POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Mateusz Mazur
Nr albumu: 180352
Poziom kształcenia: Studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność: Algorytmy i technologie internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Wykrywanie i wizualizacja znaczących informacji w logach aplikacji

Tytuł pracy w języku angielskim: Discovery and visualization of meaningful information in applications logs

Opiekun pracy: dr inż. Krzysztof Manuszewski

Abstrakt

W niniejszej rozprawie magisterskiej przedstawiono problematykę automatycznej analizy plików logów w kontekście detekcji anomalii w systemach informatycznych. W dobie rosnącej złożoności oprogramowania oraz infrastruktury IT, efektywne metody diagnostyczne odgrywają kluczową rolę w zapewnieniu stabilności i niezawodności systemów. Analiza logów, stanowiących bogate źródło informacji o działaniu aplikacji, jawi się jako obiecujące podejście do automatyzacji procesu identyfikacji potencjalnych problemów. W pracy zaproponowano nowatorski algorytm, którego celem jest analiza różnic pomiędzy dwoma plikami logów: jednym reprezentującym prawidłowe działanie systemu (baseline) i drugim, w którym podejrzewa się anomalię (checked). Istotnym elementem proponowanego rozwiązania jest skuteczne rozpoznanie i porównanie sekwencji zdarzeń zarejestrowanych w plikach logów, nawet jeśli użytkownik nie posiada wiedzy domenowej na ich temat. W tym celu zaproponowano wykorzystanie zaawansowanych technik analizy danych, w tym metod opartych o bitoniczną macierz Monga. Uzyskany w ten sposób system przetestowano na istniejących zbiorach logów, dzięki czemu potwierdzono jego skuteczność.

Abstract

This paper addresses the problem of automatic log file analysis in the context of anomaly detection in computer systems. In the face of increasing complexity of software and IT infrastructure, effective diagnostic methods play a key role in ensuring system stability and reliability. Log analysis, which provides a rich source of information about application behavior, is a promising approach to automate the process of finding potential problems. This paper proposes a novel algorithm aimed at analyzing discrepancies between two log files: one representing the correct system behavior (baseline) and the other suspected of containing anomalies (checked). A crucial aspect of the proposed solution is the effective recognition and comparison of event sequences recorded in log files, even without prior domain knowledge about their specific format. To achieve this, the algorithm utilizes advanced data analysis techniques, including methods based on the bitonic Monge array. The resulting system was tested on existing log datasets, confirming its effectiveness in anomaly detection.

Oznaczenia i Definicje

- **BMM - Bitoniczna Macierz Monga** - *Opisana w sekcji 2.2.3.*
- **Log** - *Informacja odnotowana przez system, informująca o jakimś wydarzeniu. Jest bezpośrednio powiązana z kodem źródłowym programu. Zwykle jest reprezentowana jako tekstowa linia w pliku logów (ang. Log File).*
- **Wzorzec** - *Tekst opisujący wygląd linii logu. Dana linia może albo pasować do wzorca albo do niego nie pasować.*
- **Znacznik czasowy** - *Tekstowy fragment linii logu oznaczający, kiedy linia została zapisana lub kiedy odpowiadające jej zdarzenie wystąpiło.*
- **Parser** - *Program/Algorytm który analizuje dane wejściowe na podstawie określonego zestawu reguł, zwykle z celem przekształcenia danych wejściowych w strukturę bardziej użyteczną do dalszego przetwarzania lub manipulacji.*

SPIS TREŚCI

1. WSTĘP

Współczesne systemy informatyczne są złożone i często rozproszone pomiędzy różne niezależne komponenty, które działają asynchronicznie. Sprawia to, że zrozumienie zachowań i interakcji w tych systemach staje się dużym wyzwaniem, zwłaszcza podczas diagnozowania złożonych problemów. Ze względu na ograniczenia tradycyjnych, złożonych narzędzi do debugowania [11], inżynierowie często wykorzystują prostszą, lecz skuteczną technikę zapisywania śladów aplikacji w plikach tekstowych wzbogaconych o dane takie jak znaczniki czasowe. Te pliki, zwane logami lub plikami logów, są kluczowe do monitorowania, diagnozowania i rozumienia działania współczesnych systemów informatycznych.

Pliki logów zazwyczaj przyjmują formę częściowo ustrukturyzowanych plików tekstowych, gdzie każda nowa linia odpowiada nowemu zdarzeniu wewnątrz aplikacji. Takie zdarzenie zazwyczaj również bezpośrednio wskazuje na konkretną lokalizację w kodzie źródłowym aplikacji. Dodatkowo, każda linia może zawierać dodatkowe informacje (1.1), takie jak znacznik czasowy, rodzaj zdarzenia, lokalizację, numer wątku, itp. W pracy przyjęto również, że zdarzenia (linie) w plikach z logami są uporządkowane w sposób chronologiczny. Oznacza to więc, że dla dowolnej linii wewnątrz pliku, każda kolejna linia wystąpić może w tej samej lub kolejnej chwili czasu - *nigdy wcześniej*. Największy problem tkwi w braku ujednoliconego sposobu strukturyzowania plików z logami, więc w praktyce większość aplikacji przyjmuje zupełnie inne struktury plików logów[7], m. in. w zależności od:

- Języka programowania - Wiele z języków posiada swoje własne, domyślne, systemy logowania zdarzeń. Są one jednak zwykle specyficzne dla danego środowiska.
- Użytych bibliotek dających możliwość logowania - Większe biblioteki często projektują własne systemy zapisu zdarzeń, tak aby jak najlepiej odpowiadały specyfice, rozwiązywanego systemu. Czasem umożliwiają one dostosowanie formatu logów, często jednak nie dają one takiej możliwości.
- Systemu operacyjnego - Współczesne systemy operacyjne często posiadają tak zwane dzienniki zapisujące zdarzenia w systemie. Różnią się one jednak często między wersjami, dystrybucjami oraz nie są takie same dla różnych systemów operacyjnych.
- Dostępnych zasobów - W zależności od miejsca działania systemu, na logi nałożone mogą zostać różne ograniczenia - zarówno czasowe jak i pojemnościowe. Przykładowo w systemach wbudowanych (*ang. embedded*), często zapisywane są ID zdarzeń, które następnie dekodowane są przez zewnętrzny program.

Logi w nowoczesnych aplikacjach stanowią nieocenione źródło przydatnych danych. Oferują głęboki wgląd w działanie aplikacji, bez generowania dodatkowych kosztów. Jest tak, ponieważ systemy logowania są bardzo proste i ich wyłączenie zwykle nie jest potrzebne nawet w środowisku produkcyjnym. Jednocześnie ich wszechstronność sprawia, że znajdują zastosowanie w różnych rozwiązaniach technologicznych, od zintegrowanych systemów wbudowanych (*ang. embedded*), po zaawansowane rozproszone mikro-serwisy oraz inne usługi internetowe. Ta uniwersalność i bogactwo zawartych informacji sprawiają, że logi są przedmiotem intensywnych badań, zwłaszcza w kontekście ekstrakcji istotnych danych. Te, pomóc mogą w procesach diagnostycznych, monitoringu wydajności oraz wykrywaniu potencjalnych problemów w programach.

Analiza logów stanowi kluczowy element utrzymania i diagnozowania współczesnych systemów informatycznych. Logi służą jako cenne źródło informacji dla programistów, administratorów systemów

i zespołów wsparcia (*ang. Helpdesk*), umożliwiając im zyskanie wglądu w zachowanie aplikacji i skuteczne identyfikowanie problemów, bez potrzeby ponownego uruchamiania systemu. Jest to szczególnie ważna właściwość przy diagnozowaniu błędów występujących bardzo rzadko, zwanych po angielsku *sporadical*-ami. Często skutkuje to szybszym rozwiązaniem problemu. Niestety, pomimo powszechności logów we współczesnych programach, nie istnieje ustandaryzowany sposób na zapis logów, co sprawia, że stworzenie generycznych rozwiązań opartych o nie jest bardzo trudne.

Większość z współcześnie działających systemów posiada dość złożony komponent odpowiadający za odpowiednie formatowanie logów, filtrowanie ich w zależności od ustawień oraz archiwizacje w celu przyszłej analizy (gdy jest to potrzebne). Sprawia, że logi zebrane z prawdziwych systemów informatycznych posiadają podobną strukturę oraz zawierają pewne wspólne, unikalne metadane. Finalnie ułatwia to proces analizy oraz diagnostyki systemów tylko i wyłącznie na podstawie zdarzeń, czyli logów, które w nich wystąpiły.

W tej pracy zaproponowano algorytm, próbujący rozwiązać część opisanych powyżej problemów, jednocześnie minimalizując ilość założeń początkowych w celu uzyskania jak najlepszej uniwersalności. System, jako swoje wejście przyjmuje dwa pliki wejściowe:

- *Baseline* - Plik ten powinien zawierać logi z przebiegu aplikacji, w momencie gdy działała ona w sposób prawidłowy. Preferowane jest, aby okres zbierania logów z systemu dla tego pliku, był podobny bądź lekko dłuższy od czasu zbierania dla pliku *Checked*.
- *Checked* - Ten plik zawierać powinien logi z przebiegu aplikacji, która zachowała się w sposób niepożądany bądź niespodziewany. Plik ten porównywany będzie z plikiem *Baseline*.

Algorytm został zaprojektowany tak, aby skutecznie działać bez konieczności dodatkowej konfiguracji, nawet gdy użytkownik nie dysponuje specjalistyczną wiedzą. System umożliwia jednak dostrojenie dodatkowych parametrów, co pozwala na wykorzystanie wiedzy eksperckiej w danej dziedzinie i może jeszcze bardziej zwiększyć jego skuteczność. Postawionym zadaniem systemu jest asysta użytkownika, tak aby znalezienie istotnych informacji było ułatwione, w stosunku do innych, bardziej manualnych metod wykorzystywanych przez administratorów.

Automatyczna analiza logów nie jest zadaniem łatwym. Jednym z kluczowych problemów jest rozpoznawanie wzorców w danych semi-strukturalnych. Oczywiście, struktura ta nie jest wcześniej znana osobie implementującej algorytm. Jak zostanie pokazane na przykładzie, nawet proste logi mogą być zapisywane na wiele sposobów, co wymaga od algorytmów umiejętności rozpoznawania i interpretowania różnych formatów zapisu tych samych informacji. Proces ten może być dodatkowo skomplikowany przez obecność różnych metadanych oraz konieczność interpretacji kontekstu, w jakim logi zostały wygenerowane.

Jest możliwe do oszacowania, jak bardzo prawdopodobne jest, że dana linia logu spowodowała defekt systemu, znając tylko logi systemu, gdy nie wystąpił w nim defekt. Analizując różnice między logami z poprawnego działania systemu a logami systemu z defektem, wykryć można anomalie wskazujące na potencjalne źródła problemów. Taka analiza pozwala na szybkie i skuteczne diagnozowanie przyczyn awarii, co jest kluczowe dla utrzymania stabilności i wydajności współczesnych systemów informatycznych. W tej pracy zaproponowany zostanie przykładowy system sprawdzający wyżej zapostulowaną tezę zgodnie z przyjętymi założeniami. Jego dokładne działanie i struktura opisane zostały w rozdziale 3.

1.1. Struktura logów

Jak wspomniano wcześniej, pliki logów często przyjmują format semi-strukturalny. Oznacza to, że w tym samym pliku, linie logu powinny posiadać podobną strukturę (ale nie wzorzec !). Struktura opisuje jak wygląda linia oraz jakie informacje muszą w niej się znajdować, pomijając samą treść komunikatu. Dane struktury obejmują informacje takie jak obecny czas, rodzaj zdarzenia (*ang. Severity*), nazwę modułu, etc.

Wyzwaniem w analizie logów jest brak uniwersalnej metody definiowania i wykrywania ich struktury. Sprawia to, że zadanie polegające na jej automatycznym zrozumieniu przez program komputerowy jest niełatwe. Struktura w tym kontekście oznacza pewną część wspólną dla wszystkich linii logów. Z tego powodu, wprowadzone zostało jest pojęcie wzorca. Oznacza on pewien wzór, zgodnie z którym zdefiniowana jest dana linjka logu - zwykle sprowadzający się do elementów zmiennych oraz stałych.

Problematykę różnorodnych struktur logów opisać można na dość prostym przykładzie, na którego potrzeby sfabrykowano przykładową linię na rysunku 1.1, oznaczoną dalej jako l' w równaniu 1.1. Ma ona dość łatwą strukturę z dwoma metadanymi: datą oraz informacją o ważności.

```
01.01.2024 INFO Client connected
```

Rysunek 1.1: Przykładowa linia logu

Warto zastanowić się, jaki wzorzec (*ang. pattern*) algorytm powinien wykryć w podanym (1.1) przykładzie. Dość prostym wzorem mógłby być: `<timestamp> <severity> <*> connected`, gdzie nazwy wokół trójkątnych nawiasów reprezentują parametry dla danej linii logu. Oznacza to więc, że dla dowolnej linii logu l, parser $\mathbb{P}$ powinien umieć znaleźć w niej wzór oraz argumenty wzoru dla danej linii $p, x = \mathbb{P}(l)$, z wykorzystaniem których odtworzyć można daną linię logu $l = p(x)$. Operacje te, wykorzystując podany przykład, opisane zostały poniżej, w równaniu 1.1.

$$
\begin{aligned}
l' &= \texttt{'01.01.2024 INFO Client connected'} \\
p', x' = \mathbb{P}(l') &= \texttt{'<timestamp> <severity> <*> connected'}, [\texttt{'01.01.2024'}, \texttt{'INFO'}, \texttt{'Client'}] \\
l' &= p'(x')
\end{aligned}
\tag{1.1}
$$

Łatwo wyobrazić sobie linię logu l^*, która reprezentuje identyczną informację co linia l', jednak ma zupełnie inną strukturę. Dla przykładu z rysunku 1.1, stworzyć można następujące, semantycznie identyczne linie. Różnią się one od siebie tylko metadanymi, notacją lub kolejnością, tak jak pokazano to na rysunku 1.2. Mimo to, z punktu widzenia algorytmów do analizowania logów, wszystkie omawiane przykładowe logi odpowiadają identycznemu zdarzeniu w aplikacji. Sprawia to, że funkcja $\mathbb{P}$ odpowiada w pewnym sensie za znajdywanie klastrów logów o takim samym wzorcu.

```
01.01.2024 INFO Client connected
01/01/24 Information [thread 0] - Client connected
Info - 01 January 2024 - Client connected
```

Rysunek 1.2: Przykładowa linia logu - inne warianty

Ilustruje to duży problem przy projektowaniu algorytmu, którego celem jest wykrycie wzorców bez dodatkowych informacji o strukturze logów. Każdy plik może posiadać zupełnie inne znaczniki czasowe (*ang. timestamp*), czyli opis czasu kiedy wystąpiło zdarzenie. Dodatkowo może on posiadać inne

bardziej lub mniej istotne metadane, takie jak: krytyczność zdarzenia (*ang. Severity*), ID wątku/procesu i inne parametry specyficzne dla analizowanego systemu. Wszystko to komplikuje się jeszcze bardziej w przypadku specyficznych aplikacji domenowych.

Problem struktury linii logów zostanie zgłębiony dokładniej w kolejnych sekcjach, jednak jest bardzo ważny przy zrozumieniu tematyki pracy.

