POLITECHNIKA WARSZAWSKA

Metody eksploracji danych w odkrywaniu wiedzy

PROJEKT

„Implementacja algorytmu PrefixSpan do odkrywania wzorców sekwencyjnych”

Autor:

Jakub Stępniak

Nr indeksu: 297389

Prowadzący:

Dr inż. Dominik Ryżko

Warszawa 2023/24

1. Wprowadzenie i definicja problemu

Sekwencja to uporządkowany zbiór elementów, gdzie kolejność występowania tych elementów ma kluczowe znaczenie. Przykłady sekwencji można odnaleźć w różnych dziedzinach, od genetyki (np. sekwencje DNA), przez analizę finansową (np. serie czasowe cen akcji), aż po przetwarzanie języka naturalnego (np. zdania jako sekwencje słów).

Aby ująć problem w bardziej formalny sposób. Najmniejszą składową sekwencji jest element (ang. Item). Elementem, może być wszystko w zależności od problematyki jaką się zajmujemy.

Wiele elementów może występować razem tworząc zbiory elementów (ang. Itemset), wewnątrz zbioru elementy nie mogą się powtarzać.

$$I = \{a, b, c, d, e\}$$

gdzie:
- *a,b,c,d,e* – kolejne elementy
- *I* – zestaw elementów

Sekwencja jest to uporządkowana lista zbiorów elementów:

$$S =< X_1, X_2, \dots, X_n >; \quad gdzie\ X_j \subseteq I\ dla\ j \in \{1, 2, \dots, n\}$$

Wewnątrz sekwencji możemy wyróżnić Podsekwencję (ang.subsequence). Podsekwencja jest to część jakiejś sekwencji, dla której każdy zestaw elementów zawiera się w odpowiadającym jemu zbiorze elementów całej sekwencji.

W kontekście baz danych, sekwencyjna baza danych to specjalizowany typ bazy danych zaprojektowany do przechowywania, zarządzania i analizowania danych sekwencyjnych. Takie bazy danych muszą efektywnie radzić sobie z dużymi ilościami uporządkowanych danych oraz zapewniać narzędzia do ich zapytań i analizy.

$$D = \{S_1, S_2, \dots, S_n\}$$

Przykład sekwencyjnej bazy danych:

Sequence_id	Sequence
10	$\langle a(abc)(ac)d(cf)\rangle$
20	$\langle (ad)c(bc)(ae)\rangle$
30	$\langle (ef)(ab)(df)cb\rangle$
40	$\langle eg(af)cbc\rangle$

Rysunek 1 Sekwencyjna baza danych [1]

Problem wyszukiwania sekwencji w takich bazach danych polega na znalezieniu wzorców lub częstych podsekwencji w zbiorze danych, które spełniają określone kryteria. Możemy na przykład szukać powtarzających się wzorców zakupów w danych transakcyjnych klientów supermarketu, aby lepiej zrozumieć ich zachowania zakupowe i optymalizować strategie marketingowe.

Zastosowania wyszukiwania sekwencji są wszechstronne i obejmują między innymi bioinformatykę, gdzie analiza sekwencji DNA może pomóc w identyfikacji genów związanych z pewnymi chorobami, analizę finansową do przewidywania trendów rynkowych, czy bezpieczeństwo komputerowe, gdzie sekwencje typów ataków mogą wskazywać na potencjalne zagrożenia.

Głównym problemem przy wyszukiwaniu wzorców sekwencyjnych jest czas potrzebny na ich uzyskanie. W podejściu naiwnym, aby uzyskać wyniki, należy porównać każdy wzorzec z każdym innym we wszystkich rekordach bazy danych. W bazach, które bez problemu zawierają miliony rekordów, takie podejście uniemożliwia uzyskanie jakichkolwiek efektywnych wyników. W związku z tym, zostało opracowanych wiele algorytmów pozwalających na wydajniejsze przeprowadzanie tego procesu. Jednym z proponowanych rozwiązań jest algorytm PrefixSpan.

2. Charakterystyka PrefixSpan

Aby zrozumieć działanie algorytmu PrefixSpan musimy dodatkowo wprowadzić pojęcie Projekcji Bazy Danych. Jest to stworzenie kopii bazy bez pewnych elementów. Dokonujemy tego w rekurencyjny sposób zwiększając ilość elementów tworzących podsekwencję. Bazy te są coraz mniejsze przez to, że usuwamy podsekwencje nie spełniające warunku wsparcia w każdej iteracji oraz usuwamy same sprawdzane podsekwencję oraz wszystkie jej poprzedniki.

SID	Sequence
1	$\langle\{a,b\},\{c\},\{f,g\},\{g\},\{e\}\rangle$
2	$\langle\{a,d\},\{c\},\{b\},\{a,b,e,f\}\rangle$
3	$\langle\{a\},\{b\},\{f,g\},\{e\}$
4	$\langle\{b\},\{f,g\}\rangle$

Rysunek 2 Oryginalna sekwencyjna baza danych [2]

SID	Sequence
1	$\langle\{\_,b\},\{c\},\{f,g\},\{g\},\{e\}\rangle$
2	$\langle\{\_,d\},\{c\},\{b\},\{a,b,e,f\}\rangle$
3	$\langle\{b\},\{f,g\},\{e\}$
4	$\langle\{b\},\{f,g\}\rangle$

Rysunek 3 Projekcja sekwencyjnej bazy danych wzorca <{a}> [2]

SID	Sequence
1	$\langle\{c\},\{f,g\},\{g\},\{e\}\rangle$
2	$\langle\{\_,e,f\}\rangle$

Rysunek 4 Projekcja sekwencyjnej bazy danych wzorca <{a, b}> [2]

Pseudokod algorytmu przedstawia się następująco:

Input : A sequence database S, and the minimum support threshold min_sup

Output : The complete set of sequential patterns

Method: Call *PrefixSpan*$(\langle\ \rangle, 0, S)$

Subroutine: *PrefixSpan* $(\alpha, l, S|_\alpha)$

Parameters: α : a sequential pattern; l : the length of α ; $S|_\alpha$: the α-projected database, if $\alpha \neq \langle\ \rangle$; otherwise, the sequence database S.

