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

Do zrobienia: *Abstrakt*

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 linijka w pliku logów (ang. Log File).*
- **Wzorzec** - *Tekst opisujący jak wyglądać może linijka logu. Dana linijka może albo pasować do wzorca albo do niego nie pasować.*
- **Parser** - *Algorytm/Program, który znajduje strukturę w wejściowych danych tekstowych*

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. Biorąc pod uwagę ograniczenia tradycyjnych często bardzo złożonych narzędzi do rozwiązywania problemów[9], w przypadku omawianych środowisk inżynierowie często uciekają się do znacznie prostszej, a jednak dość skutecznej techniki: przechwytywania i zapisywania śladów aplikacji jako plików tekstowych. Te, wzbogacone są o dodatkowe dane, takie jak znaczniki oznaczające czas. Te pliki, powszechnie znane jako “logi” lub “pliki logów”, służą jako kluczowe zasoby do monitorowania, diagnozowania i rozumienia wewnętrznych mechanizmów współczesnych systemów informatycznych.

Pliki logów zazwyczaj przyjmują formę półstrukturalnych 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 czasu, istotność informacji, lokalizację, numer wątku, itp. W pracy przyjęte zostało również, że wydarzenia (linijki) w plikach z logami są uporządkowane w sposób chronologiczny. Oznacza to więc, że dla dowolnej linijki wewnątrz pliku, każda następna linijka wystąpić mogła w tej samej lub kolejnej chwili czasu - *nigdy wcześniej*. Problem jednak polega na tym, że nie ma ujednoliconego sposobu strukturyzowania plików z logami, więc w praktyce większość aplikacji przyjmuje zupełnie inne struktury plików logów[5], m. in. w zależności od:

- Języka programowania - Wiele z języków posiada swoje własne, domyślne, systemy logowania wydarzeń. Są one jednak zwykle specyficzne dla danego środowiska.
- Bibliotek cyfrowych - Większe biblioteki często projektują własne systemy zapisu wydarzeń, tak aby jak najlepiej wpasowywały się w specyfikę 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 wydarzenia w systemie. Różnią się one jednak często między wersjami systemu, dystrybucjami oraz oczywiście 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 wydarzeń, który 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 gdyż 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 własność 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 "loggowanie", 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 to, 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 wydarzeń, czyli logów, które w nich wystąpiły.

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

- *Baseline* - Plik ten powinien posiadać 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 nieporządany bądź niespodziewany. Każda linijka w tym logu, w wyniku działania systemu, będzie posiadała odpowiadającą sobie linijkę w pliku *baseline* (o ile zostanie znaleziona).

Dodatkowo użytkownik ma możliwość dostrojenia dodatkowych parametrów, w celu dalszego usprawnienia skuteczności systemu. Umożliwia to wykorzystanie wiedzy domenowej użytkownika gdy jest to możliwe. W przeciwnym wypadku, czyli gdy użytkownik nie posiada wiedzy domenowej lub nie ma jak jej podać, algorytm również powinien sobie poradzić z postawionym problemem. 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 jednak zadaniem trywialnym. 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 linijka 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, linijki logu powinny posiadać podobną strukturę (ale nie wzorzec !). Struktura ta opisuje to jak wygląda linijka oraz jakie informacje muszą w niej się znajdywać, pomijając samą treść komunikatu. W praktyce dane takie obejmują informacje takie jak obecny czas, istotność wydarzenia (*ang. Severity*), nazwę modułu, etc.

Problem jednak polega na tym, że nie istnieje uniwersalny sposób na definiowanie lub zwykrywanie powyżej omawianej struktury logów. Sprawia to, że zadanie polegające na jej automatycznym zrozumieniu przez program komputerowy jest nietrywialne. Struktura w tym kontekście oznacza pewną część wspólną dla wszystkich linijek logów. Z tego powodu, używane też jest pojęcia wzorca. Oznacza on pewien wzór zgodnie z którym zdefiniowana jest dana linjka logu - zwykle sprowadzający się do części zmiennych oraz stałych danego tekstu.

