

Wydział Informatyki, Elektroniki i Telekomunikacji

Katedra Informatyki



**AGH**

# Eksploracja Danych

## Analiza wzorców i sekwencji

Michał Suder  
Janusz Węgrzyn

## Spis treści

1. Wprowadzenie .....	3
2. Metody .....	3
2.1. Metody do wyszukiwania wzorców częstych .....	3
2.2. Metody do wyszukiwania domkniętych wzorów częstych .....	3
2.3. Metody do wyszukiwania wzorców sekwencji .....	4
2.4. Metody do wyszukiwania domkniętych wzorów sekwencji .....	4
3. Realizacja .....	4
3.1. KNIME .....	4
3.2. Runnery .....	9
4. Eksperymenty .....	9
4.1. Grupa pierwsza .....	10
4.2. Grupa druga .....	10
4.3. Zbiory danych .....	11
5. Wyniki .....	13
5.1. Metody do wyszukiwania wzorców częstych .....	13
5.2. Metody do wyszukiwania domkniętych wzorców częstych .....	20
5.3. Metody do wyszukiwania wzorców sekwencji .....	27
5.4. Metody do wyszukiwania domkniętych wzorców sekwencji .....	34
6. Wnioski ogólne .....	41
7. Podsumowanie .....	42
8. Bibliografia .....	42

## 1. Wprowadzenie

W ramach eksploracji danych istnieje wiele różnych technik odkrywania wiedzy. Jedną z najczęściej stosowanych jest wyszukiwanie wzorców w ramach podanego zbioru wejściowego transakcji. Można wyszczególnić cztery typy wzorców: wzorce częste, domknięte wzorce częste, wzorce sekwencji, domknięte wzorce sekwencji. Dla każdego z tych typów istnieje wiele różnych algorytmów o różnym podejściu do przetwarzania zbiorów wejściowych. Dodatkowo, zbiory danych wejściowych mogą różnić się między sobą charakterystykami takimi jak liczba czy średnia długość transakcji czy też liczba unikalnych elementów. Celem tego projektu jest przeprowadzenie eksperymentów i porównanie wybranych algorytmów w ramach poszczególnych typów.

## 2. Metody

Metody wyszukiwania wzorców częstych można podzielić na 4 grupy zależne od celu analizy:

- Wzorce częste
- Domknięte wzorce częste
- Wzorce sekwencji
- Domknięte wzorce sekwencji

Dla każdej grupy wybrany został zbiór algorytmów, które zostały następnie zaimplementowane i przetestowane. Obydwie te rzeczy zostały opisane w kolejnych rozdziałach.

### 2.1. Metody do wyszukiwania wzorców częstych

Metody te odpowiadają za odkrywanie wzorców częstych w ramach zbioru transakcji. Transakcja jest definiowana jako zbiór unikalnych elementów, a wzorec częsty to zbiór elementów, który zawiera się w odpowiedniej ilości transakcji. Każda z tych metod powinna dawać te same wyniki dla tego samego zbioru parametrów wejściowych. Algorytmy na wejście przyjmują tylko wartość wsparcia (czy to bezwzględnego czy względnego) oraz zbiór danych. Wybrane zostało 8 metod odkrywania wzorców częstych. Zostały one posortowane zgodnie z rokiem zaprezentowania metody.

- *Apriori* [1] (1994) [Agrawal & Srikant](#)
- *Eclat* [2] (2000) [Zaki](#)
- *FPGrowth* [3] (2004) [Han et al.](#)
- *LCMFreq* [4] (2004) [Uno et al.](#)
- *Relim* [5] (2005) [Borgelt](#)
- *H-Mine* [6] (2007) [Pei et al.](#)
- *PrePost* [7] (2012) [Deng et al.](#)
- *PrePost+* [8] (2015) [Deng et al.](#)

### 2.2. Metody do wyszukiwania domkniętych wzorców częstych

Metody te odpowiadają za odkrywanie domkniętych wzorców częstych w ramach zbioru transakcji. Różnica pomiędzy zbiorem domkniętych wzorców częstych a zbiorem wzorców częstych, jest taka, że za domknięte wzorce uznajemy tylko te, które nie mają żadnego nadwzorca o tym samym wsparciu. Każda z tych metod powinna dawać te same wyniki dla tego samego zbioru parametrów wejściowych. Algorytmy na wejście przyjmują tylko wartość supportu (czy to bezwzględnego czy względnego) oraz zbiór danych. Opracowano 5 algorytmów do znajdowania domkniętych wzorców częstych. Zostały one posortowane zgodnie z rokiem zaprezentowania metody.

- *A-Close* [9] (1999) [Pasquier et al.](#)
- *LCM* [4] (2004) [Uno et al.](#)

- *DCIClosed* [10] (2004) [Lucchese et al.](#)
- *Charm* [11] (2005) [Zaki and Hsiao](#)
- *FPClose* [12] (2005) [Grahne and Zhu](#)

### 2.3. Metody do wyszukiwania wzorców sekwencji

Metody te znajdują wzorce sekwencji w ramach zbioru sekwencji. Sekwencja jest to uporządkowany zbiór zbiorów elementów. Każda z tych metod powinna dawać te same wyniki dla tego samego zbioru parametrów wejściowych. Algorytmy na wejście przyjmują tylko wartość wsparcia (czy to bezwzględnego czy względnego) oraz zbiór danych. Opracowano 6 metod do odkrywania wzorców sekwencji. Zostały one posortowane zgodnie z rokiem zaprezentowania metody.

- *SPADE* [13] (2001) [Zaki et al.](#)
- *SPAM* [14] (2002) [Ayres et al.](#)
- *PrefixSpan* [15] (2004) [Pei et al.](#)
- *LAPIN* [16] (2007) [Yang et al.](#)
- *CMSPADE* [17] (2014) [Fournier-Viger et al.](#)
- *CMSPAM* [17] (2014) [Fournier-Viger et al.](#)

### 2.4. Metody do wyszukiwania domkniętych wzorów sekwencji

Metody te odpowiadają za odkrywanie domkniętych wzorców sekwencji. Podobnie jak to było w przypadku wzorców częstych, domknięty wzorec sekwencji to taki, który nie ma żadnego nadwzorca o tym samym suporcie. Każda z tych metod powinna dawać te same wyniki dla tego samego zbioru parametrów wejściowych. Algorytmy na wejście przyjmują tylko wartość supportu (czy to bezwzględnego czy względnego) oraz zbiór danych. Opracowano 4 metod do odkrywania wzorców sekwencji. Zostały one posortowane zgodnie z rokiem zaprezentowania metody.

- *CloSpan* [18] (2003) [Yan et al.](#)
- *BIDEPlus* [19] (2004) [Wang et al.](#)
- *ClaSP* [20] (2013) [Gomariz et al.](#)
- *CMClasp* [17] (2014) [Fournier-Viger et al.](#)

## 3. Realizacja

Wszystkie algorytmy zostały zaimplementowane przy użyciu języka Java® w wersji 8. Algorytmy zostały zaimplementowane przy pomocy biblioteki SPMF [21] autorstwa Philippe Fournier-Viger. Każdy algorytm jest reprezentowany przez własną klasę o wspólnym interfejsie, odpowiedniego dla celu analizy tj. wzorce częste, domknięte wzorce częste, wzorce sekwencji, domknięte wzorce sekwencji. Dodatkowo przygotowane zostały klasy odpowiedzialne za odczyt wejścia i zapis wyjścia w odpowiednim formacie. Każdy algorytm jest jednowątkowy i niezależny od pozostałych.

### 3.1. KNIME

Środowisko KNIME® [22] zostało wykorzystane w celu prezentacji wyniku poszczególnych algorytmów. W ramach pracy przygotowane zostało 10 węzłów w dwóch niezależnych wtyczkach zgodnych ze specyfikacją KNIME®. W samym środowisku przygotowanych zostało 5 projektów prezentujących możliwości utworzonych węzłów.

#### 3.1.1. Węzły algorytmów

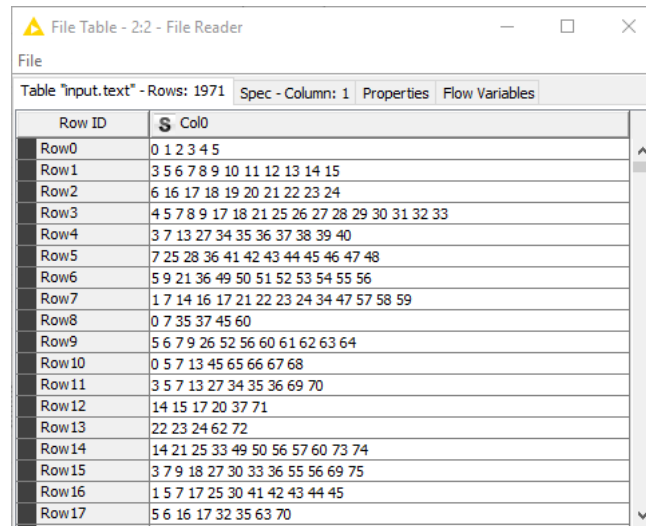
Dla każdego celu analizy przygotowany został oddzielny węzeł. Węzły zostały nazwane jako:

- Frequent Pattern Extractor
- Closed Frequent Pattern Extractor

## Analiza wzorców i sekwencji

- Frequent Sequence Extractor
- Closed Frequent Sequence Extractor

Na wejście przyjmowana jest tabela z jedną kolumną o typie tekstowym. W każdym wierszu zapisana jest jedna transakcja. Na wyjściu zwracana jest tabela z trzema kolumnami. W pierwszej kolumnie jest zapis wzorca w postaci tekstowej, w drugiej kolumnie wsparcie bezwzględne, a w trzeciej wsparcie względne dla wzorca.



File Table - 2:2 - File Reader

File

Table "input.text" - Rows: 1971 Spec - Column: 1 Properties Flow Variables

Row ID	\$ Col0
Row0	0 1 2 3 4 5
Row1	3 5 6 7 8 9 10 11 12 13 14 15
Row2	6 16 17 18 19 20 21 22 23 24
Row3	4 5 7 8 9 17 18 21 25 26 27 28 29 30 31 32 33
Row4	3 7 13 27 34 35 36 37 38 39 40
Row5	7 25 28 36 41 42 43 44 45 46 47 48
Row6	5 9 21 36 49 50 51 52 53 54 55 56
Row7	1 7 14 16 17 21 22 23 24 34 47 57 58 59
Row8	0 7 35 37 45 60
Row9	5 6 7 9 26 52 56 60 61 62 63 64
Row10	0 5 7 13 45 65 66 67 68
Row11	3 5 7 13 27 34 35 36 69 70
Row12	14 15 17 20 37 71
Row13	22 23 24 62 72
Row14	14 21 25 33 49 50 56 57 60 73 74
Row15	3 7 9 18 27 30 33 36 55 56 69 75
Row16	1 5 7 17 25 30 41 42 43 44 45
Row17	5 6 16 17 32 35 63 70

Rysunek 1 - Przykładowe wejście do algorytmu wyszukiwania wzorców częstych

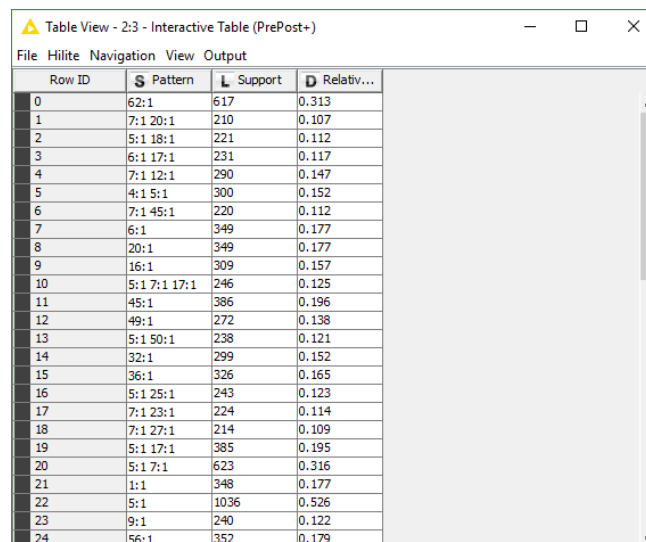


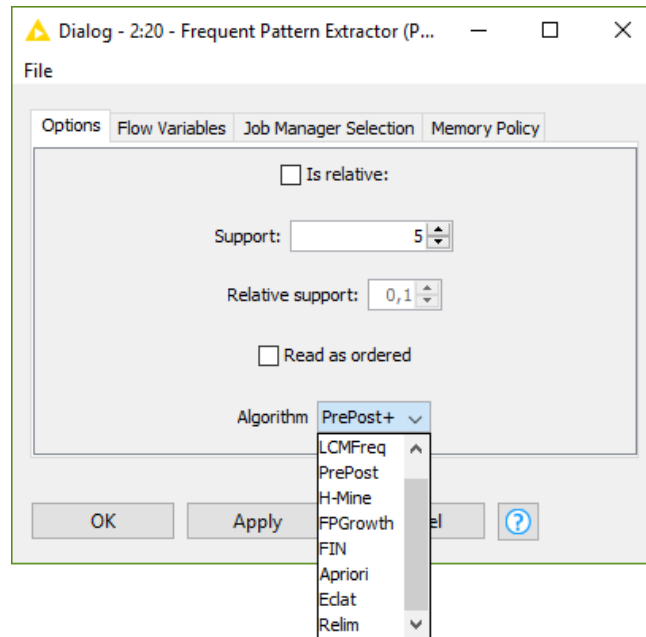
Table View - 2:3 - Interactive Table (PrePost+)

File Hilite Navigation View Output

Row ID	\$ Pattern	L Support	D Relativ...
0	62:1	617	0.313
1	7:1 20:1	210	0.107
2	5:1 18:1	221	0.112
3	6:1 17:1	231	0.117
4	7:1 12:1	290	0.147
5	4:1 5:1	300	0.152
6	7:1 45:1	220	0.112
7	6:1	349	0.177
8	20:1	349	0.177
9	16:1	309	0.157
10	5:1 7:1 17:1	246	0.125
11	45:1	386	0.196
12	49:1	272	0.138
13	5:1 50:1	238	0.121
14	32:1	299	0.152
15	36:1	326	0.165
16	5:1 25:1	243	0.123
17	7:1 23:1	224	0.114
18	7:1 27:1	214	0.109
19	5:1 17:1	385	0.195
20	5:1 7:1	623	0.316
21	1:1	348	0.177
22	5:1	1036	0.526
23	9:1	240	0.122
24	56:1	352	0.179

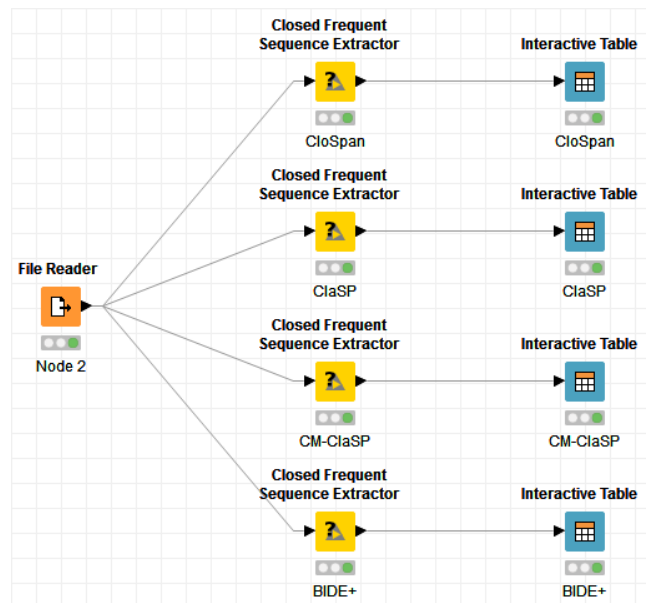
Rysunek 2 - Przykładowe wyjście dla algorytmu wyszukiwania wzorców częstych

Każdy przygotowany węzeł pozwala na wybór metody użytej w celu znalezienia celu analizy. Dodatkowo możliwe jest zdefiniowanie wsparcia bezwzględnego lub wsparcia względnego, dla którego ma być przeprowadzona analiza.



Rysunek 3 - Przykładowe okno konfiguracji węzła

W ramach pracy przygotowane zostały projekty dla każdego celu analizy oddzielnie. Każdy projekt zawiera węzeł odpowiedzialny za czytanie pliku zawierającego zbiór danych oraz po parze węzłów eksploracji i prezentacji dla każdego algorytmu z danej grupy.



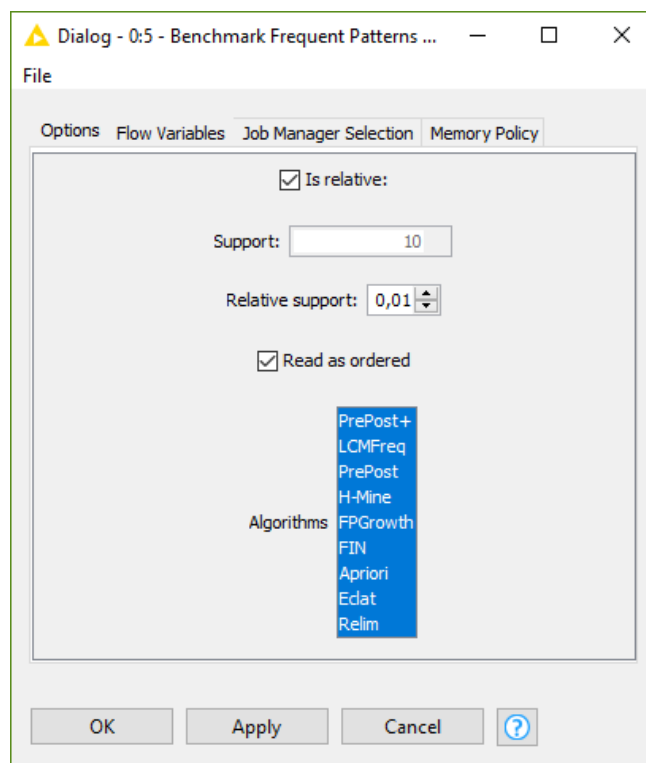
Rysunek 4 - Przykładowy projekt dla algorytmów do wyszukiwania domkniętych wzorców sekwencji

## 3.1.2. Podstawowe węzły benchmarkowe

W ramach środowiska KNIME przygotowane zostały również węzły benchmarkowe, które pozwalają na przeprowadzenie testów wybranych algorytmów z danej grupy. Każdy węzeł pozwala na wybranie algorytmów do testowania oraz wprowadzenie parametrów użytych w eksploracji danych. Przygotowane zostały następujące węzły:

- Benchmark Frequent Pattern Extractor
- Benchmark Closed Frequent Pattern Extractor
- Benchmark Frequent Sequence Extractor

- Benchmark Closed Frequent Sequence Extractor



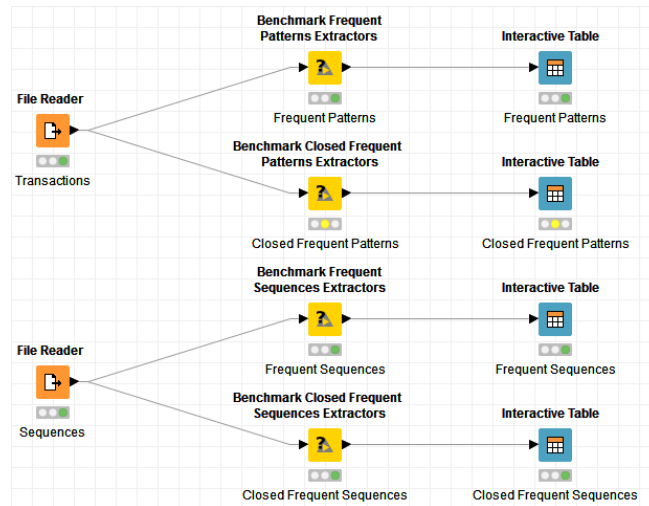
Rysunek 5 - Przykładowa konfiguracja węzła benchmarkowego

Każdy z wymienionych węzłów przyjmuje tabelę z jedną kolumną o typie tekstowym. W każdym wierszu zapisana jest jedna transakcja. Na wyjściu zwracana jest czterokolumnowa tabela. Kluczem wiersza jest nazwa użytego algorytmu. W pierwszej kolumnie znajduje się czas wykonania, w drugiej liczba znalezionych wzorców, w trzeciej poprawność wyniku a w czwartej dodatkowy komunikat w przypadku wystąpienia błędu.

Row ID	Executi...	Pattern...	Correct...	Message
PrePost+	00:00:00.172	0	true	
LCMFreq	00:00:00.246	0	true	
PrePost	00:00:00.170	0	true	
H-Mine	00:00:00.189	0	true	
FPGrowth	00:00:00.260	0	true	
FIN	00:00:08.632	0	false	
Apriori	00:04:22.566	0	true	
Eclat	00:00:00.167	0	true	
Relim	00:00:00.220	0	true	

Rysunek 6 - Przykładowe wyjście z węzła benchmarkowego

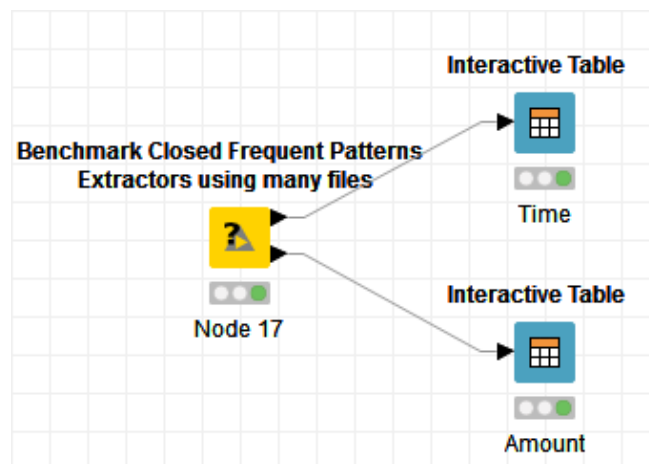
Przygotowany został przykładowy projekt wykorzystujący wszystkie podstawowe węzły benchmarkowe.



Rysunek 7 - Projekt wykorzystujący podstawowe węzły benchmarkowe

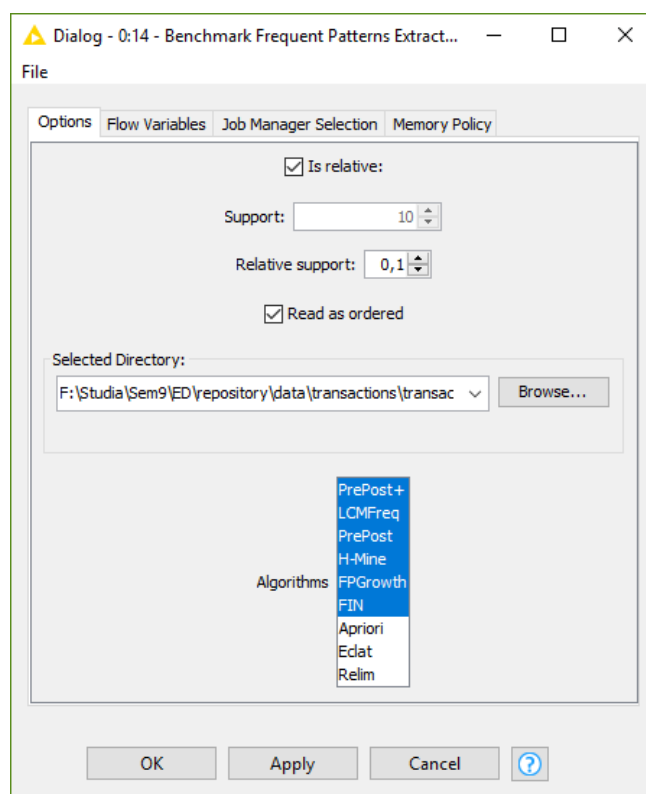
## 3.1.3. Zaawansowane węzły benchmarkowe

Oprócz podstawowych węzłów benchmarkowych opisanych powyżej, zaimplementowane zostały węzły pozwalające na wykonanie eksperymentów na wielu plikach jednocześnie. Węzły te pozwalają na wybór algorytmów użytych w eksperymencie oraz podanie ścieżki katalogu, w którym znajdują się pliki testowych zbiorów danych. Z węzła są dwa wyjścia. Pierwsze zwraca tabelę z czasami wykonania algorytmu dla każdego pliku. Drugie zwraca ilość znalezionych wzorców dla każdego algorytmu. Obydwie tabele w kolumnach mają kolejne algorytmy, a w wierszach nazwy kolejnych plików użytych jako dane testowe.