Method:

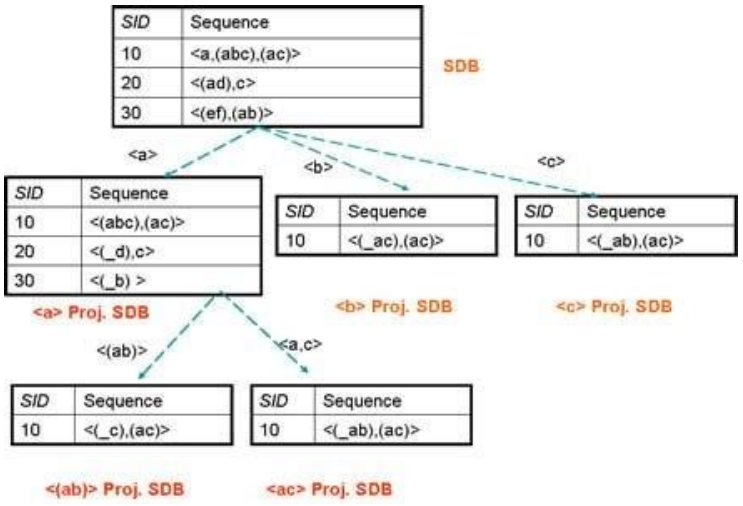
1. Scan $S|_\alpha$ once, find the set of frequent items b such that
   a) b can be assembled to the last element of α to form a sequential pattern; or
   b) $\langle b \rangle$ can be appended to α to form a sequential pattern.
2. For each frequent item b, append it to α to form a sequential pattern α', and output α';
3. For each α', construct α'-projected database $S|_{\alpha'}$, and call *PrefixSpan* $(\alpha', l+1, S|_{\alpha'})$.

Rysunek 5 PrefixSpan Pseudocode [3]

Dane wejściowe algorytmu to sekwencyjna baza danych oraz parametr określony przez użytkownika, który definiuje minimalne wsparcie.

Wynikiem działania algorytmu jest zestaw podsekwencji, których wsparcie jest równe lub większe niż wartość określona przez użytkownika.

Algorytm wykonuje swoje zadanie w sposób rekurencyjny, konstruując drzewo podsekwencji, przy czym w kolejnych węzłach pomija już sprawdzone podsekwencje, oraz dokonuje przycinania, to znaczy eliminuje podsekwencje, które mają wsparcie mniejsze niż ustalony próg. Dla każdego węzła drzewa tworzona jest nowa projekcja bazy danych sekwencyjnej, a proces jest kontynuowany aż do osiągnięcia zbioru pustego podsekwencji, które spełniają wymogi minimalnego wsparcia.



Rysunek 6 Depth-first search w PrefixSpan [4]

To co wyróżnia ten algorytm to efektywność działania nawet na dużych zbiorach (działa znacznie szybciej niż np. klasyczny GSP), łatwość implementacji oraz dodawania własnych zależności takich jak minimalna przerwa w sekwencji czy jej długość. Dużym jej plusem bezpośrednio wpływającym na wydajność jest to, że przeprowadza poszukiwania wzorców które faktycznie istnieją w bazie danych przez co nie traci zasobów na niepotrzebne obliczenia.

Minusem tego algorytmu, jest duże zużycie pamięci poprzez tworzenie nowej bazy danych przy każdej projekcji, co może być jednak poprawione poprzez modyfikację tego mechanizmu.

3. Opis implementacji

Główna część implementacji jest to klasa PrefixSpan:

```python
class PrefixSpan:
    def __init__(self, input_filepath: Path, output_filepath: Path, min_support: int):
        self.input_filepath = input_filepath
        self.output_filepath = output_filepath
        self.min_support = min_support
        self.sequences = self.read_data(input_filepath.as_posix())
        self.frequent_patterns = []

    @staticmethod
    def read_data(filename: str):
        with open(filename, 'r') as file:
            sequences = []
            for line in file:
                sequence = [item.split() for item in line.strip().split('-1')[:-1]]
                sequences.append(sequence)
            return sequences

    def prefix_span(self, prefix, projected_db):
        # Count all items and their supports in the projected_db
        item_counts = {}
        for sequence in projected_db:
            found_items = set()
            for itemset in sequence:
                for item in itemset:
                    if item not in found_items:
                        if item in item_counts:
                            item_counts[item] += 1
                        else:
                            item_counts[item] = 1
                        found_items.add(item)

        # Filter items by minimum support and recursively explore extensions
        frequent_items = [(item, count) for item, count in item_counts.items() if count >= self.min_support]
        for item, _ in sorted(frequent_items, key=lambda x: x[1], reverse=True):
            new_prefix = prefix + [item]
            self.frequent_patterns.append((new_prefix, item_counts[item]))
            suffix_db = self._build_suffix_db(new_prefix, projected_db)
            self.prefix_span(new_prefix, suffix_db)

    def _build_suffix_db(self, prefix, projected_db):
        suffix_db = []
        for sequence in projected_db:
            for index, itemset in enumerate(sequence):
                if prefix[-1] in itemset:
                    suffix = []
                    for future_index in range(index + 1, len(sequence)):
                        suffix.append(sequence[future_index])
                    if suffix:
                        suffix_db.append(suffix)
                    break
        return suffix_db
```

Rysunek 7 Pierwsza część implementacji algorytmu PrefixSpan

```
def run(self):
    initial_db = [seq for seq in self.sequences]
    self.prefix_span([], initial_db)
    return self.frequent_patterns

def write_frequent_patterns_to_file(self):
    if self.frequent_patterns:
      with open(self.output_filepath.as_posix(), 'w') as file:
          for pattern, support in self.frequent_patterns:
              file.write(' '.join(pattern) + ' -1 #SUP: ' + str(support) + '\n')
    else:
        logger.warning("Trying to save outputfile if there is no frequent found, probably algorithm was not run")
```

Rysunek 8 Druga część implementacji PrefixSpan

Inicjalizacja obiektu ma za zadanie przygotowanie go, ustawienie odpowiednich zmiennych klasowych oraz wczytania sekwencji z pliku.