Problematykę różnorodnych struktur logów opisać można na dość prostym przykładzie, na którego potrzeby sfabrykowano przykładową linijkę 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 linijka 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 linijki logu. Oznacza to więc, że dla dowolnej linijki logu l, parser $\mathbb{P}$ powinien umieć znaleźć w niej wzór $p = \mathbb{P}(l)$, przyjmujący jako argument parametry x, z wykorzystaniem których odtworzyć można daną linijkę 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 linijkę logu l^*, która reprezentuje identyczną informację co linijka l', jednak ma zupełnie inną strukturę. Dla przykładu z rysunku 1.1, stworzyć można następujące, logicznie identyczne linijki. 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 wydarzeniu 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 linijka logu - inne warianty

Ilustruje to dość spory problem przy projektowaniu algorytmu, którego celem jest wykrycie wzorców bez dodatkowych informacji o strukturze logów. Każdy plik może posiadać zupełnie inne “time-

stamp”y, zwane również znacznikami czasowymi, czyli opis czasu kiedy wystąpiło wydarzenie. Dodatkowo może on posiadać inne bardziej lub mniej istotne metadane, takie jak: istotność wydarzenia (*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 linijek logów zostanie zgłebiony dokładniej w kolejnych sekcjach, jednak jest bardzo ważny przy zrozumieniu tematyki pracy. Powyższe oznaczenia wykorzystane będą przy dalszych rozważaniach, dotyczących opisywanego dalej algorytmu oraz poszczególnych pod-problemów.

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 i/lub wskazaniu, w którym miejscu należy szukać "winnej" linijki logu. Dodatkowo mogą one określać stopień swojej pewności w kwestii takiej predykcji lub wydobywać dodatkowe metadane z tekstu.

Przy problemach tego typu, naukowcy często skupiają się na różnych rozwiązaniach opartych o machine learning. Współcześnie często wykorzystują one sieci neurownowe oparte o zróżnicowane architektury [17, 20, 11]. Tego typu rozwiązania często dobrze radzą sobie w systemach na których były uczone, jednak w systemach nie spotkanych wcześniej, ich generalizacja często umie okazać się dość słaba. Dodatkowo, sporym problemem sieci neuronowych jest ich niewyjaśnialność[16] (*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. Z ich wykorzystaniem, zadaniem znalezienia optymalnego rozwiązania zajmuje się komputer zamiast człowieka, wszystko dzięki algorytmowi propagacji wstecznej[8] (*ang. Backpropagation*). Po drugie, zwykle gdy sieć zostanie wytrenowana i posiada odpowiednią architekturę, umie ona osiągać bardzo dobrą średnią celność (*ang. ang. Accuracy*).

W kolejnych sekcjach, omówione zostaną różne podziedziny detekcji anomalii na podstawie logów, korzystając z których, zaprojektować można kompletny system wykrywający anomalię. Każda sekcja opisana poniżej związana będzie z finalnym komponentem systemu opisanego w dalszych rozdziałach tej 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

2.1.1. Znajdywanie wzorców

Pierwszym typem algorytmów wykorzystywanych w tego typu systemach, są algorytmy automatycznie odnajdujące strukturę logów (patrz. 1.1) [12, 6]. Zwykle są one oparte o uczenie maszynowe, jednak nie jest to wymaganiem. 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 [12], 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, to kryterium może polegać na ich dużej, bądź małej ilości. Zwykle algorytmy o większej ilości hiperparemterów lepiej można dostosowywać do danego typu logu, 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 [18]. Z jej wykorzystaniem opisać można jak zmieni się czas wykonania alogrytmu w zależności od wielkości danych wejściowych.

Do przykładów algorytmów ekstrakcji wzorców należy algorytm Drain [7], będący algorytmem typu "online", który radzi sobie dość dobrze z bardzo różnymi rodzajami logów, szczególnie po dostosowaniu jego hiperparametrów. Algorytm ten posiada dwa takie parametry: `depth` (głębokość drzewa) oraz `st` (Simmilarity treeshold - Granica 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 natomiast 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 linijka musi być identyczna aby pasować do danego wzorca. Jego zmniejszanie sprawia że porównanie staje się bardziej "rozymte" - częściej linijki nie pasujące do końca do wzorca, zostaną i tak do niego dopasowane.