Rysunek 8 - Przykładowe użycie zaawansowanego węzła benchmarkowego





Rysunek 9 - Przykładowa konfiguracja zaawansowanego węzła benchmarkowego

## 3.2. Runnery

Oprócz przygotowania węzłów w ramach środowiska KNIME®, przygotowane zostały odpowiednie klasy pozwalające na uruchomienie eksperymentów w niezależnym środowisku. Dzięki takiemu rozwiązaniu, w momencie wykonywania eksperymentów nie trzeba brać pod uwagę narzutu związanego z wykorzystaniem frameworka KNIME®. Dla każdego eksperymentu przygotowane zostały niezależne programy uruchamiające, dzięki temu jest możliwość równoległego wykonywania testów. W ramach jednego uruchomienia zbierane są informacje na temat

- Czasu wykonania [ms]
- Maksymalnej zajętości pamięciowej [kB]
- Rozmiaru znalezionej odpowiedzi
- Znalezione rozwiązanie

Program pozwala na wybór przypadku testowego oraz liczbę powtórzeń. Wszystkie wyniki są zapisywane niezależnie dla każdego powtórzenia w oddzielnym pliku, za wyjątkiem rozwiązania, które jest zapisywane tylko przy pierwszym wykonaniu. Każdy plik z wynikami jest zapisany w formacie CSV z średnikiem jako separator. Plik zawiera nagłówek, w którym zapisane są nazwy algorytmów. W kolejnych wierszach zapisana jest wartość zmiennej użytej w pojedynczym uruchomieniu i kolejne wyniki dla poszczególnych algorytmów, zapisane zgodnie z nagłówkiem.

## 4. Eksperymenty

W ramach pracy przygotowanych zostało 16 przypadków testowych, które można podzielić na dwie grupy. W ramach drugiej dokonywane są pomiary w zależności od rozmiaru wejściowego zbioru danych. Wszystkie eksperymenty zostały powtórzone 5 razy w celu minimalizacji losowości. Zbiory danych, które zostały użyte w ramach eksperymentów zostały opisane w kolejnym rozdziale. Eksperymenty są uruchamiane za pomocą klas runnerów opisanych w poprzednim rozdziale.

#### 4.1. Grupa pierwsza

W ramach pierwszej grupy, zawierającej 12 przypadków, dokonywane są pomiary czasu i pamięci w zależności od zadanego supportu. Przygotowanych zostało 6 zbiorów danych wraz z odpowiednimi zakresami supportu w celu najlepszego zobrazowania zachowania poszczególnych algorytmów w zależności od malejącego supportu. Dokładne parametry przypadków testowych zostały zaprezentowane w Tabeli 1. Wraz ze zmniejszeniem wsparcia rośnie liczba znajdowanych wzorców częstych, co jest jednym z głównych kryteriów wyznaczania złożoności obliczeniowej.

Lp.	Cel analizy	Zbiór danych	Maksymalny support	Minimalny support	Krok
1.	Wzorce częste	<i>Kddcup99</i>	0.8	0.5	0.025
2.		<i>Mushroom</i>	0.5	0.1	0.05
3.		<i>OnlineRetail</i>	0.01	0.001	0.001
4.	Domknięte wzorce częste	<i>Kddcup99</i>	0.8	0.5	0.025
5.		<i>Mushroom</i>	0.5	0.1	0.05
6.		<i>OnlineRetail</i>	0.01	0.001	0.001
7.	Wzorce sekwencji	<i>Kosarak Seq</i>	0.2	0.025	0.025
8.		<i>Bible</i>	0.2	0.025	0.025
9.		<i>Leviathan</i>	0.01	0.001	0.001
10.	Domknięte wzorce sekwencji	<i>Kosarak Seq</i>	0.2	0.025	0.025
11.		<i>Bible</i>	0.2	0.025	0.025
12.		<i>Leviathan</i>	0.01	0.001	0.001

Tabela 1 - Przypadki testowe wraz z zakresem i krokiem supportu

#### 4.2. Grupa druga

W ramach drugiej grupy, zawierającej 4 przypadki, wykonane zostały pomiary czasu i pamięci w zależności od rozmiaru zbioru testowego. Rozmiar zbioru testowego jest drugim czynnikiem wpływającym na złożoność algorytmu. Dokładne parametry przypadków testowych zostały zaprezentowane w Tabeli 2. Support został dobrany w ten sposób, aby liczba znajdowanych wzorców częstych była zbliżona dla kolejnych rozmiarów.

Lp.	Cel analizy	Zbiór danych	Minimalny rozmiar	Maksymalny rozmiar	Krok
1.	Wzorce częste	<i>Kosarak</i>	5000	40000	5000
2.	Domknięte wzorce częste				
3.	Wzorce sekwencji	<i>Kosarak Seq</i>			
4.	Domknięte wzorce sekwencji				

Tabela 2 - Przypadki testowe wraz z zakresem i krokiem wejściowego zbioru danych

Zastosowany zbiór danych ma specyficzną własność, która powoduje, że stosunek liczby znajdowanych wzorców do supportu jest prawie stały niezależnie od rozmiaru danych. Liczba znajdowanych wzorców w zależności od supportu i rozmiaru została zaprezentowana w tabeli Tabela 3 dla zbioru *Kosarak* i tabeli Tabela 4 dla zbioru *Kosarak Seq*.

Rozmiar	Support	Liczba znajdowanych wzorców
5000	0.0035	4009
10000	0.0034	4065
15000	0.0035	3968
20000	0.0034	3961
25000	0.00335	4003
30000	0.0033	4009
35000	0.0033	4107
40000	0.0033	4064

Tabela 3 - Parametry testowe dla zbioru Kosarak

Rozmiar	Support	Liczba znajdowanych wzorców
5000	0.0035	2374
10000	0.0034	2840
15000	0.0035	3032
20000	0.0034	3116
25000	0.00335	3225
30000	0.0033	3333
35000	0.0033	3460
40000	0.0033	3462

Tabela 4 - Parametry testowe dla zbioru Kosarak Seq

### 4.3. Zbiory danych

W ramach eksperymentów wybranych i dostosowanych zostało 6 zbiorów danych. Zbiory te mają różne właściwości i charakterystyki. Wybór tych zbiorów był podyktowany ich dostępnością, częstością użycia w innych opracowaniach i artykułach oraz różnorodnością właściwości. Zbiory danych zostały podzielone na dwie kategorie, odpowiadające zastosowaniu ich w odpowiednich przypadkach testowych.

#### 4.3.1. Wzorce częste

Testowanie algorytmów do odkrywania wzorców częstych wymaga zbiorów danych, które są podzielone na transakcje i zawierają odpowiednio niewielką liczbę unikalnych elementów. Dzięki takiemu połączeniu możliwe jest znajdowanie często powtarzających się zbiorów elementów w ramach transakcji. Dodatkowo wymagane jest, aby elementy w ramach jednej transakcji były unikalne.

Nazwa	Liczba transakcji	Unikalne elementy	Średnia długość	Odchylenie standardowe
Kddcup99	1 000 000	135	16,000	0,000
Mushroom	8 124	119	23,000	0,000
OnlineRetail	541 909	2604	4,364	1,119

##### 4.3.1.1. Kddcup99 [23]

Zbiór danych użyty w ramach „The Third International Knowledge Discovery and Data Mining Tools Competition”, które zostały zorganizowane wspólnie z KDD-99 „The Fifth International Conference on Knowledge Discovery and Data Mining”. Oryginalny zbiór powstał on na podstawie zapisów połączeń TCP z serwerów wewnętrznej sieci U.S. Air Force Lan z siedmiu tygodniu. Ruch był

symulowany na podstawie realnego ruchu sieciowego. Dane zawierają 1000000 transakcji i 135 unikalnych elementów. Transakcje są jednakowej długości. Każda transakcja opisuje pojedyncze połączenie wraz z informacjami na temat tego połączenia. W zbiorze występuje wysoka powtarzalność dużych fragmentów transakcji, co powoduje dużą liczbę wzorców przy wysokim suporcie.

#### 4.3.1.2. Mushroom [24]

Zbiór danych opisujących hipotetyczne przykłady grzybów z 23 gatunków z rodzin Agaricus i Lepiota. Każdy wpis reprezentuje pojedynczy przykład grzyba, który jest opisany za pomocą 22 parametrów, takich jak rozmiar kapelusza lub liczba pierścieni. Zbiór ten zawiera 8124 wpisy i 119 unikalnych elementów. Transakcje są jednakowej długości. W odróżnieniu od zbioru *kddcup99*, do jednej właściwości może być przypisana wartość z rozłącznego zbioru możliwych wartości.

#### 4.3.1.3. OnlineRetail [25]

Zbiór danych stworzony na podstawie zapisu transakcji z sprzedaży w sieci przez firmę zajmującą się głównie sprzedażą unikalnych prezentów na dowolną okoliczność. Sprzedaż była rejestrowana w dniach od 01-12-2010 do 09-12-2011. Zbiór danych zawiera 541909 transakcji i 2603 unikalnych elementów. W odróżnieniu od dwóch poprzednich zbiorów, transakcje są różnej długości.

#### 4.3.1.4. Kosarak [26]

Zbiór danych przygotowany przez Ferenc Bodon i zawierający dane z ciągów kliknięć na Węgierskich stronach z wiadomościami. Oryginalny zbiór danych zawiera 990000 wpisów. W ramach eksperymentów rozmiar ten został zmniejszony do 40000 transakcji. Średnia długość transakcji wynosi 8.1 elementów przy standardowych odchyleniu 23.6. Zbiór ten jest wykorzystywany tylko w przypadku drugiej grupy eksperymentów z racji własności, która powoduje, że stosunek liczby znajdowanych wzorów do supportu jest prawie stały niezależnie od rozmiaru danych.

### 4.3.2. Sekwencje częste

Podobnie jak w testowaniu algorytmów do wyszukiwania wzorców częstych, algorytmy do wyszukiwania sekwencji częstych wymagają podziału na transakcje, nazywane sekwencjami. Każda sekwencja składa się z następujących po sobie zbiorów elementów. Zbiory te nazywane są grupami. Elementy w ramach jednej grupy muszą być unikalne.

Nazwa	Liczba transakcji	Unikalne elementy	Średnia długość	Odchylenie standardowe
Bible	36 376	13 918	21,636	12,257
Kosarak	40 000	17 645	8,023	21,365
Leviathan	5 834	9 025	33,811	18,609

#### 4.3.2.1. Bible [27]

Zbiór danych przygotowany na podstawie biblii. Każde zdanie tworzy oddzielną sekwencję. Każde słowo tworzy oddzielną grupę w sekwencji. Zbiór zawiera 36369 sekwencji i 13905 unikalnych elementów. Średnia długość sekwencji to 21.7 grup. Średnia liczba unikalnych elementów w sekwencji to 17.84.

#### 4.3.2.2. Kosarak Seq [26]

Odpowiednik zbioru danych *Kosarak* dla sekwencji. Zbiór ten powstał na podstawie podzielenia transakcji na jednoelementowe grupy, które utworzyły w ten sposób sekwencje kliknięć. Zbiór ten został wykorzystany zarówno w pierwszej grupie eksperymentów jak i drugiej.

#### 4.3.2.3. *Leviathan* [28]

Zbiór danych powstały w wyniku przekształcenia noweli pt. „*Leviathan*” autorstwa Thomasa Hobbes z roku 1651. Każde słowo tworzy jednoelementową grupę. Każde zdanie tworzy jedną sekwencję. Zbiór zawiera 5834 sekwencji i 9025 unikalnych elementów. Średnia długość sekwencji to 33.8. Średnia liczba unikalnych elementów w sekwencji to 26.34.

## 5. Wyniki

W ramach każdego eksperymentu zbierane były wartości czasu wykonania, maksymalnej zajętości pamięciowej w trakcie działania algorytmu oraz rozmiaru wyników. Dodatkowo dla każdego pojedynczego uruchomienia zostały zebrane i porównane wyniki algorytmów. Po zebraniu wszystkich rezultatów została wyliczona średnia wszystkich wartości w poszczególnych uruchomieniach. Wartości odchyień standardowych są rzędu 0.1%, więc nie zostały oznaczone na diagramach w celu zachowania czytelności.

Eksperymenty były uruchamiane niezależnie i zostały wykonywane na maszynie z procesorem Intel Core™ i5-4670K CPU @ 3.40GHz i pamięcią DDR3 16GB @ 2400MHz, z czego na środowisko uruchomieniowe przeznaczone zostało 12GB. Rzeczywista maksymalna wartość pamięci RAM, która mogła być przeznaczona na wykonanie algorytmu wynosi 10000kB. Wynika to z faktu wartości progowych wbudowanych w maszynę wirtualną Javy, po przekroczeniu której uruchamiany jest mechanizm „Garbage Collector”. W wartości pamięci nie zostały wliczone rozmiary nieprzetworzonych danych wejściowych oraz pamięci potrzebnej na przechowywanie wewnętrznych struktur Javy.

Z racji specyfiki maszyny wirtualnej Javy, która sama zarządza pamięcią za pomocą mechanizmu „Garbage Collection”, w zużytą pamięć przez algorytm wliczane są wszystkie obiekty pomocnicze, które zostały utworzone na potrzeby działania algorytmu. Wszystkie wyniki, które wskazują jednoznacznie na zadziałanie mechanizmu GC zostały odrzucone i uznane za nieprawidłowe dla potrzeb eksperymentów.

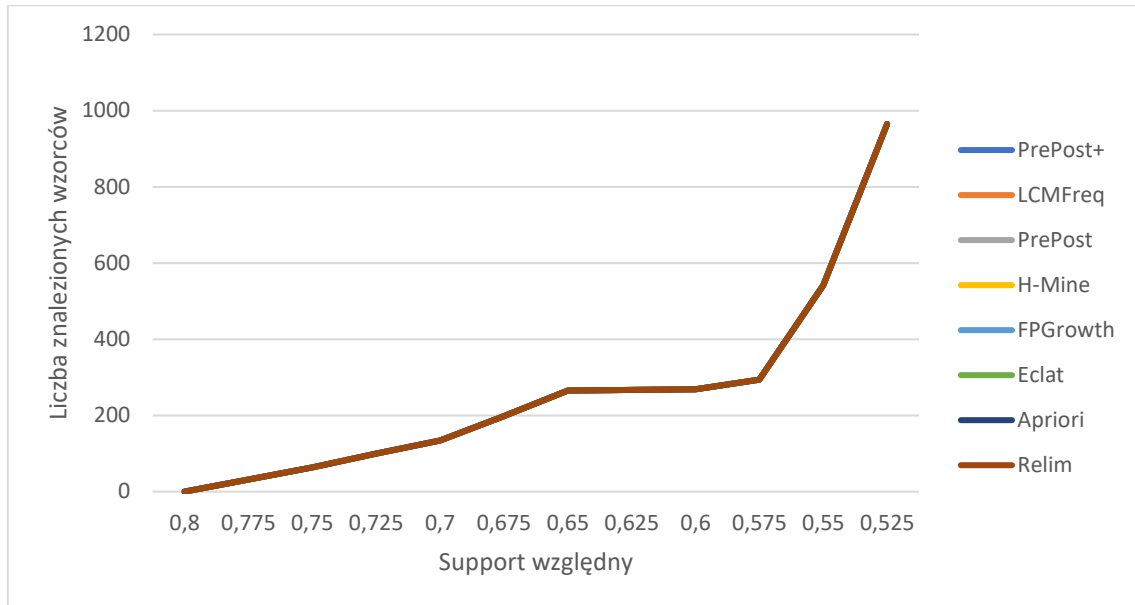
W ramach każdej kategorii algorytmów zaprezentowane zostaną wykresy zależności czasu i pamięci od supportu względnego, dla pierwszej grupy, oraz wykresy zależności czasu i pamięci od rozmiaru danych wejściowych dla grupy drugiej. Dla pierwszej grupy eksperymentów na osi głównej wykresu znajduje się wartość supportu względnego posortowana w malejącym porządku. W przypadku drugiej grupy, na osi głównej znajduje się rozmiar danych wejściowych posortowane rosnąco.

Kolory symbolizujące algorytm są unikalne w ramach kategorii metody. Diagramy czasowe zostały zaprezentowane z użyciem skali logarytmicznej o podstawie 10 w celu zaprezentowania dużych różnic pomiędzy kolejnymi krokami.

### 5.1. Metody do wyszukiwania wzorców częstych

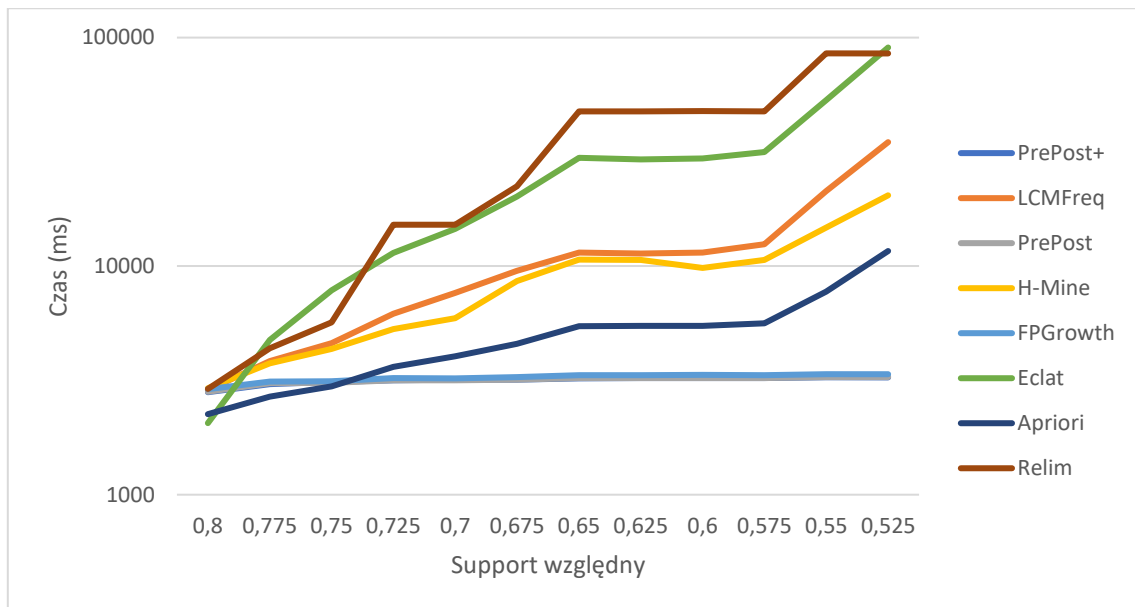
W ramach eksperymentów wykonane zostały trzy testy z grupy pierwszej i jeden test z grupy drugiej. Zostały przetestowane wszystkie wybrane algorytmy. Wszystkie algorytmy dawały te same wyniki dla tego samego zestawu danych wejściowych.

### 5.1.1. Grupa pierwsza – Zbiór danych kddcup99



Wykres 1 - Zależność liczby znalezionych wzorców od supportu względnego

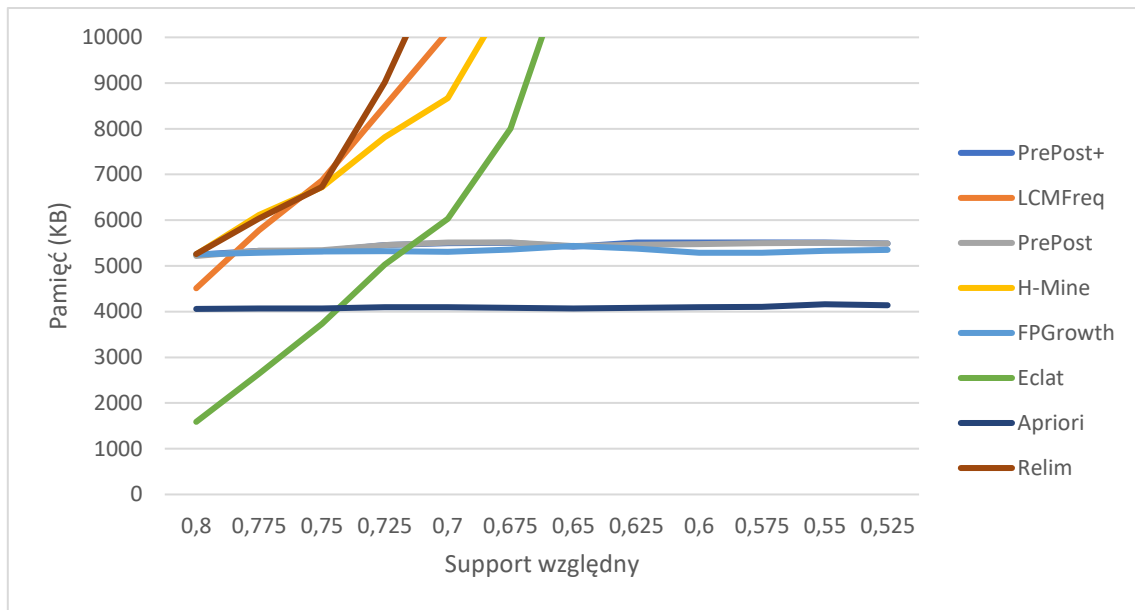
Wykres 1 prezentuje zależność liczby znalezionych wzorców względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę wzorców częstych.



Wykres 2 - Zależność czasu wykonania w milisekundach od supportu względnego

Wykres 2 prezentuje zależność czasu wykonania w milisekundach do użytej wartości wsparcia względnego. W ramach tego testu najgorszym algorytmem jest *Relim* i *Eclat*. Najlepszymi algorytmami zostały *FPGrowth*, *PrePost* i *PrePost+*, które nie wykazuje praktycznie żadnego wzrostu czasu. Wszystkie pozostałe algorytmy wykazują wzrost wykładniczy. Ciekawym jest fakt, że algorytm *Apriori* osiągnął drugi czas.

