowy związane jest oczywiście z wykorzystaniem metody `strippedProduct()`.

wariant wywołania	H	H/P	SH	SH/P	PRSH	RSH
wariant holds	Holds	Holds	Stripped Holds	Stripped Holds	Part Reduced Stripped Holds	Reduced Stripped Holds
przycinanie	nie	tak	nie	tak	nie	nie

W przypadku testów na dużych zbiorach, mierzony jest również czas obliczeń dla każdego wywołania programu za pomocą funkcji `time` dostępnej domyślnie w powłoce *bash*. Parametr decyzyjny wybierany był na podstawie opisu zbioru danych z [3].

Wykorzystane pliki testowe i ich podziały:

- *nursery.csv* - 9 atrybutów, 12961 wierszy, podziały na wersję zawierającą 4500, 6500, 7000, 9000 pierwszych wierszy (plus pełny plik),
- *letter-recognition.csv* - 17 atrybutów, 20001 wierszy, podziały na wersję zawierającą 100, 200, 500, 1000, 2000, 5000, 10000, 15000 pierwszych wierszy (plus pełny plik),
- *krkopt.csv* - 7 atrybutów, 28057 wierszy, podziały na wersję zawierającą 2000, 5000, 10000, 15000, 20000 i 25000 wierszy (plus pełny plik).

Skrypty realizujące podział, testowanie i pomiary czasu znajdują się odpowiednio w plikach *nurseryTests.sh*, *letterRecognitionTests.sh* i *krkoptTests.sh*.

4.2 Wynik testu na przykładowym zbiorze

Poniższe wywołanie (albo wywołanie z dowolną prawidłową kombinacją opcji dotyczących metod `holds()` i `product()`):

```
./fun tests/przyklad.csv 6 "1,2,3,4,5"
```

zwraca wynik:

```
Iteration 1...
Iteration 2...
Iteration 3...
Iteration 4...

Results:
a(1) e(4)
c(3) e(4)
```

Wynik ten został sprawdzony poprzez ręczne wykonanie algorytmu i jest poprawny. W połączeniu z faktem, że algorytm był implementowany bardzo starannie, indukcyjnie zakładamy, że wyniki dla pozostałych testów również będą poprawne.

4.3 Testy na dużych zbiorach

Poniższe wykresy oraz tabele przedstawiają wyniki pomiaru czasu wykonania dla dużych zbiorów danych. Wykresy wykonane są w skali liniowej. Wyniki jakościowe testów spisywane są przez skrypty do odpowiednich plików w katalogu *results*.

4.3.1 Nursery

Jako atrybuty badane zostały wybrane wszystkie atrybuty poza decyzyjnym, tzn. 8 atrybutów. Jedyną znalezioną zależnością funkcyjną była zależność od całego zbioru atrybutów, co jest zgodne z przewidywaniami ze względu na charakter danych.

wiersze	H	H/P	SH	SH/P	PRSH	RSH
4500	407	401	402	402	399	336
6500	602	637	609	606	608	507
7000	665	650	658	668	657	571
9000	894	890	890	913	897	761
12961	1387	1389	1373	1382	1379	1211

Rysunek 2: Czasy wykonania [ms] dla zbioru danych nursery.

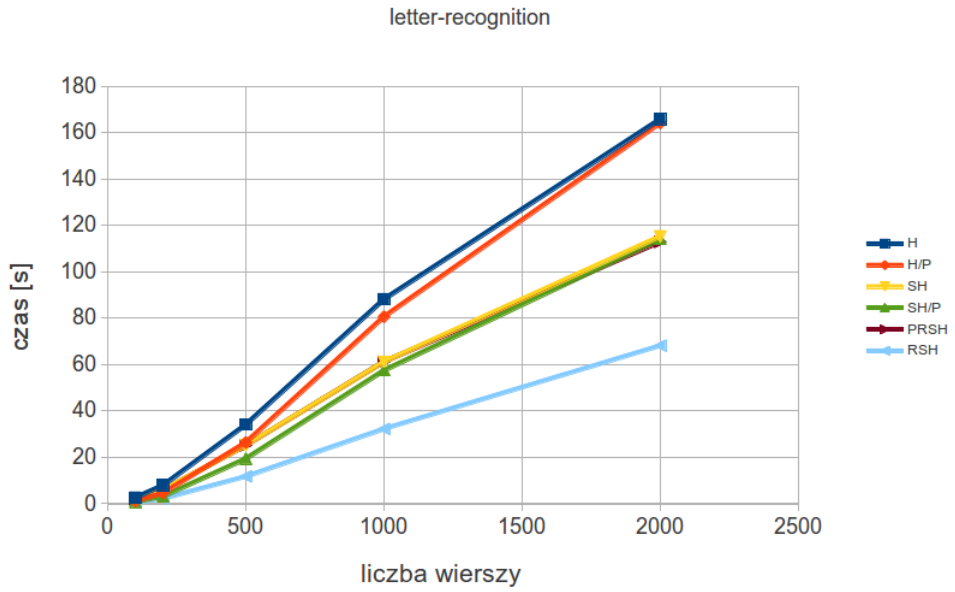
Rysunek 3: Wykres czasów wykonania dla zbioru danych nursery.