2. PRZEGLĄD LITERATURY

2.1. Detekcja anomalii

Problem poruszany w tej pracy, zaliczyć można do działu informatyki nazywanego detekcją anomalii (ang. Anomaly Detection) na podstawie logów. Zadaniem systemów rozwiązujących taki problem jest określenie czy dany plik logów zawiera informację o źródle anomalii, następnie wskazaniu w którym miejscu należy szukać linii powiązanych z tą anomalią. Dodatkowo mogą one określać stopień swojej pewności predykcji lub wydobywać dodatkowe metadane z tekstu.

Przy problemach tego typu, naukowcy często skupiają się na różnych rozwiązaniach opartych o uczenie maszynowe (*ang. machine learning*). Współcześnie często wykorzystują one sieci neuronowe oparte o zróżnicowane architektury [19, 22, 13]. Tego typu rozwiązania często dobrze radzą sobie w systemach, na których były uczone, jednak w systemach nie spotkanych wcześniej, ich użyteczność umie być znacznie mniejsza. Dodatkowo, dużym problemem sieci neuronowych jest ich niewyjaśnialność[18] (*ang. Explainability*), co sprawia że ciężko jest zagwarantować ich stabilność i skuteczność.

Sieci neurownowe mają również swoje olbrzymie zalety. Po pierwsze, nie wymagają one od programisty pomysłu na złożone, często prawie niemożliwe, rozwiązanie problemu. Dzięki ich wykorzystaniu, proces znalezienia optymalnego rozwiązania problemu wykonywany jest przez algorytm uczenia maszynowego - algorytm propagacji wstecznej [10] (*ang. Backpropagation*). Po drugie, zwykle gdy sieć zostanie wytrenowana i posiada odpowiednią architekturę, umie ona osiągać bardzo dobre wyniki.

W kolejnych sekcjach, omówione zostaną różne poddziedziny detekcji anomalii na podstawie logów, korzystając z których, zaprojektować można kompletny system wykrywający anomalie (3). Każda sekcja opisana poniżej związana będzie z finalnym komponentem systemu opisanego w dalszych rozdziałach pracy. W szczególności:

- Znajdywanie wzorców, opisane w sekcji 2.1.1
- Znajdywanie czasu w logach, opisane w sekcji 2.1.3
- Dopasowanie dwóch chronologicznie uporządkowanych zbiorów, opisane w sekcji 2.2

Przy okazji omawiania tematu detekcji anomalii, wspomnieć należy o bardzo prężnie działającej w tej dziedzinie organizacji naukowej: *LogPai*[29]. Badaczne należący do niej są twórcami omawianych później parserów Drain[9] oraz Brain[23]. Są oni również autorami repozytorium testowego dla parserów logów: *LogParser*[14].

2.1.1. Znajdywanie wzorców

Pierwszym typem algorytmów wykorzystywanych w tego typu systemach, są algorytmy automatycznie odnajdujące strukturę logów (patrz: 1.1) [14, 8]. Zwykle są one oparte o uczenie maszynowe, jednak nie jest to wymagane. Dodatkowo dzielą się one zwyczajowo na “offline” - takie, które wymagają przetworzenia całego pliku oraz “online” - takie, które umieją znajdywać strukturę podanej linii, bez potrzeby przetworzenia całego pliku. W algorytmach znajdujących strukturę logu istnieje parę potencjalnych celów optymalizacji:

- **Jakość uzyskanych struktur (wzorców)** - Jest to najbardziej oczywiste kryterium, jednak również dość ciężkie to ewaluacji. Istnieją systemy oceniające algorytmy pod tym względem [14], jednak kryterium to jest dość subiektywne i może różnić się w zależności od potrzeb. Problem ten jest szczególnie widoczny w przypadku problemu detekcji anomalii, gdzie celem nie jest *koniecznie* odtworzenie struktury logów opisanej w kodzie źródłowym programu.

- **Ilość hiperparametrów** - W zależności od rozwiązywanego problemu, kryterium może zależeć od ich dużej, bądź małej ilości. Zwykle algorytmy o większej ilości hiperparemterów lepiej można dostosowywać do danego typu logów, te o mniejszej natomiast łatwiej jest skonfigurować, bądź zrobić to automatycznie.

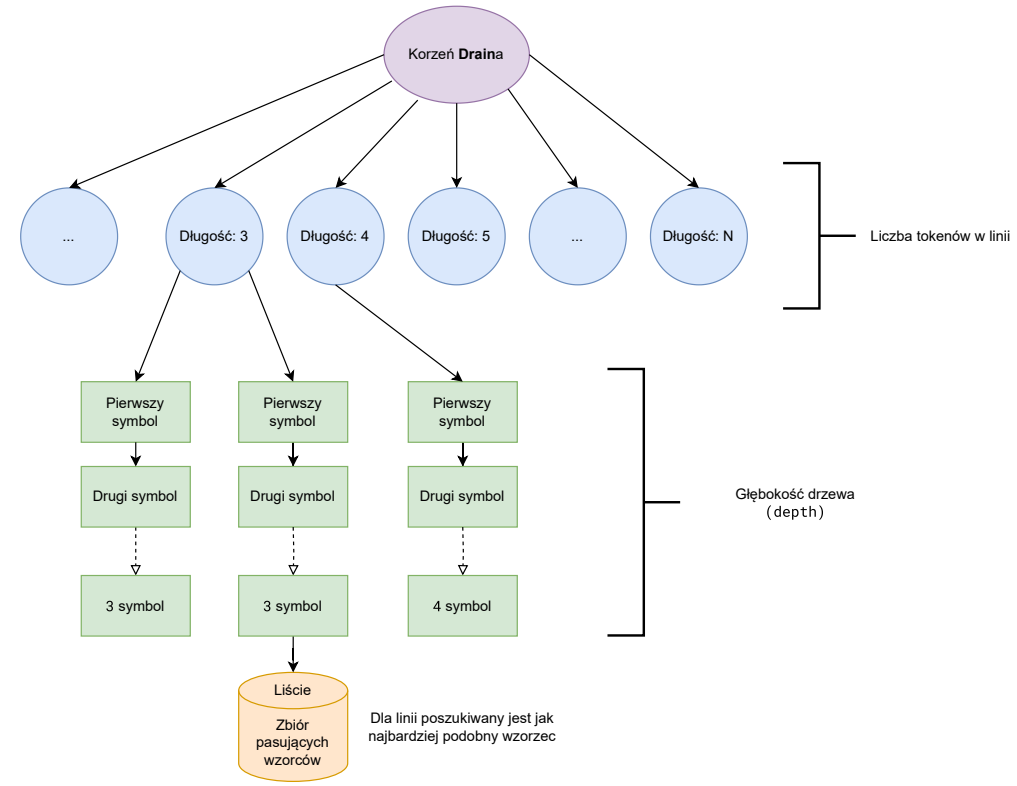
- **Złożoność obliczeniowa** - Przy analize algorytmów, często istotnym aspektem jest analiza złożoności - obliczeniowych oraz pamięciowych. Opisują one, jak zwiększanie wielkości argumentów/parametrów wpływa na wydajność danego algorytmu. Zwykle wyrażane są one za pomocą notacji dużego O [20]. Z jej wykorzystaniem opisać można jak zmieni się czas wykonania algorytmu w zależności od wielkości danych wejściowych.

Do przykładów algorytmów ekstrakcji wzorców należy algorytm Drain [9], będący algorytmem typu “online”, który radzi sobie dość dobrze z zróżnicowanymi logami, szczególnie po dostosowaniu jego hiperparametrów. Poglądowy sposób działania Draina zarysowany jest na rysunku 2.1. Algorytm posiada dwa hiperparametry: `depth` (głębokość drzewa) oraz `st` (Simmilarity threshold - Próg Podobieństwa). Parametr `depth` wpływa na to jak głębokie jest drzewo symboli, zarazem jego zwiększanie powoduje znajdywanie większej ilości różnych wzorów oraz przyspiesza ich znajdywanie kosztem pamięci, natomiast zmiejszanie tego parametru ma efekt bezpośrednio przeciwny.

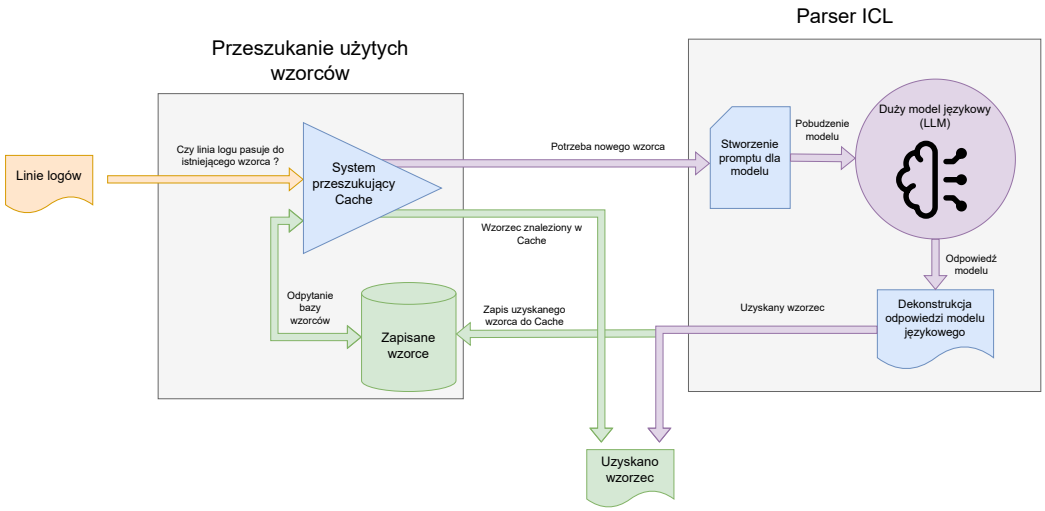
Parametr `st` służy do konfiguracji operacji porównywania z sobą wzorców w liściach drzewa wzorców. Przyjmuje on wartość z zakresu $(0, 1\rangle$, w przypadku wartości 1 oznacza, że linia musi być identyczna, aby pasować do danego wzorca. Jego zmniejszanie sprawia że, porównanie staje się bardziej rozymte. Sprawia to, że częściej linie nie pasujące perfekcyjnie do wzorca, zostaną i tak do niego dopasowane.

Następcą algorytmu Drain, jest Brain[23], zaprojektowany przez tych samych autorów w 2023 roku. Działa on podobnie do Draina, jednak graf wykorzystany wewnętrznie przez algorytm jest znacznie bardziej złożony. Krawędzie wewnątrz drzewa są dwukierunkowe co sprawia, że wzorce znalezione przez algorytm składają się z współdzielonych wierzchołków, które reprezentują kolejne fragmenty wzorca. Fragment może być jednym z dwóch: albo parametrem albo stałym symbolem. Według wyników z oryginalnej pracy, algorytm lepiej radzi sobie z bardziej złożonymi logami, niż jego poprzednik. Brain posiada tylko jeden parametr konfiguracyjny `threshold`, wykorzystywany przy poruszaniu się po grafie wzorców - często jednak nie ma on aż tak dużego znaczenia i może przyjąć stałą wartość.

Innym interesującym podejściem do parsowania logów, jest system LILAC[26]. System ten zbudowany jest w oparciu o Duży Model Językowy[27] (*ang. LLM - Large Language Model*), którego zadaniem jest ekstrakcja wzorców z linii. System, aby ograniczyć ilość wykorzystań modelu, na początku stara się znaleźć wzorce wcześniej “wymyślone” przez model językowy, gdy się to nie udaje, wtedy dopiero wykorzystany zostaje LLM. Schemat działania LILAC jest również zwizualizowany na rysunku 2.2.



Rysunek 2.1: Schemat działania Drain



Rysunek 2.2: Schemat działania systemu LILAC

Istnieje też wiele innych algorytmów ekstrakcji wzorców[15, 31, 16]. Różnią się one od siebie działaniem, jakością wyników, szybkością i wieloma innymi właściwościami. W tej pracy, nie zostaną one szczególnie omówione, jedną są one również interesującymi rozwiązaniami tego problemu.

2.1.2. Automatyczna detekcja anomalii

Automatyczna detekcja anomalii jest kluczowym zagadnieniem w analizie logów. Współczesne metody w tym obszarze często wykorzystują sieci neuronowe [19, 22, 13] ze względu na ich zdolność do przetwarzania dużych zbiorów danych i adaptacji do różnorodnych wzorców. Należy jednak pamiętać, że sieci neuronowe wymagają dużych zbiorów treningowych i ich interpretowalność bywa ograniczona. Oprócz sieci neuronowych, w automatycznej detekcji anomalii stosuje się również metody klasyczne, takie jak algorytmy oparte na statystyce czy analizie skupień. Metody te charakteryzują się zazwyczaj mniejszą złożonością obliczeniową i większą interpretowalnością, ale mogą być mniej skuteczne w przypadku złożonych wzorców anomalii.

Metody automatycznej detekcji anomalii zazwyczaj składają się z trzech głównych etapów:

1. **Przetworzenie wstępne danych**: Wstępne przetworzenie lini logów jest niezbędne przed podaniem ich do sieci neuronowej lub innego algorytmu. Proces ten może obejmować różne techniki, takie jak automatyczne wykrywanie wzorców[22], analiza semantyczna[17], itp. Przykładowo, automatyczne wykrywanie wzorców pozwala na identyfikację struktury logów, co ułatwia ich dalszą analizę przez sieć neuronową.

2. **Wykorzystanie sieci neuronowej lub innej metody**: Na tym etapie przetworzone dane są podawane do sieci neuronowej bądź innego algorytmu, które przeprowadzają ewaluację danych. Sieci neuronowe mogą mieć różne architektury, takie jak LSTM (*ang. Long Short-Term Memory*) czy GAN (*ang. Generative Adversarial Networks*), które są dostosowane do specyficznych potrzeb detekcji anomalii. Sieci uczą się rozpoznawać wzorce normalnego zachowania systemu oraz identyfikować odstępstwa, które mogą wskazywać na anomalie.

3. **Zebranie i opracowanie wyników**: Po przeprowadzeniu analizy przez sieć neuronową bądź inny algorytm, wyniki są zbierane i przetwarzane w celu wygenerowania końcowych wniosków. Ten krok może obejmować ocenę pewności predykcji, identyfikację konkretnej lini logu odpowiedzialnej za anomalię oraz wydobycie dodatkowych metadanych, które mogą być przydatne przy dalszej analizie.

Podejścia oparte na sieciach neuronowych oferują znaczące korzyści, takie jak zdolność do automatycznego uczenia się na podstawie danych. Niemniej jednak, istnieją również wyzwania związane z ich stosowaniem. Jednym z głównych problemów wykorzystania sieci neuronowych, jest złożoność procesu interpretacji wyników generowanych przez sieci (niewyjaśnialność), co utrudnia zapewnienie stabilności i niezawodności takich systemów. Ponadto, sieci neuronowe mogą mieć problemy z generalizacją do nowych, wcześniej nieznanych danych, co może ograniczać ich możliwość wykorzystania w niektórych systemach produkcyjnych (*ang. production*).