Algorytm uruchamiany metodą ***run().***

Wewnątrz tej metody zaczynamy rekurencyjny algorytm PrefixSpan na wczytanych w poprzednim kroku sekwencjach. W pierwszym kroku zliczamy wsparcie dla wszystkich elementów projektowanej bazy danych. Następnie sprawdzamy warunek minimalnego wsparcia i filtrujemy w ten sposób elementy rozważane do tworzenia sekwencji o minimalnych wsparciu. Następnym krokiem jest stworzenie projekcji bazy danych bez sprawdzanego już elementu (w przyszłych interacjach podsekwencji) oraz elementów (podsekwencji), które nie spełniają już w tym momencie warunku minimalnego wsparcia. Na nowo utworzonej bazie danych przeprowadzamy ponownie ten sam algorytm.

Algorytm kontynuujemy, aż do uzyskania pustej podsekwerncji spełniającej warunki minimalnego wsparcia.

Wykorzystanie tego algorytmu wygląda w ten sposób:

```
prefix_span = PrefixSpan(input_sequence_filepath, output_filepath, min_support)
start_time = time.perf_counter()
prefix_span.run()
end_time = time.perf_counter()
print(f"Time of custom prefix time: {end_time - start_time} with min support: {min_support}")
prefix_span.write_frequent_patterns_to_file()
```

Rysunek 9 Przykład wykorzystania implementacji algorytmu PrefixSpan

Inną częścią implementacji było wykorzystanie gotowego algorytmu z paczki SMPF, zaimplementowanej w Javie wraz z wieloma optymalizacjami:

```
def calculate_percentage_min_support(min_support: int, input_filepath: Path) -> float:
    with open(input_filepath, 'r') as file:
        seq_length = len(file)
        return (min_support * 100) / seq_length if seq_length > 0 else 0
```

Rysunek 10 Funkcja obliczająca procentowy udział minimalnego wsparcia w liczbie wszystkich sekwencji

```
min_support_perc = calculate_percentage_min_support(min_support, input_sequence_filepath)
spmf_prefix_span = Spmf("PrefixSpan", input_filename=input_sequence_filepath.as_posix(),
            output_filename=spmf_output_filepath.as_posix(), arguments=[0.0001])
start_time = time.perf_counter()
spmf_prefix_span.run()
end_time = time.perf_counter()
print(f"Time of custom prefix time: {end_time - start_time} with min support: {min_support}")
```

Rysunek 11 Przykład wykorzystania gotowej implementacji algorytmu PrefixSpan z paczki SMPF

Aby wykorzystać parametr min_support oznaczający minimalne wsparcie musiałem dla każdego pliku wyznaczyć ilość sekwencji, którą wykorzystałem do podzielenia tego parametru. Wykorzystując ten procent uzyskałem analogiczny efekt jak z parametrem minimalnego wsparcia z moją implementacją.

4. Instrukcja dla użytkownika

Są dwa sposoby wykorzystania implementacji stworzonej podczas projektu, w zależności czy chcemy wykorzystywać własne zasoby obliczeniowe czy chmurę od firmy Google zapewnianej w środowisku Google Colab. Różnica jest kluczowa ze względy na moc zasobów. Jeśli ktoś dysponuje dobrym sprzętem daje on zdecydowanie lepsze możliwości niż darmowy Google Colab, który ma tylko 12 GR (który w tym ćwiczeniu jest kluczowy) jednak pozwala na samo sprawdzenie działania algorytmów na małej ilości sekwencji.

1. Własne zasoby

Zależności:

- Ubuntu (najlepiej 22.04)

- Python >=3.10

Pozwolenie na od użytkownika na działania wymagające sudo (administratora) bądź ręczne zainstalowanie zależności zwiąanych z biblioteką SMPF (gotową implementacją w Javie).

1. Wchodzimy do folderu przechowującego repozytorium
2. Wpisujemy komendę „./create_venv.sh”
3. Wpisujemy komendę „jupyter notebook”
4. Wybieramy notatnik o nazwie: „PrefixSpan.ipynb”
5. Wykonujemy eksperymenty jakie chcemy korzystając z komórek informujących nas co, która komurka robi

2. Google Colab

Dodajemy plik z repozytorium o nazwie „ColabPreficSpan.ipynb” do Google Colab na swoim dysku. Następnie odpalamy wszystkie komórki, w którym między innymi łączymi się ze swoim Dyskiem Google, a następnie pobieramy na niego potrzebne dane wraz z ich rozpakowaniem.

5. Wykorzystane zbiory danych

Zbiory danych na których eksperymenty będą przeprowadzane muszą się charakteryzować różnorodnością, aby móc lepiej zauważyć zależności w działaniu algorytmu względem typu szukanej sekwencji. Główną różnicą w zbiorach danych to rodzaj oraz ilość elementów tworzących sekwencję, średnia długość sekwencji oraz wielkość samego zbioru.

Aby zapewnić jak największą różnorodność planuję wykorzystać zbiory:

Nazwa zbioru	Opis	Liczba sekwencji	Liczba elementów	Średnia długość sekwencji
Kosorak	To jest bardzo duży zbiór danych zawierający 990 000 sekwencji danych o ścieżkach kliknięć z węgierskiego portalu informacyjnego.	990000	41270	8,1
Sign	Zbiór danych zawierający wypowiedzi języka migowego, który obejmuje około 800 sekwencji.	800	267	51,997
Fifa	Dane o ścieżkach kliknięć z witryny internetowej gry FIFA 98.	20450	2990	34,74
E-Shop	Dane o ścieżkach kliknięć dla zakupów online. Zbiór danych zawiera sekwencje kliknięć (ścieżki kliknięć) z internetowego sklepu oferującego odzież dla kobiet w ciąży.	24026	317	62,98
MicroblogPCU	Zbiory danych dotyczące spamu w mikroblogach.	429	50505	296,37

Tabela 1 Zbiory danych planowane do eksperymentów [5]

6. Wyniki eksperymentów

Pierwszy wynik jaki uzyskałem to ogromne zużycie pamięci RAM przez algorytm. Dla odpowiednio wielkiej ilości sekwencji, jej długości oraz małego minimalnego wsparcia moja implementacja algorytmu chodziła bardzo długo do momentu wyczerpania się pamięci RAM (32GB). Gotowa implementacja SMPF zachowywała się zdecydowanie lepiej pod względem pamięci oraz szybkości. Nawet przy ilości sekwencji, która powodowała wyczerpanie się pamięci dla własnej implementacji, PrefixSpan od SMPF mimo, że działał dość długo to nie było widać zauważalnych skoków pamięci.