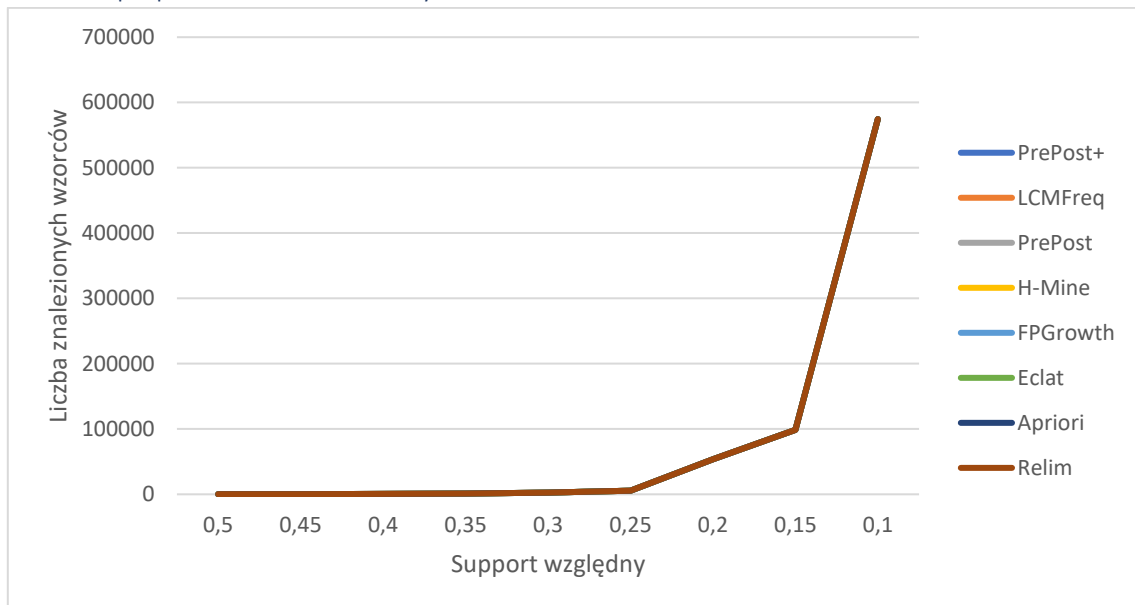
## Analiza wzorców i sekwencji



Wykres 3 - Zależność maksymalnej pamięci od supportu względnego

Wykres 3 prezentuje zależność maksymalnej użytej pamięci względem użytej wartości wsparcia względnego. Da się zaobserwować, że część algorytmów, których działanie opiera się na jednorazowym przeglądnięciu bazy i wygenerowaniu struktur pomocniczych, czyli *PrePost+*, *FPGrowth* i *PrePost*, nie wykazuje wzrostu potrzebnej pamięci. Dodatkowo, algorytm *Apriori*, który nie generuje żadnych dodatkowych struktur, również ma stałe zapotrzebowanie na pamięć. Co więcej jest to najlepszy algorytm. Pozostałe algorytmy wykazują wykładniczy wzrost.

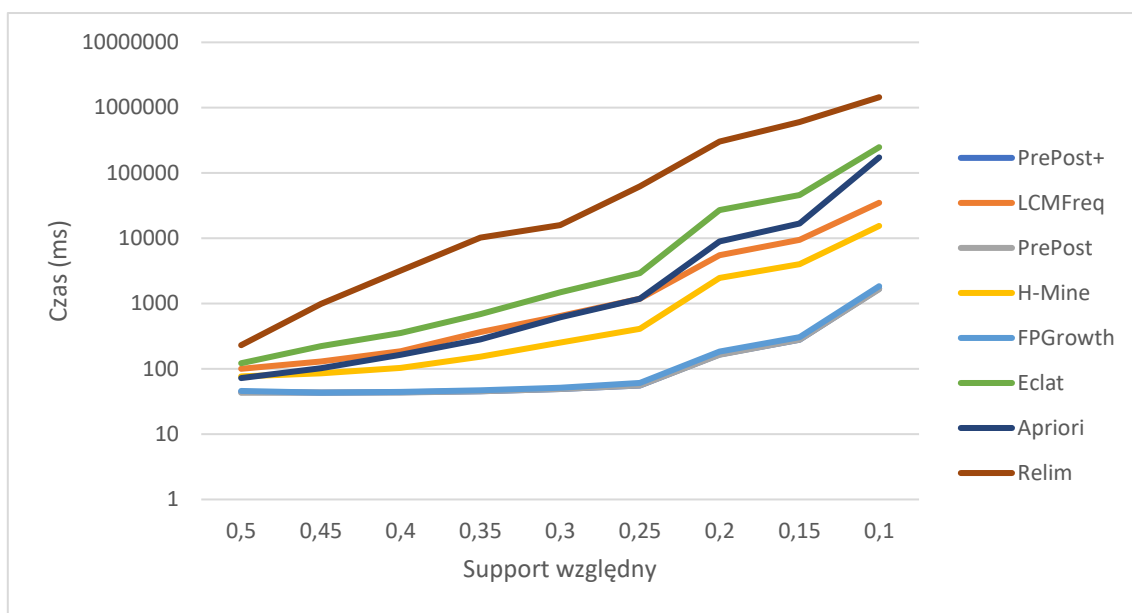
### 5.1.2. Grupa pierwsza – Zbiór danych mushroom



Wykres 4 - Zależność liczby znalezionych wzorców od supportu względnego

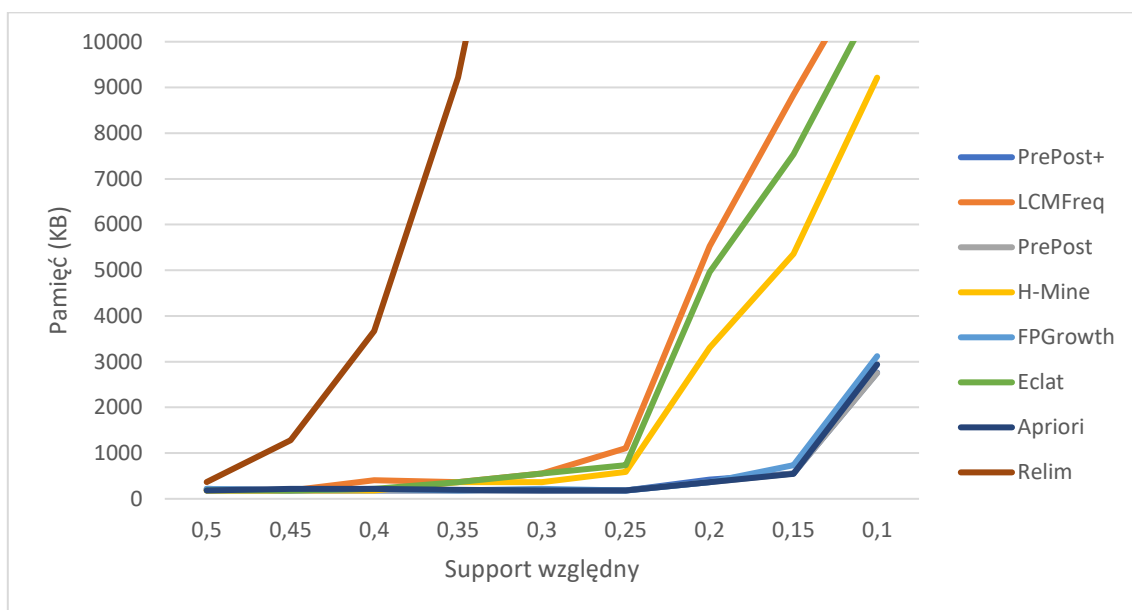
Wykres 4 prezentuje zależność liczby znalezionych wzorców od zadanego wsparcia dla zbioru danych Mushroom. Wszystkie algorytmy znalazły tę samą liczbę wzorców częstych, można więc uznać, że działają poprawnie.

## Analiza wzorców i sekwencji



Wykres 5 - Zależność czasu wykonania w milisekundach od supportu względnego

Wykres 6 prezentuje stosunek czasu wykonania w milisekundach do użytej wartości supportu względnego. Ponownie najlepszymi algorytmami są *PrePost+*, *FPGrowth* i *PrePost*. Mają one znaczną przewagę nad pozostałymi. Dodatkowo w początkowej fazie wykazują najmniejszy przyrost. Dla pozostałych algorytmów wzrost ten cały czas.

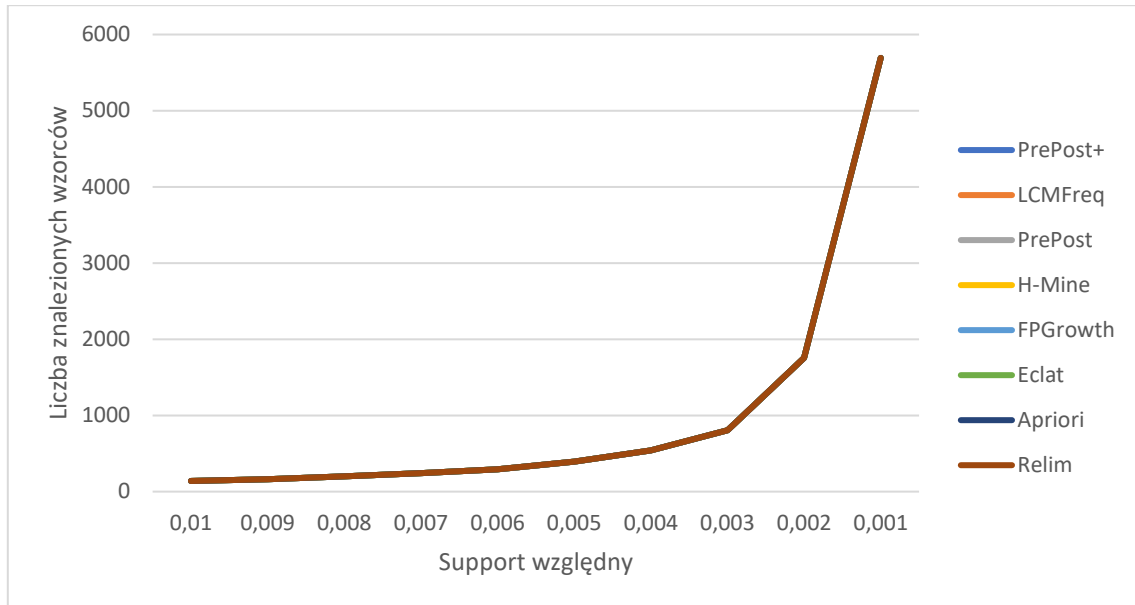


Wykres 6 - Zależność maksymalnej pamięci od supportu względnego

Wykres 6 pokazuje zależność maksymalnej użytej pamięci względem podanej wartości wsparcia względnego. Widzimy tutaj rozróżnienie między grupą najlepszych algorytmów dla bardzo dużej ilości znajdowanych wzorców. Najlepszym algorytmem jest algorytm *PrePost+* i *PrePost*. Zaraz za nimi plasuje się algorytm *Apriori* i *FPGrowth*. Pozostałe algorytmy wykazują gwałtowny wzrost, w szczególności algorytm *Relim*.

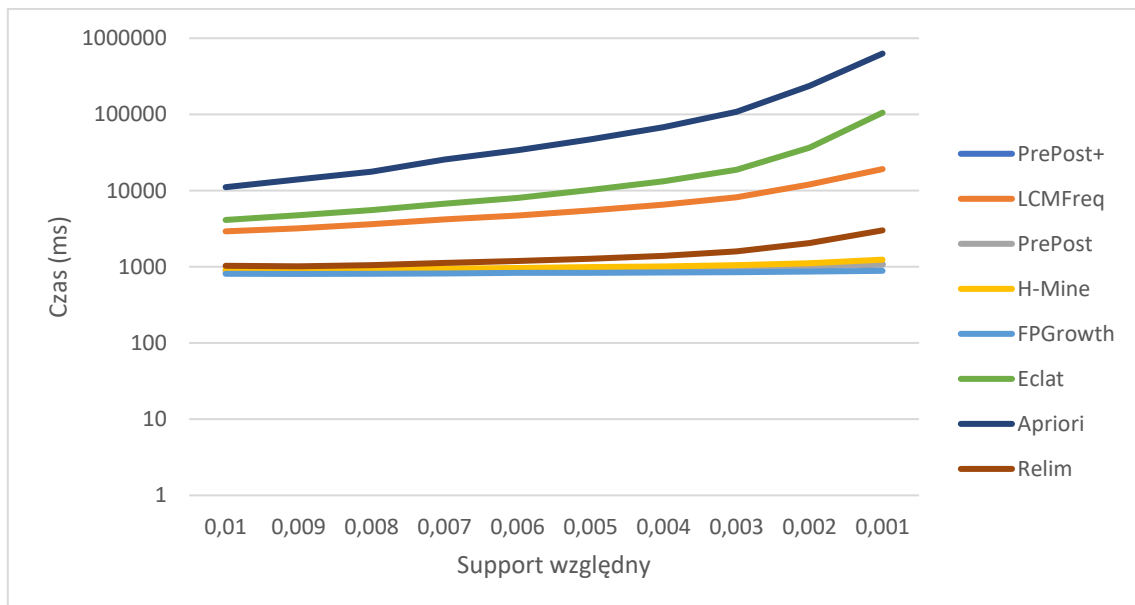


### 5.1.3. Grupa pierwsza – Zbiór danych OnlineRetail



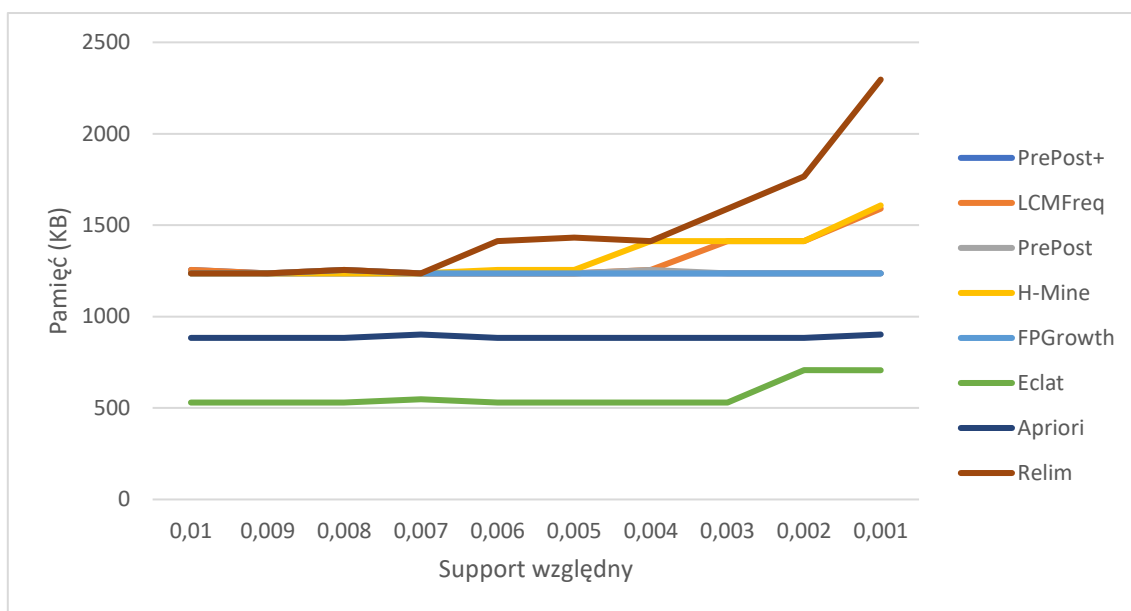
Wykres 7 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 7 prezentuje zależność liczby znalezionych wzorców od zadanego wsparcia dla zbioru danych OnlineRetail. Wszystkie algorytmy znalazły tę samą liczbę wzorców częstych, można więc uznać, że działają poprawnie.



Wykres 8 - Zależność czasu wykonania w milisekundach od supportu względnego

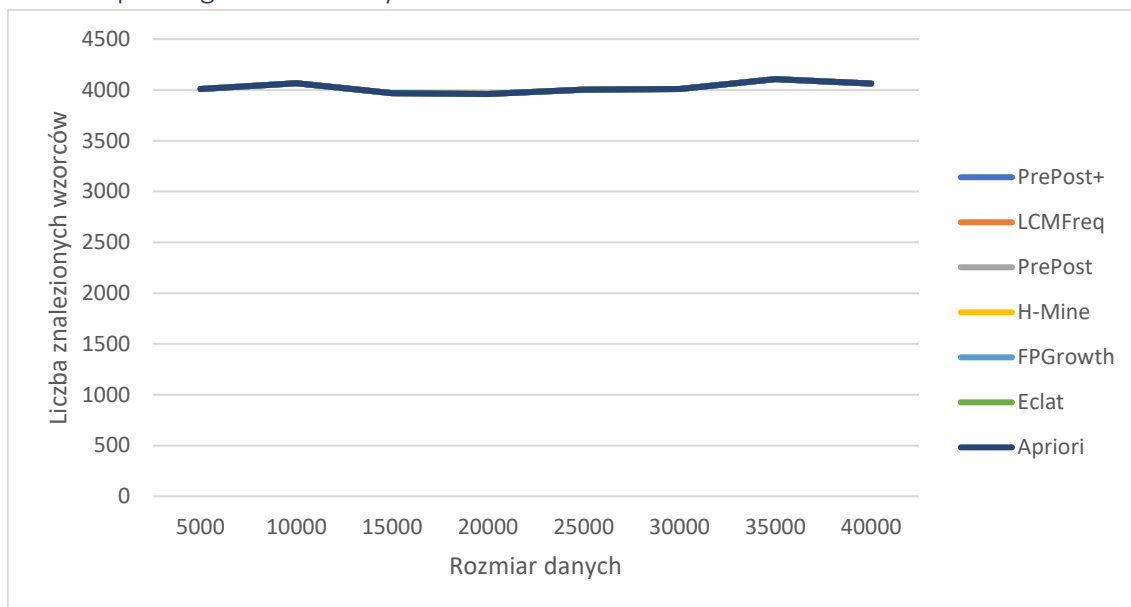
Wykres 8 ilustruje zależność czasu wykonania od wsparcia względnego dla zbioru danych OnlineRetail. Ponownie, najlepszymi algorytmami okazały się *PrePost+*, *FPGrowth* i *PrePost*. Zaraz za nimi plasują się algorytmy *H-Mine* i *Relim*. Dla dwóch poprzednich zbiorów *Relim* osiągnął najgorszy rezultat, więc jest to duże zaskoczenie.



Wykres 9 - Zależność maksymalnej pamięci od supportu względnego

Na Wykres 9 widoczny jest przyrost zapotrzebowania pamięci wraz z zmniejszaniem wartości wsparcia względnego. Najlepszym algorytmem jest algorytm *Eclat* co potwierdza teorię, że metoda ta ma stosunkowo małe zapotrzebowanie na pamięć przy niewielkiej ilości znajdowanych wzorców. Jednakże, zużycie to bardzo szybko rośnie co można zaobserwować na Wykres 3 i Wykres 6. Ponownie algorytm *Apriori* osiąga bardzo dobre rezultaty. Zachowanie metoda *Relim* potwierdza teorię o największym zapotrzebowaniu na pamięć.

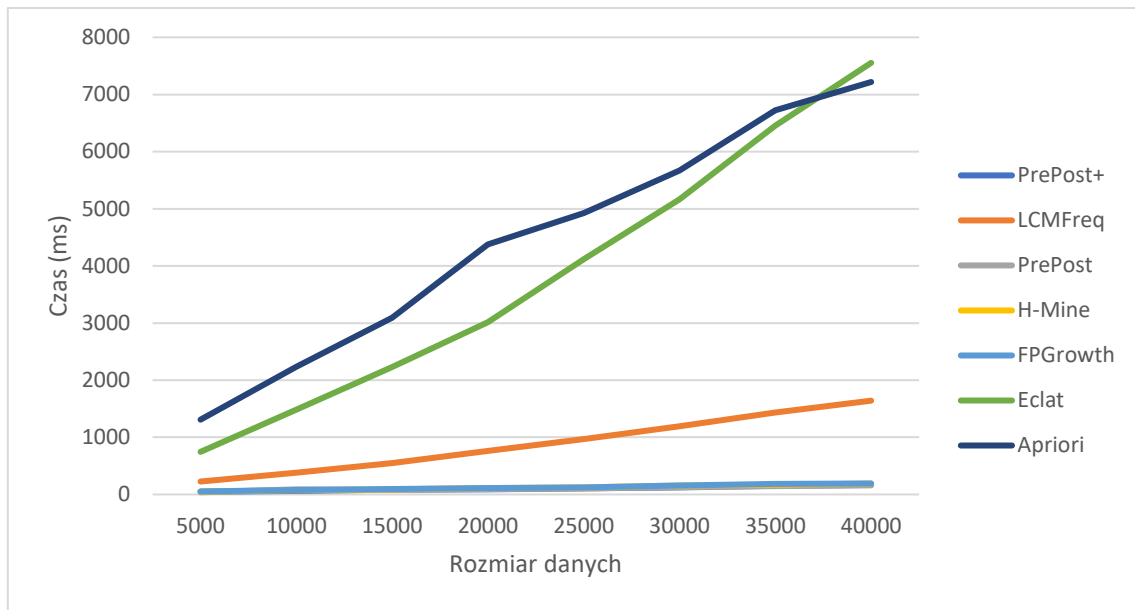
#### 5.1.4. Grupa druga – Zbiór danych Kosarak



Wykres 10 - Zależność liczby znalezionych wzorców od rozmiaru danych

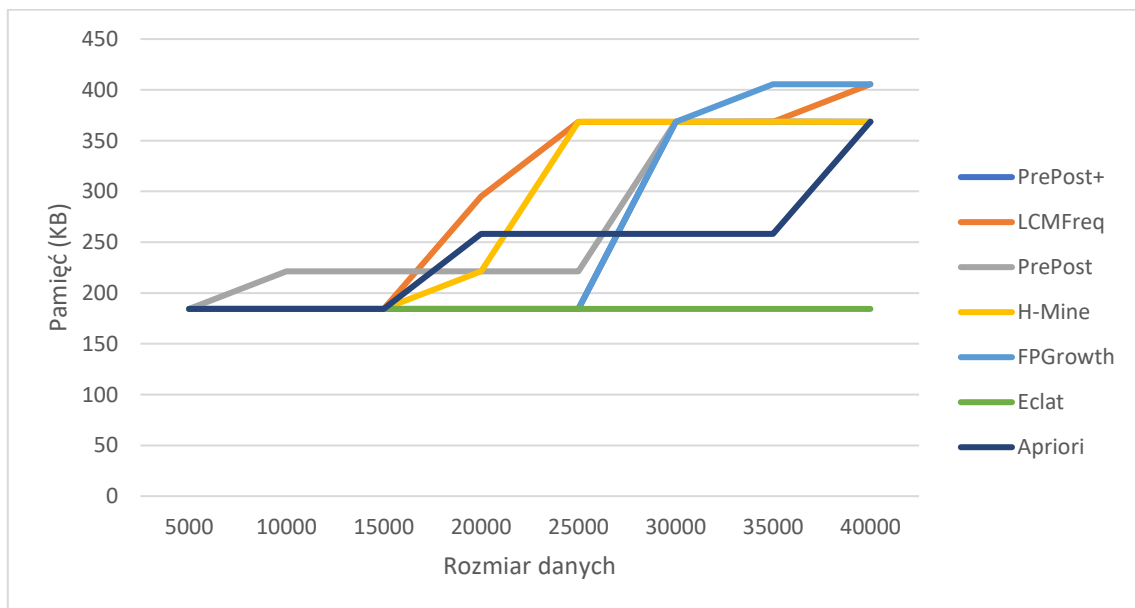
Wykres 10 prezentuje zależność liczby znalezionych wzorców względem rozmiaru zbioru wejściowego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę wzorców częstych. Dodatkowo liczba znajdowanych wzorców jest niezależna od rozmiaru zbioru wejściowego.

## Analiza wzorców i sekwencji



Wykres 11 - Zależność czasu wykonania w milisekundach od rozmiaru danych

Wykres 11 obrazuje czas wykonania algorytmu względem rosnącego rozmiaru zbioru wejściowego. Wszystkie algorytmy wykazują liniowy wzrost czasu. Najmniejszy wzrost jak i generalny czas wykonania mają metody *PrePost+*, *FPGrowth*, *PrePost* i *H-Mine*. Najszybszy wzrost zalicza algorytm *Eclat*.