2.1.3. Automatyczne znajdywanie znaczników czasowych

Automatyczne znajdywanie znaczników czasowych (*ang. Timestamp*) nie jest głównym tematem tej pracy, jednak jest dość istotne i pełni kluczową rolę dla algorytmu opisanego w dalszych sekcjach. Problem wyrażony jest w dość intuicyjny sposób: jak, nie mając żadnej wiedzy domenowej o danym systemie, automatycznie określić w jakim momencie wystąpiło dane zdarzenie (linia logu) w stosunku do pozostałych zdarzeń w pliku. Informacja ta zwykle przechowywana jest w części tekstu logu, zwanej znacznikiem czasowym - zwykle na początku lini.

W dalszych częściach pracy opisujących ten problem, przyjęto parę założeń dotyczących znaczników czasowych w logach. Założenia te oparte są o empiryczne doświadczenia oraz dostępną wiedzę w

dziedzinie rodzajów logów systemowych [21], zgodnie z którymi poniższe założenia są dość uniwersalnie spełniane. Podejście takie miało na celu maksymalizację rzeczywistej użyteczności projektowanego algorytmu.

1. Linie wewnątrz jednego pliku zawsze ułożone są chronologicznie. Nie jest możliwe, aby linia występująca po innej linii, w rzeczywistości wydarzyła się wcześniej.
2. Linie mogą posiadać znaczniki czasowe. Znacznik czasowy, to część logu, znajdujaca się w jego początkowej części, opisująca zgodnie z konwencją danego systemu, kiedy wystąpiło dane zdarzenie. Nie jest wiadome, jaki jest dokładny format takiego znacznika oraz jaka jest jego dokładność bez dodatkowej wiedzy domenowej.
3. Każdy plik z logami posiada przynajmniej jedną linię logu posiadającą znacznik czasowy. Jeżeli linia logu nie posiada takiego znacznika, przyjmuje się, że wystąpiła ona w tym samym momencie co poprzednie zdarzenie o znanym czasie wystąpienia.

Powyższe trzy założenia są fundamentalne przy dalszym projektowaniu algorytmu służącego do detekcji ważnych zdarzeń. W pozostałych sekcjach wykorzystane są one przy budowaniu dalszych założeń dotyczących omawianego problemu. Idealnie, dla omawianego problemu, każda linia logu powinna posiadać precyzyjny znacznik czasowy.

2.2. Najlepsze dopasowanie

Plik logu traktować można jak uporządkowany zbiór linii $l \in L$. Dodatkowo przyjęto, że zdefiniowany został parser logów $\mathbb{P}$. Każda linia poddana została procesowi parsowania, w wyniku czego uzyskano ciąg par: wzorca p oraz linii logu l. Linie o tym samym wzorcu są następnie grupowane i dopasowywane względem czasu. W przypadku problemu omawianego w pracy, jednym z podproblemów staje się optymalne dopasowanie do siebie dwóch zbiorów, tak aby każda linia z jednego logu posiadała odpowiadającą linię z drugiego. Przyjęto że dobrą miarą dopasowania dwóch lini do siebie, jest dystans czasowy pomiędzy wystąpieniem linii.

Problem ten więc przedstawić można jako minimalne dopasowanie do siebie dwóch rozłącznych zbiorów: A oraz B, gdzie pomiędzy każdą parą elementów z obu zbiorów zdefiniowana jest funkcja odległości d. Dziedzina takiej funkcji opisana jest w równaniu 2.1, gdzie c to liczba z $\mathcal{R}_+$. Zbiory A oraz B oznaczają odpowiednio pliki *baseline* oraz *checked*, gdzie każda linia to kolejny element zbioru [1].

$$d : A \times B \rightarrow \mathcal{R}_+ \tag{2.1}$$

Jako funkcję odległości d przyjęto dystans pomiędzy dwoma czasami, gdzie dla każdej z lini zdefiniowana jest relacja $t(x)$ (2.2) określająca jej czas wystąpienia, relatywnie do czasu wystąpienia pierwszej linii, zawsze: $t(\text{pierwsza linia pliku}) = 0$. Równanie definiujące funkcję dystansu d opisano poniżej (2.3).

$$t : X \rightarrow \mathcal{R}_+, \quad X = A \cup B \tag{2.2}$$

$$d(a,b) = |t(a) - t(b)|, \quad a \in A, b \in B \tag{2.3}$$

[1] W reszcie tej sekcji odnosić się będę do tych dwóch zbiorów - A i B, zamiast do linii i ich czasów, w celu uproszczenia analizy. Są one jednak zupełnie analogiczne do plików z logami i obie notacje mogą być stosowane zamiennie.

Podsumowując, zbiory A i B są w pełni analogiczne do logów, będą więc one wykorzystywane w dalszych rozważaniach opisanych w tej sekcji. Zbiory te zawierają elementy uporządkowane zgodnie z wartością funkcji t. Dodatkowo, w celu uproszczenia wyrażeń matematycznych, przyjęta jest konwencja indeksowania tych zbiorów. Wyrażenie A_i oznacza i-ty element zbioru A. Dodatkowo, w dalszej części sekcji wykorzystane będą zmienne i oraz j, reprezentujące indeksy dla zbiorów A oraz B. Przyjmują one więc kolejne wartości: $i \in \{1, 2, \ldots, |A|\}$ i $j \in \{1, 2, \ldots, |B|\}$.

2.2.1. Algorytm Węgierski

Klasycznym rozwiązaniem takiego problemu, jest zastosowanie algorytmu węgierskiego [1]. Algorytm ten w swojej oryginalnej postaci rozwiązuje problem dopasowania dla każdego pracownika i optymalnej pracy j. Optymalność pracy opisana jest z wykorzystaniem macierzy kwalifikacji Q, tak że wartość $Q[i, j]$ opisuje kwalifikacje pracownika i do pracy j. Zadaniem jest znalezienie par pracowników oraz prac, tak aby uzyskać jak największą sumę kwalifikacji. Zakładając więc, że zbiór S posiada takie pary, to problem można opisać formalnie w równaniu 2.4

$$\min \sum_{(i',j') \in S} Q[i', j'] \tag{2.4}$$

Algorytm więgierski, znajduje najlepsze takie dopasowanie w czasie $O(n^4)$. Ma on też wiele udoskonaleń opisanych w kolejnych pracach [3, 2], uzyskując złożoność obliczeniową $O(n^3)$.

Powoduje to, że algorytm węgierski i jego pochodne, stają się najbardziej oczywistymi podejściami do rozwiązania omawianego problemu. W przypadku problemu dopasowania zbiorów A oraz B, macierz Q opisana jest zgodnie z równaniem 2.5

$$Q[i, j] = d(A_i, B_j), \quad |A| = |B| \tag{2.5}$$

Pierwszym z widocznych problemów staje się tutaj przypadek, w którym moce zbiorów A oraz B, są różne od siebie - $|A| \neq |B|$. W takim wypadku, utworzyć należy kwadratową macierz Q, a miejsca nie pokryte przez relację A z B, należy uzupełnić wartościami “niewybieralnymi” przez algorytm, przykładowo jak w wzorze 2.6. Warto zwrócić uwagę, że niektóre z wartości “niewybieralnych” zostaną wybrane przez algorytm węgierski, jednak oznaczają one *brak* dopasowanej lini.

$$Q[i, j] = \begin{cases} -d(A_i, B_j) & i \leq |A| \wedge j \leq |B| \\ 0 & i > |A| \vee j > |B| \end{cases} \tag{2.6}$$

Sprawia to, że złożoność pamięciowa jest kwadratowa $O((max(|A|, |B|))^2)$, względem mocy większego zbioru z pary zbiorów: A oraz B. Przez to, algorytm staje się problematyczny do wykorzystania, szczególnie na dużych zbiorach danych. Wiąże się to też oczywiście z niepotrzebnym przetwarzaniem “niewybieralnych” relacji wewnątrz macierzy Q.

2.2.2. Graf dwudzielny

Łatwo zauważyć, że omawianą relację dwóch logów zapisać można w postaci grafu dwudzielnego $G(V, E)$, gdzie $V \in A \cup B$. W grafie tym, każda krawędź posiada wagę reprezentującą odległość pomiędzy elementami z odpowiednio zbioru A oraz B. Oznacza to, że waga krawędzi $e_{a,b}$ między $a \in A$ oraz $b \in B$ wynosi $v(e_{a,b}) = d(a, b)$. W takiej sytuacji problemem najlepszego dopasowania staje znalezienie grafu $G'(V, E')$, takiego, że przy ograniczeniu 2.7 minimalizowana jest wartość 2.8.

$$\begin{cases} \forall_{a\in A}\ \exists_{b\in B}\ \exists e_{a,b} & |A| < |B| \\ \forall_{b\in B}\ \exists_{a\in A}\ \exists e_{a,b} & |B| < |A| \\ \forall_{b\in B}\ \forall_{a\in A}\ \exists e_{a,b} & |A| = |B| \end{cases} \tag{2.7}$$

$$\sum_{e_{a,b}\in E'} v(e_{a,b}) = \sum_{e_{a,b}\in E'} d(a,b) \tag{2.8}$$

Ten model w dość intuicyjny sposób reprezentuje strukturę lini logu oraz relacje między nimi. Ułatwia on również wizualizację problemu oraz projektowanie heurystyk na potrzeby algorytmu detekcji anomalii. Szczególnie ciekawe są dwie własności grafu opisane poniżej. Ich wykorzystanie ogranicza oczekiwane wyniki algorytmu najlepszego dopasowania.

1. **Gdy oba zbiory mają tą samą moc** - Każdy element wchodzi w relację z każdym, minimalizując wyżej opisaną wartość.
2. **Gdy zbiór A jest większy od B** (i odwrotna sytuacja) - Każdy element z zbioru B wchodzi relację z innym elementem z zbioru A, również minimalizując wyższe kryterium.

2.2.3. Bitoniczna Macierz Monga

Bitoniczna Macierz Monga[2] to macierz relacji między dwoma uszeregowanymi rosnąco zbiorami elementów, zgodnie z metryką t (2.2, 2.3). Kryterium, które należy spełnić aby relację między zbiorami opisać można było z wykorzystaniem takiej macierzy, opisane zostało w równaniu 2.9. Jeżeli macierz taką stworzyć można na podstawie dwóch zbiorów, to jest możliwe dopasowanie ich elementów[4], tak aby zminimalizować "odległość"[3] między takimi elementami z złożonością obliczeniową $O(Nlog(M))$. Gdzie $N \leq M$ i odpowiadają wielkościom dwóch porównywanych zbiorów.

$$\forall_{a_1,a_2\in A}\forall_{b_1,b_2\in B} \begin{cases} t(a_1) \leq t(a_2) \wedge t(b_1) \leq t(b_2) \\ d(a_1,b_1) + d(a_2,b_2) \leq d(a_1,b_2) + d(a_2,b_1) \end{cases} \tag{2.9}$$

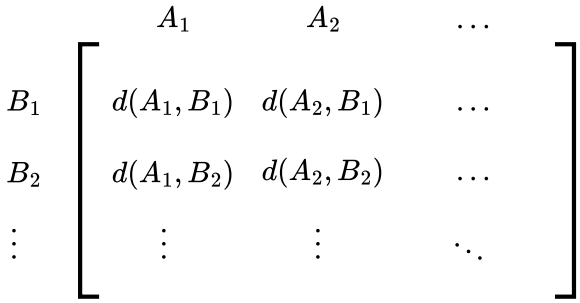
Linie logów w pliku spełniają równanie 2.9, oznacza to więc, że macierz tą wykorzystać można do znalezienie optymalnego dopasowania zbiorów $A, N = |A|$ oraz $B, M = |B|$. Dowód na to wyprowadzony został w sekcji 2.2.4. Co szczególnie atrakcyjne, macierz Monga nie musi być macierzą kwadratową, co sprawia że dobrze pasuje do omawianego problemu.

Macierz M stworzyć można w sposób analogiczny do tego opisanego dla macierzy Q w algorytmie węgierskim 2.5. Każdemu elementowi z zbioru A przypisać należy indeks wiersza i, natomiast każdemu elementowi z zbioru B należy przypisać indeks kolumny. Następnie dla każdej kombinacji i z j wypełnić macierz M według 2.10. Dla uzyskanej macierzy istnieje algorytm znajdujący takie dopasowanie ze złożonością opisaną wyrażeniem 2.11. Dodatkowo na rysunku 2.3 zwizualizowano relację między zbiorami A i B z wykorzystaniem macierzy M.

$$M[i,j] = d(A_i, B_j) \tag{2.10}$$

[2] W dalszych częściach pracy, opisywana jako BMM.

[3] W przypadku tej pracy, odległość między elementami w macierzy monga oznacza znormalizowaną względem pierwszego zdarzenia (w każdym z logów) odległość czasową.

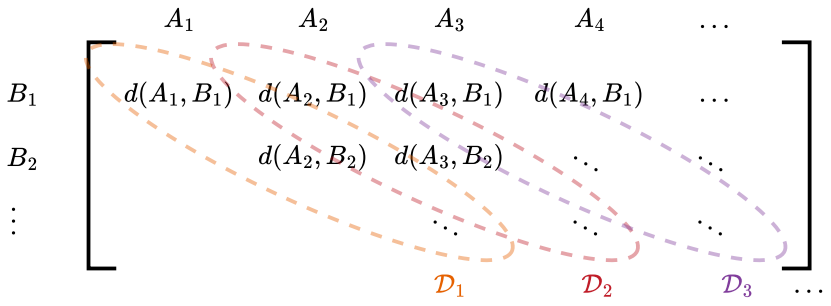


Rysunek 2.3: Macierz Monga M dla zbiorów A i B

$$O\left(\min(|A|, |B|) \cdot \log(\max(|A|, |B|))\right) \tag{2.11}$$

Szczegółowe działanie algorytmu, jak i dowód na jego skuteczność oraz złożoność opisane zostały w oryginalnej pracy[4]. Jego założeniem, jest poruszanie się po diagonalach $\mathcal{D}$ macierzy M, tak jak pokazano na rysunku 2.4. Wynika to z właściwości optymalnego dopasowania w BMM, w której wartości najlepszego dopasowania zawsze znajdują się w diagonalach macierzy. W ten sposób, w zależności od różnicy $||A| - |B||$, uzyskujemy $r = 1 + ||A| - |B||$ diaglonali macierzy M - $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_r$. Każda z takich diagonali opisana jest równaniem 2.12 i posiada tyle samo elementów co mniejszy z wymiarów macierzy monga - $\min(|A|, |B|)$ - przyjmuje się, że jest to pierwszy wymiar - ilość wierszy. Algorytm jest zaprojektowany zgodnie z techniką "divide and conquer", aby w czasie logarytmicznym znaleźć najlepsze dopasowanie dwóch zbiorów. Dzieli on diagonale na dwie grupy, zaczynając od "środka" macierzy, następnie powtarza ten krok dla obu grup, aż diagonale posiadają tylko optymalne dopasowania. Sprawia to, że algorytm ten ma również potencjał do implementacji wielowątkowej.

$$\mathcal{D}_l = \{M[1, l], M[2, l+1], M[3, l+2], \ldots, M[n = |B|, l+n]\} \tag{2.12}$$



Rysunek 2.4: Diagonale $\mathcal{D}_l$ w macierzy M

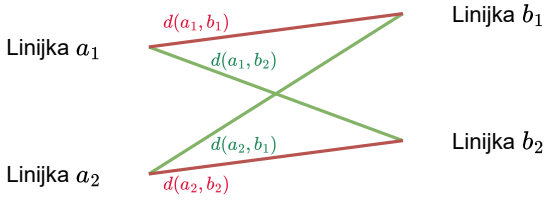
2.2.4. Dowód bitoniczności macierzy odległości dla dopasowania logów