Kolejnym wynikiem jest potwierdzenie poprawności mojego algorytmu. Uzyskuję identyczne wyniki dla gotowego algorytmu od SMPF oraz mojej implementacji.

Zbiór danych SIGN

Dla minimalnego wsparcia = 10

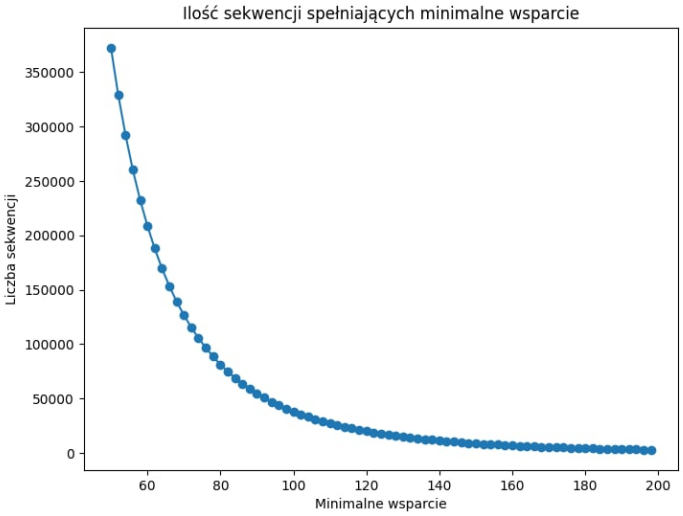
Własne implementacja: Error z powodu braku pamięci RAM

SMPF implementacja:

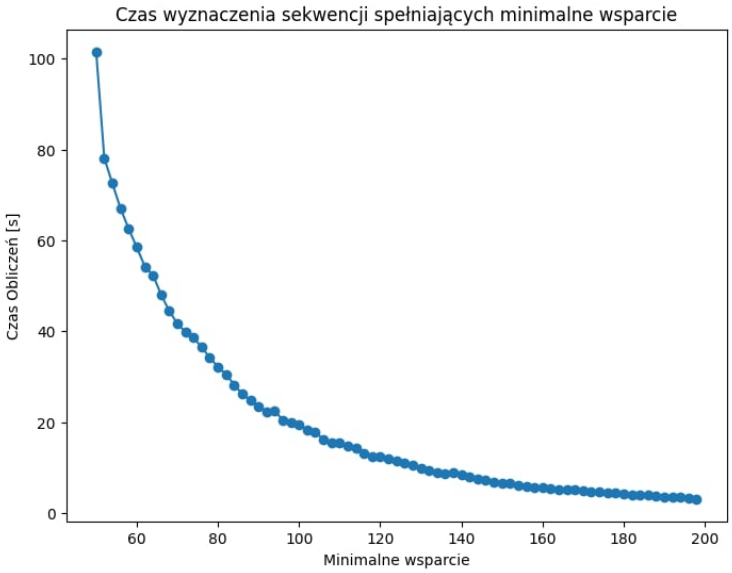
```
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 163988 ms
 Frequent sequences count : 49256533
 Max memory (mb) : 451.3256149291992
 minsup = 10 sequences.
 Pattern count : 49256533
 ============================================================
```

Rysunek 12 Wyniki uzyskane za pomocą implementacji od SMPF dla minimalnego wsparcia = 10

Dla minimalnego wsparcia zaczynającego się od 50 do 200 własna implementacja przedstawia następujące wyniki:



Rysunek 13 Ilość wykrytych sekwencji w zależności od parametru minimalnego wsparcia dla mojej implementacji algorytmu



Rysunek 14 Ilość wykrytych sekwencji w zależności od parametru minimalnego wsparcia dla mojej implementacji algorytmu

Dla implementacji SMPF przy minimalnym wsparciu równym 50 otrzymaliśmy:

```
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 4542 ms
 Frequent sequences count : 372610
 Max memory (mb) : 364.91123962402344
 minsup = 50 sequences.
 Pattern count : 372610
```

Rysunek 15 Wyniki uzyskane za pomocą implementacji od SMPF dla minimalnego wsparcia = 50

W obydwu przypadkach otrzymaliśmy taka samą liczbę sekwencji przy znacznie krótszym czasie.