Wykres 12 - Zależność maksymalnej pamięci od rozmiaru danych

Wykres 12 prezentuje zależność zapotrzebowania na pamięć w zależności od rozmiaru zbioru wejściowego i przy stałej ilości znajdowanych wzorców. Algorytm *Eclat* nie wykazuje żadnego wzrostu zapotrzebowania na pamięć. Najwcześniej rośnie dla algorytmu *PrePost*, jednakże finalnie osiąga ona wartość równą algorytmom *PrePost+*, *Apriori* czy *H-Mine*. Najwięcej pamięci żużył algorytm *FPGrowth*, jednakże są to wartości na poziomie pojedynczego bloku pamięci zajmowanego przez maszynę wirtualną Java.

### 5.1.5. Wnioski

Wszystkie metody znalazły tyle samo wzorców częstych dla kolejnych wartości supportu czy rozmiaru danych, co zostało zobrazowane na Wykres 1, Wykres 4, Wykres 7 i Wykres 10. Na tej podstawie można wyciągnąć wniosek, że wybrane algorytmy prawidłowo znajdują wzorce częste niezależnie od rozmiaru danych wejściowych, wartości wsparcia czy też ilości prawdopodobnych wzorców.

Większość algorytmów wykazuje wzrost czasu w zależności od wartości wsparcia. Jest to spowodowane rosnącą liczbą wzorców częstych wraz z zmniejszaniem wartości supportu. Zależność ta ma charakter wykładniczy.

Algorytmy *FPGrowth* i *PrePost* oraz jego ulepszona odmiana *PrePost+* wykazują bardzo niewielki wzrost czasu wykonania, niezależny od wartości supportu. Dopiero przy gwałtownym wzroście liczby znajdowanych wzorców, jak to ma miejsce w przypadku zbioru danych *mushroom*, zauważamy przyrost czasu. Z pozostałych algorytmów najlepszym jest algorytm *H-Mine*. Ciekawostką jest również dobry wynik metody *Apriori*, która została wymyślona jako pierwsza. Tylko w przypadku zbioru danych *OnlineRetail* jest ona najgorsza. Jest to spowodowane zarówno średnią długością transakcji, dzięki czemu znajdowane są dłuższe wzorce, jak i liczbą unikalnych elementów.

W przypadku zależności od rozmiaru danych wejściowych, wszystkie algorytmy wykazują charakter wzrostowy. Wiąże się to bezpośrednio z ilością przetwarzanych transakcji na wejściu. Zależność ta ma charakter liniowy. Najlepszymi algorytmami są algorytmy *PrePost*, *PrePost+*, *FPGrowth* i *H-Mine*.

Podobne wnioski można wyciągnąć dla zależności pamięci maksymalnej. Algorytmy oparte na pojedynczym przetwarzaniu zbioru wejściowego i przekształcaniu go do struktur wewnętrznych osiągają praktycznie stałą wartość pamięci. Dopiero przy bardzo dużym wzroście liczby wzorców częstych, jak to ma się w przypadku zbioru danych *mushroom*, zauważamy wykładniczy wzrost maksymalnej pamięci.

Wyjątkiem od wyżej wymienionej reguły jest algorytm *Apriori*. We wszystkich przypadkach metoda ta ma stałą wartość maksymalnej zużytej pamięci. Dodatkowo algorytm ten potrzebuje mniej pamięci niż najlepsze algorytmy pod względem czasowym. Jest to spowodowane faktem, że metoda ta nie generuje żadnych dodatkowych struktur danych.

Algorytm *Eclat* jest najlepszym algorytmem pod względem maksymalnej pamięci dla małej liczby znajdowanych wzorców. Jednakże, przy dużym wzroście liczby wzorców, metoda ta gwałtownie zwiększa wymagania pamięciowe.

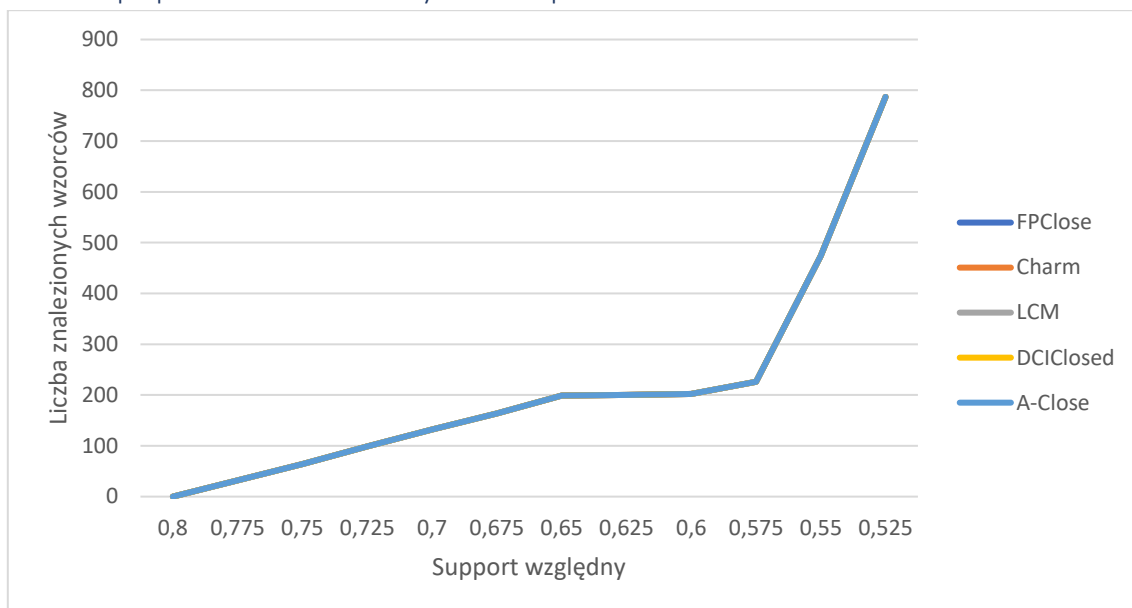
W przypadku eksperymentu z rosnącą liczbą transakcji, wszystkie algorytmy liniowo zwiększają swoje zapotrzebowanie na pamięć operacyjną.

Nie ma jednej najlepszej metody. W przypadku ogólnym najlepszymi algorytmami są *PrePost+*, *FPGrowth* i *PrePost*. Jeżeli bardzo ważnym jest niskie zapotrzebowanie pamięci to, w przypadku małej liczby wzorów (do 500), proponowanym algorytmem jest *Eclat*, w przeciwnym wypadku *Apriori*. Jednakże zysk jest na poziomie 30%, a metody te są znacznie wolniejsze, nawet 1000-krotnie, od proponowanych na początku

## 5.2. Metody do wyszukiwania domkniętych wzorców częstych

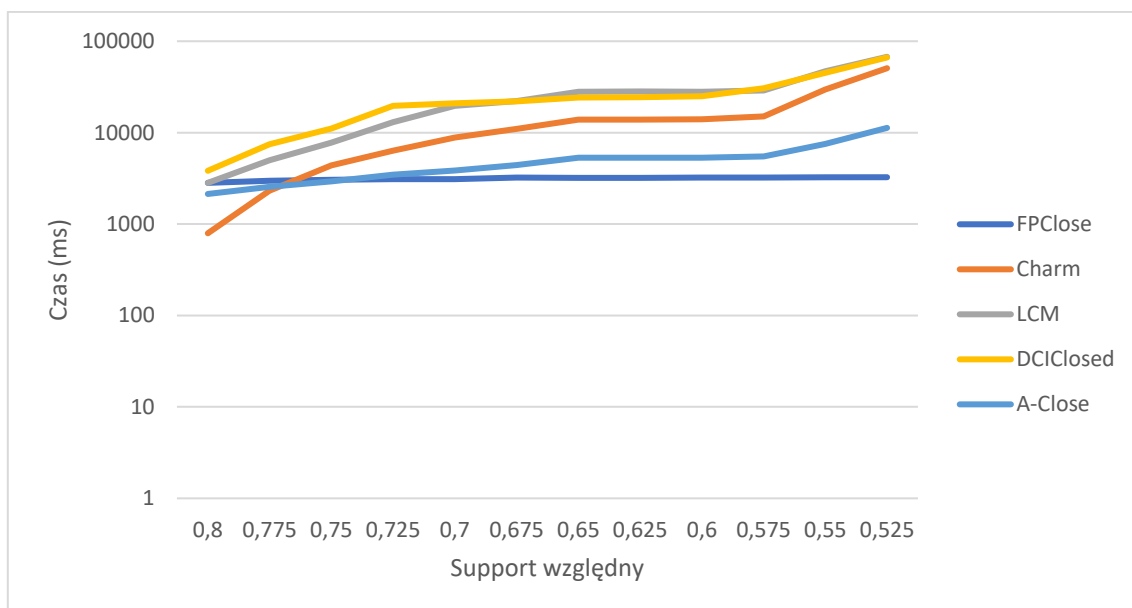
W ramach eksperymentów wykonane zostały trzy testy z grupy pierwszej i jeden test z grupy drugiej. Zostały przetestowane wszystkie wybrane algorytmy. Wszystkie algorytmy dawały te same wyniki dla tego samego zestawu danych wejściowych.

### 5.2.1. Grupa pierwsza – Zbiór danych kddcup99



Wykres 13 - Zależność liczby znalezionych wzorców od supportu względnego

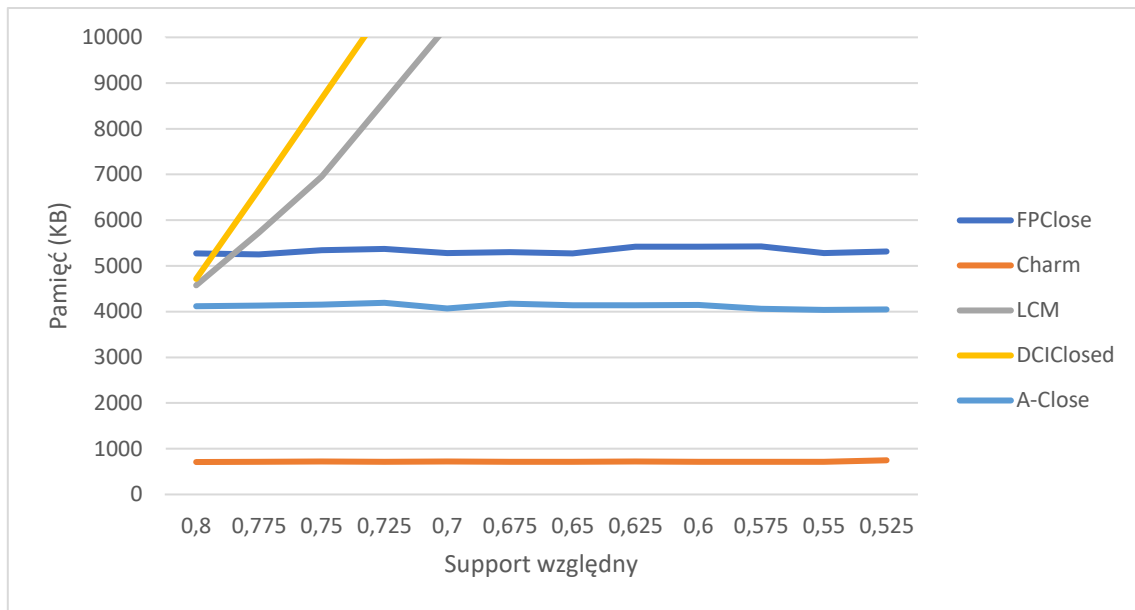
Wykres 13 prezentuje zależność liczby znalezionych wzorców od zadanego wsparcia dla zbioru danych kddcup99. Wszystkie algorytmy znalazły tą samą liczbę wzorców częstych, można więc uznać, że działają poprawnie.



Wykres 14 - Zależność czasu wykonania w milisekundach od supportu względnego

Na wykresie Wykres 14 widzimy stosunek czasu wykonania algorytmu do użytej wartości wsparcia względnego dla zbioru wejściowego kddcup99. W tym przypadku można jednoznacznie stwierdzić, że najlepszym algorytmem okazał się algorytm *FPClose*. Minimalnie lepszym wydaje się algorytm *Charm*, który ma najlepszy czas dla pierwszej wartości supportu, jednakże osiąga on najszybszy wzrost czasu dla kolejnych. Pozostałe algorytmy wykazują liniowo-wykładniczy wzrost. Ciekawostką jest algorytm *A-Close*, który jest odpowiednikiem algorytmu *Apriori* dla domkniętych wzorców częstych, ponieważ osiągnął on drugi rezultat.

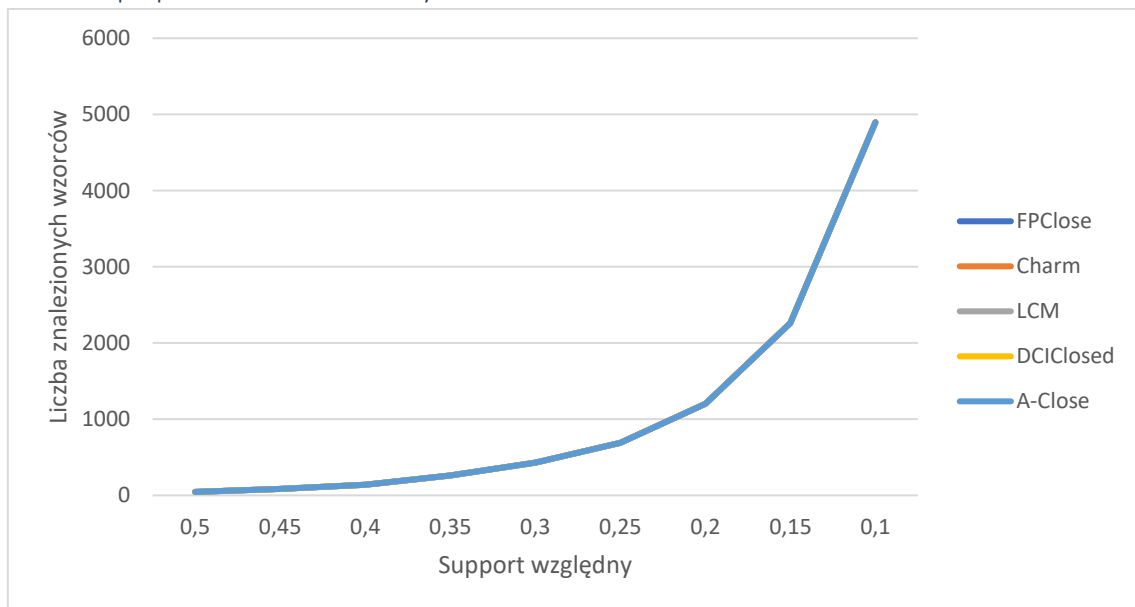
## Analiza wzorców i sekwencji



Wykres 15 - Zależność maksymalnej pamięci od supportu względnego

W przypadku Wykres 15, widzimy, że algorytmy *Charm*, *A-Close* i *FPClose* osiągnęły zerowy przyrost zapotrzebowania na pamięć, względem wartości supportu. Pozostałe dwa algorytmy wykazują wzrost liniowy i bardzo szybko osiągają maksymalną wartość pamięci.

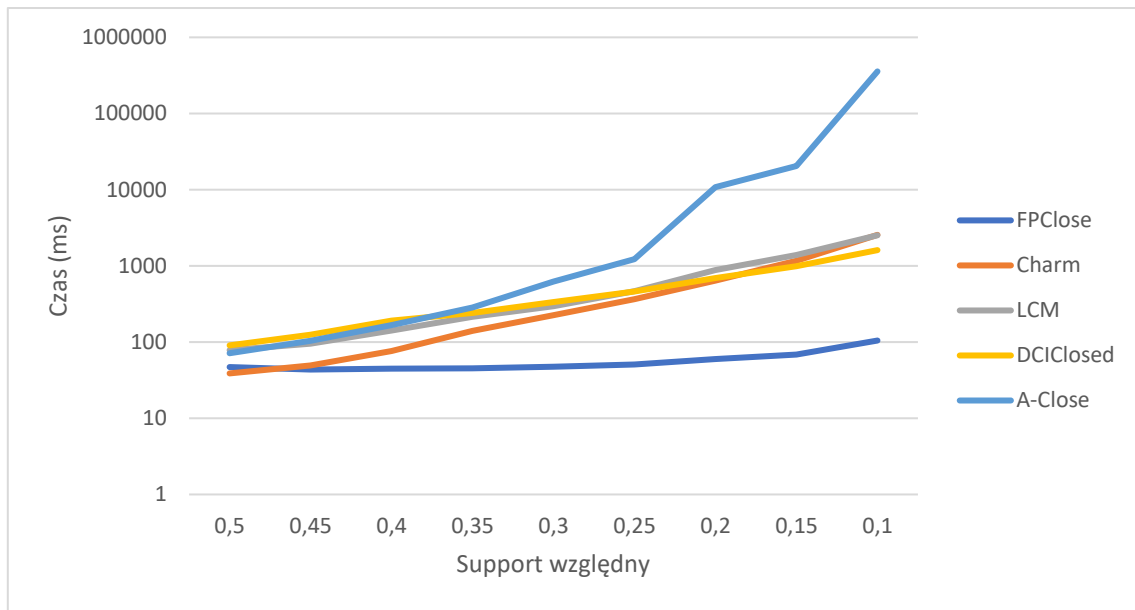
### 5.2.2. Grupa pierwsza – Zbiór danych mushroom



Wykres 16 - Zależność liczby znalezionych wzorców od supportu względnego

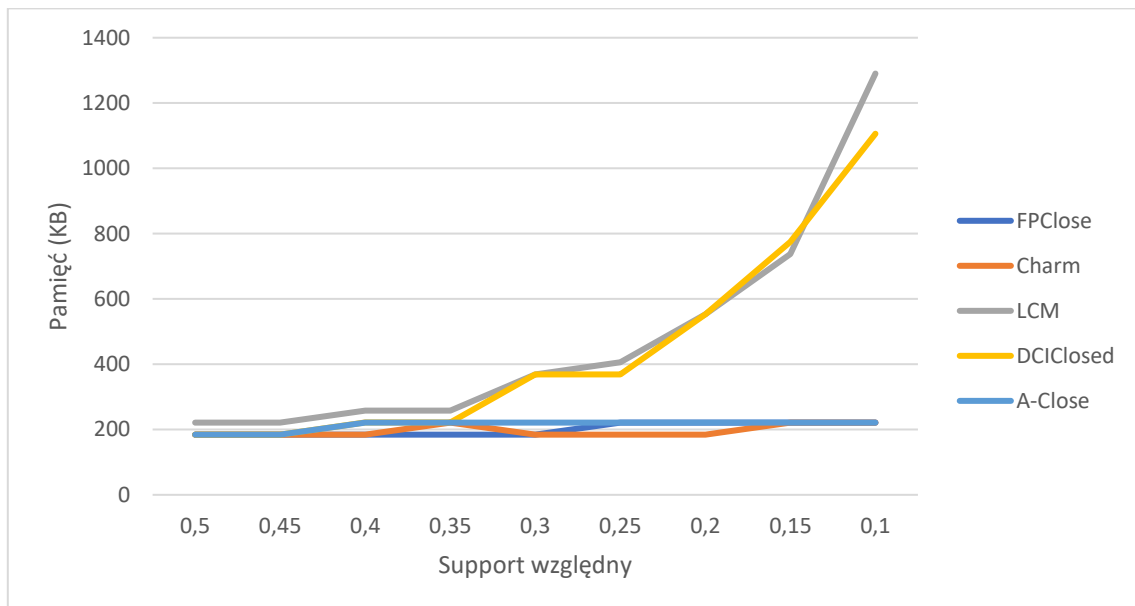
Wykres 16 prezentuje zależność liczby znalezionych wzorców względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę wzorców częstych.

## Analiza wzorców i sekwencji



Wykres 17 - Zależność czasu wykonania w milisekundach od supportu względnego

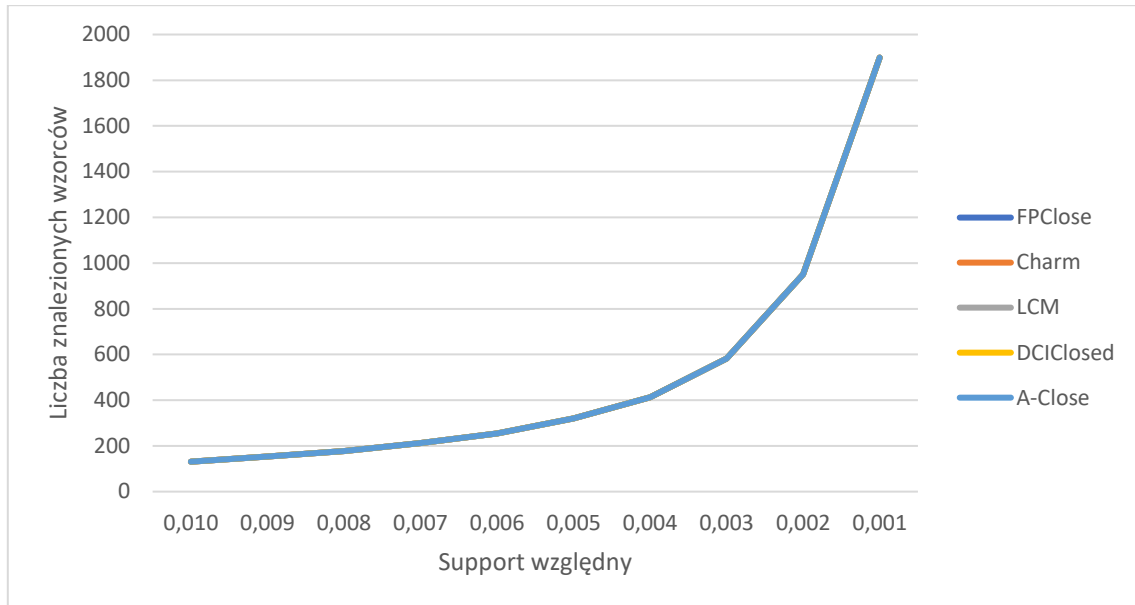
Na Wykres 17 zaprezentowany jest zależność czasu wykonania od użytej wartości wsparcia względnego. Zdecydowanie najlepszym algorytmem jest algorytm *FPClose*. Następnie są trzy algorytmy o zbliżonym czasie wykonania, czyli *DCIClosed*, *Charm* i *LCM*. Najgorszym jest metoda *A-Close*. Każdy algorytm ma wzrost wykładniczy.



Wykres 18 - Zależność maksymalnej pamięci od supportu względnego

Wykres 18 przedstawia zależność użycia pamięci od użytego supportu względnego. Ponownie trzy algorytmy osiągnęły stałą wartość maksymalnej pamięci. Pozostałe dwa, czyli *DCIClosed* i *LCM* wykazują wzrost wykładniczy.

### 5.2.3. Grupa pierwsza – Zbiór danych OnlineRetail



Wykres 19 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 19 prezentuje zależność liczby znalezionych wzorców względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę wzorców częstych.