Zgodnie z założeniami przyjętymi we wstępie do pracy, prawdą jest wyrażenie opisane w równaniu 2.13. Wynika to z ważnej właściwości plików z logami - czyli chronologiczności. W języku nieformalnym, oznacza to, że zdarzenie a_2 musiało wystąpić w tej samej chwili lub w chwili późniejszej niż zdarzenie a_1. Analogiczną właściwość mają elementy z zbioru B, ponieważ mają takie samo źródło.

$$\forall_{a_1,a_2 \in A} \forall_{b_1,b_2 \in B} \quad \begin{cases} t(a_1) \leq t(a_2) \\ t(b_1) \leq t(b_2) \end{cases} \tag{2.13}$$

Aby macierz M była Bitoniczną Macierzą Monga, spełniony musi być warunek 2.14. Poniżej, przedstawiony został dowód na prawdziwość tego warunku. Intuicyjnie, równanie przedstawia dość prostą zależność, różnice czasu "na krzyż" zawsze będą większe, tak jak pokazano to na rysunku 2.5. Dla obu plików, weźmy dwie pary elementów: a_1 i a_2 oraz b_1 i b_2, te o niższym indeksie występują na pewno przed elementami o większym indeksie. Aby macierz była BMM, suma odległości czasowych $d(a_1, b_1) + d(a_2, b_2)$, musi być mniejsza od sumy $d(a_1, b_2) + d(a_2, b_1)$.

$$\forall_{a_1,a_2 \in A} \forall_{b_1,b_2 \in B} \quad d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1) \tag{2.14}$$
$$\text{Gdzie: } d(x, y) = |t(x) - t(y)|$$



Rysunek 2.5: Geometryczna interpretacja równania 2.14

W równaniu 2.14, największą trudność sprawiają wartości bezwzględne. Wynikają one z absolutnych dystansów między elementami zbiorów - zawsze znaleźć należy różnicę między większym a mniejszym czasem. Ujemny czas, szczególnie w omawianym kontekście, nie posiada logicznej intepretacji. Problem ten rozwiązać można poprzez nałożenie dodatkowych ograniczeń na relację między elementami a oraz b. Wszystkie takie sześć wariantów opisano w równaniu 2.15. Dodatkowo w celu uproszenia nierówności, pominięto uwzględnienie kwantyfikatorów dla elementów a_1, a_2, b_1 oraz b_2 - przyjęte zostało, że są one takie same jak w równaniu 2.13 oraz równaniu 2.14.

$$\begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_2) \\ t(a_1) \geq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \leq t(b_2) \\ t(a_1) \geq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \leq t(b_2) \\ t(a_2) \geq t(b_1) \\ t(a_1) \leq t(b_1) \end{cases} \tag{2.15}$$
$$\vee \quad t(a_2) \leq t(b_1) \qquad \vee \quad t(a_1) \geq t(b_2)$$

W sekcjach poniżej przeanalizowano wszystkie wyżej opiasne warianty w porządku chronologicznym. Każdy z wariantów reprezentuje inną relację zdarzeń: a_1, a_2, b_1 i b_2. W każdym z nich, dowiedzione jest, że spełnione zostało równanie 2.16 zgodnie z warunkami danego wariantu. Dowodzi to, że prawdą jest równanie 2.16, co zarazem kończy dowód.

$$|t(a_2) - t(b_2)| + |t(a_1) - t(b_1)| \leq |t(a_2) - t(b_1)| + |t(a_1) - t(b_2)| \tag{2.16}$$

Wariant 1

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$b_1 \text{ -> } a_1 \text{ -> } b_2 \text{ -> } a_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$\begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_2) \\ t(a_1) \geq t(b_1) \end{cases}$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$\begin{aligned} t(a_2) - t(b_2) + t(a_1) - t(b_1) &\leq t(a_2) - t(b_1) - t(a_1) + t(b_2) \\ -t(b_2) + t(a_1) &\leq -t(a_1) + t(b_2) \\ t(a_1) &\leq t(b_2) \end{aligned}$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

Wariant 2

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$a_1 \text{ -> } b_1 \text{ -> } b_2 \text{ -> } a_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$\begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_1) \end{cases}$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$t(a_2) - t(b_2) - t(a_1) + t(b_1) \leq t(a_2) - t(b_1) - t(a_1) + t(b_2)$$
$$-t(b_2) + t(b_1) \leq -t(b_1) + t(b_2)$$
$$t(b_1) \leq t(b_2)$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

Wariant 3

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$b_1 \text{ -> } a_1 \text{ -> } a_2 \text{ -> } b_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$\begin{cases} t(a_2) \leq t(b_2) \\ t(a_1) \geq t(b_1) \end{cases}$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$-t(a_2) + t(b_2) + t(a_1) - t(b_1) \leq t(a_2) - t(b_1) - t(a_1) + t(b_2)$$
$$-t(a_2) + t(a_1) \leq t(a_2) - t(a_1)$$
$$t(a_1) \leq t(a_2)$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

Wariant 4

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$a_1 \text{ -> } b_1 \text{ -> } a_2 \text{ -> } b_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$\begin{cases} t(a_2) \leq t(b_2) \\ t(a_2) \geq t(b_1) \\ t(a_1) \leq t(b_1) \end{cases}$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane

zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$-t(a_2) + t(b_2) - t(a_1) + t(b_1) \leq t(a_2) - t(b_1) - t(a_1) + t(b_2)$$
$$-t(a_2) + t(b_1) \leq t(a_2) - t(b_1)$$
$$t(b_1) \leq t(a_2)$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

Wariant 5

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$a_1 \text{ -> } a_2 \text{ -> } b_1 \text{ -> } b_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$t(a_2) \leq t(b_1)$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$-t(a_2) + t(b_2) - t(a_1) + t(b_1) \leq -t(a_2) + t(b_1) - t(a_1) + t(b_2)$$
$$0 \leq 0$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

Wariant 6

W tym wariancie przyjęto następującą kolejność zdarzeń, czyli linii logów, która została opisana poniżej. Są to dodatkowe ograniczenia na dowodzoną nierówność:

$$b_1 \text{ -> } b_2 \text{ -> } a_1 \text{ -> } a_2$$

Powyższą kolejność opisać można w postaci warunków w formalnym języku matematycznym. Ich opis został przedstawiony poniżej:

$$t(a_1) \geq t(b_2)$$

Powyższe warunki potraktować można jako dodatkowe założenia. Poprzez przyjęcie ich przy wyrażeniu 2.16, sprawdzić można czy badana kolejność zdarzeń je spełnia. W ten sposób uzyskane zostały, a następnie uproszczone, wyrażenia dowodzące prawdziwości twierdzenia 2.16 przy przyjętej kolejności wydarzeń:

$$t(a_2) - t(b_2) + t(a_1) - t(b_1) \leq t(a_2) - t(b_1) + t(a_1) - t(b_2)$$

$$0 \leq 0$$

■

Zatem nierówność $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$ jest spełniona.

3. ZAPROPONOWANE ROZWIĄZANIE

Zaproponowany system ma za zadanie wyznaczyć, które linie logu w pliku checked są dla niego unikalne (czyli podejrzane) i/lub, które linie w pliku baseline nie mają odpowiedników w pliku checked. System podzielić można na trzy główne komponenty:

- **Ekstrakcja wzorca** - Zadaniem tego komponentu jest wyznaczenie wspólnych (między dwoma plikami) wzorców, które dopasowane są do każdej z linii z obu logów. Teoria związana z zagadnieniem wyciągania wzorców z logów opisana została dokładniej w sekcji 1.1.
- **Ekstrakcja i normalizacja czasu** - Po znalezieniu wzorca dla linii, kolejnym krokiem jest ekstrakcja czasu jej wystąpienia. Problem ten jest dość złożony i mało przebadany, ze względnu na swoją nietypowość.
- **Minimalne dopasowanie** - Dwa pliki logów, w których każde zdarzenie ma przypisany czas wystąpienia oraz identyfikator wzorca, są ze sobą dopasowywane. Dopasowanie odbywa się poprzez porównanie czasów wystąpienia zdarzeń o tym samym identyfikatorze wzorca w obu plikach. Pozwala to określić różnice w czasie wystąpienia odpowiadających sobie zdarzeń. Analiza tych różnic umożliwia identyfikację ważnych wydarzeń.

3.1. Idea działania systemu detekcji istotnych zdarzeń

Zadaniem algorytmu jest analiza zależności pomiędzy liniami pliku `checked` oraz pliku `baseline`. Oba pliki przez algorytm interpretowane są jako listy linii tekstu - logów. Dla każdej linii jest możliwe dokonanie ekstrakcji wzorca oraz znacznika czasowego (przy wykorzystaniu odpowiednich parserów). Algorytm, dla każdej linii dokonuje obu ekstrakcji, następnie z wykorzystaniem algorytmu najlepszego dopasowania względem czasu wystąpienia, wybiera linie, które *nie* znalazły dopasowania. Intuicyjnie, oznacza to brak analogicznego zdarzenia w drugim pliku. Prawdopodobnie zdarzenie to jest więc bezpośrednio powiązane z diagnozowanym defektem. Możliwy jest również wariant, w którym algorytm zapisuje różnice pomiędzy czasami logów, następnie wykorzystuje je do analizy z wykorzystaniem dowolnych heurystyk.

Poniżej znajduje się pseudokod 3.1, który na wysokim poziomie abstrakcji, w lekko uproszczony sposób, opisuje działanie opracowanego w tej pracy algorytmu, zwanego też w innych miejscach "systemem". Algorytm przedstawiony jest jako funkcja zdefiniowana słowem kluczowym `def`, która jako wejście przyjmuje dwa pliki logów oraz zwraca listę logów *istotnych*. Istotne logi w rozumieniu tego systemu to takie, które nie mają dopasowania pomiędzy plikami. Kod wykorzystuje funkcje/metody pomocnicze, będące algorytmami rozwiązującymi poszczególne podproblemy, które opisano w punktach poniżej:

1. `parse_pattern` - Jest to funkcja, która na podstawie podanej linii znajduje wzorzec do niej pasujący. W algorytmie uwzględniono samo unikalne ID wzorca, więc funkcja mapuje dowolną linię logu na odpowiednie ID wzorca. ID wzorców są współdzielone dla obu plików. Problem ten został dokładniej opisany w sekcji 2.1.1.
2. `parse_timestamp` - Jest to funkcja, która dla dowolnej linii logu, podaje czas w którym wystąpiła. W zależności od implementacji, może ona wykorzystywać wiedzę domenową dotyczącą analizowanego systemu, informację od użytkownika lub być zupełnie uniwersalna. Problem ten został dokładniej opisany w sekcji 2.1.3.

3. `bitonic_monge_array_match` - Jest to funkcja, która na podstawie dwóch list lini logów i czasów ich wystąpienia, pochodzących z dwóch różnych plików, znajduje ich jak najlepsze dopasowanie, względem różnicy czasu. Problem ten został dokładniej opisany w sekcji 2.2.3.

4. `map`, `set`, `list` - Struktury danych, odpowiednio: mapa/słownik (tabela mieszająca), zbiór unikalnych elementów oraz indeksowalna lista elementów.

```
def find_important_lines(checked_lines, baseline_lines):
    checked_by_pattern  = map()  # Pattern ID -> Line in checked file
    baseline_by_pattern = map()  # Pattern ID -> Line in baseline file
    all_pattern_ids     = set()  # Unique set of Pattern IDs
    all_unmatched_lines = list() # Here, unmatched lines will be stored

    for line in checked_lines: # Iteration over each line in checked file
        pattern_id = parse_pattern(line)
        timestamp  = parse_timestamp(line)
        all_pattern_ids.add(pattern_id)
        checked_by_pattern.push(pattern_id, (timestamp, line))

    for line in baseline_lines: # Iteration over each line in baseline file
        pattern_id = parse_pattern(line)
        timestamp  = parse_timestamp(line)
        all_pattern_ids.add(pattern_id)
        baseline_by_pattern.push(pattern_id, (timestamp, line))

    # For each pattern ID in all pattern IDs,
    # take lines from both files with given pattern ID,
    # and perform matching on them with BMM, based on timestamps.
    for pattern_id in all_pattern_ids:
        checked_lines   = checked_by_pattern.get_all(pattern_id)
        baseline_lines  = baseline_by_pattern.get_all(pattern_id)
        unmatched_lines = bitonic_monge_array_match(baseline_lines, checked_lines)
        all_unmatched_lines.push_flatten(unmatched_lines)

    return all_unmatched_lines
```

Kod 3.1: Uproszczony Pseudokod Systemu

Wykorzystując pseudokod 3.1 dość łatwo oszacować można złożoność obliczeniową algorytmu, tak jak pokazano to na równaniu 3.1. Oszacowana jest ona na postawie jednej właściwości O: czas wykonywania pętli `for` jest liniowy względem wielkości zbioru. Dodatkowo traktuje ona wykorzystane w algorytmie systemy jako “black box” o parametryzowalnej złożoności obliczeniowej. Taka złożoność nie jest jednak bardzo przydatna w praktyce, gdyż posiada ona aż trzy dane wejściowe, które w dodatku również reprezentują złożoności obliczeniowe zewnętrznych algorytmów. Aby poprawić tą sytuację wykonać należy dokładniejszą analizę względem wielkości tylko danych wejściowych oraz parametryzowalnych algorytmów ekstrakcji. W ten sposób uzyskana zostanie faktyczna złożoność względem wejścia algorytmu.

$$
\begin{gathered}
O_{Alg} = X + Y + Z \\
X \text{ - Złożoność przetworzenia pliku checked} \\
Y \text{ - Złożoność przetworzenia pliku baseline} \\
Z \text{ - Złożoność dopasowania z użyciem BMM}
\end{gathered}
\tag{3.1}
$$

Istnieją dwie złożoności "zewnętrzne" dla złożoności omawianego algorytmu. Reprezentują one parametry w formie dowolnych algorytmów, możliwych do wykorzystania w zaproponowanym systemie. Są to oczywiście algorytmy ekstrakcji wzorców (TE) oraz znaczników czasowych (PE). Te, mogą mieć dowolne złożoności zależne od wybranych przez implementującego rozwiązań. Przykładowo, możliwe jest wykorzystanie mechanizmu zapamiętywania (*ang. Caching*) [12], z wykorzystaniem którego złożoności tych algorytmów mogą być nawet $O(1)$. Złożoności te, wraz z innymi oznaczeniami przydatnymi do dalszych rozważań, zostały zestawione poniżej:

- C - Zbiór linii z pliku `checked`
- B - Zbiór linii z pliku `baseline`
- $C_{\texttt{pid=}x}$ - Wszystkie linie z Checked o ID patternu $= x$
- $B_{\texttt{pid=}x}$ - Wszystkie linie z Baseline o ID patternu $= x$
- O_{TE} - Złożoność algorytmu ekstrakcji czasu (np. parser słownikowy, Drain)
- O_{PE} - Złożoność algorytmu ekstrakcji wzorca
- O_{PRE} - Złożoność preprocessing logów - $O_{TE} + O_{PE}$
- $O_{Alg}(C, B)$ - Złożoność całego systemu/algorytmu
- $O_{BMM}(N, M)$ - Złożoność algorytmu BMM z sekcji 2.2.3 względem dwóch ciągów logów N i M
- $r \in (0, 1\rangle$ - Wartość oczekiwana liczby wzorców na linię, prawdopodobieństwo że linia będzie nowym wzorcem. Założone zostało, że prawdopodobieństwa tego że linie będą nowym wzorcem są od siebie **niezależne**. Parametr ten w rzeczywistości jest inny w zależności od analizowanych logów.
- $k \in (0, 1\rangle$ - Współczynnik określający maksymalną ilość linii pasujących do tego samego wzorca. Przykładowo, gdy $k = 1$ to wszystkie linie należą do tego samego wzorca. Parametr ten w rzeczywistości jest inny w zależności od analizowanych logów.