MicroblogPCU

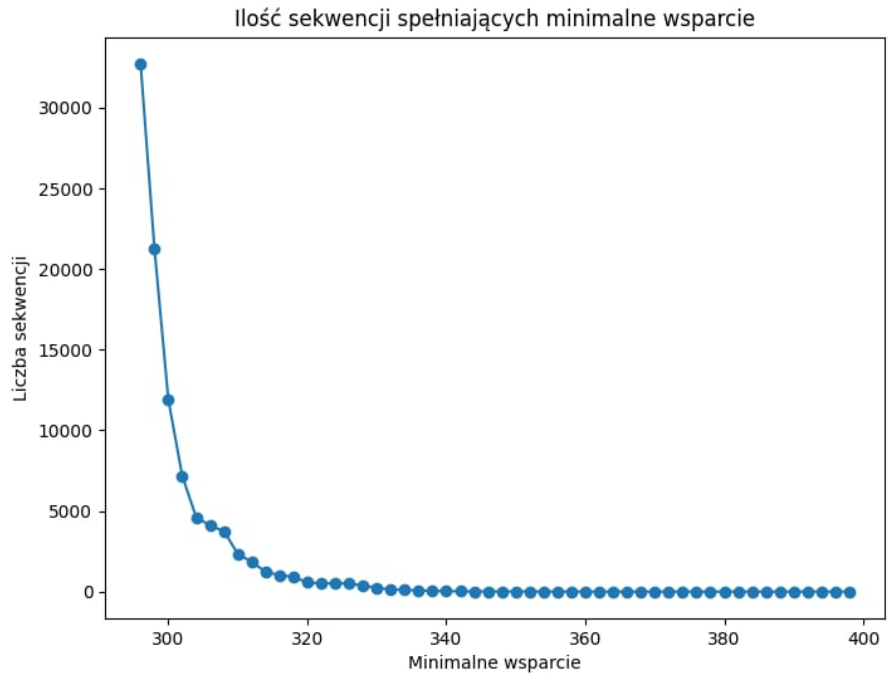
Zbiór danych MicroblogPCU charakteryzuje się mniejszą ilością sekwencji jednak znacznie dłuższych. Dla stosunkowo mały progów minimalnego wsparcia takich jak 50 (przy średniej długości sekwencji równej 296) nie zauważałem nagłego zużycia pamięci RAM, jednak sam proces obliczeń trwał niezwykle długo. Eksperymenty zacząłem więc od progu, który zarazem był średnią – 296. Otrzymałem następujące wyniki.

```
>/home/kuba/repo/MED_PrefixSpan/spmf.jar
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 843950 ms
 Frequent sequences count : 1740007
 Max memory (mb) : 564.2021636962891
 minsup = 296 sequences.
 Pattern count : 1740007
================================================
```

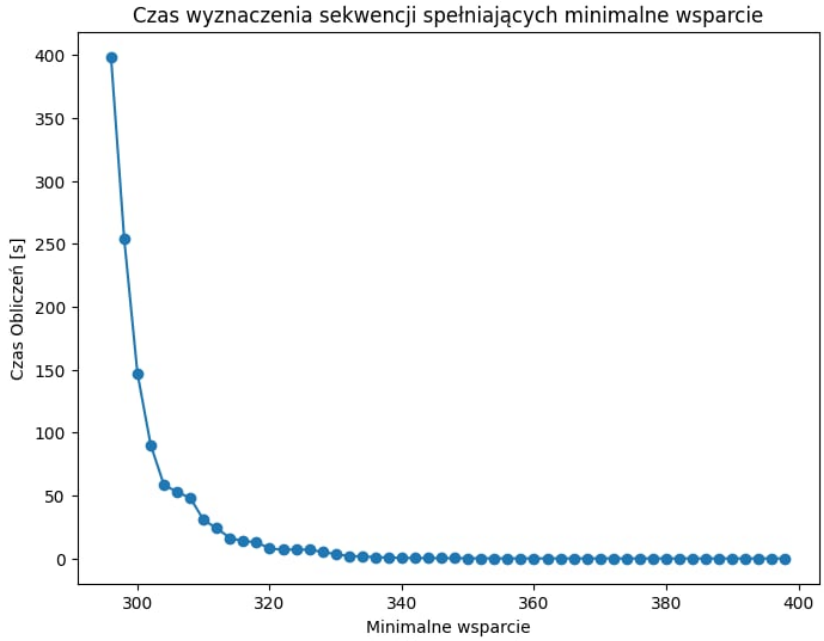
Rysunek 16 Wyniki dla gotowej implementacji PrefixSpan dla wsparcia = 296

```
Time of custom prefix time: 380.45945496097556 with min support: 296, found: 32728 sequences
```

Rysunek 17 Czas eksperymentów dla mojej implementacji przy progu wsparcia 296



Rysunek 18 Liczba znalezionych sekwencji w zależności od progu minimalnego wsparcia dla własnej implementacji



Rysunek 19 Czas obliczeń w zależności od progu minimalnego wsparcia dla własnej implementacji

Fifa

Zbiór danych Fifa różni się od poprzednich głównie tym, że jest w nim znacznie więcej sekwencji. Po próbie obliczeń na wszystkich możliwych musiałem zrezygnować ze względu na bardzo długi czas i rosnące zużycie pamięci RAM, badania ograniczyłem do pierwszych 1000 sekwencji.

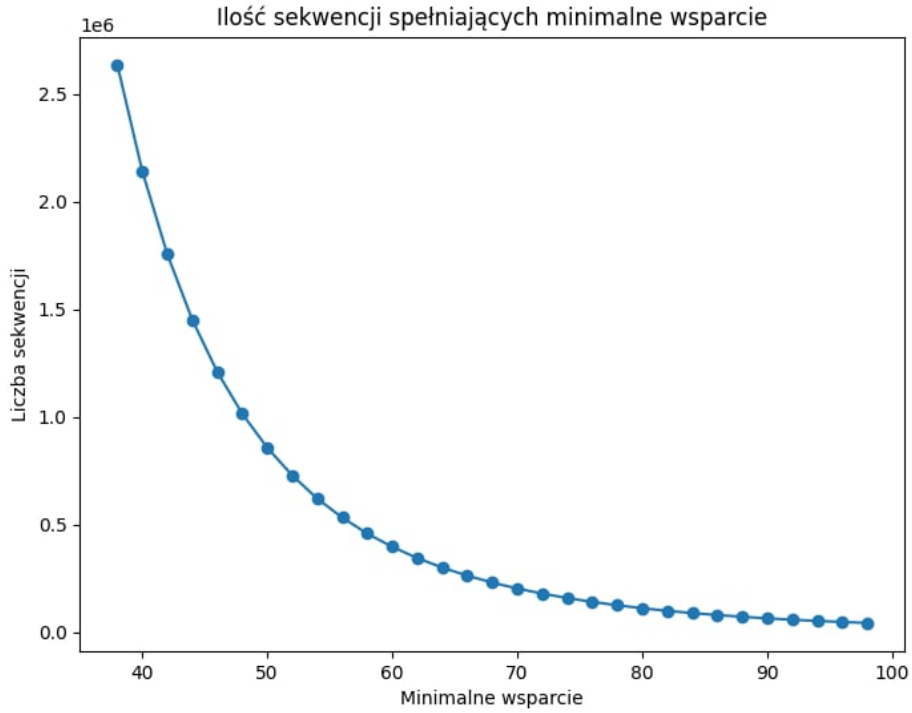
Otrzymałem następujące wyniki:

```
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 47454 ms
 Frequent sequences count : 2635822
 Max memory (mb) : 439.6333923339844
 minsup = 38 sequences.
 Pattern count : 2635822
================================================


Time of smpf prefix time: 47.65360784999211 with min support: 38
```