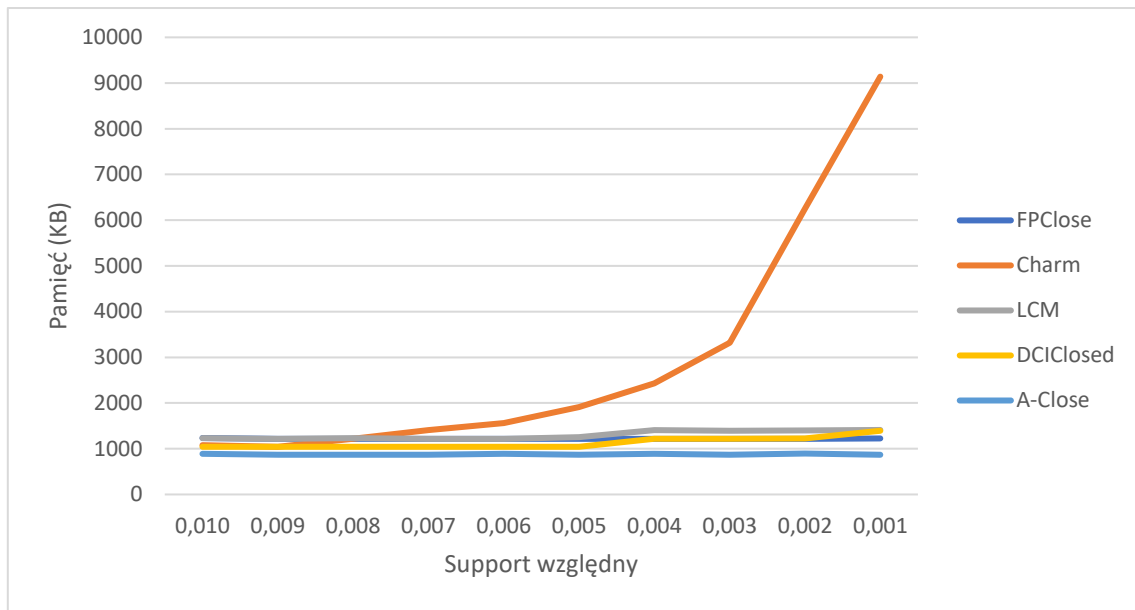


Wykres 20 - Zależność czasu wykonania w milisekundach od supportu względnego

Wykres 20 przedstawia wpływ wartości wsparcia względnego na czas wykonania metody dla zbioru danych OnlineRetail. Podobnie jak przy poprzednich zbiorach (Wykres 14, Wykres 17) najlepszym okazał się algorytm *FPClose*. Pozostałe algorytmy są znacznie gorsze.



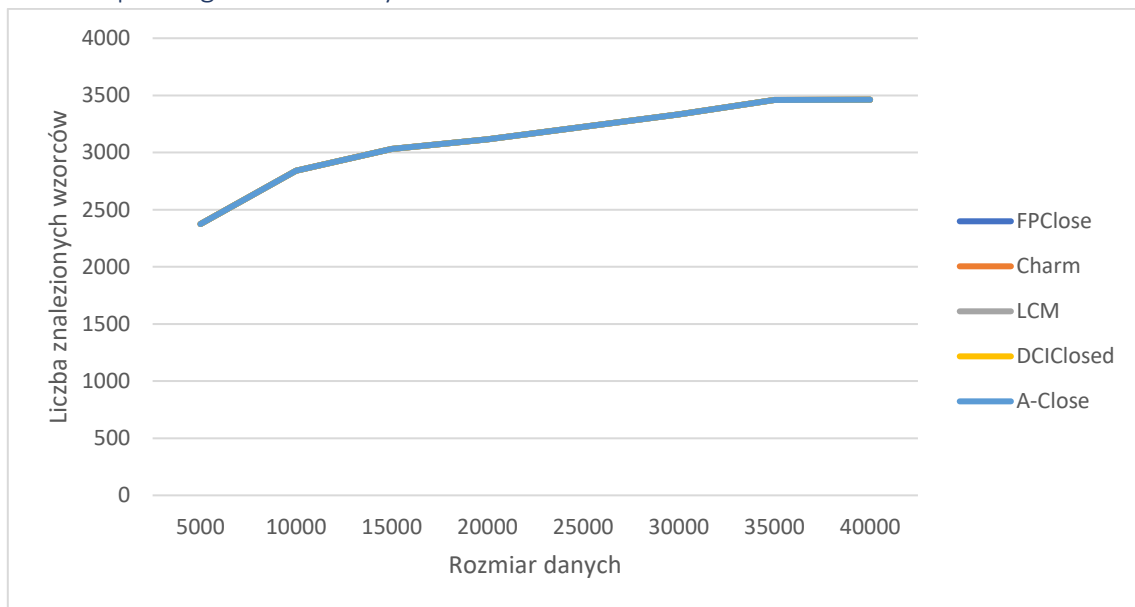
## Analiza wzorców i sekwencji



Wykres 21 - Zależność maksymalnej pamięci od supportu względnego

W przypadku Wykres 21, możemy zauważyć gwałtowny wzrost zapotrzebowania na pamięć przez algorytm *Charm*. Jest to spowodowane różnorodnością zbioru danych OnlineRetail. Pozostałe algorytmy zachowują się zgodnie z przewidywaniami.

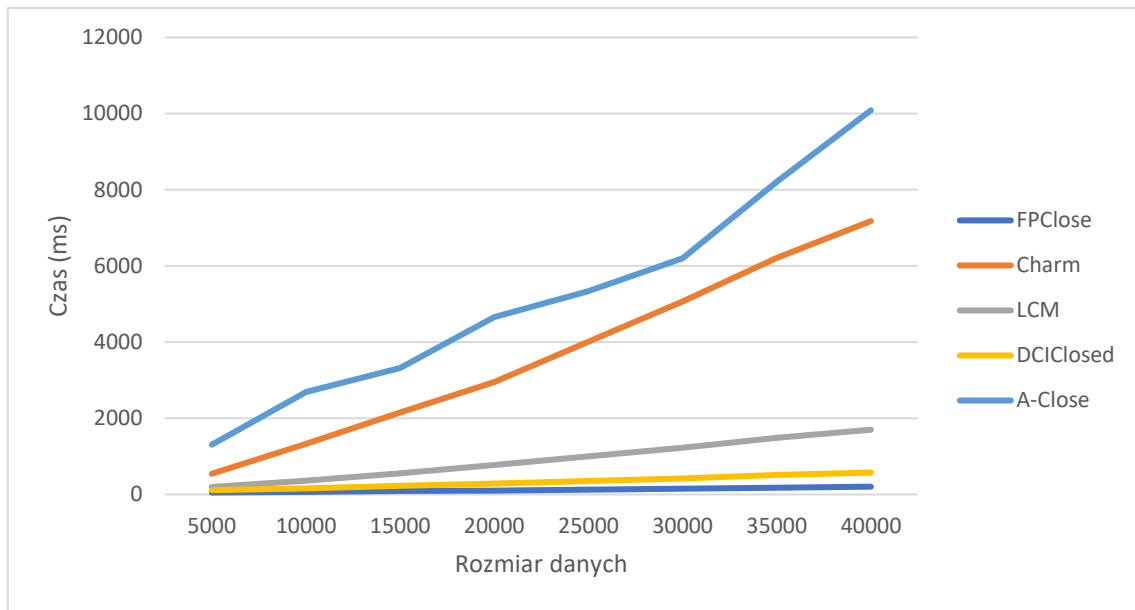
### 5.2.4. Grupa druga – Zbiór danych Kosarak



Wykres 22 - Zależność liczby znalezionych wzorców od rozmiaru danych

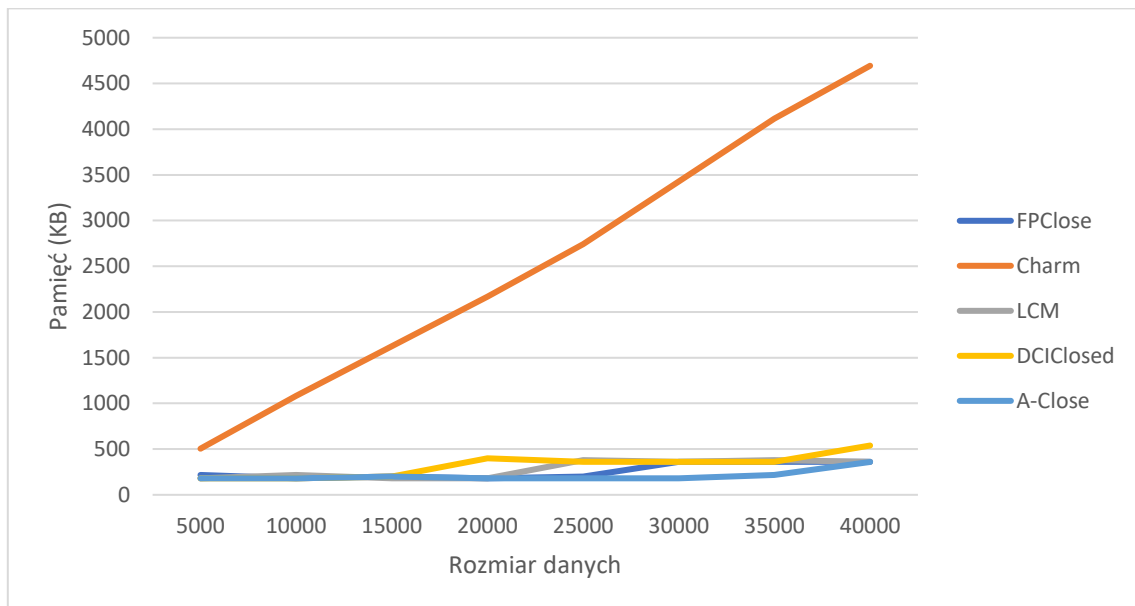
Wykres 22 prezentuje zależność liczby znalezionych wzorców od rozmiaru wejściowego zbioru danych Kosarak. Wszystkie algorytmy znalazły tę samą liczbę wzorców częstych, można więc uznać, że działają poprawnie.

## Analiza wzorców i sekwencji



Wykres 23 - Zależność czasu wykonania w milisekundach od rozmiaru danych

Wykres 23 przedstawia zależność czasu wykonania od rozmiaru danych wejściowych. Najlepszym czasem wykonania i najmniejszym przyrostem charakteryzuje się algorytm *FPClose*. Niewiele gorszy jest algorytm *DCIClosed*. Najgorszym algorytmem są algorytmy *A-Close* i *Charm*. Wszystkie algorytmy wykazują liniową zależność czasu wykonania od rozmiaru zbioru wejściowego.



Wykres 24 - Zależność maksymalnej pamięci od rozmiaru danych

Wykres 24 przedstawia zależność zapotrzebowania na pamięć operacyjną w stosunku do rozmiaru danych wejściowych. Wszystkie algorytmy wykazują liniowy wzrost zapotrzebowania na pamięć operacyjną. Najmniejszy stopień wzrostu wykazuje algorytm *FPClose*, niewiele większy, algorytmy *Charm*, *LCM* i *DCIClosed*. Największy przyrost wykazuje metoda *Charm*. Jest to o tyle zaskakujące, że metoda ta osiągała stosunkowo dobry rezultat w przypadku zbiorów danych *kddcup99* i *mushroom*.

### 5.2.5. Wnioski

Wszystkie metody znalazły tyle samo domkniętych wzorców częstych dla kolejnych wartości supportu czy rozmiaru danych. Na tej podstawie można wyciągnąć wniosek, że wybrane algorytmy prawidłowo znajdują domknięte wzorce częste niezależnie od rozmiaru danych wejściowych, wartości wsparcia czy też ilości prawdopodobnych wzorców.

Wszystkie algorytmy wykazuje wzrost czasu w zależności od wartości wsparcia. Jest to spowodowane rosnącą liczbą wzorców częstych wraz z zmniejszaniem wartości supportu. W większości przypadków zależność ta ma charakter wykładniczy.

Algorytm *FPClose* wykazuje praktycznie zerowy wzrost czasu wykonania, niezależny od wartości supportu. Dopiero przy większej liczbie znajdowanych wzorców, jak to ma miejsce w przypadku zbioru danych *mushroom*, zauważamy przyrost czasu. Pozostałe algorytmy mają zbliżony do siebie nawzajem czas wykonania.

Zdecydowanie najgorszym algorytmem jest *A-Close*, odmiana algorytmu *Apriori* do znajdowania domkniętych wzorców częstych. Dodatkowo metoda ta wykazuje znacznie większy wzrost czasu w stosunku do wartości wsparcia.

W przypadku zależności od rozmiaru danych wejściowych, wszystkie algorytmy wykazują charakter wzrostowy. Wiąże się to bezpośrednio z ilością przetwarzanych transakcji na wejściu. Zależność ta ma charakter liniowy. Najlepszym algorytmem jest *FPClose*.

Dla danych o stałej długości transakcji, czyli *mushroom* i *kddcup99*, najgorszymi algorytmami pod kątem wymagań pamięciowych są metody *DCIClosed* i *LCM*. Nawet dla niewielkiej liczby znajdowanych wzorców osiągały one maksymalną wartość pamięci. Również w przypadku zbioru danych *OnlineRetail* osiągają one gorszy wynik niż pozostałe. Jedynym wyjątkiem w przypadku tego ostatniego zbioru jest algorytm *Charm*, którego zapotrzebowanie rośnie wykładniczo do malejącego wsparcia.

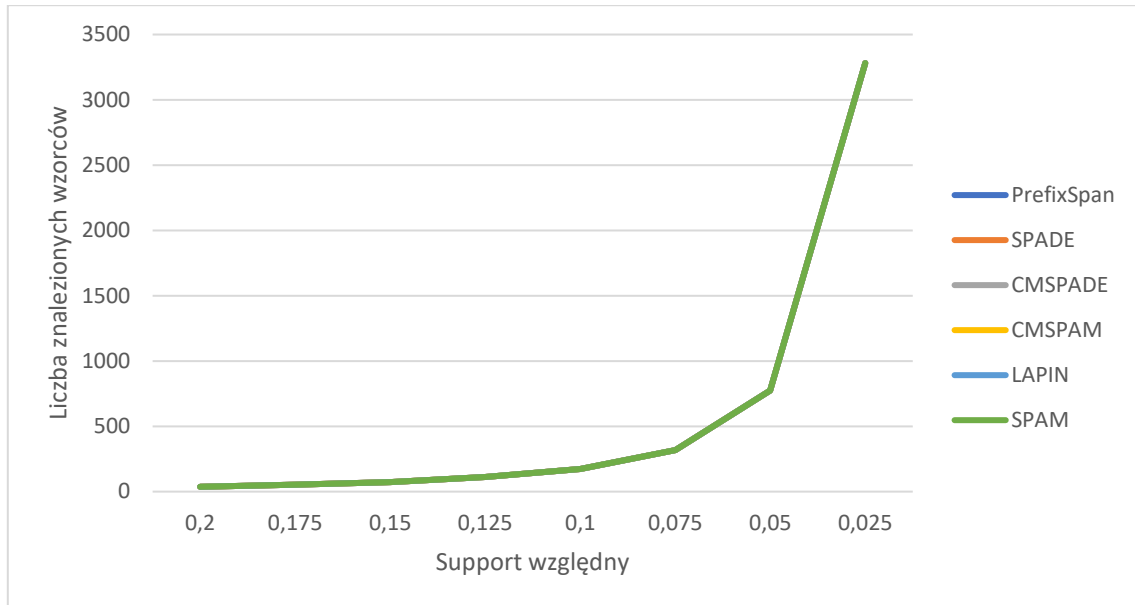
W przypadku eksperymentu z rosnącą liczbą transakcji, wszystkie algorytmy liniowo zwiększają swoje zapotrzebowanie na pamięć operacyjną. Jednakże największy przyrost, rzędu 500MB na 5000 nowych transakcji, osiąga algorytm *Charm*. Dla pozostałych metod wzrost ten jest niewielki.

W przypadku tej kategorii da się jednoznacznie wyznaczyć najlepszy algorytm. W praktycznie każdym przypadku najlepszą metodą okazał się *FPClose*. Algorytm ten wykazuje najmniejszy wzrost czasu wykonania przy wzroście liczby wzorców. Dodatkowo wykorzystuje stałą ilość pamięci niezależnie od wartości wsparcia. Również w eksperymencie ze zwiększającym się rozmiarem zbioru wejściowego wzrost zużycia pamięci jest niewielki.

### 5.3. Metody do wyszukiwania wzorców sekwencji

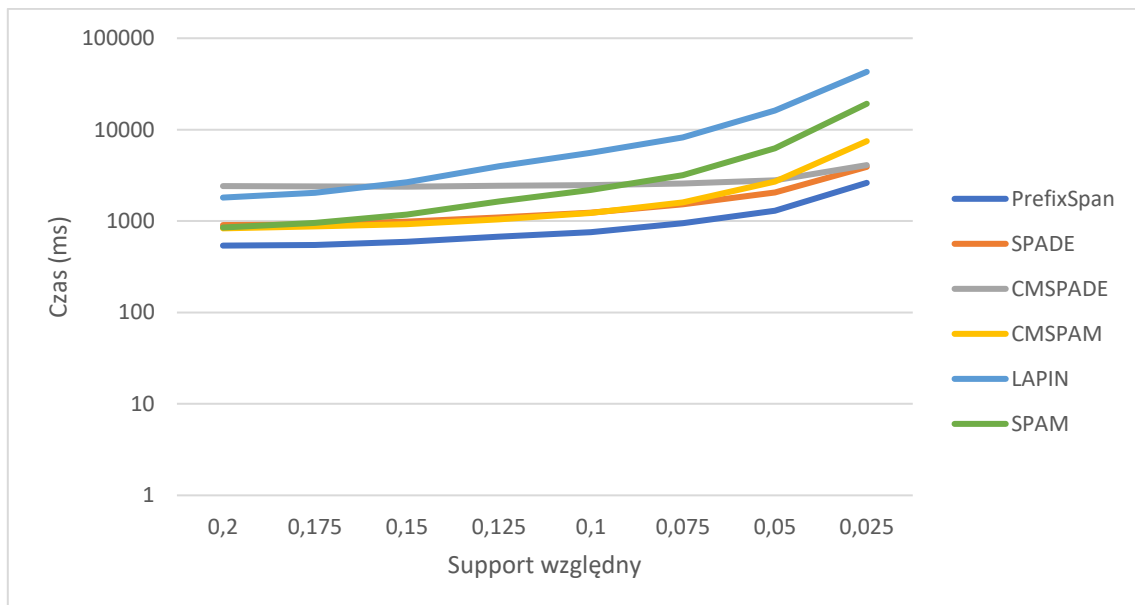
W ramach eksperymentów wykonane zostały trzy testy z grupy pierwszej i jeden testy z grupy drugiej. Zostały przetestowane wszystkie wybrane algorytmy. Wszystkie algorytmy dawały te same wyniki dla tego samego zestawu danych wejściowych.

### 5.3.1. Grupa pierwsza – Zbiór danych Bible



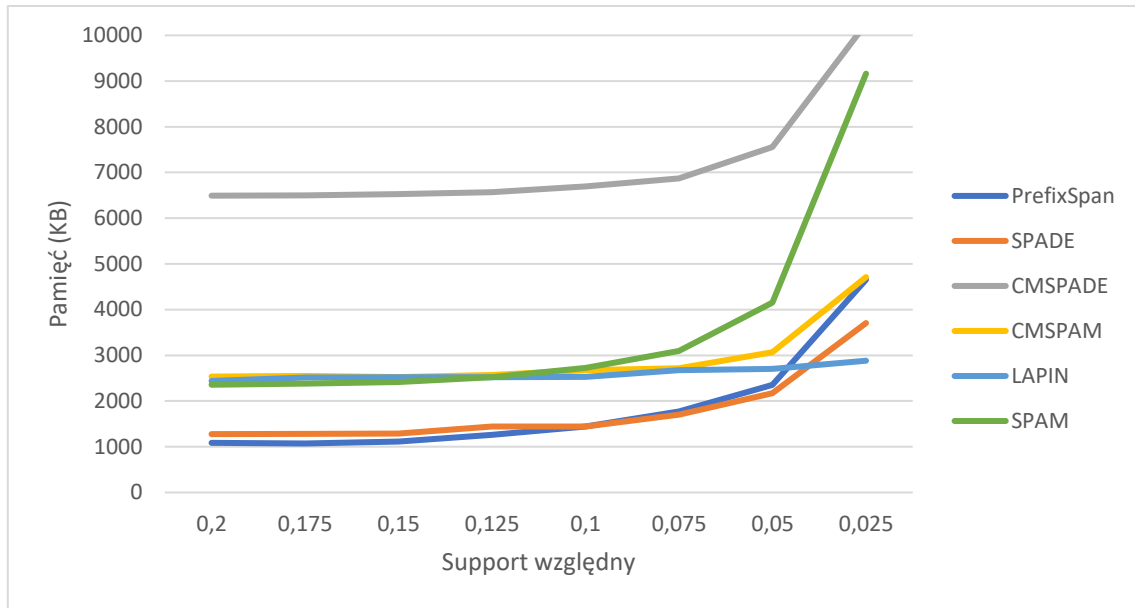
Wykres 25 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 25 prezentuje zależność liczby znalezionych sekwencji od zadanego wsparcia dla zbioru danych Bible. Wszystkie algorytmy znalazły tę samą liczbę sekwencji częstych, można więc uznać, że działają poprawnie.



Wykres 26 - Zależność czasu wykonania w milisekundach od supportu względnego

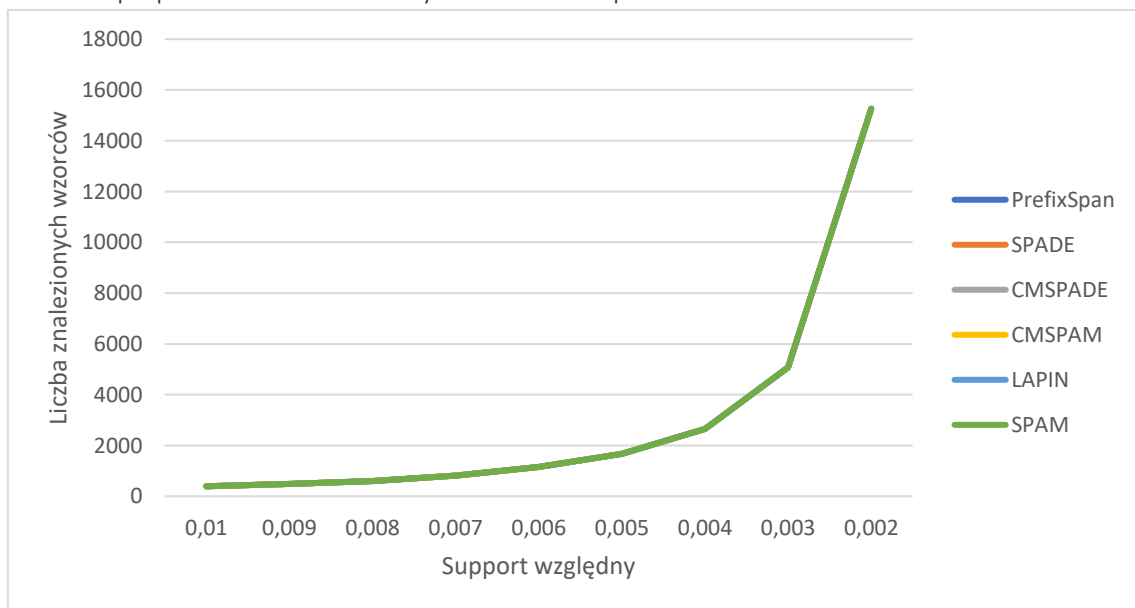
Wykres 26 prezentuje zależność czasu wykonania od supportu względnego dla rodziny algorytmów do szukania sekwencji częstych. Można tutaj jednoznacznie ustalić hierarchię. Najlepszym algorytmem jest *PrefixSpan*. Algorytmy *CMSPAM* i *SPAM* w pierwszym etapie osiągają lepsze rezultaty niż *CMSPADE* i *SPADE*, jednakże w pewnym momencie następuje wzrost czasu wykonania dla tych algorytmów i role się odwracają. Najgorszym algorytmem jest *LAPIN*.