Najprostsze do oszacowania są dwie pierwsze złożoności związane z preprocessingiem: X i Y. Każda linia jest przetwarzana przez dwa algorytmy: ekstrakcji czasu (TE) oraz ekstrakcji wzorca (PE). Ta operacja zastosowana musi być dla zbioru B i C. W ten sposób uzyskano wyrażenia 3.2.

$$\begin{aligned} X &= O(|C|) \cdot O_{TE}(C) \cdot O_{PE}(C) = O(|C|) \cdot O_{PRE}(C) \\ Y &= O(|B|) \cdot O_{TE}(B) \cdot O_{PE}(B) = O(|B|) \cdot O_{PRE}(B) \end{aligned} \tag{3.2}$$

Większą trudność sprawia jednak oszacowanie złożoności Z, związanej z algorytmem dopasowania BMM. Początkowo oszacować należy ilość wzorców W, czyli ilość wykonanych dopasowań w grupach: $W \leq r \cdot (|C| + |B|)$. Dla uproszczenia i ze względu na własności O, dalej przyjęte zostanie $W = r \cdot (|C| + |B|)$, mimo że oczekiwane jest współdzielenie wzorców pomiędzy plikami (aby osiągnąć przypadek takiej równości, wzorce pomiędzy C i B musiałby być zupełnie inne).

$$\begin{aligned} Z &= \underbrace{O_{BMM}(C_{\texttt{pid=}1}, B_{\texttt{pid=}1}) + O_{BMM}(C_{\texttt{pid=}2}, B_{\texttt{pid=}2}) + \ldots + O_{BMM}(C_{\texttt{pid=}W}, B_{\texttt{pid=}W})}_{W \text{ elementów, id wzorców indeksowane od 1}} \\ O_{BMM}(N, M) &= \begin{cases} O(|N| \log |M|) & |N| \geq |M| \\ O(|M| \log |N|) & |N| < |M| \end{cases} \end{aligned} \tag{3.3}$$

Wyrażenie 3.3 przedstawia złożoność jako sumę W złożoności, gdzie każda z nich to jedno dopasowanie BMM. Z wykorzystaniem parametru modelu k, stwierdzić można, że najgorsza możliwa złożoność do uzyskania jako cześć skończonej sumy, to $O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha})$, gdzie $\alpha \in 1, 2, \ldots, W$ to takie ID, że: $|B_{\texttt{pid}=\alpha}| = k \cdot |B| \wedge |C_{\texttt{pid}=\alpha}| = k \cdot |C|$.

Warto zauważyć, że złożoność dopasowania BMM składa się z iloczynu dwóch elementów: N i $\log N$. Oba te elementy można potraktować jako funkcje względem N. Dodatkowo, kluczowym jest zauważenie, że obie funkcje są rosnące i dodatnie dla $N \geq 1$. Oznacza to więc, że $O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha}) = O_{BMM}(k \cdot |C|, l \cdot |B|)$, gdyż jest to najbardziej ekstremalny punkt funkcji O_{BMM}.

$$
\begin{aligned}
Z &= \underbrace{O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha}) + O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha}) + \ldots + O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha})}_{W \text{ elementów}} \\
&= O(W) \cdot O_{BMM}(C_{\texttt{pid}=\alpha}, B_{\texttt{pid}=\alpha}) = O(W) \cdot O_{BMM}(k \cdot |C|, k \cdot |B|) \\
&= O(r(|C| + |B|)) \cdot \begin{cases} O(|C| \log(k \cdot |B|)) & |C| \geq |B| \\ O(|B| \log(k \cdot |C|)) & |C| < |B| \end{cases}
\end{aligned}
\tag{3.4}
$$

W ten sposób uzyskano wyrażenia 3.4, które wyznaczają ostatecznie złożoność komponentu Z algorytmu. Warto zauważyć, że złożoność ma dwa dodatkowe parametry: $k, r \in (0, 1\rangle$, opisujące specyfikę domenową porównywanych logów. Uwzględnienie ich umożliwia zauważenie dwóch kluczowych zależności:

1. Duże zróżnicowanie wzorców, spowoduje zwiększenie ilości kroków, co zarazem zwolni liniowo algorytm.
2. Wzorce uwzględniające duży procent wszystkich linii, spowodują dłuższy czas wykonywania dopasowania BMM.

Podsumowując, wyrażenie 3.5 przedstawia pełną złożoność obliczeniową systemu detekcji anomalii. Zostało ono podzielone na logiczne etapy pracy algorytmu, podobnie do tych przedstawionych w sekcji 3.1.1.

$$
\begin{aligned}
O_{Alg}(C, B) = {} & O(|C|) \cdot O_{PRE}(C) \quad \Big\} \textit{ Preprocessing Checked} \\
& + O(|B|) \cdot O_{PRE}(B) \quad \Big\} \textit{ Preprocessing Baseline} \\
& + O(r(|C| + |B|)) \cdot \begin{cases} O(|C| \log(k \cdot |B|)) & |C| \geq |B| \\ O(|B| \log(k \cdot |C|)) & |C| < |B| \end{cases} \Bigg\} \textit{ Dopasowanie logów z BMM}
\end{aligned}
\tag{3.5}
$$

Złożoność 3.5 oznacza, że zaproponowany algorytm jest dość odporny na dysproporcje ($||C| - |B||$) w ilości linii logów pomiędzy plikiem `baseline` oraz `checked`. Zgodnie z oczekiwaniami, przy zwiększaniu jednego z plików, głównym problemem pozostaną algorytmy: ekstrakcji wzorców oraz ekstrakcji czasów. Wynika to z rzeczywistego małego współczynnika k, który znacznie skraca czas dopasowanie z BMM. Samo dopasowanie linii, czyli operację najcięższą do pominięcia dzięki mechanizmom takim jak cache, ma bardzo "niegroźną" złożoność względem długości plików z logami. Parametr r, ze względu na bycie stałą wartością skalarną, jest pomijalny.

3.1.1. Przykładowy przebieg algorytmu

W tej podsekcji opisany został jeden przykładowy przebieg algorytmu opisanego w sekcji 3.1. Dane dobrane w nim zostały w sposób arbitralny, mając jednocześnie na celu pokazanie interesujących przypadków znajdywalnych przez system. Kroki przebiegu opisane zostały w fazach na rysunkach: 3.1, 3.2, 3.3, 3.4

Etap 1 - Wczytanie danych

Linie pliku **Baseline**	Linie pliku **Checked**
[0.0] Error: Exception 1	[0.4] Error: Exception 2
[1.0] Error: Exception 2	[0.8] Error: Exception 1
[2.0] Warning: Program 4 slowed down	[1.5] Warning: Program 1 slowed down
[3.0] Periodic Service Diagnostics	[2.0] Warning: Program 4 slowed down
[4.5] Warning: Program 2 slowed down	[3.0] Periodic Service Diagnostics
[6.0] Periodic Service Diagnostics	[4.0] Warning: Program 2 slowed down
	[6.0] Periodic Service Diagnostics

Rysunek 3.1: Etap pierwszy przykładowego przebiegu algorytmu

3.1: Przedstawia pierwszą fazę, polega ona na zebraniu linii z logu. Dodatkowo zakłada ekstrakcję znaczników czasowych z każdej linii. Jak widać, wczytane zostały dwa pliki: `baseline` oraz `checked`.

Etap 2 - Parsowanie logów

Wzorce logów		
Schemat wzorca	**ID**	**Przykład**
Error: Exception <NUM>	1	Error: Exception 2
Warning: Program <NUM> slowed down	2	Warning: Program 2 slowed down
Periodic Service Diagnostics	3	Periodic Service Diagnostics

Rysunek 3.2: Etap drugi przykładowego przebiegu algorytmu

3.2 Przedstawia drugą fazę, polega ona na ekstrakcji wzorca logu, z wykorzystaniem parsera (słownikowego, Drain, etc) logów. Każdy wzorzec dostaje współdzielone między oboma plikami logów unikalne ID. Z wykorzystaniem tych ID wykonywane będą dopasowywania między liniami logów.

Etap 3 - Podział na grupy

Dla ID = 1, wzorzec: Error: Exception <NUM>

Baseline

Znacznik czasu	Linia
0.0	Error: Exception 1
1.0	Error: Exception 2

Checked

Znacznik czasu	Linia
0.4	Error: Exception 1
0.8	Error: Exception 2

Dla ID = 2, wzorzec: Warning: Program <NUM> slowed down

Baseline

Znacznik czasu	Linia
2.0	Warning: Program 4 slowed down
4.5	Warning: Program 2 slowed down

Checked

Znacznik czasu	Linia
1.5	Warning: Program 1 slowed down
2.0	Warning: Program 4 slowed down
4.0	Warning: Program 2 slowed down

Dla ID = 3, wzorzec: Periodic Service Diagnostics

Baseline

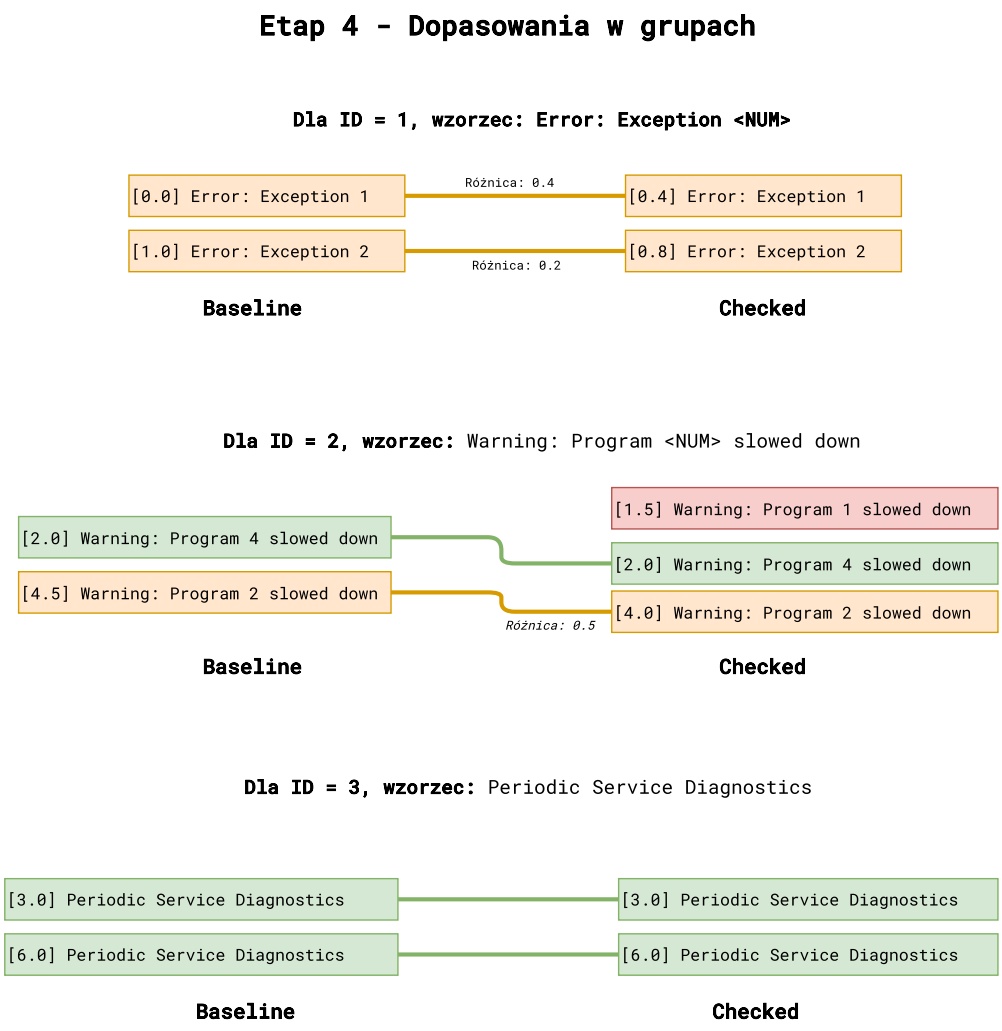
Znacznik czasu	Linia
3.0	Periodic Service Diagnostics
6.0	Periodic Service Diagnostics

Checked

Znacznik czasu	Linia
3.0	Periodic Service Diagnostics
6.0	Periodic Service Diagnostics

Rysunek 3.3: Etap trzeci przykładowego przebiegu algorytmu

3.3: Ta faza polega na podziale wszystki linii, z obu plików, na grupy o tych samych ID wzorców (każda grupa posiada unikalne ID). Dla tego przypadku uzyskano 3 unikalne wzorce.



Rysunek 3.4: Etap czwarty przykładowego przebiegu algorytmu

3.4: Ostatni etap - wynikiem jego działania jest uzyskanie najlepszego możliwego dopasowania dwóch plików z liniami. Każda linia może nie mieć dopasowania lub je mieć, dodatkowo dopasowanie może być perfekcyjne czasowo (można uwzględnić tutaj parametr tolerancji) lub nieperfekcyjne. Są to trzy podstawowe rodzaje dopasowań, które zaproponowany algorytm jest w stanie wyznaczyć.

Na zrzucie ekranu (3.5) znajduje się rozwiązanie tego problemu z wykorzystaniem narzędzia `differ` (patrz: 4.1). Pokazuje on wynik jego pracy dla wyżej przedstawionego zestawu logów - w celu weryfikacji przykładu. Ustawiony był na pokazanie jedynie istotnych różnic / defektów, pomijając linie prawidłowe. Jak widać, jego wynik jest zgodny z przedstawionym przebiegiem algorytmu.

```
1 | Incorrect Time | [0.4] Error: Exception 2 | Baseline: [0.0] Error: Exception 1
2 | Incorrect Time | [0.8] Error: Exception 1 | Baseline: [1.0] Error: Exception 2
3 | Additional | [1.5] Warning: Program 1 slowed down
6 | Incorrect Time | [4.0] Warning: Program 2 slowed down | Baseline: [4.5] Warning: Program 2 slowed down
```

Rysunek 3.5: Wynik narzędzia `differ` dla danych z przykładowego przebiegu.

3.2. Parser Słownikowy

Sercem algorymu opisanego powyżej jest algorytm znajdywania wzorców PE. W tej pracy zaproponowano prosty algorytm o nazwie *Parser Słownikowy*, oparty o słownik słów w języku użytym w aplikacji. Algorytm ten nazywany jest w innych miejscach pracy "Dict Parser", jest to jego tłumaczenie na język angielski - tak został nazwany przy implementacji.

Parser zbudowany jest na prostym założeniu. Gdy token jest słowem należącym do wybranego słownika, np. słownika słów angielskich z odmianami, jest on stały w danym wzorcu. Token taki nazwany został symbolem[1]. W przeciwnym przypadku dany token traktowany jest jako parametr, oznacza on więc element zmienny w danym wzorcu. Logika algorytmu opisana została w punktach poniżej.