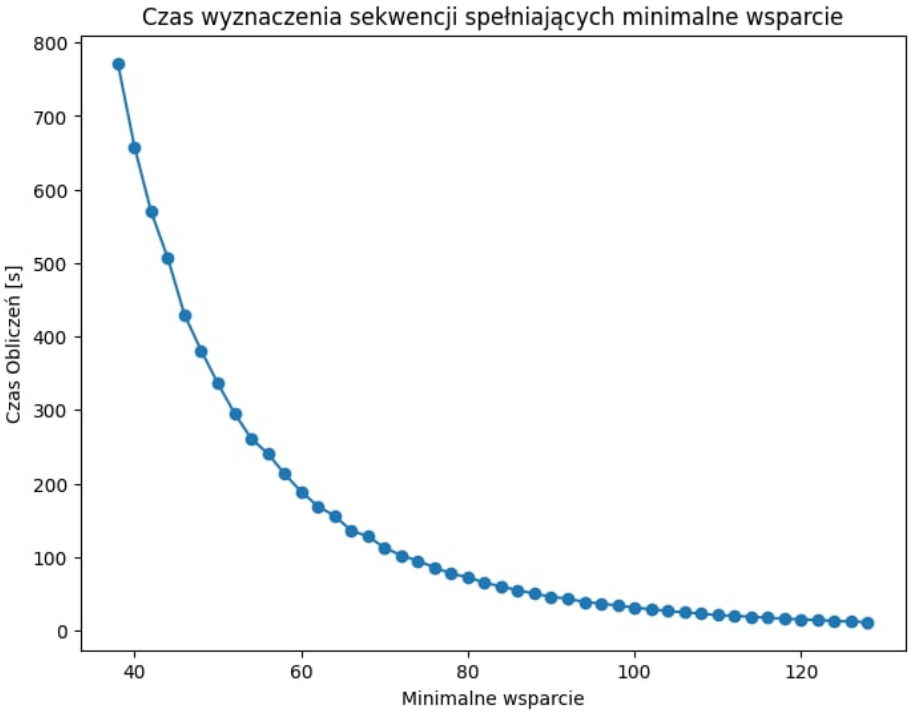
Rysunek 20 Wyniki otrzymane dla PrefixSpan zaimplementowanego w paczce SMPF dla minimalnego wsparcia równego 38

```
Time of custom prefix time: 751.5815233410103 with min support: 38, found: 2635822 sequences
```

Rysunek 21 Wyniki dla własnej implementacji algorytmu



Rysunek 22 Liczba znalezionych sekwencji w zależności od minimalnego wsparcia dla własnej implementacji



Rysunek 23 Czas obliczeń w zależności od minimalnego wsparcia dla własnej implementacji

Kosorak

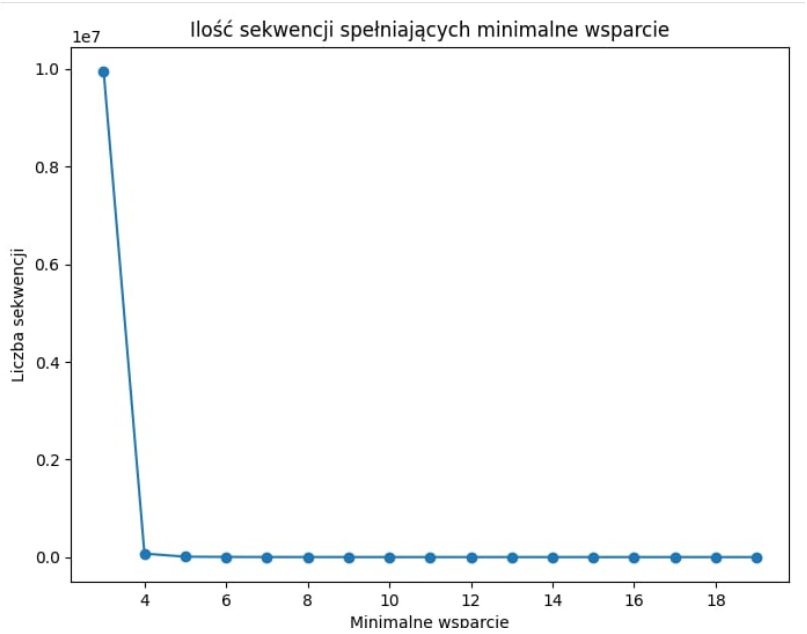
Zbiór danych Kosorak charakteryzował się bardzo dużą ilością danych przy bardzo krótkich sekwencjach. Zdecydowałem się jednak zmniejszyć ilość badanych sekwencji podobnie jak w poprzednim przypadku do 1000 aby łatwiej było porównać je z poprzednimi wynikami. Poniżej przedstawiam otrzymane rezultaty.

```
>/home/kuba/repo/MED_PrefixSpan/spmf.jar
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 15335 ms
 Frequent sequences count : 9952946
 Max memory (mb) : 435.55570220947266
 minsup = 3 sequences.
 Pattern count : 9952946
================================================


Time of smpf prefix time: 15.576141186000314 with min support: 3
```