Wykres 27 - Zależność maksymalnej pamięci od supportu względnego

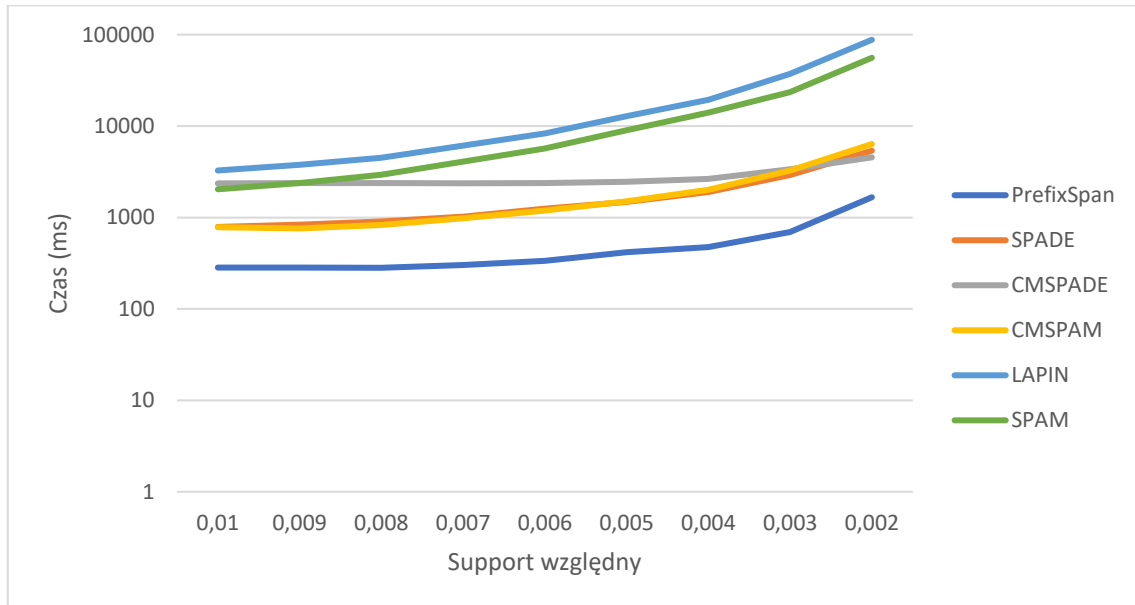
Wykres 27 obrazuje stosunek zapotrzebowania na pamięć operacyjną przez poszczególne metody w stosunku do wartości wsparcia względnego. Największe zapotrzebowanie wykazuje algorytm *CMSPADE*. Algorytm *SPAM* w pierwszym etapie osiąga wynik równy algorytmowi *LAPIN*, jednakże w momencie wzrostu liczby wyszukiwanych wzorców, gwałtownie zwiększa swoje zapotrzebowanie. Algorytm *CMSPAM* zachowuje się podobnie do swojego pierwowzoru, z tym, że stopień wzrostu jest mniejszy. Najlepsze zapotrzebowanie na pamięć operacyjną prezentują algorytmy *SPADE* i *PrefixSpan*. Dopiero przy bardzo dużym wzroście znajdowanych sekwencji, rośnie zapotrzebowanie, jednakże wzrost ten jest umiarkowany. Jedynie metoda *LAPIN* wykazuje stały stosunek, dla pozostałych algorytmów jest to zależność wykładnicza.

## 5.3.2. Grupa pierwsza – Zbiór danych KosarakSeq



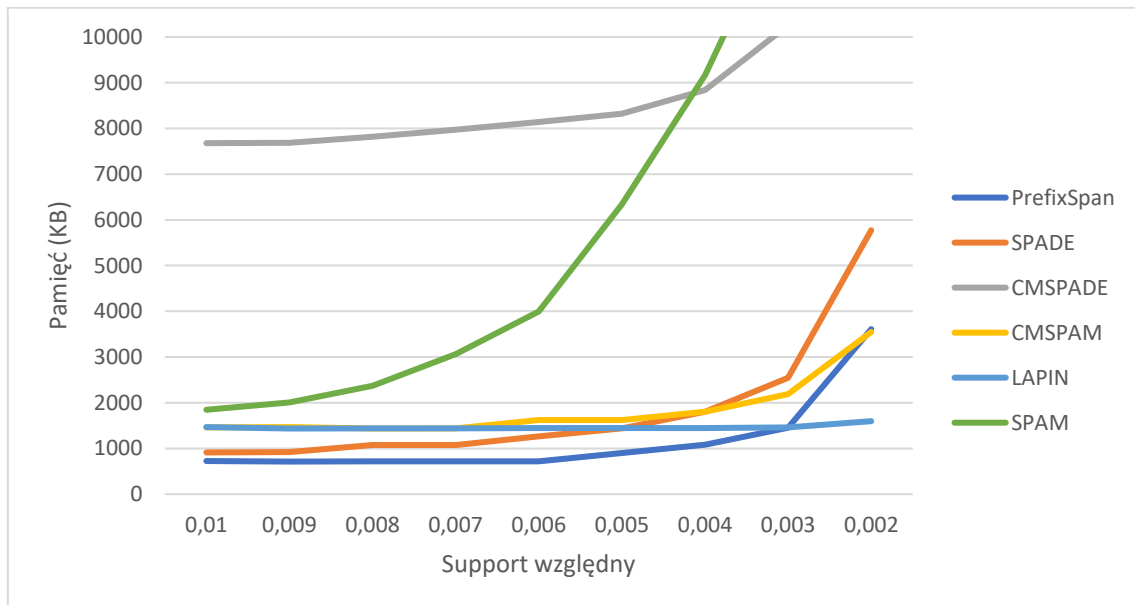
Wykres 28 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 28 prezentuje zależność liczby znalezionych sekwencji względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę sekwencji częstych.



Wykres 29 - Zależność czasu wykonania w milisekundach od supportu względnego

Wykres 29 przedstawia zależność czasu wykonania metody względem użytego wsparcia względnego dla zbioru danych KosarakSeq. Podobnie jak dla poprzedniego zbioru danych najlepszym czasem wykazuje się metoda *PrefixSpan*. W tym przypadku algorytm *CMSPADE* dopiero w ostatnim przypadku ma lepszy czas wykonania względem *CMSPAM* i *SPADE*. Ponownie najgorszym algorytmem jest *LAPIN*.

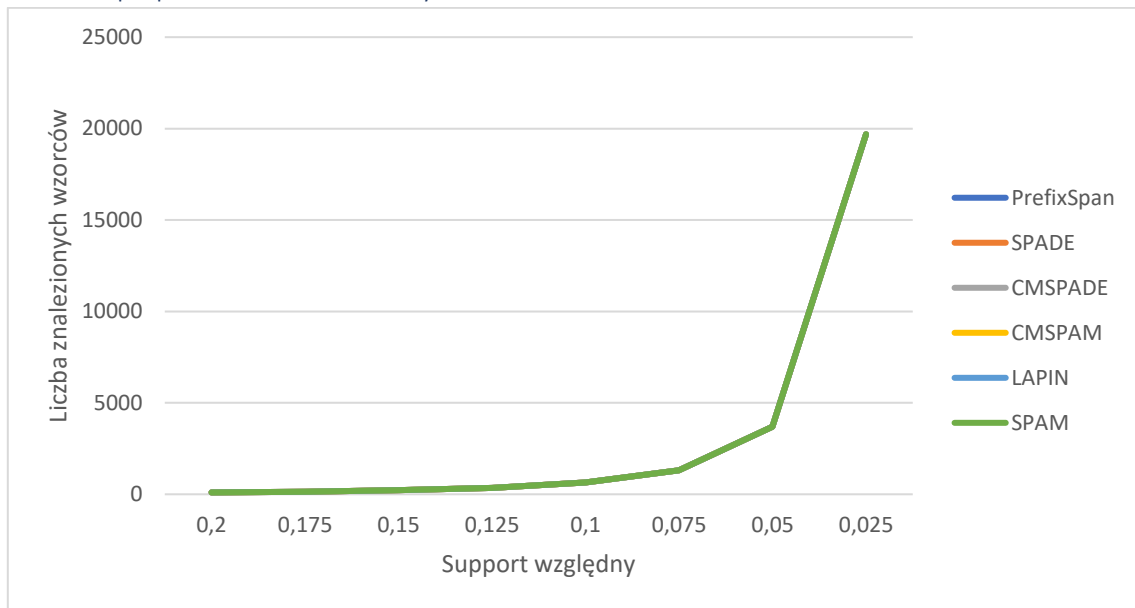


Wykres 30 - Zależność maksymalnej pamięci od supportu względnego

Na Wykres 30 zaprezentowana jest zależność zapotrzebowania na pamięć operacyjną względem wartości wsparcia względnego. Ponownie algorytm *LAPIN* ma prawie stałą wartość pamięci. Dopiero przy bardzo dużej ilości znajdowanych sekwencji, można zauważyć niewielki wzrost. Pozostałe

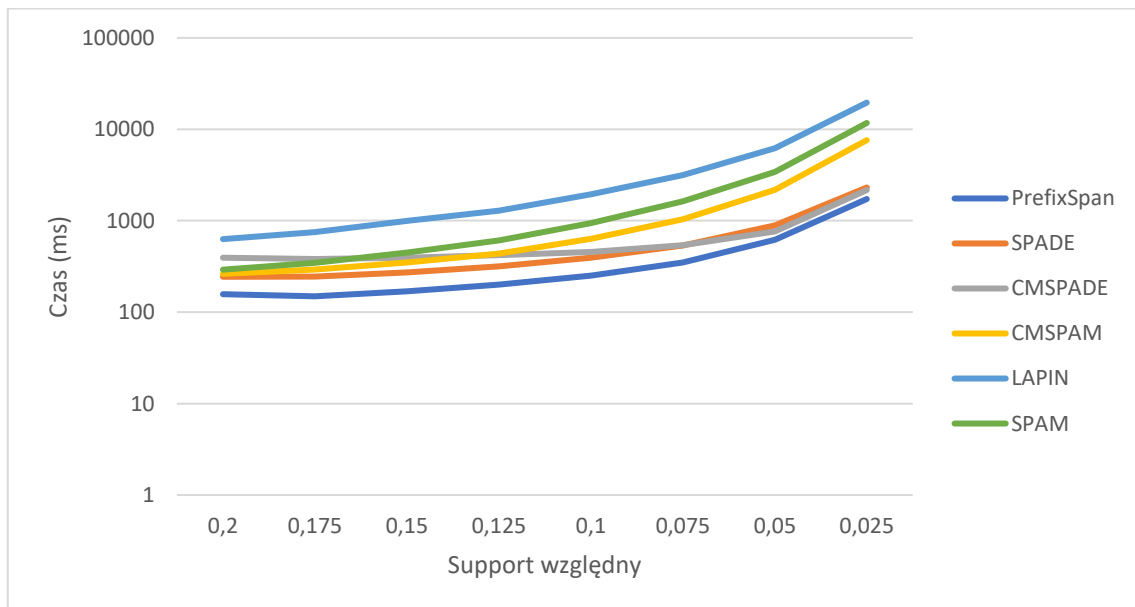
algorytmy zachowują wykładniczy wzrost zapotrzebowania. Najszybszym wzrostem wykazuje się metoda *SPAM*. Najlepszy algorytm to *PrefixSpan*.

### 5.3.3. Grupa pierwsza – Zbiór danych Leviathan



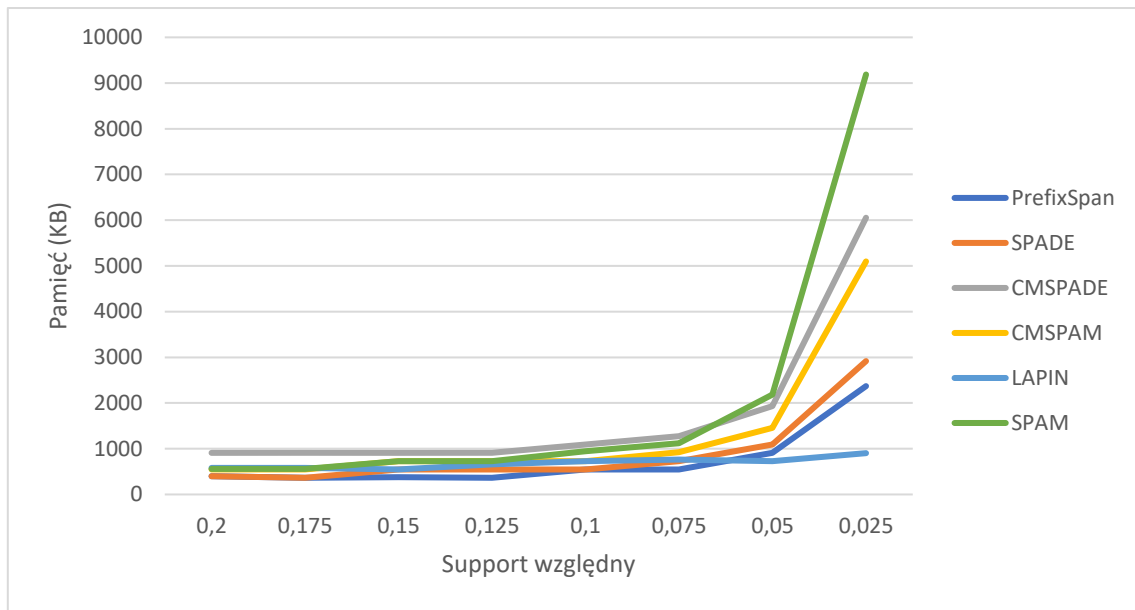
Wykres 31 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 31 prezentuje zależność liczby znalezionych sekwencji względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę sekwencji częstych.



Wykres 32 - Zależność czasu wykonania w milisekundach od supportu względnego

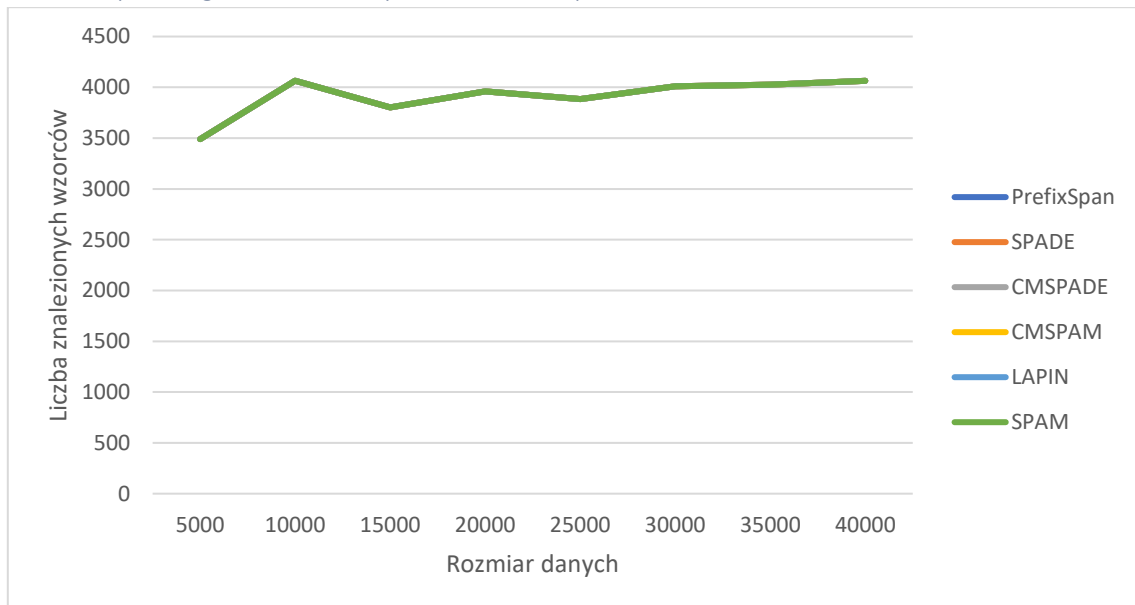
Wykres 32 obrazuje zależność czasu wykonania od supportu względnego dla zbioru danych Leviathan. Algorytmy *SPAM* i *CMSPAM* już na samym początku są gorsze od *SPADE* i *CMSPADE*. Ponownie najlepszym algorytmem jest *PrefixSpan*, a najgorszym *LAPIN*.



Wykres 33 - Zależność maksymalnej pamięci od supportu względnego

Na Wykres 33 zobrazowana jest zależność maksymalnej użytej pamięci do wartości supportu względnego. Podobnie jak w przypadku Wykres 30 i Wykres 27 najlepszym algorytmem *LAPIN* osiągnął stałą wartość pamięci operacyjnej. Pozostałe algorytmy wykazują zależność wykładniczą względem supportu. Najgorszym wzrostem przedstawia się algorytm *SPAM*. Potem w kolejności od najgorszego znajdują się algorytmy *CMSPADE*, *CMSPAM*, *SPADE* i algorytm *PrefixSpan*.

#### 5.3.4. Grupa druga – Zbiór danych KosarakSeq

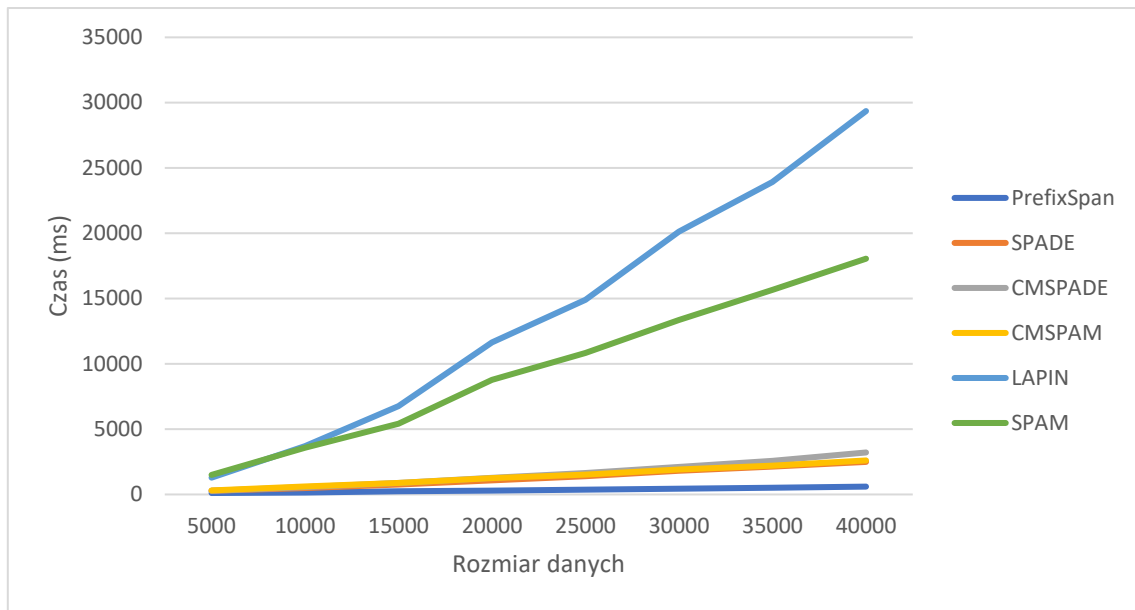


Wykres 34 - Zależność liczby znalezionych wzorców od rozmiaru danych

Wykres 34 prezentuje zależność liczby znalezionych sekwencji od rozmiaru zbioru wejściowego Kosarak. Wszystkie algorytmy znalazły tę samą liczbę sekwencji częstych, można więc uznać, że działają poprawnie.

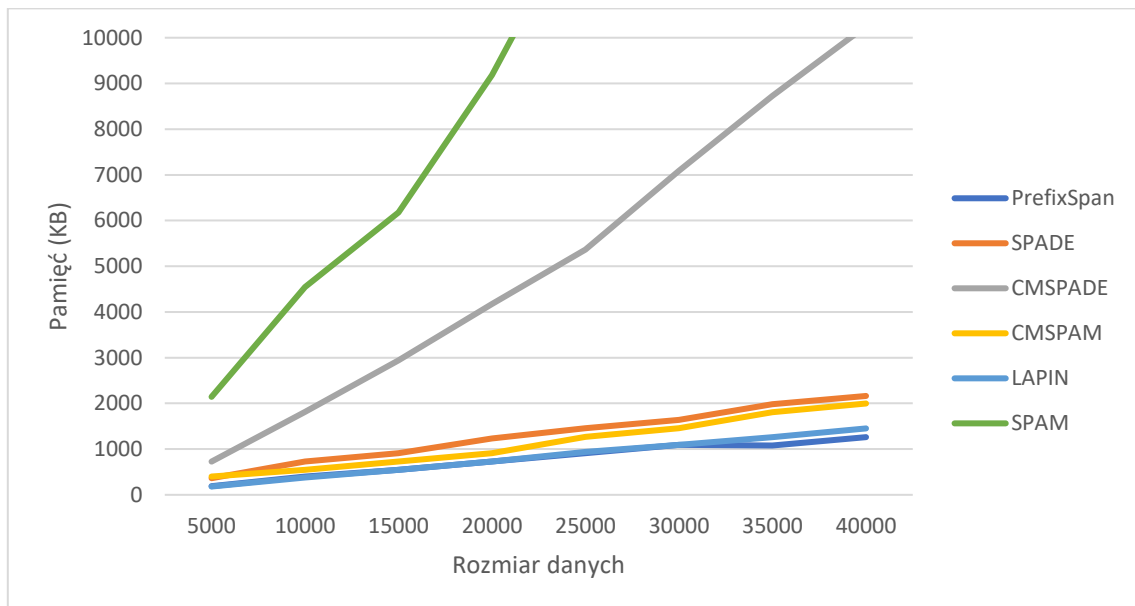


## Analiza wzorców i sekwencji



Wykres 35 - Zależność czasu wykonania w milisekundach od rozmiaru danych

Wykres 35 przedstawia stosunek czasu wykonania algorytmu do rozmiaru danych wejściowych. Podobnie jak w przypadku zmiennego supportu, najgorszym algorytmem okazał się *LAPIN*. Najlepszym algorytmem jest *PrefixSpan*. Niewiele gorszymi są algorytmy *CMSPADE*, *CMSPAM* i *SPADE*. Wszystkie algorytmy wykazują liniowy wzrost czasu wykonania względem rozmiaru danych wejściowych.



Wykres 36 - Zależność maksymalnej pamięci od rozmiaru danych

W przypadku zależności maksymalnej pamięci do rozmiaru danych najlepszym okazał się algorytm *PrefixSpan* na równi z *LAPIN*. Największy wzrost wykazuje algorytm *SPAM*. Algorytmy *CMSPAM* i *SPADE* są niewiele gorsze od najlepszego. Dziwny jest stosunkowo szybki wzrost algorytmu *CMSPADE*, który jest ulepszoną wersją metody *SPADE*, a jest od niego znacząco gorszy. Wszystkie te zależności można zaobserwować na Wykres 36.

### 5.3.5. Wnioski

Wszystkie metody znalazły tyle samo wzorców sekwencji dla kolejnych wartości supportu czy rozmiaru danych. Na tej podstawie można wyciągnąć wniosek, że wybrane algorytmy prawidłowo znajdują wzorce sekwencji niezależnie od rozmiaru danych wejściowych, wartości wsparcia czy też ilości prawdopodobnych wzorców.

Wszystkie algorytmy wykazuje wzrost czasu w zależności od wartości wsparcia. Jest to spowodowane rosnącą liczbą wzorców częstych wraz z zmniejszaniem wartości supportu. W większości przypadków zależność ta ma charakter wykładniczy.

Algorytm *PrefixSpan* w każdym eksperymencie osiąga najlepszy czas wykonania. Wykazuje też stosunkowo mały przyrost czasu w stosunku do wzrostu ilości znajdowanych wzorców. Również w przypadku wymagań dotyczących pamięci plasuje się w czołówce metod. Pamięć rośnie liniowo względem liczby znajdowanych wzorców i jest to najmniejszy przyrost wśród wybranych algorytmów. W przypadku eksperymentu z zmienną liczbą transakcji metoda osiąga najmniejszy stopień przyrostu maksymalnej pamięci oraz czasu wykonania.