Dla wejściowej linii tekstu (logu):

(a) Gdy jest to możliwe, znajdź znacznik czasowy. Usuń tą część tekstu.

(b) Dokonaj tokenizacji - podziel tekst na listę symbolów, na podstawie listy separatorów takich jak białe znaki (*ang. whitespace*). Dodatkowo dokonaj "oczyszczenia" symbolu poprzez usunięcie z niego znaków interpunkcyjnych oraz innych "niepotrzebnych" znaków. Szczegóły tego kroku opisane zostały w sekcji 3.2.1.

(c) Dla każdego symbolu dokonaj wyszukania w słowniku (*ang. lookup*). Jeżeli słowo jest w słowniku, oznacz tą część wzorca jako niezmienną. W innym przypadku, oznacz część wzorca jako zmienną.

(d) Sprawdź czy taki wzorzec już istnieje, gdy nie nadaj mu unikalne ID.

Jest on zaprojektowany tak, aby wykorzystywać metody ekstrakcji znacznika czasowego oraz tokenizacji, które opisane zostały w innych rozdziałach tej pracy. Dodatkowo ze względu na swoją prostotę, posiada on łatwą do wyznaczenia złożoność obliczeniową. W celu jej oszacowania wykorzystany zostanie poniższy pseudokod (3.2) opisujący algorytm.

```
dictionary = PredefinedDictionary()

def dict_parse(line):
    tokens = tokenize(line)
    for token in tokens:
        is_real_word = dictionary.lookup(token)
        if is_real_word:
            yield token as Symbol    # non-changable
        else:
            yield token as Parameter # changable
```

Kod 3.2: Pseudokod Parsera Słownikowego

Na podstawie fragmentu 3.2 widać, że górna granica złożoności jest opisana wyrażeniem 3.6. Czas wykonywania algorytmu jest ograniczony względem ilości tokenów oraz czasu przeszukania słownika, wszystkie pozostałe kroki algorytmu są $O(1)$. Oznacza to więc, że przy wykorzystaniu optymalnych struktur danych oraz algorytmów do szukania słów w słowniku, uzyskać można liniową złożoność.

$$O(N_{tokens}) \cdot O_{lookup} \tag{3.6}$$

[1] Nazwenictwo to zaczerpnięte zostało z innych rozwiązań tego typu, jak Drain czy Brain

Przy implementacji opisanej w kolejnym rozdziale jako funkcję wyszukania wykorzystano przeszukanie binarne o złożoności czasowej $O_{lookup} = O(\log N_{words\_in\_dict})$ oraz pamięciowej $O(N_{words\_in\_dict})$. Jest to optymalne podejście dla słownika słów angielskich, ze względu na jego stosunkowo mały rozmiar N - czyli ilość słów i ich wariantów w słowniku. Warto jednak zauważyć, że w zależności od warunków pracy algorytmu, inne struktury, takie jak tablica mieszająca (*ang. hash table*), mogą być wydajniejsze.

3.2.1. Tokenizacja

Tokenizacja wykorzystana w algorytmach, ma na celu wyznaczenie tokenów (*ang. token*) z fragmentu tekstu, takiego jak linia logu. Każdy kolejny token traktowany jest jako kolejna informacja przekazana w tekście. Informacja taka może być słowem, ale również liczbą lub innym specyficznym ciągiem znaków. W punktach poniżej zdefiniowano kroki omawianego algorytmu:

(a) Niech:

  (a) `whitespaces` - Będzie listą znaków uznawanych za białe (*ang. whitespace*). Znaki te traktowane są jako separatory informacji.

  (b) `punctuation` - Będzie listą znaków interpunkcyjnych, będą to znaki odrzucane z tokenów, gdy znajdują się na ich początku lub końcu.

(b) Podziel tekst na tokeny z użyciem znaków z `whitespace`. Każdy token jest oddzielony od siebie jednym lub większą ilością znaków z `whitespace`.

(c) Z każdego tokenu usuń znaki interpunkcyjne, zdefiniowane w liście `punctuation`, które znadują się na jego końcu lub początku.

(d) Usuń tokeny o zerowej długości (bez znaków), następnie zwróć listę uzyskanych tokenów.

3.3. Naiwny ekstraktor znaczników czasowych

Kluczowym założeniem algorytmu detekcji anomalii (3.1) jest posiadanie informacji o czasie wystąpienia danej lini logu. Jest to problem skomplikowany i często bardzo specyficzny dla danej domeny. W tej sekcji opisany zostały prosty algorytm rozwiązujący ten problem, na potrzebę algorytmu detekcji anomalii. Nie daje on jednak gwarancji prawidłowej ekstrakcji czasu dla wszystkich rodzajów logów.

Przy projekcie rozwiązania warto opisać przyjęte założenia, gdyż są one kluczowe do zrozumienia omawianego problemu:

- Algorytm opiera się o wzorce dat. Wzorzec daty opisuje to jak sformatowana jest dana data. Przykładowo data `17.01.2000`, opisana może być wzorcem `%dd-%mm-%yyyy`. Warto zauważyć, że definicja struktury wzorca daty jest zależna od implementacji algorytmu.
- Użytkownik powinien mieć możliwość definicji wzorów dat, zgodnie z posiadaną wiedzą domenową. W przeciwnym przypadku algorytm może, ale nie musi, zidentyfikować czas wystąpienia danej linjki.

Idea działania algorytmu ekstrakcji znaczników czasowych jest bardzo prosta. Linię wejściową poddajemy tokenizacji z wykorzystaniem tokenizera opisanego w sekcji 3.2.1. Jego wykorzystanie jest szczególnie przydatne, ponieważ wynik tokenizacji może być współdzielony z algorytmem parsera słownikowego opisanego w sekcji 3.2.

Kroki algorytmu, dla wejściowej linii logu są następujące:

(a) Niech T będzie ciągiem uzyskanych tokenów uzyskanych poprzez tokenizację linii. T_n oznacza n-ty token, a T_1 to pierwszy z tokenów.

(b) Gdy długość ciągu T jest większa od n_{max} [2], usunąć należy wszystkie elementy $T_{n'},\ n' \in \{n_{max}+1, n_{max}+2, \dots n\}$.

(c) Niech $T'_n = T_1 + T_2 + \dots + T_n,\ n \leq n_{max}$ będzie cząstkową sumą tokenów

  (a) Operacja + na tokenach oznacza konkatenację dwóch tokenów z znakiem `' '`. Na przykład: `'12.01.2024' + '12:00' = '12.01.2024 12:00'`.

(d) Niech $T''_{n,m}$ będzie rozszerzeniem ciągu T'_n o dodatkowe warianty potencjalnego znacznika czasowego:

  (a) Niech $S = \{(\texttt{'['}, \texttt{']'}),\ (\texttt{'('}, \texttt{')'}),\ \dots\}$ będzie ciągiem par znaków które otaczać mogą znacznik czasowy.

  (b) Otoczenie tokenu przez element z ciągu S_x oznacza, że pierwszy znak tokenu jest równy pierwszemu z elementów S_x, natomiast ostatni znak tokenu jest równy drugiemu z elementów S_x. Token bez otoczenia, oznacza taki token, z którego usunięte zostało otoczenie S_x.

  (c) Niech zmienna m będzie indeksem w ciągu $S,\ m \leq |S|$. Gdy $m = 0$ przyjęte zostało że znacznik nie jest otoczony.

  (d)

$$T''_{n,m} = \begin{cases} T'_n & m = 0 \\ T'_n \text{ bez otoczenia } S_m & m > 0 \wedge T'_n \text{ jest otoczony przez znaki } S_m \\ \text{niezdefiniowane} & m > 0 \wedge T'_n \text{ nie jest otoczony przez znaki } S_m \end{cases} \tag{3.7}$$

(e) Znajdź największe n^*, dla którego istnieje takie m^* że:

  (a) Ciąg znaków T''_{n^*,m^*} jest zdefiniowany

  (b) Istnieje wzorzec daty pasujący do potencjalnego znacznika czasowego T''_{n^*,m^*}

(f) Znacznikiem czasowym jest T''_{n^*,m^*}, wystarczy dokonać na nim ekstrakcji daty z wykorzystaniem pasującego wzorca.

Zaprojektowany algorytm daje gwarancję złożoności obliczeniowej $O(n_{max} \cdot |S|)$. Przy parametrach n_{max} i S zdefiniowanych jako stałe w algorytmie, tak jak zostało to zrobione w zaproponowanej implementacji, sprawia że złożoność algorytmu można opisać jako $O(1)$.

[2] Przy implementacji przyjęto że $n_{max} = 6$

4. EKSPERYMENTY

System automatycznej detekcji anomalii opisany w rozdziale 3 został zrealizowany w ramach implementacji wraz z opisanym parserem słownikowym oraz naiwnym ekstraktorem znaczników czasowych. Wszystko zaimplementowane zostało w języku Kotlin[6] na platformie JVM. Implementacja ta została wykorzystana do stworzenia prostego narzędzia CLI (*ang. Command Line Interface*) oraz zestawu narzędzi testowych (*ang. benchmark*), które wykorzystane zostały do eksperymentów. Projekt eksperymentów został stworzony w oparciu o dane od LogHub[24].

Implementacja znajduje się online pod adresem Mazurel/analyzer. Kod źródłowy udostępniony został pod licencją MIT[28]. Więcej informacji o implementacji znajduje się w 4.1

4.1. Implementacja

Adres repozytorium kodu: https://github.com/Mazurel/analyzer

Kod napisany w celu realizacji opisanych poniżej eksperymentów, podzielony jest na parę logicznych grup, a zarazem modułów systemu budowania Gradle[5]:

- `core` - Moduł, w którym zaimplementowane zostały algorytmy opisane w rozdziale 3. Udostępnia on dość wysokopoziomowe API do wykorzystania przez inne aplikacje, w tym benchmarki, testy, etc. W szczególności, zaimplementowane zostały:
  - *Parser logów* - Parser słownikowy (3.2)
  - *Ekstraktor znaczników czasowych* - Naiwny ekstraktor znaczników czasowych (3.3)
  - *Najlepsze dopasowanie* - Algorytm najlepszego dopasowania w BMM (2.2.3)
- `e2e-tests` - Moduł testowy, sprawdzający zachowanie programu `differ`, dla różnych przypadków par logów `baseline` i `checked`, które można napotkać. Jego zadaniem jest weryfikacja podstawowych zachowań programu, tak aby spełniał minimalne oczekiwania przyjęte podczas projektowania systemu (3). Zestawy testowe zdefiniowane w tym module, wypisane zostały również w tabeli 4.17, każdy z krótkim opisem w języku angielskim.
- `benchmark` - Moduł wykorzystujący dostępne parsery (`dict`, `drain`) w celu sprawdzenia ich wydajności. Wykorzystuje zbiory danych i metodologię stworzoną do pracy logparser[14]. Wyniki uzyskane z wykorzystaniem tego modułu znajdują się w 4.2.1.
- `differ` - Narzędzie CLI (*ang. Command Line Interface*) działające zgodnie z algorytmem detekcji anomalii opisanym w 3. Na podstawie dwóch logów `checked` oraz `baseline` znajduje różnice pomiędzy plikami. Podaje informację jak logi `checked` różnią się od logów `baseline`. Przykładowym wynikiem działania programu jest rysunek 3.5.
- `drain-java-core` - Moduł zapożyczony z repozytorium `drain-java`[25], w celu zestawienia działania parsera słownikowego oraz Draina. Wykorzystany jest przy module `benchmark`, chociaż dość łatwo może być wykorzystany również w aplikacji `differ`.

4.2. Testy jakościowe

4.2.1. Jakość ekstrakcji wzorców

W ramach pracy nad systemem detekcji anomalii, przetestowano oba zaimplementowane parsery: Drain i Słownikowy. Testy, nazywane również “benchmarkami” (*ang. benchmark*), oparte są o zbiory danych i metodologię z pracy Logparser[14]. Wykorzystane zostało większość zbiorów przygotowanych w pracy Logparser, następnie zebrano opracowane przez twórców metryki oraz zestawiono w tabelach poniżej (4.2.1). Oba parsery były testowane w identyczny sposób, z następującymi konfiguracjami:

- Słownikowy - Wykorzystany został słownik języka angielskiego z odmianami.
- Drain - Ustawiono domyślne wartości: $\texttt{st} = 0.4$, $\texttt{depth} = 4$

Dodatkowo, stworzono tabelę 4.1 zestawiającą wyniki dla parsera słownikowego wraz z wynikami innych parserów, oficjalnie podanymi przez ich twórców. Dane dla parsera słownikowego zaczerpnięte zostały z danych benchmarkowych.

	Dokładność			Wynik F1 Score		
Dataset	**Słownikowy**	**Drain**	**Brain**	**Słownikowy**	**Drain**	**Brain**
HDFS	0.808	0.997	0.997	0.9863	0.9999	0.9999
Hadoop	0.728	0.947	0.949	0.8895	0.9999	0.9987
Spark	0.922	0.920	0.997	0.9946	0.9919	0.9999
Zookeeper	0.922	0.966	0.987	0.9997	0.9995	0.9998
BGL	0.794	0.962	0.986	0.9822	0.9996	0.9999
HPC	0.819	0.887	0.945	0.9837	0.9906	0.9977
Thunderbird	0.636	0.955	0.971	0.9998	0.9993	0.9999
Windows	0.987	0.997	0.997	0.9999	0.9999	0.9999
Linux	0.093	0.690	0.996	0.3155	0.9924	0.9999
Android	0.430	0.911	0.960	0.8611	0.9959	0.9968
HealthApp	0.126	0.780	1.000	0.6726	0.9184	1.0000
Proxifier	0.002	0.526	1.000	0.4881	0.7849	1.0000
OpenSSH	0.434	0.787	1.000	0.9420	0.9992	1.0000
OpenStack	0.470	0.732	1.000	0.7415	0.9925	1.0000
Mac	0.723	0.786	0.942	0.9532	0.9754	0.9958

Tabela 4.1: Zestawienie wyników jakościowych dla różnych parserów
Zielony - najlepszy wynik; Żółty - drugi najlepszy; czarny - trzeci

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.7862	0.95	0.8604	0.4305	True
DRAIN	0.6685	0.9964	0.8001	0.397	True

Tabela 4.2: Wyniki jakościowe dla zbioru danych Android

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9999	0.9622	0.9807	0.794	False
DRAIN	0.9976	0.9998	0.9987	0.9335	False

Tabela 4.3: Wyniki jakościowe dla zbioru danych BGL

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.973	0.9863	0.8085	True
DRAIN	1.0	1.0	1.0	0.9975	True

Tabela 4.4: Wyniki jakościowe dla zbioru danych HDFS

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9887	0.9788	0.9837	0.8195	True
DRAIN	0.7607	0.98	0.8566	0.5115	True

Tabela 4.5: Wyniki jakościowe dla zbioru danych HPC

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9999	0.801	0.8895	0.728	True
DRAIN	0.9689	0.801	0.877	0.453	True

Tabela 4.6: Wyniki jakościowe dla zbioru danych Hadoop

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.5066	1.0	0.6726	0.126	False
DRAIN	0.9994	0.4326	0.6038	0.28	False

Tabela 4.7: Wyniki jakościowe dla zbioru danych HealthApp

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9998	0.1889	0.3178	0.093	True
DRAIN	0.9821	0.6169	0.7578	0.0645	True

Tabela 4.8: Wyniki jakościowe dla zbioru danych Linux

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9222	0.9862	0.9532	0.723	True
DRAIN	0.9475	0.9946	0.9705	0.7895	True

Tabela 4.9: Wyniki jakościowe dla zbioru danych Mac

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.8903	0.942	0.434	True
DRAIN	0.9951	0.9995	0.9973	0.4525	True

Tabela 4.10: Wyniki jakościowe dla zbioru danych OpenSSH

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9983	0.5897	0.7415	0.47	False
DRAIN	0.7999	0.831	0.8152	0.1215	False

Tabela 4.11: Wyniki jakościowe dla zbioru danych OpenStack

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.3228	0.4881	0.0025	True
DRAIN	0.9987	0.1988	0.3316	0.0	True

Tabela 4.12: Wyniki jakościowe dla zbioru danych Proxifier

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.9893	0.9946	0.922	True
DRAIN	0.9839	1.0	0.9919	0.92	True

Tabela 4.13: Wyniki jakościowe dla zbioru danych Spark

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	0.9999	0.9967	0.9983	0.636	False
DRAIN	0.9876	0.9989	0.9932	0.7985	False

Tabela 4.14: Wyniki jakościowe dla zbioru danych Thunderbird

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.9999	0.9999	0.9875	True
DRAIN	0.9889	0.9776	0.9832	0.4235	True

Tabela 4.15: Wyniki jakościowe dla zbioru danych Windows

parser	precission	recall	f_score	accuracy	timestamp_extracted
DICT	1.0	0.9971	0.9985	0.9225	True
DRAIN	0.9984	0.9999	0.9991	0.9305	True

Tabela 4.16: Wyniki jakościowe dla zbioru danych Zookeeper

4.2.2. Spreparowane przypadki

W celu przetestowania skuteczności zaimplementowanego systemu wykorzystano opracowane narzędzie `differ`. Przypadki te mają na celu zweryfikowanie prostych problemów, z którymi system zawsze powinien sobie poradzić. Przykładem tych sytuacji mogą być: losowo pojawiające się wydarzenia, wydarzenie przesunięte w czasie, nowe wydarzenia (nowy wzorzec), etc. W tabeli 4.17 zestawione zostały wszystkie takie uwzględnione przypadki.