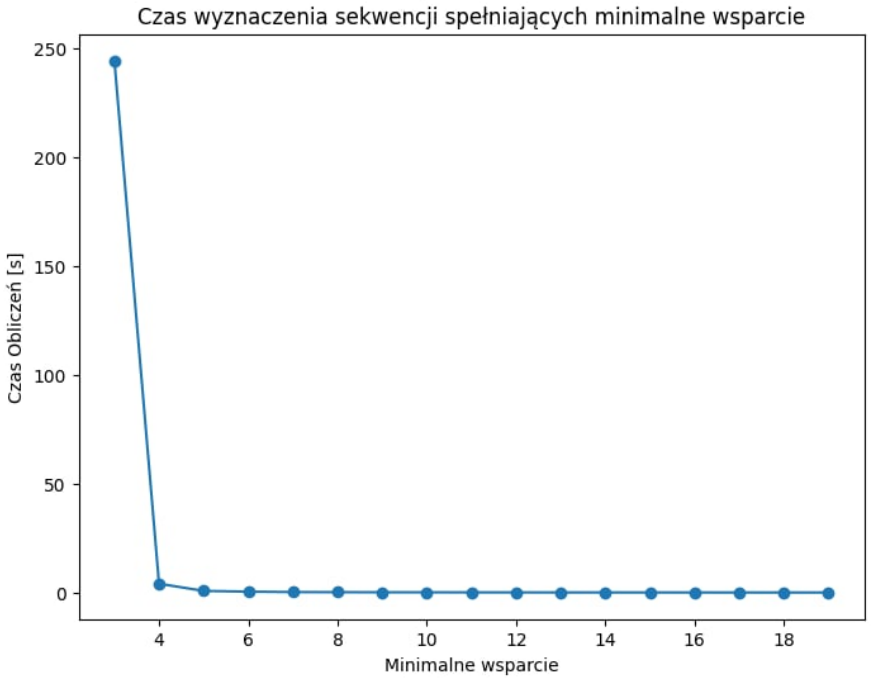
Rysunek 24 Rezultaty dla implementacji z paczki SMPF

```
Time of custom prefix time: 249.26075228099944 with min support: 3, found: 9952946 sequences
```

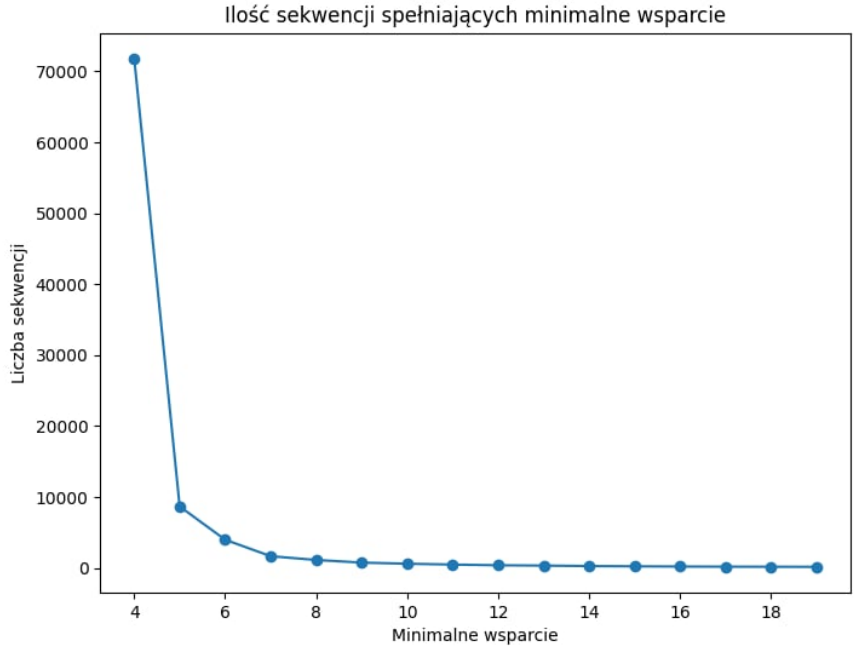
Rysunek 25 Rezultaty dla własnej implementacji



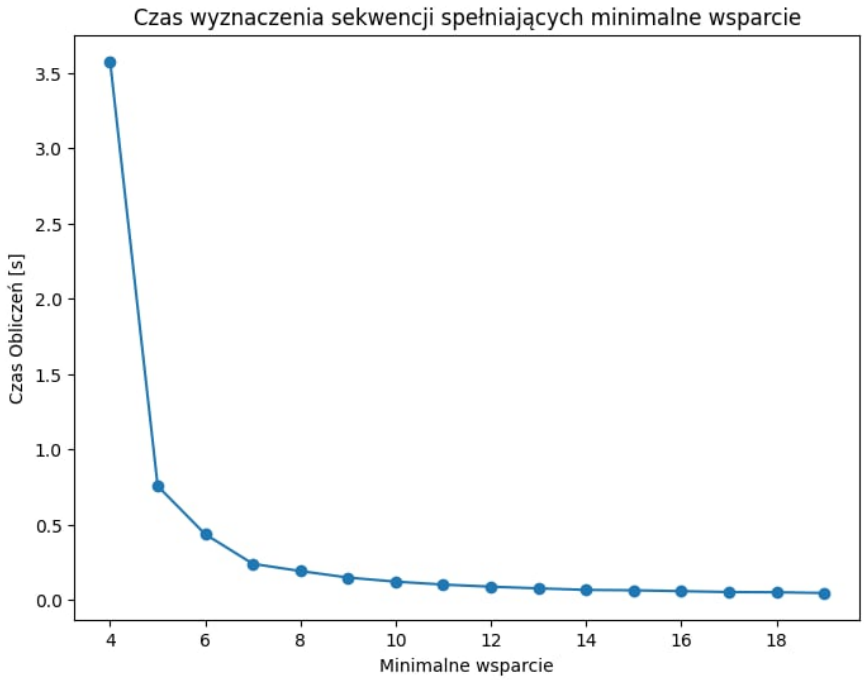
Rysunek 26 Liczba znalezionych sekwencji dla minimalnego progu równego 3



Rysunek 27 Czas obliczeń dla minimalnego progu równego 3



Rysunek 28 Liczba znalezionych sekwencji dla minimalnego progu równego 4



Rysunek 29 Czas obliczeń dla minimalnego progu równego 4

E-Shop

Zbiór danych charakteryzuję się małą różnorodnością tworzących go elementów w porównaniu do ilości sekwencji. Tutaj również ograniczyłem się jednak tylko do pierwszych 1000 sekwencji. Co ciekawe średnia, która wynosi 62 sekwencji w tym przypadku została mocno zmieniona ponieważ wyniki w akceptowalnym czasie otrzymywałem dla progu wsparcia powyżej 200 co pozwala wysnuć wniosek, że początek tego zbioru jest pełen długich sekwencji w przeciwieństwie do jego drugiej części.