Algorytm *LAPIN* wykazuje ciekawe zachowanie w kontekście maksymalnej pamięci. Niezależnie od wartości supportu zużycie pamięci jest stałe. Własność ta zaczyna grać rolę dopiero przy dużej liczbie wzorców. Jednakże zaleta ta jest okupiona największym czasem wykonywania. W eksperymencie ze zmienną liczbą transakcji algorytm wykazuje minimalnie większy przyrost pamięci niż metoda *PrefixSpan* i największy przyrost czasu względem danych.

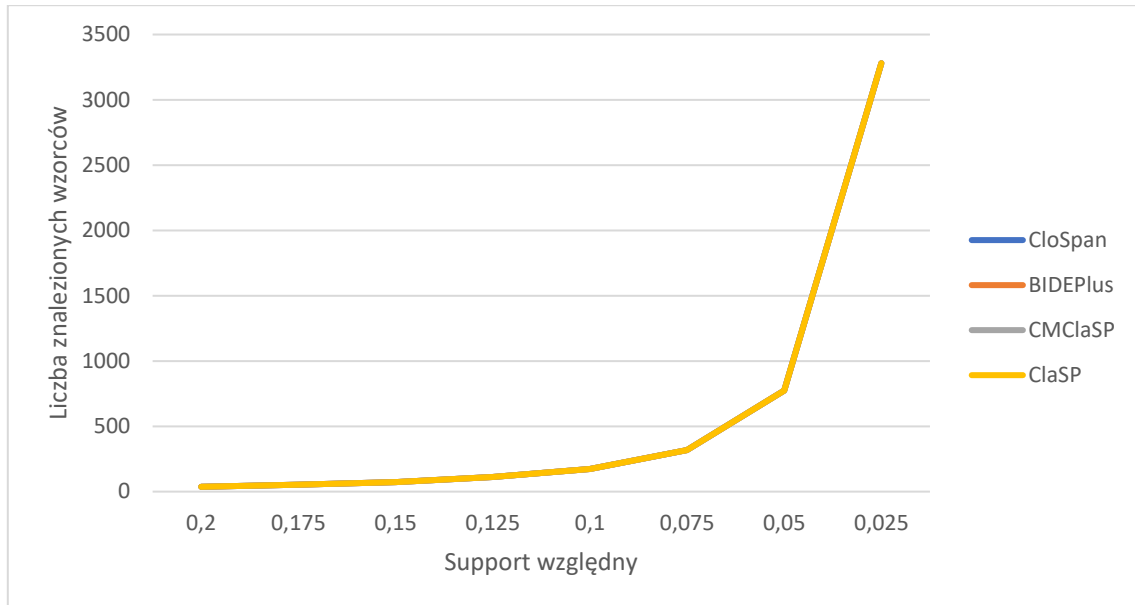
Algorytm *CMSPAM* w każdym przypadku osiąga rezultaty lepsze niż pierwowzór *SPAM*. Jednakże w większości przypadków jest on wolniejszy niż metoda *SPADE*. Ulepszenie, które spełniło swoje zadanie w *CMSPAM* w żaden sposób nie poprawiło działania algorytmu *SPADE*, a większości przypadków spowodowało wzrost czasu wykonania i znaczny wzrost potrzebnej pamięci.

W przypadku tej kategorii najlepszym wyborem jest algorytm *PrefixSpan*. Niewielki wzrost potrzebnej pamięci i czasu wykonania w stosunku do liczby transakcji oraz najlepszy czas wykonania oraz najmniejszy przyrost w stosunku do liczby znajdowanych wzorców sprawia, że jest on najlepszym algorytmem spośród wybranych.

### 5.4. Metody do wyszukiwania domkniętych wzorców sekwencji

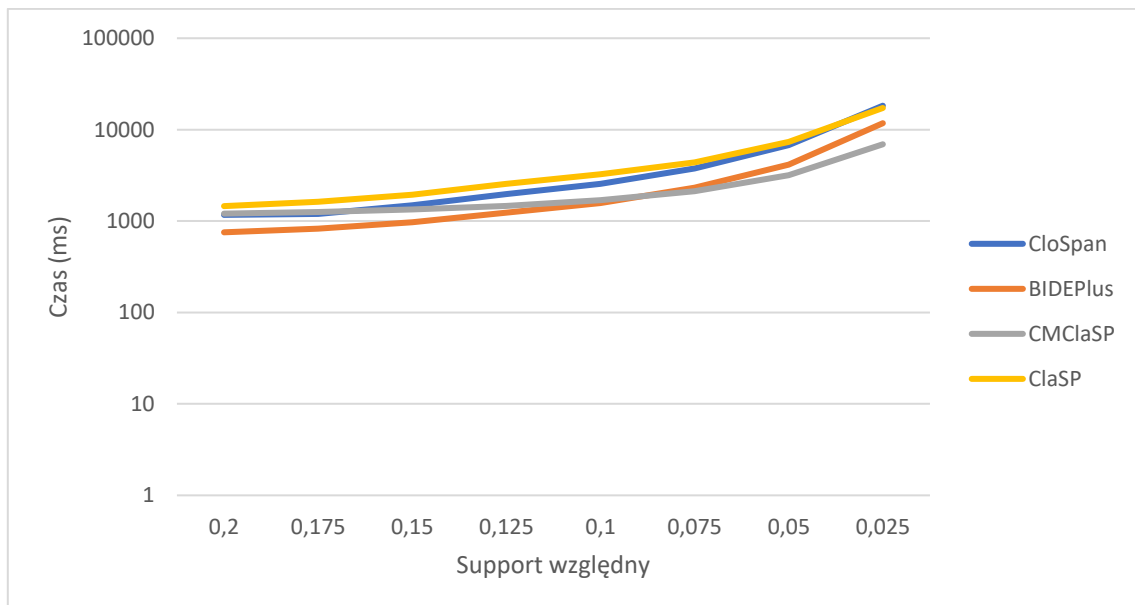
W ramach eksperymentów wykonane zostały trzy testy z grupy pierwszej i jeden testy z grupy drugiej. Zostały przetestowane wszystkie wybrane algorytmy. Wszystkie algorytmy dawały te same wyniki dla tego samego zestawu danych wejściowych.

#### 5.4.1. Grupa pierwsza – Zbiór danych Bible



Wykres 37 - Zależność liczby znalezionych wzorców od supportu względnego

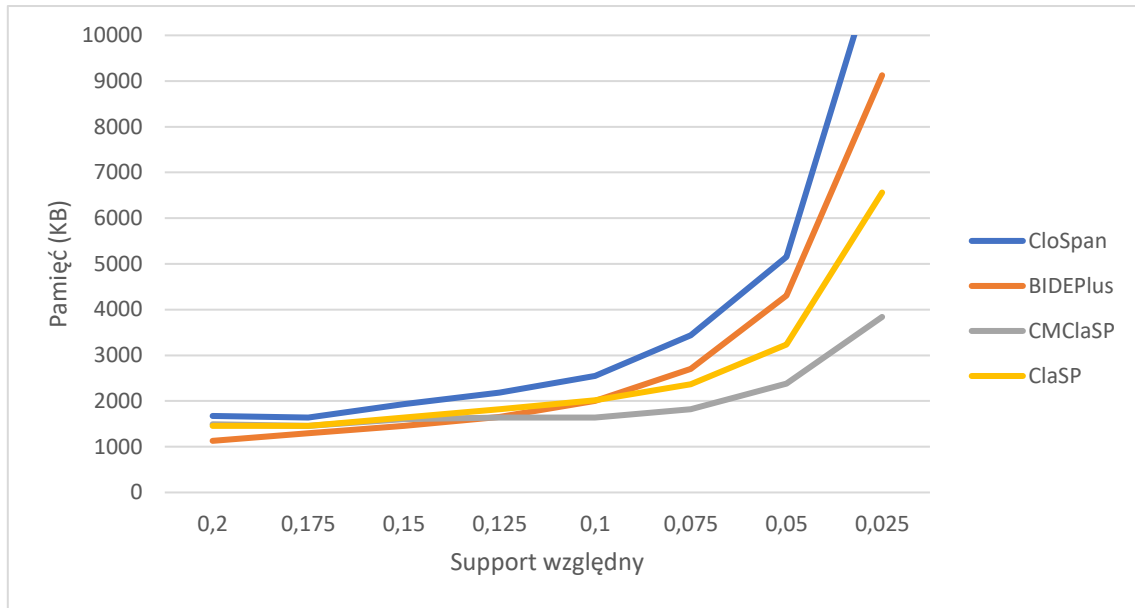
Wykres 37 prezentuje zależność liczby znalezionych sekwencji od rozmiaru zbioru wejściowego Bible. Wszystkie algorytmy znalazły tę samą liczbę sekwencji częstych, można więc uznać, że działają poprawnie.



Wykres 38 - Zależność czasu wykonania w milisekundach od supportu względnego

Na Wykres 38 została zaprezentowana zależność czasu wykonania od wsparcia względnego. Wszystkie algorytmy osiągają zbliżony do siebie czas wykonania. Początkowo najlepszy jest algorytm *BIDE+*, jednakże przy wzroście liczby znajdowanych sekwencji, na prowadzenie wysuwa się algorytm *CMClasP*. Pozostałe dwa algorytmy, czyli *CloSpan* i *ClaSP* na równi zajmują ostatnie miejsce.

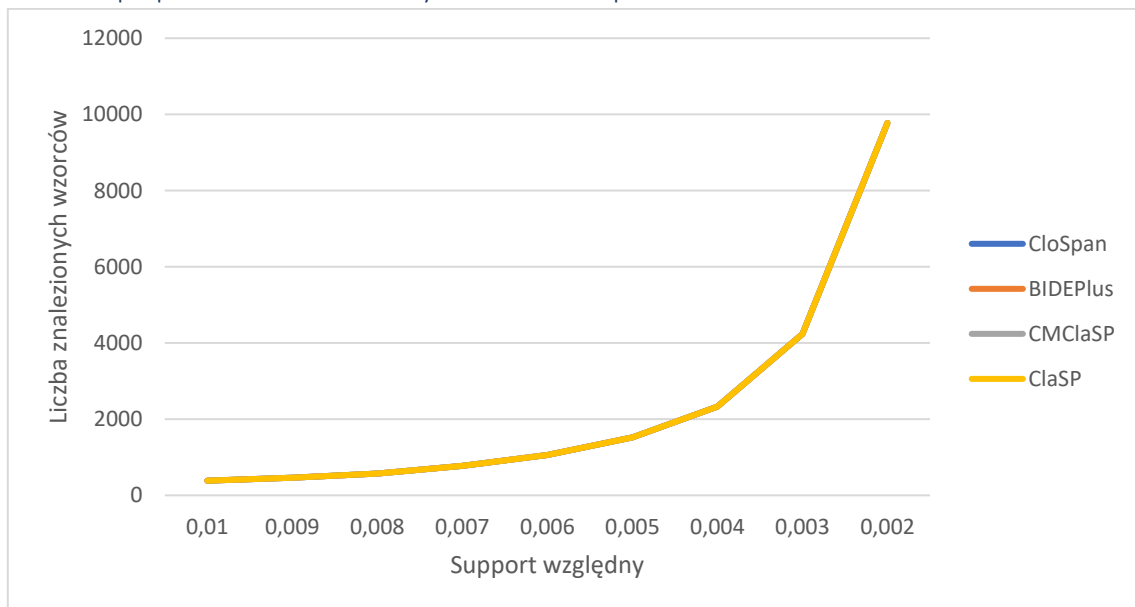
## Analiza wzorców i sekwencji



Wykres 39 - Zależność maksymalnej pamięci od supportu względnego

Wykres 39 prezentuje stosunek zapotrzebowania pamięci w zależności od użytej wartości wsparcia względnego. Wszystkie algorytmy wykazują wykładniczy wzrost. Najmniejszy wzrost wykazuje algorytm *CMClasP*, co utwierdza go na prowadzeniu. Najgorszy jest algorytm *CloSpan*.

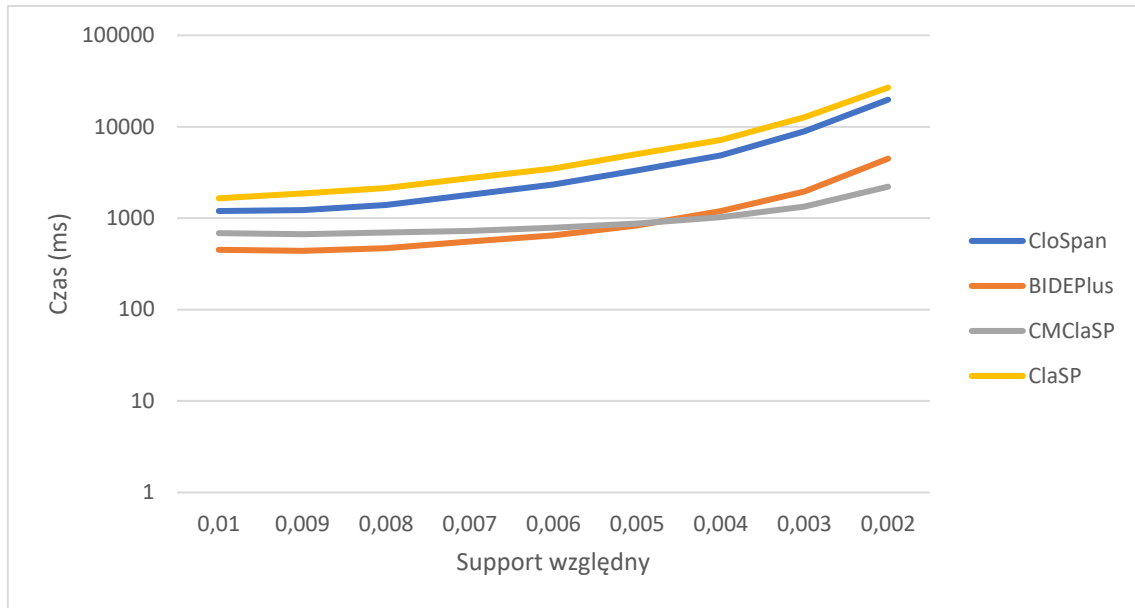
### 5.4.2. Grupa pierwsza – Zbiór danych KosarakSeq



Wykres 40 - Zależność liczby znalezionych wzorców od supportu względnego

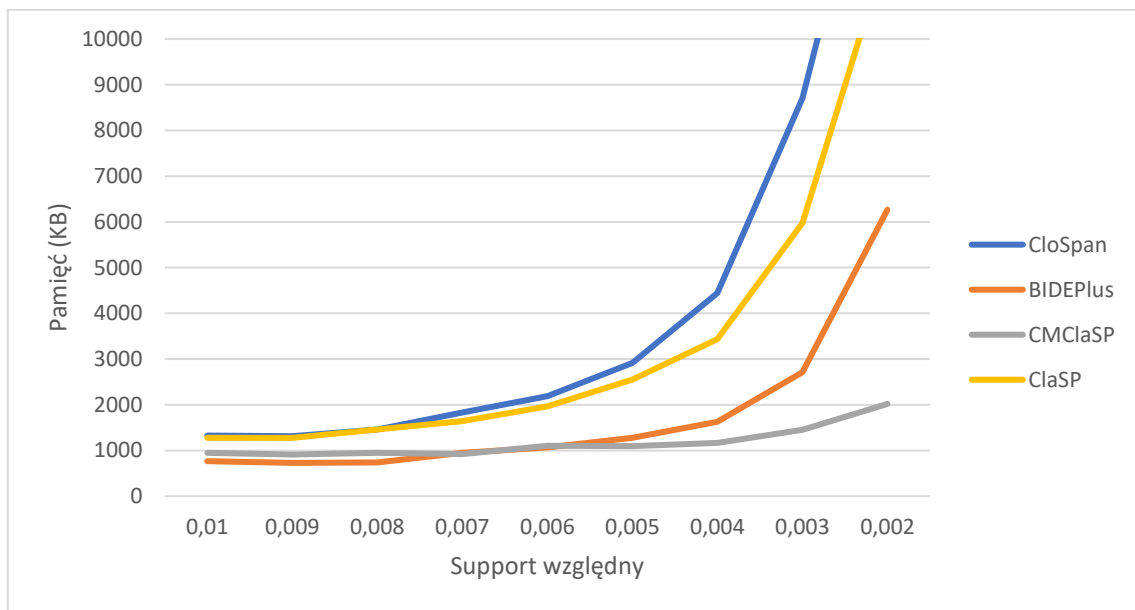
Wykres 40 prezentuje zależność liczby znalezionych sekwencji od zadanego wsparcia dla zbioru danych KosarakSeq. Wszystkie algorytmy znalazły tę samą liczbę sekwencji częstych, można więc uznać, że działają poprawnie.

## Analiza wzorców i sekwencji



Wykres 41 - Zależność czasu wykonania w milisekundach od supportu względnego

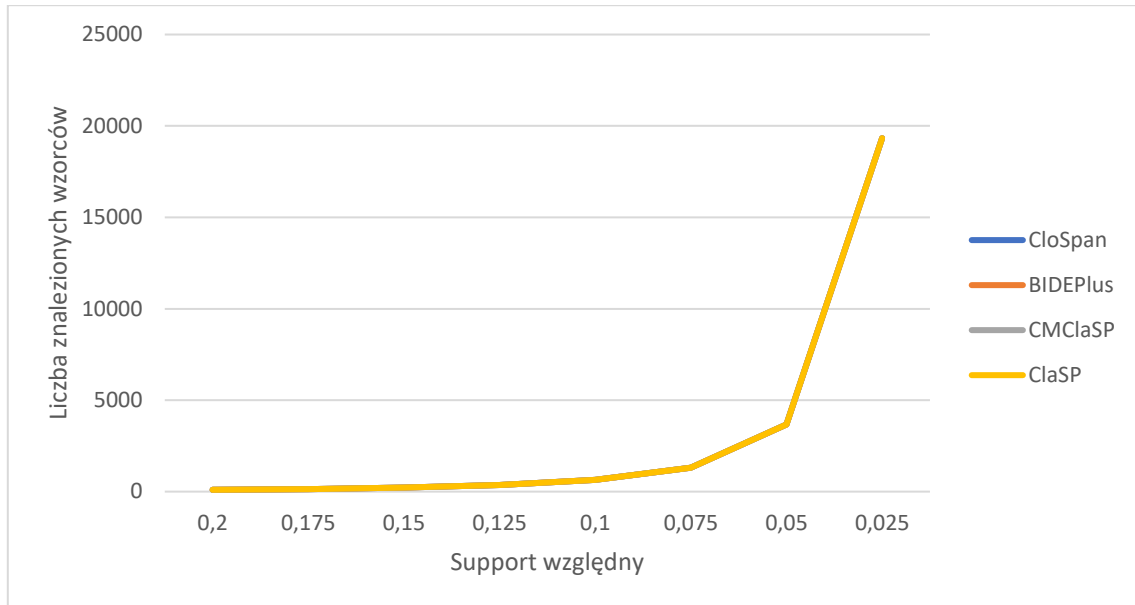
Wykres 41 obrazuje stosunek czasu wykonania od wartości supportu dla zbioru danych KosarakSeq. Podobnie jak w poprzednim przypadku, dla małej ilości znajdowanych wzorców najlepszy jest algorytm *BIDE+*, a dla większej ilości *CMClasP*. W tym przypadku najgorszy jest algorytm *ClaSP*.



Wykres 42 - Zależność maksymalnej pamięci od supportu względnego

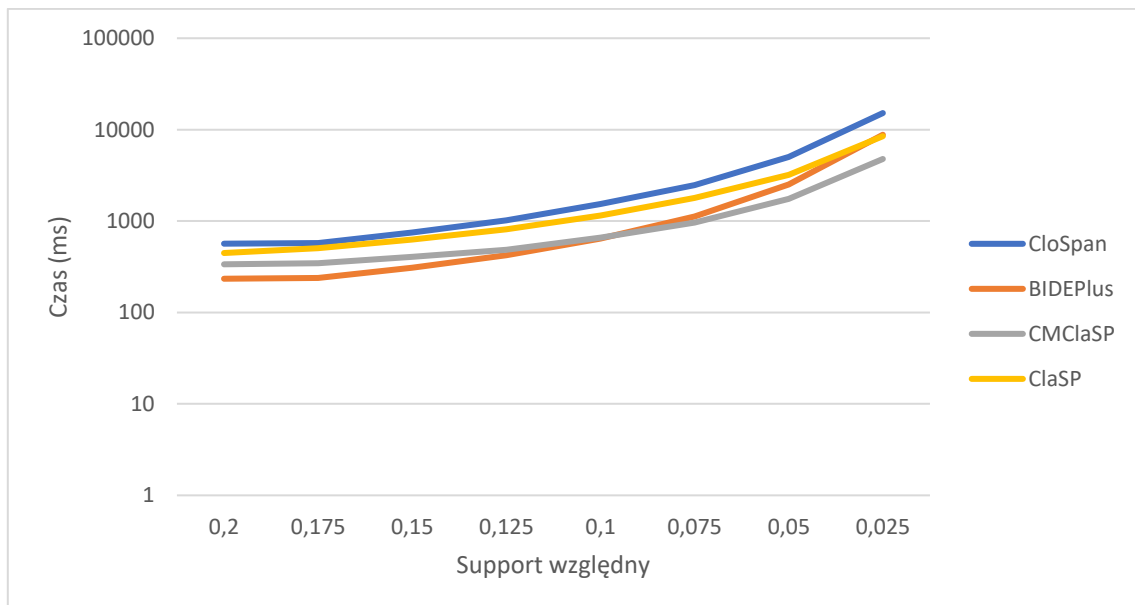
Wykres 42 przedstawia zależność zapotrzebowania na pamięć w stosunku do wartości supportu. W tym przypadku najlepszym jest algorytm *CMClasP*. Potem, w kolejności od najlepszego, znajdują się algorytmy *BIDE+*, *ClaSP* i *CloSpan*. Wszystkie algorytmy wykazują wzrost wykładniczy.

#### 5.4.3. Grupa pierwsza – Zbiór danych Leviathan



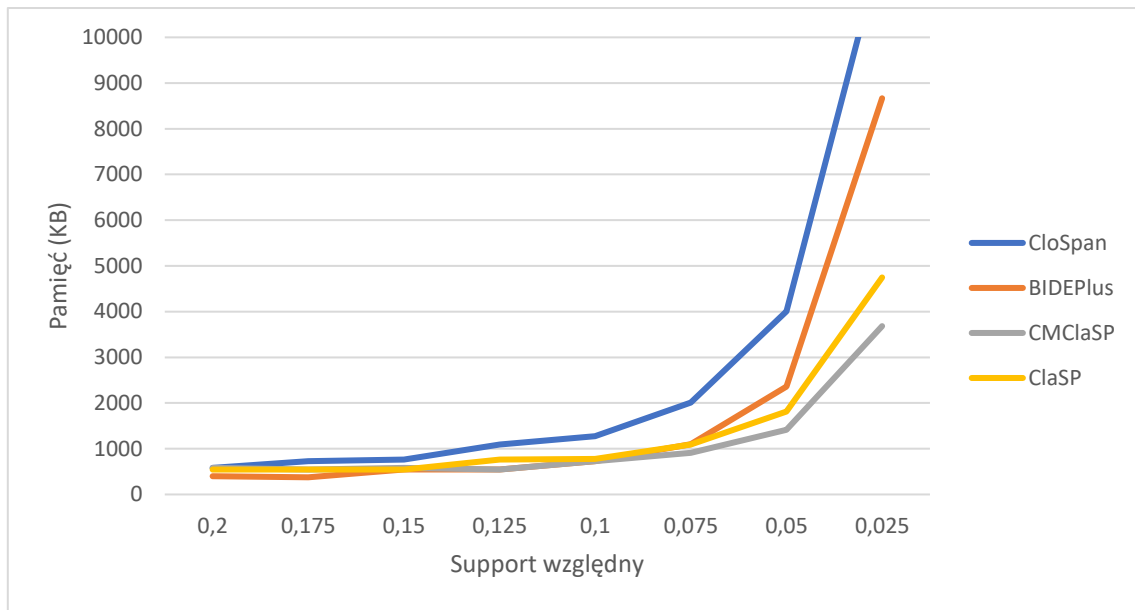
Wykres 43 - Zależność liczby znalezionych wzorców od supportu względnego

Wykres 43 prezentuje zależność liczby znalezionych sekwencji od zadanego wsparcia dla zbioru danych Leviathan. Wszystkie algorytmy znalazły tę samą liczbę sekwencji częstych, można więc uznać, że działają poprawnie.