Treść przypadków testowych odnaleźć można w repozytorium kodu, w folderze `e2e-tests/tests/`. Każdy podfolder tego folderu to oddzielny przypadek testowy. W ramach przypadku testowego znaleźć się muszą pliki tekstowe: `baseline.txt`, `checked.txt` oraz `expected.txt`. Dodatkowo posiadać on może również plik tekstowy `DESCRIPTION` zawierający opis przypadku testowego.

Nazwa testu	Opis (po angielsku)
`EndOfLogRemoved`	Checks if when two files are the same, except last few lines, these lines should get reported.
`MiddleOfLogRemoved`	Checks wheter when middle of the checked logs will be removed, it will be discovered correctly.
`MultipleRandomInserts`	Checks if one inserted event will be discovered
`OneLineAdded`	Checks if randomly inserted events will be discovered
`OneLineChanged`	This is a case described in the sample flow section in the thesis.
`SampleFromThesis`	Checks if file is properly parsed and change in log line is detected.
`InterestingCase1`	Checks if switched event timings will be found. Only events that were moved around should be detected.
`LineModified`	This case modifies one line in a major way. Additionally, both files have some lines removed.

Tabela 4.17: Tabela zestawiająca sprefabrykowane testy

4.2.3. Logi prawdziwych systemów

Najtrudniejszym do zwalidowania elementem zaproponowanego systemu, jest jego rzeczywista skuteczność na logach pochodzących z aplikacji i serwisów produkcyjnych. Nie ma dobrze opracowanej metodologii i zbioru danych przystosowanych do walidacji opracowanego formatu danych wyjściowych. Z tego powodu zdecydowano się na manualną walidację i opisanie przykładów wyniku pracy systemu na różnych konkretnych przykładach. Przy zbieraniu przykładów posłużono się zbiorami opisanymi i zebranymi w pracy *"A Critical Review of Common Log Data Sets"*[30]. Dodatkowo zaprezentowano logi lokalnie zebrane przez autora w celu dodatkowej praktycznej weryfikacji programu.

Listingi poniżej prezentują fragmenty wyjścia generowanego przez narzędzie `differ`. Każda kolejna linia odpowiada linii z pliku `checked`. Dodatkowo podzielona jest ona logicznie znakiem `|`: pierwszy fragment to numer linii, drugi to komunikat programu `differ`, trzeci to treści linijki. Dodatkowo, gdy

dopasowanie nie jest perfekcyjne czasowo, jako czwarta części linii wyświetlana jest odpowiednia linia z pliku `baseline`. Każdy fragment jest dodatkowo opisany, by wytłumaczyć jego wartość testową.

```
...
2510 | Incorrect Time | 2015-10-17 21:40:51,780 INFO [IPC Server handler 13 on 49594]
    org.apache.hadoop.mapred.TaskAttemptListenerImpl: Progress of TaskAttempt <...> is :
     0.7788796 | 2015-10-17 21:38:31,181 INFO [IPC Server handler 29 on 22927] org.
    apache.hadoop.mapred.TaskAttemptListenerImpl: Progress of TaskAttempt <...> is :
    0.76604986
2511 | Incorrect Time | 2015-10-17 21:40:54,796 INFO [IPC Server handler 13 on 49594]
    org.apache.hadoop.mapred.TaskAttemptListenerImpl: Progress of TaskAttempt <...> is :
     0.7795243 | 2015-10-17 21:38:34,212 INFO [IPC Server handler 17 on 22927] org.
    apache.hadoop.mapred.TaskAttemptListenerImpl: Progress of TaskAttempt <...> is :
    0.7671452
2512 | Additional | 2015-10-17 21:41:25,122 WARN [LeaseRenewer:msrabi@msra-sa-41:9000]
    org.apache.hadoop.ipc.Client: Address change detected. Old: msra-sa
    -41/10.190.173.170:9000 New: msra-sa-41:9000
2513 | Additional | 2015-10-17 21:41:25,137 WARN [LeaseRenewer:msrabi@msra-sa-41:9000]
    org.apache.hadoop.hdfs.LeaseRenewer: Failed to renew lease for [
    DFSClient_NONMAPREDUCE_483047941_1] for 46 seconds. Will retry shortly ...
2514 | Additional | java.net.NoRouteToHostException: No Route to Host from MININT-
    FNANLI5/127.0.0.1 to msra-sa-41:9000 failed on socket timeout exception: java.net.
    NoRouteToHostException: No route to host: no further information; For more details
    see: http://wiki.apache.org/hadoop/NoRouteToHost
2515 | Additional |   at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native
    Method)
2516 | Additional |   at sun.reflect.NativeConstructorAccessorImpl.newInstance(
    NativeConstructorAccessorImpl.java:57)
2517 | Additional |   at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(
    DelegatingConstructorAccessorImpl.java:45)
...
```

Kod 4.1: Przykład Hadoop 1:
baseline: `application_1445087491445_0001`, checked: `application_1445087491445_0002`

Przykład 4.1 przedstawia prostą detekcję błędu, który wystąpił w pliku `checked`, a nie wystąpił w pliku `baseline`. Jest to przykład detekcji prostego przypadku błędu, z którym system radzi sobie bez większego problemu.

```
...
2151 | Ok | 2015-10-19 14:21:56,768 INFO [main] org.apache.hadoop.mapred.MapTask:
    bufstart = 0; bufvoid = 104857600 | Baseline: 2015-10-19 14:21:46,443 INFO [main]
    org.apache.hadoop.mapred.MapTask: bufstart = 0; bufvoid = 104857600
2152 | Ok | 2015-10-19 14:21:56,768 INFO [main] org.apache.hadoop.mapred.MapTask:
    kvstart = 26214396; length = 6553600 | Baseline: 2015-10-19 14:21:46,443 INFO [main]
     org.apache.hadoop.mapred.MapTask: kvstart = 26214396; length = 6553600
2153 | Ok | 2015-10-19 14:21:56,815 INFO [main] org.apache.hadoop.mapred.MapTask: Map
    output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer | Baseline
    : 2015-10-19 14:21:46,458 INFO [main] org.apache.hadoop.mapred.MapTask: Map output
    collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
2154 | Additional | 2015-10-19 14:22:29,612 INFO [main] org.apache.hadoop.mapred.MapTask
    : Spilling map output
2155 | Additional | 2015-10-19 14:22:29,612 INFO [main] org.apache.hadoop.mapred.MapTask
    : bufstart = 0; bufend = 34174195; bufvoid = 104857600
2156 | Additional | 2015-10-19 14:22:29,612 INFO [main] org.apache.hadoop.mapred.MapTask
    : kvstart = 26214396(104857584); kvend = 13786432(55145728); length =
    12427965/6553600
```

```
2157 | Incorrect Time | 2015-10-19 14:22:29,612 INFO [main] org.apache.hadoop.mapred.
    MapTask: (EQUATOR) 44659953 kvi 11164984(44659936) | Baseline: 2015-10-19
    14:21:49,115 INFO [main] org.apache.hadoop.mapred.MapTask: (EQUATOR) 44659953 kvi
    11164984(44659936)
2158 | Incorrect Time | 2015-10-19 14:23:01,974 INFO [SpillThread] org.apache.hadoop.
    mapred.MapTask: Finished spill 0 | Baseline: 2015-10-19 14:22:00,037 INFO [
    SpillThread] org.apache.hadoop.mapred.MapTask: Finished spill 0
...
```

Kod 4.2: Przykład Hadoop 2:
baseline: `application_1445182159119_0001`, checked: `application_1445182159119_0002`

Przykład 4.2 przedstawia interesujący przypadek, w którym w pliku `checked` zauważono 3 dodatkowe linie w stosunku do pliku `baseline`. Oznacza to, że linie `2154 - 2156` mogą być szczególnie ciekawe dla osoby analizującej system.

```
...
545 | Ok | [ 0.356258] pci 0000:00:01.3: bridge window [mem 0x00100000-0x000fffff 64bit
    pref] to [bus 01] add_size 200000 add_align 100000 | Baseline: [ 0.314452] pci
    0000:00:01.3: bridge window [mem 0x00100000-0x000fffff 64bit pref] to [bus 01]
    add_size 200000 add_align 100000
546 | Ok | [ 0.356262] pci 0000:00:01.3: bridge window [mem 0x00100000-0x000fffff] to [
    bus 01] add_size 200000 add_align 100000 | Baseline: [ 0.314455] pci 0000:00:01.3:
    bridge window [mem 0x00100000-0x000fffff] to [bus 01] add_size 200000 add_align
    100000
547 | Ok | [ 0.356266] pci 0000:00:02.2: bridge window [io 0x1000-0x0fff] to [bus 02]
    add_size 1000 | Baseline: [ 0.314457] pci 0000:00:02.2: bridge window [io 0x1000-0
    x0fff] to [bus 02] add_size 1000
548 | Ok | [ 0.356269] pci 0000:00:02.2: bridge window [mem 0x00100000-0x000fffff] to [
    bus 02] add_size 200000 add_align 100000 | Baseline: [ 0.314458] pci 0000:00:02.2:
    bridge window [mem 0x00100000-0x000fffff] to [bus 02] add_size 200000 add_align
    100000
549 | Additional | [ 0.356285] pci 0000:00:01.3: bridge window [mem 0xe0500000-0
    xe06fffff]: assigned
550 | Additional | [ 0.356289] pci 0000:00:01.3: bridge window [mem 0xe0700000-0
    xe08fffff 64bit pref]: assigned
551 | Additional | [ 0.356291] pci 0000:00:02.2: bridge window [mem 0xe0900000-0
    xe0afffff]: assigned
552 | Additional | [ 0.356295] pci 0000:00:01.3: bridge window [io 0x1000-0x1fff]:
    assigned
553 | Additional | [ 0.356298] pci 0000:00:02.2: bridge window [io 0x2000-0x2fff]:
    assigned
554 | Additional | [ 0.356302] pci 0000:00:01.3: PCI bridge to [bus 01]
555 | Ok | [ 0.356307] pci 0000:00:01.3: bridge window [io 0x1000-0x1fff] | Baseline: [
    0.314488] pci 0000:00:01.3: bridge window [io 0x1000-0x1fff]
556 | Ok | [ 0.356311] pci 0000:00:01.3: bridge window [mem 0xe0500000-0xe06fffff] |
    Baseline: [ 0.314491] pci 0000:00:01.3: bridge window [mem 0xe0500000-0
...
```

Kod 4.3: Lokalne logi dmesg - dodatkowe linie

Przykład 4.3 jest dość podobny do 4.2. Przedstawia on również detekcję dodatkowych linii `549 - 554`, tym razem podczas procesu uruchamiania się systemu operacyjnego. Jak widać, po oznaczeniu dodatkowych linii, system dalej dopasował linie pasujące do tych z pliku `baseline`.

```
1 | Additional | [ 0.000000] Linux version 6.10.9-200.fc40.x86_64 (...) (gcc (GCC)
    14.2.1 20240801 (...), GNU ld version ...) #1 SMP PREEMPT_DYNAMIC Sun Sep 8 17:23:55
```

```
     UTC 2024
2 | Ok | [ 0.000000] Command line: BOOT_IMAGE=(hd0,gpt3)/vmlinuz-... root=UUID=... ro
    rootflags=subvol=root rd.luks.uuid=... rhgb quiet | Baseline: [ 0.000000] Command
    line: BOOT_IMAGE=(hd0,gpt3)/vmlinuz-... root=UUID=... ro rootflags=subvol=root rd.
    luks.uuid=... rhgb quiet
3 | Perfect | [ 0.000000] BIOS-provided physical RAM map:
4 | Perfect | [ 0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009ffff] usable
5 | Perfect | [ 0.000000] BIOS-e820: [mem 0x00000000000a0000-0x00000000000fffff]
    reserved
6 | Perfect | [ 0.000000] BIOS-e820: [mem 0x0000000000100000-0x0000000009bfefff] usable
7 | Perfect | [ 0.000000] BIOS-e820: [mem 0x0000000009bff000-0x0000000009ffffff]
    reserved
8 | Perfect | [ 0.000000] BIOS-e820: [mem 0x000000000a000000-0x000000000a1fffff] usable
9 | Perfect | [ 0.000000] BIOS-e820: [mem 0x000000000a200000-0x000000000a20cfff] ACPI
    NVS
10 ...
```

Kod 4.4: Lokalne logi dmesg - perfekcyjne dopasowanie

Przykład 4.4 przedstawia początkowe linie logu, dla których znalezione zostało parę perfekcyjnych dopasowań. Oznacza to, że oba podane logi są praktycznie identyczne na samym początku - zgodnie z oczekiwaniami, system operacyjny na takiej samej architekturze uruchamia się podobnie.

4.3. Wnioski

Wnioski podzielone zostały na dwie podsekcje dotyczące: parsowania logów i jakości systemu detekcji anomalii.