Naturalną kontynuuacją algorytmu Drain, jest algorytm Brain[21], zaprojektowany przez tych samych autorów co Drain w 2023 roku. Działa on podobnie do Braina, jednak graf wykorzystany wewnętrznie przez algorytm jest znacznie bardziej złożony. Krawędzie wewnątrz drzewa są dwukierunkowe, sprawia to, że wzorce znalezione przez algorytm są składają się z współdzielonych wierzchołków, które reprezentują kolejne fragmenty wzorca. Fragment może być jednym z dwóch: albo parametrem albo stałym symbolem. Według wyników pracy autorów, algorytm ten lepiej radzi sobie z bardziej złożonymi logami, niż jego poprzednik.

Istnieje też wiele innych algorytmów ekstrakcji wzorców[13, 22, 14]. 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 dziedzinie analizy logów. Obecnie, większość nowoczesnych metod w tej dziedzinie opiera się na wykorzystaniu sieci neuronowych[17, 20, 11]., które cechują się zdolnością do przetwarzania dużych ilości danych oraz adaptacji do różnorodnych wzorców.

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

1. **Przetworzenie wstępne danych**: Wstępne przetworzenie linijek logów jest niezbędne przed podaniem ich do sieci neuronowej. Proces ten może obejmować różne techniki, takie jak automatyczne wykrywanie wzorców[20], analiza semantyczna[15], itp. Przykładowo, automatyczne wykrywanie wzorców pozwala na identyfikację struktury logów, co ułatwia ich dalszą analizę przez sieć neuronową.

2. **Wykorzystanie sieci neuronowej**: Na tym etapie przetworzone dane są podawane do sieci neuronowej, która przeprowadza ich ewaluację. 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 te 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, wyniki są zbierane i przetwarzane w celu wygenerowania końcowych wniosków. Ten krok może obejmować ocenę pewności predykcji, identyfikację konkretnej linijki logu odpowiedzialnej za anomalię oraz wydobycie dodatkowych metadanych, które mogą być przydatne w dalszej analizie.

Podejścia oparte na sieciach neuronowych oferują znaczące korzyści, takie jak zdolność do automatycznego uczenia się z danych oraz wysoka dokładność predykcji. Niemniej jednak, istnieją również wyzwania związane z ich stosowaniem. Jednym z głównych problemów jest trudność w interpretacji wyników generowanych przez sieci neuronowe, co utrudnia zapewnienie stabilności i niezawodności systemu. Ponadto, sieci neuronowe mogą mieć problemy z generalizacją do nowych, wcześniej nieznanych danych, co może ograniczać ich skuteczność w niektórych przypadkach.

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 przy algorytmie opisanym w dalszych sekcjach tej pracy. Problem wyrażony jest w dość łatwy sposób: jak, nie mając żadnej wiedzy domenowej o danym systemie, automatycznie określić w jakim momencie wystąpiło dane wydarzenie (linijka logu) w stosunku do pozostałych wydarzeń w pliku. Informacja ta zwykle przechowywana jest w pewnej części tekstu logu, zwanej znacznikiem czasowym - zwykle na początku linijki.

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 [19], zgodnie z którymi poniższe założenia są dość uniwersalnie spełniane. Podejście takie miało na celu maksymalizację rzeczywistej użyteczności projektowanego algorytmu.

1. Linijki wewnątrz jednego pliku zawsze ułożone są chronologicznie. Nie jest możliwe aby linijka występująca po innej linijce, w rzeczywistości wydarzyła się wcześniej.

2. Linijki mogą posiadać znaczniki czasowe. Znacznik czasowy, to część logu, znajdujaca się w jego początkowej części, opisująca zgodnie z konwencją danego systemu, kiedy wystąpiło dane wydarzenie. Nie jest wiadome jaki jest dokładny format takiego znacznika oraz jaka jest jego dokładność bez dodatkowej widzy domenowej.

3. Każdy plik z logami posiada przynajmniej jedną linijkę logu posiadającą znacznik czasowy. Jeżeli linijka logu nie posiada takiego znacznika, przyjmuje się że wystąpiła ona w tym samym momencie co poprzednie wydarzenie 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. Oczywiście, najkorzystniejszą sytuacją dla omawianego

problemu, jest taka, w której jak najwięcej linijek posiada znaczniki czasowe o jak największej dokładności.