Wykres 44 - Zależność czasu wykonania w milisekundach od supportu względnego

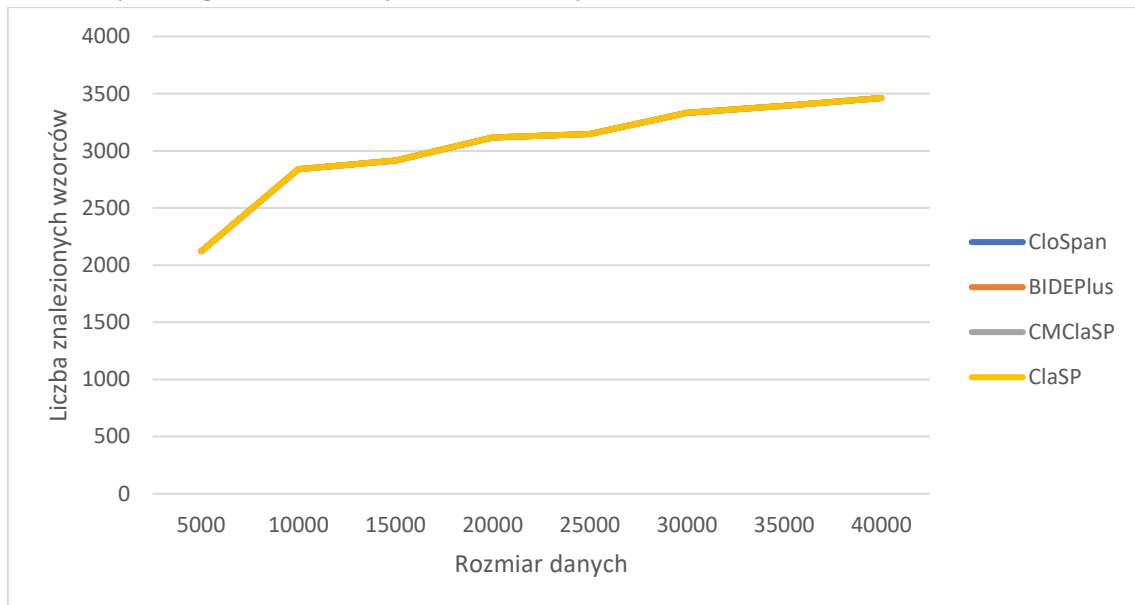
Wykres 44 obrazuje wzrost czasu wykonania wraz z malejącą wartością wsparcia. Podobnie jak w przypadku Wykres 41 i Wykres 38, można zauważyć granicę, po przekroczeniu której algorytm *BIDE+* przestaje być najlepszy na rzecz metody *CMClasP*. Algorytm *CloSpan* jest najgorszy. Algorytm *ClaSP* wykazuje mniejszy wzrost niż algorytm *BIDE+*, w wyniku czego algorytm ten może osiągnąć lepsze rezultaty niż *BIDE+* wraz z dalszym maleniem supportu.



Wykres 45 - Zależność maksymalnej pamięci od supportu względnego

Wykres 45 ilustruje stosunek zapotrzebowania na pamięć operacyjną względem wartości supportu względnego. Podobnie jak w poprzednich przypadkach, najmniejszym zapotrzebowaniem jak i najmniejszym wzrostem charakteryzuje się algorytm *CMClasp*. Algorytm *ClaSP* osiąga porównywalne wyniki, jednakże wykazuje się minimalnie większym wzrostem. Najgorszy jest algorytm *CloSpan*, którego wzrost jest największy.

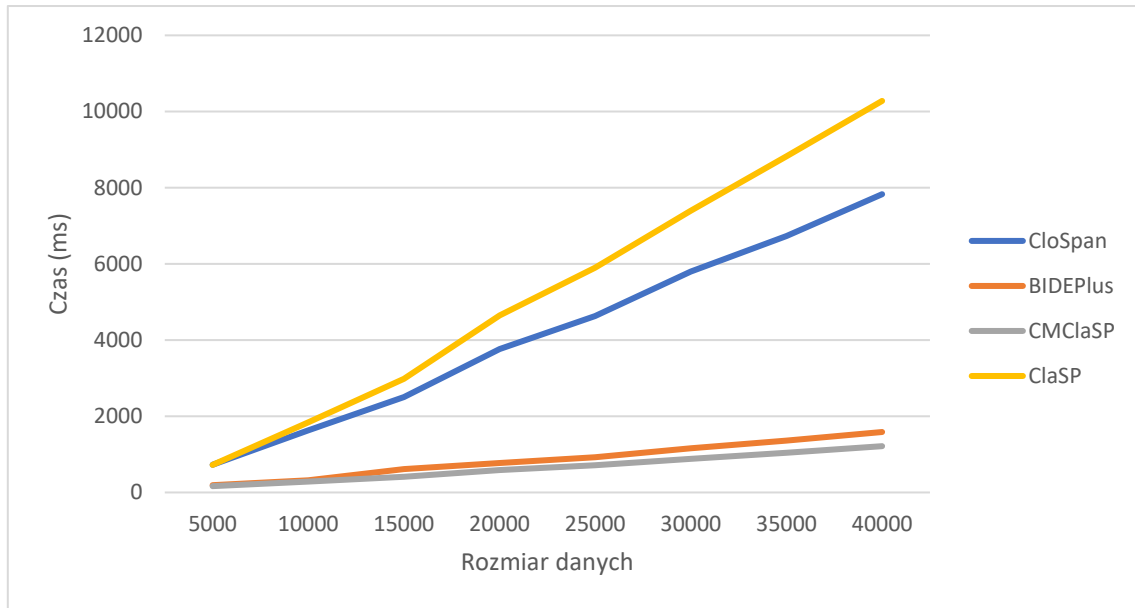
#### 5.4.4. Grupa druga – Zbiór danych KosarakSeq



Wykres 46 - Zależność liczby znalezionych wzorców od rozmiaru danych

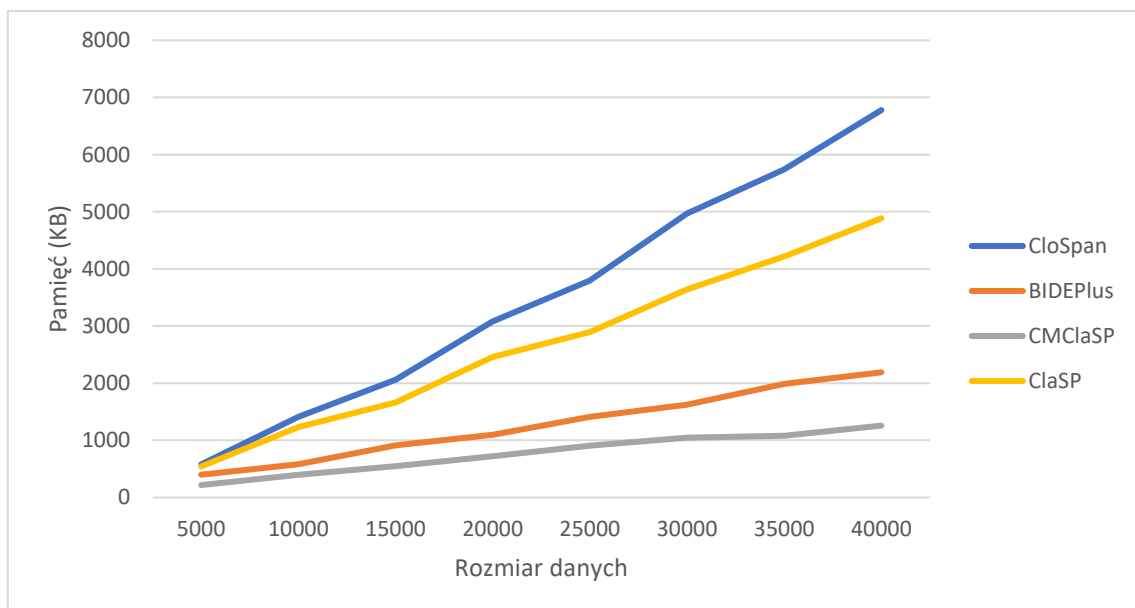
Wykres 46 prezentuje zależność liczby znalezionych sekwencji względem użytej wartości wsparcia względnego. Wszystkie użyte algorytmy działają poprawnie i znajdują tę samą liczbę sekwencji częstych.

## Analiza wzorców i sekwencji



Wykres 47 - Zależność czasu wykonania w milisekundach od rozmiaru danych

Na Wykres 47 możemy zaobserwować wzrost czasu wykonania metody względem rozmiaru wejściowego zbioru danych. Dla każdego algorytmu wzrost ten jest liniowy. Najmniejszą wartość pamięci jak i najmniejszy wzrost wykazuje algorytm *CMClasP*. Niewiele gorszy okazuje się algorytm *BIDE+*. Zaskakującym jest algorytm *ClaSP*, który osiąga najgorsze rezultaty zarówno pod względem wartości jak i wzrostu.



Wykres 48 - Zależność maksymalnej pamięci od rozmiaru danych

Wykres 48 przedstawia zależność maksymalnej osiągniętej pamięci operacyjnej od rozmiaru wejściowego zbioru danych dla zbioru KosarakSeq. W przypadku pamięci najgorszym algorytmem okazał się *CloSpan*. Algorytm *CMClasP* ponownie jest najlepszym algorytmem zarówno pod względem wartości jak i stopnia wzrostu. Wszystkie algorytmy wykazują liniową zależność maksymalnej pamięci od rozmiaru danych.



#### 5.4.5. Wnioski

Wszystkie metody znalazły tyle samo domkniętych wzorców sekwencji dla kolejnych wartości supportu czy rozmiaru danych. Na tej podstawie można wyciągnąć wniosek, że wybrane algorytmy prawidłowo znajdują wzorce sekwencji niezależnie od rozmiaru danych wejściowych, wartości wsparcia czy też ilości prawdopodobnych wzorców.

Wszystkie algorytmy wykazuje wzrost czasu w zależności od wartości wsparcia. Jest to spowodowane rosnącą liczbą wzorców częstych wraz z zmniejszaniem wartości supportu. W większości przypadków zależność ta ma charakter wykładniczy.

Wszystkie metody w liniowy sposób zwiększają zapotrzebowanie na pamięć operacyjną jak i liniowy stosunek czasu wykonania do liczby transakcji w zbiorze wejściowym.

Algorytm *CMClasp*, będący ulepszoną wersją algorytmu *Clasp*, ma najmniejsze zużycie pamięci ze wszystkich 4 metod oraz najmniejszy przyrost w stosunku do przyrostu liczby wzorców. W przypadku rosnącej liczby transakcji algorytm ten ma najmniejszy współczynnik przyrostu pamięci maksymalnej oraz czasu wykonania.

Algorytm *BIDEPlus* w porównaniu do *CMClasp* osiąga lepsze wyniki przy małej liczbie znajdowanych wzorców. Jednakże maksymalne zużycie pamięci jest wyższe niezależnie do wartości supportu oraz rośnie szybciej względem *CMClasp*.

Pozostałe dwa algorytmy, czyli *CloSpan* i *Clasp* osiągają zbliżone do siebie osiągają zbliżone do siebie czasy wykonania, z minimalną przewagą na korzyść *CloSpan*. Jednakże wydać znaczenie większe różnice w zajętości pamięci operacyjnej. Metoda *CloSpan* zdecydowanie zużywa najwięcej pamięci i osiąga największy przyrost zarówno w stosunku do liczby znajdowanych wzorów jak i liczby transakcji podanych na wejściu.

W przypadku tej kategorii najlepszą metodą jest *CMClasp*. Stosunkowo niewielki wzrost czasu wykonania i pamięci maksymalnej przy wzroście liczby transakcji oraz najlepszy czas wykonania plasuje ten algorytm na pierwszym miejscu. Z drugiej strony liczba algorytmów do wyszukiwania domkniętych wzorców sekwencji jest niewielka w porównaniu do liczby metod odkrywania wzorców częstych sugeruje powstanie nowych i lepszych metod w przyszłości.

## 6. Wnioski ogólne

Wszystkie metody znalazły tyle samo domkniętych wzorców dla kolejnych wartości supportu czy rozmiaru danych (Wykres 1, Wykres 4, Wykres 7, Wykres 10, Wykres 13, Wykres 16, Wykres 19, Wykres 22, Wykres 25, Wykres 28, Wykres 31, Wykres 34, Wykres 37, Wykres 40, Wykres 43, Wykres 46). Na tej podstawie można wyciągnąć wniosek, że wybrane algorytmy prawidłowo znajdują wzorce niezależnie od rozmiaru danych wejściowych, wartości wsparcia czy też ilości prawdopodobnych wzorców. Niezależnie od celu analizy większość metod wykazuje liniowy wzrost czasu wykonania i maksymalnej pamięci operacyjnej do ilości transakcji w zbiorze wejściowych.

W przypadku metod do wyszukiwania wzorców częstych można wymienić 3 najlepsze algorytmy: *PrePost+*, *FPGrowth* i *PrePost*. Algorytmy te wykazują bardzo niewielki wzrost czasu wykonania (Wykres 2, Wykres 5, Wykres 8) i maksymalnej pamięci operacyjnej w stosunku do liczby znajdowanych wzorców (Wykres 3, Wykres 6, Wykres 9). Dopiero przy bardzo dużym wzroście liczby wzorców można zaobserwować wzrost obydwoch mierzonych składników. Wzrost ten jest wykładniczy. W większości przypadków algorytm *Apriori* osiąga najlepsze rezultaty pod względem zużycia pamięci, jednakże czas wykonania algorytm jest nieakceptowalnie długi.

Dla metod do wyszukiwania domkniętych wzorców częstych najlepszym algorytmem jest *FPClose*. Metoda ta ma najmniejszy przyrost czasu wykonania przy wzroście liczby wzorców (Wykres 14, Wykres 17, Wykres 20). Wzrost ten ma charakter wykładniczy. W przypadku maksymalnego zapotrzebowania na pamięć operacyjną zaobserwowany został minimalny wzrost liniowy względem liczby wartości wsparcia i tym samym logarytmiczny względem liczby wzorców (Wykres 15, Wykres 18, Wykres 21). Jednakże metoda ta jest stosunkowo starą metodą (2005) oraz istnieją już nowsze i wydajniejsze metody niż *FPGrowth*, na którym bazuje *FPClose*, należy spodziewać się w niedalekiej przyszłości nowych metod do odkrywania wzorów częstych.

Dla kolejnego typu odkrywania wzorców, czyli odkrywanie sekwencji częstych najlepszym algorytmem jest *PrefixSpan*. Metoda ta ma niewielki wzrost potrzebnej pamięci (Wykres 27, Wykres 30, Wykres 33) i czasu wykonania w stosunku do liczby transakcji (Wykres 26, Wykres 29, Wykres 32). Dodatkowo, algorytm osiągnął najlepszy czas wykonania w ramach wszystkich testów. Potrzebna pamięć rośnie liniowo w stosunku do liczby znajdowanych wzorów i jest to najmniejszy przyrost wśród pozostałych algorytmów. Zużycie pamięci jest najmniejsze.

Ostatnim omawianym typem są metody do odrywania domkniętych sekwencji częstych. Dla tego typu najlepszy jest algorytm *CMClasp*. Algorytm ten ma najmniejsze zużycie pamięci (Wykres 39, Wykres 42, Wykres 45) i najkrótszy czas wykonania ze wszystkich użytych (Wykres 38, Wykres 41, Wykres 44). Dodatkowo przyrost zużycia pamięci i czasu wykonania w stosunku do liczby znajdowanych wzorców również jest najmniejszy. Najlepszy wzrost czasu wykonania i pamięci maksymalnej przy wzroście liczby transakcji oraz najlepszy czas wykonania plasuje ten algorytm na pierwszym miejscu. Z drugiej strony liczba algorytmów do wyszukiwania domkniętych wzorców sekwencji jest niewielka w porównaniu do liczby metod odkrywania wzorców częstych sugeruje powstanie nowych i lepszych metod w przyszłości.

## 7. Podsumowanie

W ramach przygotowanych eksperymentów przebadane zostało 23 algorytmy. Dla każdego z nich zostały uruchomione 4 eksperymenty na zbiorach wykazujących różne charakterystyki. Każdy eksperyment został powtórzony pięciokrotnie. Dzięki takiemu podejściu udało się ustalić najlepszy algorytm dla każdej metody. Dla wzorców częstych najlepszymi algorytmami zostały *PrePost+*, *FPGrowth* i *PrePost*. Dla domkniętych wzorców częstych jest to *FPClose*. Dla wzorców sekwencji *PrefixSpan*. Dla domkniętych wzorców sekwencji *CMClasp*. Wyniki eksperymentów jednoznacznie wskazywały najlepszy algorytm.

## 8. Bibliografia

- [1] R. Agrawal i R. Srikant, „Fast algorithms for mining association rules,” *Proc. of 20th Intl. Conf. on VLDB*, pp. 487-499, 1994.
- [2] M. Zaki, „Scalable Algorithms for Association Mining,” *IEEE Transactions on Knowledge and Data Engineering*, tom 12, nr 3, pp. 372-390, Maj 2000.
- [3] J. Han, J. Pei, Y. Yin i R. Mao, „Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach,” *Data Mining and Knowledge Discovery*, tom 8, nr 1, pp. 53-87, Styczeń 2004.

- [4] T. Uno, M. Kiyomi i H. Arimura, „LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets,” *Workshop on Frequent Itemset Mining Implementations*, tom 19, pp. 30-41, 2004.
- [5] C. Borgelt, „Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination,” *1st international workshop on open source data mining: frequent pattern mining implementations*, pp. 66-70, 2005.
- [6] J. Pei, H. Jiawel, L. Hongjun, N. Shojiro, T. Shiwei i Y. Dongqing, „H-Mine: Fast and space-preserving frequent pattern mining in large databases,” *IIE Transactions*, tom 39, nr 6, pp. 593-605, Czerwiec 2007.
- [7] Z.-H. Deng, Z.-H. Wang i J.-J. Jiang, „A new algorithm for fast mining frequent itemsets using N-lists,” *Science China Information Sciences*, tom 55, nr 9, pp. 2008-2030, Wrzesień 2012.
- [8] Z.-H. Deng i L. Sheng-Long, „PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning,” *Expert Systems with Applications*, tom 42, nr 13, pp. 5424-5432, 2015.
- [9] N. Pasquier, Y. Bastide, R. Taouil i L. Lakhal, „Discovering Frequent Closed Itemsets for Association Rules,” *7th International Conference on Database Theory*, pp. 398-416, 1999.
- [10] C. Lucchese, S. Orlando i R. Perego, „DCIClosed: a Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets,” *IEEE ICDM 2004 Workshop on Frequent Itemset Mining Implementations*, 2004.
- [11] M. Zaki i C.-J. Hsiao, „CHARM: Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure,” *IEEE Transactions on Knowledge and Data Engineering*, tom 17, nr 4, pp. 462-478, Kwiecień 2005.
- [12] G. Grahne i J. Zhu, „Fast Algorithms for Frequent Itemset Mining Using FP-Trees,” *IEEE Transactions on Knowledge and Data Engineering*, tom 17, nr 10, pp. 1347-1362, Październik 2005.
- [13] M. Zaki, „SPADE: An Efficient Algorithm for Mining Frequent Sequences,” *Machine Learning*, tom 42, nr 1, pp. 31-60, Styczeń 2001.
- [14] J. Ayres, J. Flannick, J. Gehrke i T. Yiu, „SPAM: Sequential PAttern Mining using A Bitmap Representation,” *8th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 429-435, 2002.
- [15] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, C. Qiming, U. Dayal i M.-C. Hsu, „Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach,” *IEEE Transactions on knowledge and data engineering*, tom 16, nr 11, pp. 1424-1440, Październik 2004.
- [16] Z. Yang, Y. Wang i M. Kitsuregawa, „Effective Sequential Pattern Mining Algorithms by Last Position Induction,” *Advances in Databases: Concepts, Systems and Applications*, pp. 1020-1023, 2007.

- [17] P. Fournier-Viger, A. Gomariz, M. Campos i R. Thomas, „Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information,” *Advances in Knowledge Discovery and Data Mining*, pp. 40-52, 2014.
- [18] X. Yan, J. Han i R. Afshar, „CloSpan: Mining Closed Sequential Patterns in Large Datasets,” *Third SIAM International Conference on Data Mining*, pp. 166-177, 2003.
- [19] J. Wang i J. Han, „BIDE: Efficient Mining of Frequent Closed Sequences,” *20th International Conference on Data Engineering*, pp. 79-91, 2004.
- [20] A. Gomariz, M. Campos, R. Marin i B. Goethals, „ClaSP: An efficient algorithm for Mining Frequent Closed Sequences,” *Advances in Knowledge Discovery and Data Mining*, pp. 50-61, 2013.
- [21] P. Fournier-Viger, „SPMF An Open-Source Data Mining Library,” 12 11 2016. [Online]. Available: <http://www.philippe-fournier-viger.com/spmf/index.php>. [Data uzyskania dostępu: 12 11 2016].
- [22] KNIME.COM AG, „KNIME®,” 10 Styczeń 2017. [Online]. Available: <https://www.knime.org/>. [Data uzyskania dostępu: 10 Styczeń 2017].
- [23] „kddcup99 transactional dataset,” 2 Grudzień 2016. [Online]. Available: <https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>. [Data uzyskania dostępu: 2016 Grudzień 2016].
- [24] „Mushroom transactional dataset,” 2 Grudzień 2016. [Online]. Available: <https://archive.ics.uci.edu/ml/datasets/Mushroom>. [Data uzyskania dostępu: 2 Grudzień 2016].
- [25] „OnlineRetail transactional dataset,” 2 Grudzień 2016. [Online]. Available: <https://archive.ics.uci.edu/ml/datasets/Online+Retail>. [Data uzyskania dostępu: 2 Grudzień 2016].
- [26] „Kosarak transactional dataset,” 2 Grudzień 2016. [Online]. Available: <http://fimi.ua.ac.be/data/kosarak.dat>. [Data uzyskania dostępu: 2 Grudzień 2016].
- [27] „Bible sequences dataset,” 2 Grudzień 2016. [Online]. Available: <http://www.philippe-fournier-viger.com/spmf/datasets/BIBLE.txt>. [Data uzyskania dostępu: 2 Grudzień 2016].
- [28] „Leviathan sequences dataset,” 2 Grudzień 2016. [Online]. Available: <http://www.philippe-fournier-viger.com/spmf/datasets/LEVIATHAN.txt>. [Data uzyskania dostępu: 2 Grudzień 2016].