4.3.2 Letter-recognition

Jako atrybuty badane zostały wybrane wszystkie atrybuty poza decyzyjnym, tzn. 16 atrybutów. Program znajduje duże ilości zależności funkcyjnych dla tego zbioru i zapisuje je w odpowiednim pliku w katalogu *results*.

wiersze	H	H/P	SH	SH/P	PRSH	RSH
100	2,594	1,132	2,01	0,763	1,892	0,619
200	7,935	4,606	5,985	3,179	5,917	2,181
500	34,14	26,408	25,175	19,397	25,178	11,764
1000	88,195	80,656	61,176	57,503	61,27	32,277
2000	165,89	164,225	115,345	114,104	112,954	68,176

Rysunek 4: Czasy wykonania [s] dla zbioru danych letter-recognition.



Rysunek 5: Wykres czasów wykonania dla zbioru danych letter-recognition.

4.3.3 Krkopt

Jako atrybuty badane zostały wybrane wszystkie atrybuty poza decyzyjnym, tzn. 6 atrybutów. W przypadku pliku zawierającego 2000 wierszy program stwierdził zależność funkcyjną od dowolnego atrybutu, co spowodowane było identyczną wartością atrybutu decyzyjnego dla wszystkich sprawdzonych wierszy. Gdy pojawiły się inne wartości atrybutu decyzyjnego, program zwrócił w dla każdego wywołania zależność od całego zbioru badanych atrybutów, co jest zgodne z przewidywaniami ze względu na charakter danych.

wiersze	H	H/P	SH	SH/P	PRSH	RSH
2000	30	16	14	18	19	16
5000	144	140	140	136	138	128
10000	317	368	320	292	298	294
15000	480	472	476	477	474	480
20000	666	658	654	682	652	687
25000	851	836	840	835	826	857
28057	936	944	935	937	946	959

Rysunek 6: Czasy wykonania [ms] dla zbioru danych krkopt.

Rysunek 7: Wykres czasów wykonania dla zbioru danych krkopt.

5 Wnioski

Manualna weryfikacja wyników programu na bardzo małych zbiorach danych świadczy o poprawności utworzonej implementacji. Z analizy czasów wykonania różnych wariantów algorytmu w funkcji liczby wierszy w zbiorze testowym nasuwa się wniosek, że warianty wykorzystujące podziały uproszczone (*stripped*) działają dla większości użytych zbiorów testowych szybciej niż warianty wykorzystujące podziały podstawowe, natomiast wolniej niż warianty oparte na podziałach uproszczonych zredukowanych (*reduced stripped*). Wariant *RSH* wykonywał się na zbiorze *letter-recognition* ponad dwukrotnie szybciej od wariantu *H* i około 1.5 razy szybciej od wariantu *SH*. Nie stwierdzono natomiast wyraźnego wpływu opcjonalnego przycina-

nia na szybkość działania algorytmu. Wnioski te odbiegają od przedstawionych w [1], gdzie wariant *RSH* plasował się gorzej od pozostałych. Rozbieżność tę autorzy tłumaczą różnicami w użytych technikach implementacji i/lub użytych kontenerach i strukturach danych.

Literatura

[1] Marzena Kryszkiewicz, Piotr Lasek, *FUN: Fast Discovery of Minimal Sets of Attributes Functionally Determining a Decision Attribute*

[2] Rakesh Agrawal and Ramakrishnan Srikant, *Fast algorithms for mining association rules*, Proc. of 20th Intl. Conf. on VLDB, 1994

[3] Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets.html`