Warto zauważyć w tym miejscu, że często format logów w ramach tego samego systemu i/lub systemów mu pokrwenych, jest często znany przez ich użytkowników. Oznacza to więc, że dość łatwe jest wykorzystanie ich wiedzy, aby znacznie uprościć omawiany problem. W tej pracy omówiony zostanie również taki wariant, jest on uznawany za przypadek szczególny algorytmu.

2.2. Najlepsze dopasowanie

Plik logu traktować można jak uporządkowany zbiór linijek $l \in L$. Dodatkowo przyjęto, że zdefiniowany został parser logów $\mathbb{P}$. Każda linijka poddana została procesowi parsowania, w wyniku czego uzyskano ciąg par: wzorca p oraz linijki logu l. Następnie linijki dopasowywane są w grupach, zgodnie z znalezionym wzorcem - dopasowane względem czasu, są wszystkie linijki o tym samym wzorcu. W przypadku problemu postawionego w pracy, jednym z pod problemów staje się optymalne dopasowanie do siebie dwóch takich zbiorów, tak aby każda linijka z jednego logu posiadała odpowiadającą linijkę z drugiego. W pracy tej przyjęto, że dobrą miarą dopasowania dwóch linijek do siebie, jest dystans czasowy pomiędzy wystąpieniem linijek.

Problem ten więc przedstawić można jako minimalne dopasowanie do siebie dwóch niezależnych 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 dowolna wartość, którą można porównać z inną wartością z tej samej dziedziny 2.2. Zbiory A oraz B oznaczają odpowiednio pliki *baseline* oraz *checked*, gdzie każda linijka to kolejny element zbioru [1].

$$d : a \times b \to c, \quad a \in A, b \in B, c \in C \tag{2.1}$$

$$\forall_{c_1,c_2 \in C}\ c_1 \neq c_2 \implies c_1 > c_2 \vee c_2 > c_1 \tag{2.2}$$

Jako funkcję odległości d przyjęto dystans pomiędzy dwoma czasami, gdzie dla każdej z linijki logu zdefiniowana jest relacja określająca jego czas wystąpienia, relatywnie do wspólnego punktu początkowego, jako $t(x)$ - równanie 2.3. Równanie definiujące funkcję dystansu opisano poniżej 2.4.

$$t : X \to C, \quad X = A \cup B \tag{2.3}$$

$$d(a,b) = |t(a) - t(b)|, \quad a \in A, b \in B \tag{2.4}$$

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

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

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 2.5

$$\min \sum_{(i',j')\in S} Q[i', j'] \tag{2.5}$$

Algorytm węgierski, umie znaleźć najlepsze takie dopasowanie ze złożonością obliczeniową opisaną jako $O(n^4)$. Ma on jednak 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 dopasowania. W przypadku problemu dopasowania zbiorów A oraz B, macierz Q opisana jest zgodnie z równaniem 2.6

$$Q[i, j] = d(A_i, B_j), \quad |A| = |B| \tag{2.6}$$

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 2.7. Warto zwrócić uwagę, że niektóre z wartości "niewybieralnych" zostaną wybrane przez algorytm węgierski, jednak oznaczają one *brak* dopasowanej linijki.

$$Q[i, j] = \begin{cases} d(A_i, B_j) & i \leq |A| \wedge j \leq |B| \\ \infty & i > |A| \vee j > |B| \end{cases} \tag{2.7}$$

$$O((max(|A|, |B|))^2) \tag{2.8}$$

Sprawia to, że złożoność pamięciowa jest kwadratowa (2.8), względem mocy większego zbioru z pary zbiorów: A oraz B. Przez to, algorytm ten 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 $V \in A + B$. 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 2.9 minimalizowana jest wartość 2.10.

$$\begin{cases} \forall_{a\in A}\, \exists_{b\in B}\, \exists e_{a,b} & |A| < |B| \\ \forall_{b\in B}\, \exists_{a\in A}\, \exists e_{a,b} & |B| < |A| \\ \forall_{b\in B}\, \forall_{a\in A}\, \exists e_{a,b} & |A| = |B| \end{cases} \tag{2.9}$$

$$\sum_{e_{a,b}\in E'} v(e_{a,b}) = \sum_{e_{a,b}\in E'} d(a, b) \tag{2.10}$$

Ten model w dość intuicyjny sposób reprezentuje strukturę linijek 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 (2.3, 2.4). Kryterium, które należy spełnić aby relację między zbiorami opisać można było z wykorzystaniem takiej macierzy, opisane zostało w równaniu 2.11. 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 $O(Nlog(M))$.

