Metody bioinformatyki
„Mieszaniny DNA” – dokumentacja końcowa

Piotr Jastrzębski
Marcin Nazimek

1 Treść projektu

Przy badaniu DNA dla celów kryminalistycznych określa się warianty w wybranych miejscach genomu (nazywanych markerami). Projekt zakładał napisanie aplikacji, która dla danego profilu mieszaniny (obejmującego wiele markerów) oblicza wszystkie możliwe profile dla dwu osób i dla trzech osób, a następnie określa poprawne profile drugiej osoby (zakładając, że w mieszaninie będą dwie osoby i profil pierwszej osoby jest znany), lub drugiej i trzeciej osoby (zakładając, że w mieszaninie będą trzy osoby).

2 Zrealizowana funkcjonalność

Projekt polegał na stworzeniu programu, w którym zaimplementowany będzie algorytm generacji na podstawie mieszaniny DNA profili genetycznych osób nieznanych w sytuacji, gdy znane są profile niektórych osób, których DNA występuje w mieszaninie. Przygotowany projekt wypełnia stawiane mu wymagania nawet z nawiązką. Rozszerzyliśmy zadany problem (mieszanina DNA 2 osób, 1 znana albo mieszanina DNA 3 osób, 1 znana) do problemu wyszukiwania dowolnej liczby profili podejrzanych przy dowolnej liczbie znanych osób.

Podczas parsowania pliku, jak założono, sprawdzane jest:

- czy każdy z profili znanych osób zawiera tyle samo markerów
- czy te same markery różnych osób również mają taki sam rozmiar
- czy mieszanina zawiera tyle samo markerów, co każdy z elementów.

Na dalszym etapie - już podczas generowania zbiorów wymaganych, ale jeszcze przed wyszukiwaniem profili podejrzanych, sprawdzane jest, czy możliwe jest, że dla danej liczby osób znanych i poszukiwanych zbiór wymagany jest zbyt duży. Taka sytuacja świadczyłaby o tym, że w mieszaninie znajduje się

DNA większa liczby osób niż założono. Wprowadzone zabezpieczenia pozwalają uniknąć fałszywych oskarżeń osób niezwiązanych ze sprawą i znacznie podnoszą wiarygodność wyników.

Projekt został napisany w Javie i podzielony został, zgodnie ze standardem na pakiety agregujące zbiory funkcjonalności. W pakiecie *.mbi znajdują się klasy odpowiedzialne za wywoływanie odpowiednich metod pobocznych oraz przechowywanie uzyskanych z pliku wejściowego danych, a także klasa `Main` programu. Pakiet *.test zawiera jedynie testy jednostkowe przygotowane za pomocą biblioteki *JUnit*[3]. W pakiecie *.utils zgromadzono klasy pomocnicze odpowiedzialne np. za logowanie zdarzeń lub obsługę plików. Strukturę drzewa projektu przedstawiono poniżej. W katalogu `src` znajdują się źródła, a w `data` przykładowe dane wejściowe.

- src
  - pl.edu.pw.elka.pjastrz2.mbi
    - Evidence.java
    - EvidenceContainer.java
    - Main.java
  - pl.edu.pw.elka.pjastrz2.mbi.tests
    - CombinationsWithRepetitionsTest.java
    - EvidenceContainerTest.java
    - ParameterizedCorrectPathLoadTest.java
    - ParameterizedWrongPathLoadTest.java
    - SpeedTest.java
    - VarationsWithRepetitionTest.java
  - pl.edu.pw.elka.pjastrz2.mbi.utils
    - ArrayMaker.java
    - FileOperator.java
    - ListOfIntegersComparator.java
    - ListOfListOfIntegersComparator.java
    - Log.java
    - ParsingException.java
    - PermThread.java
    - StructurePrinter.java
- ...
- data
  - input0.txt
  - ...
  - inputN.txt

3 Założenia projektowe

Program zrealizowany został jako aplikacja konsolowa, która pozwala użytkownikowi na podanie ścieżki do pliku zawierającego dane. Na wyjściu, w zależności od sytuacji, pojawia się informacja o ścieżce pod jaką zapisane zostały wyniki analizy lub informacja o błędzie poprawności pliku wejściowego. Proces analizy mieszaniny składa się z poniższych etapów:

- Wczytanie danych – profil mieszaniny, liczba osób, których DNA jest w niej obecne oraz profile osób znanych będą podane w pliku wejściowym. Sparsowane dane zostają przekazane do funkcji odpowiedzialnych za generację profili
- Analiza danych wejściowych – sprawdzanie danych pod kątem poprawności zarówno składniowej jak i logicznej
- Generacja profili – generacja profili osób, których DNA może być obecne w mieszaninie
- Wydrukowanie danych – zapis do pliku możliwych profili osób brakujących w mieszaninie

4 Algorytm generacji profili

Weźmy pod uwagę przykład kiedy mamy mieszaninę $X\,\{1, 2, 3, 4\}$ i wiemy, że są w niej zawarte DNA 3 osób. Znamy profil jednej osoby: $X\,\{1, 2\}$

1. Na podstawie zadanej mieszaniny i profili znanych osób obliczamy różnicę zbiorów: $\{1, 2, 3, 4\} - \{1, 2\} = \{3, 4\}$
2. Wiemy teraz, że wśród profili pozostałych dwóch osób musi znajdować się zarówno 3 i 4. Ich zestawienie musi pasować do jednego z poniższych wzorców:
   (3,i), (4,j)
   (3,4), (i,j)
   (i,j), (3,4)
3. Generujemy wszystkie możliwe pary permutacji:
   (1,1), (3,4)
   (1,1), (3,4)
   (1,2), (3,4)
   (1,3), (1,4)
   (1,3), (2,4)
   (1,3), (3,4)
   (1,3), (4,4)

(1,4), (2,3)
(1,4), (3,3)
(1,4), (3,4)
(2,2), (3,4)
(2,3), (2,4)
(2,3), (3,4)
(2,3), (4,4)
(2,4), (3,3)
(2,4), (3,4)
(3,3), (3,4)
(3,3), (4,4)
(3,4), (3,4)
(3,4), (4,4)

Litera X jest tutaj wyznacznikiem mówiącym o typie mieszaniny. W pliku wejściowym możemy zdefiniować dowolną liczbę różnych mieszanin jak również dowolną liczbę profili osób. Ich parowanie przebiegać będzie na podstawie znalezienia identycznego wyróżnika umieszczonego przed profilem lub mieszaniną. Wynikiem są zatem listy możliwych permutacji dla każdego typu danych wejściowych.

5 Badania i ocena przydatności

Mówiąc o złożoności czasowej przygotowanego rozwiązania nie można założyć, że zależy ona wprost od rozmiaru mieszaniny DNA. Jest to raczej funkcja uwikłana, w której główną rolę odgrywają rozmiar „zbioru obowiązkowego", a przede wszystkim rozmiar i liczba poszczególnych markerów profili. Ważna jest także liczba typów markerów, ale można założyć, że przy dobrze zrównoleglonych obliczeniach wraz ze wzrostem liczby typów, złożoność zwiększa się liniowo, ewentualnie może zostać wyrażona sumą, jak widać we wzorze 1.

Na potrzeby badania wydajności przygotowany został oddzielny test (klasa `SpeedTest`), którego zadaniem jest tylko i wyłącznie generowanie wszystkich poprawnych logicznie danych wejściowych aż do zadanych wartości rozmiaru zbioru i do maksymalnej liczby poszukiwanych profili. Test pozwala także na zestawianie uzyskanych czasów wraz z parametrami wejściowymi na wyjście.