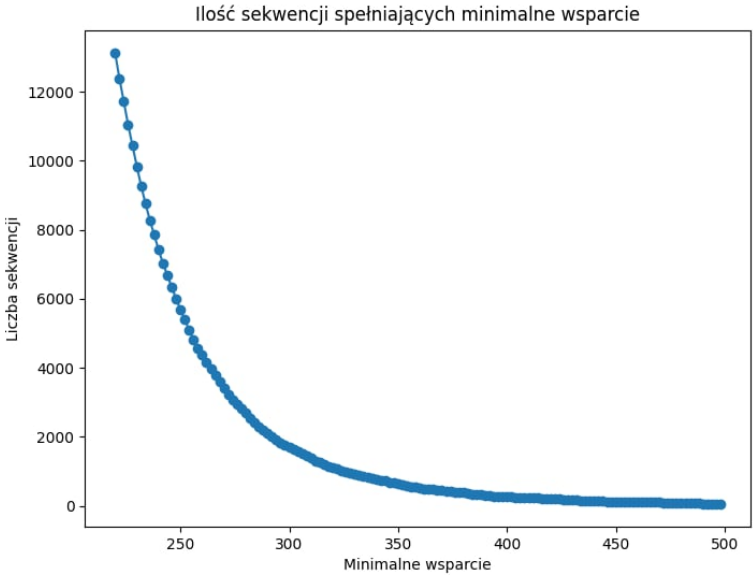
Badania rozpocząłem od progu wsparcia o wartości 220.

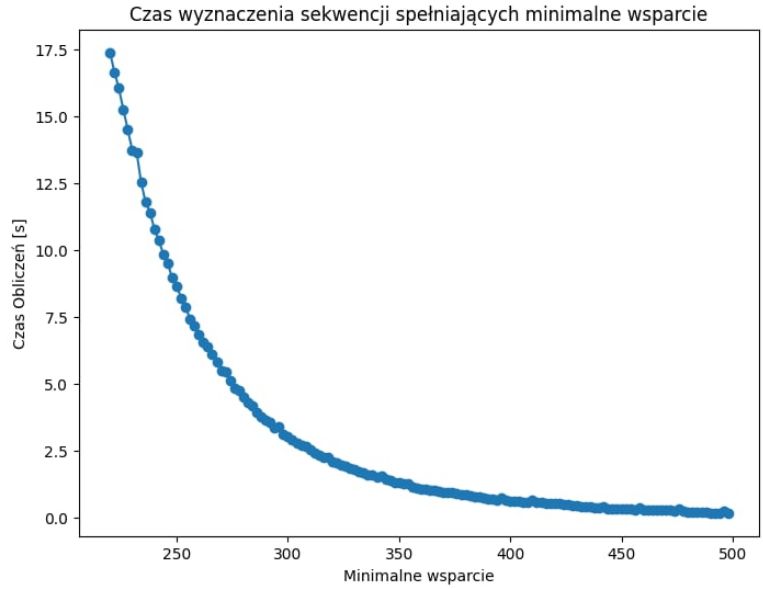
```
=============  PREFIXSPAN 0.99-2016 - STATISTICS =============
 Total time ~ 89738 ms
 Frequent sequences count : 294094
 Max memory (mb) : 446.70467376708984
 minsup = 220 sequences.
 Pattern count : 294094
================================================

Time of smpf prefix time: 89.92130546699627 with min support: 220
```

Rysunek 30 Rezultaty uzyskane poprzez gotową implementację z paczki SMPF

```
Time of custom prefix time: 17.9186023560178 with min support: 220, found: 13127 sequences
```

Rysunek 31Rezultaty uzyskane przez własną implementację





7. Wnioski

Pierwszym wnioskiem jest potwierdzenie poprawności zaimplementowanego algorytmu poprzez porównanie jego wyników wraz z gotową implementacją z paczki SMPF. Ilość znalezionych sekwencji była dokładnie taka sama w obydwu przypadkach, a zapisane pliku na lokalnym dysku potwierdziły te hipotezę.

Łatwo było również zauważyć bardzo silną zależność pamięci RAM do ilości sekwencji w bazie danych. Implementacja SMPF potrafiła sobie jednak z tym poradzić i przy wykorzystaniu jej implementacji zużycie RAMu było zdecydowanie mniejsze niż w przypadku własnej implementacji. W większości przypadków również było szybsze jednak nie w każdym.

Przy przetwarzaniu bardzo długich sekwencji gotowa implementacja dość znacznie odbiegała pod względem czasu do własnej implementacji. Widać, że przy jej tworzeniu skupiono się na radzeniu sobie z bardzo szubko rosnącą pamięcią RAM kosztem przetwarzania bardzo długich sekwencji.

Badać można było zbiory dopiero od jakiegoś progu powyżej którego czas oczekiwania był akceptowalny. Jak można zauważyć na wykresach czas oraz ilośc znalezionych sekwencji które ewidentnie są ze sobą skorelowane rosną bardzo szybko (prawdopodobnie wykładniczo) im mniejszy próg minimalnego wsparcia zastosuję.

Dalsze badania mogły by zawierać jeszcze sprawdzanie przebiegów czasowych implementacji SMPF czy również czas przetwarzania jest aż tak zależny od minimalnego przebiegu (kształt wykresu).

8. Bibliografia

1. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M., “Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach”, IEEE Trans. Knowledge and Data Engineering
2. Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, Rincy Thomas, A Survey of Sequential Pattern Mining, 2017, Ubiquitous International
3. https://www.researchgate.net/figure/Pseudo-Code-of-the-PrefixSpan-algorithm_fig9_46093566 [22.04.2024]
4. Yu Hirate, Hayato Yamana, Generalized Sequential Pattern Mining with Item Intervals, 2006, Journal Of Computers
5. https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php#d1 [23.04.2024]
6. https://www.philippe-fournier-viger.com/spmf/ [23.04.2024]