4.3.1. Wnioski dotyczące wyników parsowania logów

Najbardziej widocznym zjawiskiem w wynikach parsowania logów (4.2.1, 4.2.2), jest niespójność pomiędzy wynikami algorytmu Drain pomiędzy danymi wygenerowanymi z wykorzystaniem zaimplementowanego `benchmarka` oraz danymi podanymi przez autorów algorytmu Drain. Niespójność ta wynika z różnicy pomiędzy założeniami projektowymi Draina i Parsera Słownikowego. Parser słownikowy nie posiada parametrów konfiguracyjnych, jego założeniem jest uniwersalność. Drain natomiast posiada parę parametrów, dodatkowo twórcy przy wykonywaniu pomiarów przygotowali inne informacje, jak zbiór wyrażeń regularnych (*ang. regex*) ułatwiających Drainowi zparsowanie logów. Drain testowany poprzez system `benchmark` wykorzystywał zawsze domyślne parametry.

Wyniki z `benchmarka` wydają się wskazywać, że dokładność (*ang. accuracy*) działania parsera słownikowego, przy założonych warunkach jest podobna do Draina. W zależności od przypadku może być lepsza lub taka sama. Co interesujące jednak, parser słownikowy charakteryzuje się widocznie lepszym wynikiem F1 (*ang. F1 Score*). Oznacza to, że algorytm jest bardziej spójny przy swoich decyzjach niż Drain. Prawdopodobnie wynika to z heurystyki podobieństwa wzorców wykorzystanej przy algorytmie Drain, która nie została wykorzystana przy parserze słownikowym.

Oprócz weryfikacji parserów, zestawienia `benchmarka` pokazują również skuteczność ekstrakcji znaczników czasowych (3.3). W każdej z tabel, oprócz metryk parserów, opisano status algorytmu ekstrakcji znaczników czasowych (pole `timestamp_extracted`). Skuteczność osiągnięta przez naiwny ekstraktor znaczników czasowych, na omawianym zbiorze danych, wyniósł $^{12}/_{15} = 80\%$. Wynik ten jest raczej zadawalający biorąc pod uwagę założenia projektowe.

Dużo informacji o działaniu parsera słownikowego, dało zestawieniego jego wyników z wynikami Draina i Braina. Ważnym spostrzeżeniem przy ich analizie jest to, że Drain i Brain przy testowaniu

posiadały znacznie więcej wiedzy domenowej o type logów, niż parser słownikowy. Brain jest algorytmem, który wypadł zdecydowanie najlepiej w każdej metryce. Kolejnym algorytmem pod względem uzyskanych wyników jest Drain, który w wielu przypadkach wypadł niewiele lepiej od Dict-a (parsera słownikowego), a nawet momentami gorzej. Dowodzi to że parser słownikowy, mimo swojej prostoty, umie radzić sobie z parsowaniem logów z efektem niewiele gorszym od Draina, jednocześnie wymagając znacznie mniej konfiguracji i wiedzy domenowej.

Mimo ogólnie dobrych wyników parsera słownikowego, istotnym jest zwrócenie uwagi na jego gorsze wyniki. Przykładowo, problematycznym zbiorem danych okazał się `HealthApp`. W celu zrozumienia tego stanu rzeczy, warto zwrócić uwagę na przykładową linię logu z tego zbioru danych:

```
Linia z HealthApp:
20171223-22:15:29:648|Step_ExtSDM|30002312|calculateAltitudeWithCache totalAltitude=240

Wzorzec znaleziony przez Dict Parser:
<Param> <Param> <Number> <Word> <Word> <Number>
```

Dict Parser nie jest w stanie poradzić sobie z formatem logów wykorzystanym w `HealthApp`. Logi z tej aplikacji wykorzystują technikę “cammel Case” zgodnie z którą zamiast separować słowa znakiem białym, te oddzielone są od siebie duża literą. Przykładowo: `calculateAltitudeWithCache -> calculate altitude with cache`. Jest to przypadek, z którym parser słownikowy nie umie sobie poradzić. Warto zauważyć, że rozwinięcie tokenizera (3.2.1) o separację słów z wykorzystaniem notacji takich jak: camelCase, PascalCase, etc; mogłoby znacząco zwiększyć jego uniwersalność.

4.3.2. Wnioski dotyczące wyników detekcji anomalii

Pierwszym rodzajem eksperymentów wykonanych na systemie detekcji anomalii na podstawie logów, były te wykonane na spreparowanych przykładach logów przygotowanych na potrzeby tej pracy (4.2.2). W ten sposób zweryfikowano podstawowe przypadki, między innymi te opisane w rozdziale 3. Takie testy dają gwarancję fundamentalnych funkcjonalności zaproponowanego rozwiązania. W szczególności sprawdzone zostało, że system poradzi sobie z:

- Wykryciem różnicy w ilości oraz treści linii logów
- Wykryciem niespójności czasowej pomiędzy wydarzeniami w logach
- Redukowaniem ilości informacji przy porównywaniu logów

W celu dodatkowej weryfikacji systemu, wykonano manualne testy na zbiorach logów, co opisano w 4.2.3. Przypadki te potwierdzają skuteczność systemu przy logach z systemów produkcyjnych.

5. PODSUMOWANIE

Współczesne systemy informatyczne charakteryzują się wysokim stopniem złożoności, co stawia przed administratorami i programistami wyzwanie w zakresie ich efektywnego monitorowania i diagnostyki. Pliki logów, zawierające chronologiczny zapis zdarzeń z działania aplikacji, stanowią z tego powodu cenne źródło informacji diagnostycznych, bez generowania dodatkowych kosztów i problemów. Ze względu na brak ujednoliconego formatu i ogromną ilość generowanych danych, manualna analiza logów jest czasochłonna i narażona na błędy osoby analizującej. W pracy przeprowadzono przegląd literatury dotyczącej automatycznej analizy logów, ze szczególnym uwzględnieniem metod wykrywania anomalii z ich użyciem. Omówiono różne podejścia do tego problemu, w tym techniki uczenia maszynowego, analizę semantyczną oraz inne metody oparte o reguły i heurystyki.

W ramach pracy zaproponowano i zaimplementowano algorytm do asysty przy wykrywaniu anomalii na podstawie logów. Algorytm zbudowany jest na podstawie porównania dwóch logów: jednego reprezentującego prawidłowe działanie systemu (`baseline`) i drugiego, w którym podejrzewa się wytłumaczenie anomalii (`checked`). Kluczowym elementem algorytmu jest wykorzystanie bitonicznej macierzy Monga do efektywnego dopasowania sekwencji zdarzeń w obu plikach logów z wykorzystaniem ich znaczników czasowych. Zastosowanie tej metody pozwoliło na znaczną redukcję złożoności obliczeniowej algorytmu w porównaniu do niewyspecjalizowanych metod dopasowywania sekwencji, takich jak algorytm Węgierski. Dodatkowo zaproponowano i zaimplementowano algorytmy parsowania oraz ekstrakcji znaczników czasowych, zaprojektowanych do radzenia sobie z arbitralnymi logami bez potrzeby posiadania wiedzy domenowej.

W celu oceny skuteczności zaproponowanych rozwiązań przeprowadzono szereg eksperymentów z wykorzystaniem zarówno syntetycznych, jak i rzeczywistych zbiorów testowych, pochodzących między innymi od organizacji LogPai. Wyniki eksperymentów potwierdziły skuteczność zaimplementowanego systemu przy manualnych testach, który skutecznie znalazł potencjalnie interesujące zdarzenia. W ramach eksperymentów dokonano również szeregu testów algorytmów parsujących (w tym zaproponowanego) z wykorzystaniem danych oraz metodologii opracowanej przez organizację LogPai.

Zaproponowany system stanowi obiecujące narzędzie asystujące przy procesie wykrywania anomalii w logach. Eksperymenty na nim potwierdziły, że teza postawiona w pracy jest prawdziwa: jest możliwe do oszacowania czy linia w logu danego systemu jest istotna z punktu widzenia anomalii, w oparciu jedynie o plik z logami systemu gdy anomalia nie wystąpiła.

Pomimo obiecujących wyników, system w obecnej formie posiada pewne ograniczenia, szczególnie widoczne w tabelach benchmarku. Znacznym usprawnieniem mogłoby być rozwinięcie zaproponowanego algorytmu tokenizacji oraz parsera słownikowego. Przykładowo, algorytm powinien dzielić token na parę tokenów, w przypadku gdy ten zapisany jest w którejś ze znanych notacji programistycznych. Możliwe jest też opracowanie wielu innych rozszerzeń, które zaprojektować można w wyniku przeprowadzenia dodatkowych eksperymentów, tak, aby system radził sobie z szerszą gammą logów.

Innym interesującym kierunkiem dalszych badań, jest opracowanie algorytmu automatycznej ekstrakcji znaczników czasowych. W ramach przeglądu literatury, nie udało się znaleźć prac na ten temat, co sugeruje, że jest to obszar dotychczas niezbadany lub mało przedstawiany w literaturze. Opracowanie takiego algorytmu mogłoby znacząco zwiększyć uniwersalność proponowanego systemu analizy logów, umożliwiając jego zastosowanie do danych o nieznanym formacie znaczników czasowych. Algorytm ten mógłby wykorzystywać techniki uczenia maszynowego do rozpoznawania wzorców reprezentujących datę i czas w tekście, a następnie konwertować je do ujednoliconego formatu lub wartości.

Pozwoliłoby to na eliminację konieczności ręcznego definiowania formatu czasu przez użytkownika, co jest obecnie jednym z ograniczeń proponowanego systemu.

SPIS RYSUNKÓW

SPIS TABEL

BIBLIOGRAFIA

[1] H. W. Kuhn. „The Hungarian method for the assignment problem". W: *Naval Research Logistics Quarterly* 2.1-2 (1955), s. 83–97. DOI: https://doi.org/10.1002/nav.3800020109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[2] N. Tomizawa. „On some techniques useful for solution of transportation network problems". W: *Networks* 1.2 (1971), s. 173–194. DOI: https://doi.org/10.1002/net.3230010206. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230010206. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230010206.

[3] Jack Edmonds i Richard M. Karp. „Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". W: *J. ACM* 19.2 (kw. 1972), s. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699. URL: https://doi.org/10.1145/321694.321699.

[4] A. Aggarwal i in. „Efficient minimum cost matching using quadrangle inequality". W: *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*. 1992, s. 583–592. DOI: 10.1109/SFCS.1992.267793.

[5] Gradle Inc. *Gradle Build Tool*. https://gradle.org/. Accessed: 2024. 2007.

[6] JetBrains. *Kotlin: The Pragmatic Language for JVM, Android, JavaScript, Native, and more*. https://kotlinlang.org/. Accessed: 2024. 2010.

[7] Pratibha Sharma, Surendra Yadav i Brahmdutt Bohra. „A review study of server log formats for efficient web mining". W: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. 2015, s. 1373–1377. DOI: 10.1109/ICGCIoT.2015.7380681.

[8] Pinjia He i in. „An Evaluation Study on Log Parsing and Its Use in Log Mining". W: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016, s. 654–661. DOI: 10.1109/DSN.2016.66.

[9] Pinjia He i in. „Drain: An Online Log Parsing Approach with Fixed Depth Tree". W: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, s. 33–40. DOI: 10.1109/ICWS.2017.13.

[10] 3Blue1Brown (Grant Sanderson). *What is backpropagation really doing? | Chapter 3, Deep learning*. Accessed: 2024. 2017. URL: https://youtu.be/Ilg3gGewQ5U?si=ShgQ0c4Gii2qjkoU.

[11] Diomidis Spinellis. „Modern Debugging: The Art of Finding a Needle in a Haystack". W: (2018). DOI: 10.1145/3186278. URL: https://cacm.acm.org/research/modern-debugging/.

[12] Jian Tan i in. „On Resource Pooling and Separation for LRU Caching". W: *Proc. ACM Meas. Anal. Comput. Syst.* 2.1 (kw. 2018). DOI: 10.1145/3179408. URL: https://doi.org/10.1145/3179408.

[13] Bin Xia i in. „LogGAN: A Sequence-Based Generative Adversarial Network for Anomaly Detection Based on System Logs". W: *Science of Cyber Security*. Red. Feng Liu i in. Cham: Springer International Publishing, 2019, s. 61–76. ISBN: 978-3-030-34637-9.

[14] Jieming Zhu i in. „Tools and benchmarks for automated log parsing". W: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '19. Montreal, Quebec, Canada: IEEE Press, 2019, s. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021. URL: https://doi.org/10.1109/ICSE-SEIP.2019.00021.

[15] Hetong Dai i in. *Logram: Efficient Log Parsing Using n-Gram Dictionaries*. 2020. arXiv: 2001.03038 [cs.SE]. URL: https://arxiv.org/abs/2001.03038.

[16] Sasho Nedelkoski i in. *Self-Supervised Log Parsing*. 2020. arXiv: 2003.07905 [cs.LG]. URL: https://arxiv.org/abs/2003.07905.

[17] Van-Hoang Le i Hongyu Zhang. „Log-based Anomaly Detection Without Log Parsing". W: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, s. 492–504. DOI: 10.1109/ASE51524.2021.9678773.

[18] Wojciech Samek i in. „Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications". W: *Proceedings of the IEEE* 109.3 (2021), s. 247–278. DOI: 10.1109/JPROC.2021.3060483.

[19] Li Bo Zhao Zhijun Xu Chen. „A LSTM-Based Anomaly Detection Model for Log Analysis". W: *Journal of Signal Processing Systems*. 2021, s. 745, 751. DOI: https://doi.org/10.1007/s11265-021-01644-4.

[20] Thomas H. Cormen i in. *Introduction to Algorithms*. 4th. MIT Press, 2022, s. 1312. ISBN: 978-0-262-04630-5.

[21] Łukasz Korzeniowski i Krzysztof Goczyła. „Landscape of Automated Log Analysis: A Systematic Literature Review and Mapping Study". W: *IEEE Access* 10 (2022), s. 21892–21913. DOI: 10.1109/ACCESS.2022.3152549.

[22] Van-Hoang Le i Hongyu Zhang. „Log-based anomaly detection with deep learning: how far are we?" W: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, s. 1356–1367. ISBN: 9781450392211. DOI: 10.1145/3510003.3510155. URL: https://doi.org/10.1145/3510003.3510155.

[23] Siyu Yu i in. „Brain: Log Parsing With Bidirectional Parallel Tree". W: *IEEE Transactions on Services Computing* 16.5 (2023), s. 3224–3237. DOI: 10.1109/TSC.2023.3270566.

[24] Jieming Zhu i in. „Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics". W: *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 2023.

[25] Brice Dutheil. *drain-java: A pet project to explore log pattern extraction using DRAIN*. https://github.com/bric3/drain-java. Accessed: 2024. 2024.

[26] Zhihan Jiang i in. *LILAC: Log Parsing using LLMs with Adaptive Parsing Cache*. 2024. arXiv: 2310.01796 [cs.SE]. URL: https://arxiv.org/abs/2310.01796.

[27] Humza Naveed i in. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[28] Open Source Initiative. *The MIT License*. Accessed: 2024. 2024. URL: https://opensource.org/licenses/MIT.

[29] LogPai team. *LogPai*. Accessed: 2024. 2024. URL: https://logpai.com/.

[30] AIT Austrian Institute of Technology. *anomaly-detection-log-datasets: Scripts to analyze publicly available log data sets*. https://github.com/ait-aecid/anomaly-detection-log-datasets. Accessed: 2024. 2024.

[31] Junjielong Xu i in. „DivLog: Log Parsing with Prompt Enhanced In-Context Learning". W: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639155. URL: https://doi.org/10.1145/3597503.3639155.