Po agregacji wyników i posortowaniu ich po rosnącym czasie wykonania, na rysunku 1 widać, że podejrzenia, co do zależności złożoności obliczeniowej od ww. parametrów były słuszne. Wynika to z tego, że współczynnik oznaczony na rysunku jest w praktyce rozmiarem generowanych permutacji. Zważywszy na fakt, że w procesie generowania permutacji wyniki są także odpowiednio sortowane, aby zapewnić przejrzystość prezentacji uzyska-

	A	B	C	D	E	F
1	a	b	c	d	factor (c*d-b)	t [ms]
49	3	1	2	2	3	**1**
50	4	1	2	2	3	**1**
51	2	1	2	2	3	**1**
52	4	0	3	1	3	**1**
53	3	3	3	2	3	**1**
54	4	4	3	2	2	**1**
55	2	0	2	2	4	**2**
56	4	0	4	1	4	**2**
57	3	2	3	2	4	**3**
58	5	0	4	1	4	**3**
59	3	0	3	1	3	**3**
60	4	3	3	2	3	**4**
61	1	0	1	1	1	**5**
62	3	0	2	2	4	**13**
63	4	2	3	2	4	**13**
64	3	1	3	2	5	**18**
65	5	0	2	2	4	**20**
66	4	0	2	2	4	**34**
67	4	1	3	2	5	**49**
68	3	0	3	2	6	**107**
69	4	4	4	2	4	**312**
70	4	0	3	2	6	**312**
71	4	3	4	2	5	**992**
72	5	0	3	2	6	**1441**
73	4	2	4	2	6	**3942**
74	4	1	4	2	7	**13390**
75	4	0	4	2	8	**56305**

Rysunek 1: Wyniki symulacji (pominięto pozycje z $t \approx 0$)

nych rezultatów, można założyć, że osiągane czasy są zadowalające. Punktu ewentualnych poprawek można by poszukiwać w wykorzystanej zewnętrznej bibliotece generowania permutacji i próbie optymalizacji zagnieżdzonej pętli `for`.

$$f(x) \in O(\sum_{i \in \{A..N\}} g(a_i, b_i, c_i, d_i)), \tag{1}$$

gdzie: a to rozmiar alfabetu, b to rozmiar „zbioru obowiązkowego", c to rozmiar markera a d to liczba poszukiwanych osób.

6 Języki i narzędzia

Aplikacja została stworzona w języku Java w środowisku Eclipse[1]. Ze względu na wysoką przenośność, możliwość szybkiego jej przygotowania oraz bogate wbudowane biblioteki zdecydowaliśmy się na ten wybór. W toku pracy pomocne okazały się również zewnętrzne biblioteki *combinatoricslib 2.0*[2] (biblioteka obiektów kombinatorycznych) oraz biblioteki do testów jednostkowych: główna - *junit 4.11* wraz z *hamcrest 4.11* odpowiedzialną za budowanie testów.

7 Testowanie

Testy aplikacji zostały wykonane za pomocą biblioteki JUnit. Przy pomocy testów jednostkowych zapewniliśmy, że wyniki zwracane przez aplikację są co do założeń zgodne, a *test-driven development* pozwolił na wczesne wykrywanie ewentualnych problemów i pracę nad poszczególnymi funkcjonalnościami programu z bieżącą kontrolą postępu. Dzięki temu, przygotowanie wersji końcowej sprowadziło się w praktyce do złączenia testowanych metod w całość.

8 Podsumowanie

Podczas realizacji projektu mieliśmy okazję zapoznać się z problemem badania DNA dla celów kryminalistycznych od strony technicznej. Jak wspomniano na wstępie, przygotowany program spełnia z nawiązką wymagania określone w specyfikacji. Zdecydowaliśmy się jednak na implementację pozwalającą na wyznaczenie nieznanych profili dla dowolnej liczby osób. Niestety w parze z większą funkcjonalnością rozwiązania idzie dość duża złożoność obliczeniowa, o której wspomniano w punkcie 5. Jak pokazano, w praktyce, dla n większego niż 10 czas działania programu potrzebny do uzyskania poprawnych wyników może być bardzo długi.

Literatura

[1] *Eclipse*
`http://www.eclipse.org`

[2] *combinatoricslib – biblioteka obiektów kombinatorycznych*
`https://code.google.com/p/combinatoricslib/`

[3] *Junit – framework testów jednostkowych*
`http://junit.org`

[4] *Wikipedia*
`http://en.wikipedia.org/wiki/{DNA|DNA_marker|DNA_profiling}`