$$\forall_{a_1,a_2\in A}\forall_{b_1,b_2\in B} \begin{cases} t(a_1) \leq t(a_2) \wedge t(b_1) \leq t(b_2) \\ d(a_1,b_1) + d(a_2,b_2) \leq d(a_1,b_2) + d(a_2,b_1) \end{cases} \tag{2.11}$$

Linijki logów w pliku spełniają równanie 2.11, oznacza to więc, że macierz tą wykorzystać można do znalezienie optymalnego dopasowania zbiorów A oraz B. Dowód na to wyprowadzony został w sekcji poniżej 2.2.4. Co szczególnie atrakcyjne, macierz Monga nie musi być macierzą kwadratową, co sprawia że jest szczególnie dobrze pasująca do omawianego problemu.

Macierz M stworzyć można w sposób analogiczny do tego opisanego dla macierzy Q w algorytmie węgierskim 2.6. Każdemu elementowi z zbioru A przypisać należy indeks wiersza i, natomiast każdemu elementowi z zbioru B nalwży przypisać indeks kolumny. Następnie dla każdej kombinacji i z j wypełnić macierz M według 2.12. Dla uzyskanej macierzy istnieje algorytm znajdujący takie dopasowanie ze złożonością opisaną wyrażeniem 2.13. Dodatkowo na rysunku 2.1 zwizualizowano relację między zbiorami A i B z wykorzystaniem macierzy M.

$$\begin{array}{c} \\ B_1 \\ B_2 \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccc} A_1 & A_2 & \dots \end{array} \\ \begin{bmatrix} d(A_1,B_1) & d(A_2,B_1) & \dots \\ d(A_1,B_2) & d(A_2,B_2) & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \end{array}$$

Rysunek 2.1: Macierz Monga M dla zbiorów A i B

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

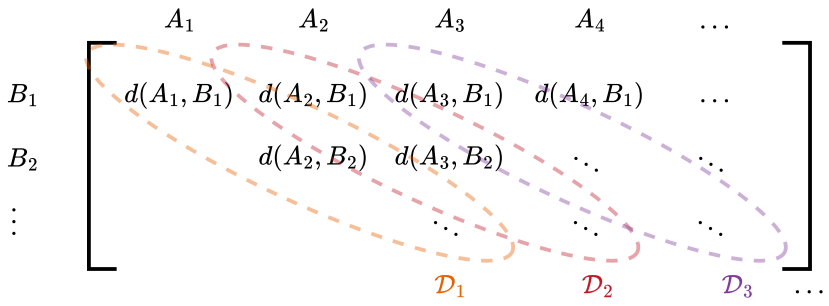
[3] W przypadku tej pracy, odległość między elementami w macierzy monga oznacza znormalizowaną względem pierwszego wydarzenia odległość czasową - spełnia ona warunki takiej macierzy.

$$M[i,j] = d(A_i, B_j) \tag{2.12}$$

$$O\left(\min(|A|,|B|) \cdot \log(\max(|A|,|B|))\right) \tag{2.13}$$

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 2.2. 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 2.14 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.

$$\mathcal{D}_l = \{M[1,l], M[2,l+1], M[3,l+2], \ldots, M[n = |B|, l+n]\} \tag{2.14}$$



Rysunek 2.2: Diagonale $\mathcal{D}_l$ w macierzy M

2.2.4. Dowód, że logi to BMM

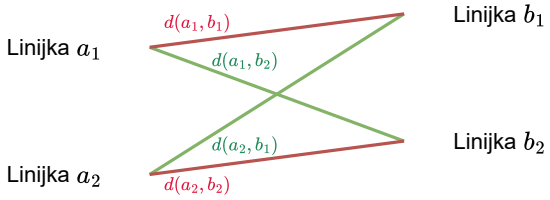
Zgodnie z założeniami przyjętymi w wstępie do pracy, prawdą jest wyrażenie opisane w równaniu 2.15. Wynika to z ważnej właściwości plików z logami - czyli chronologiczności. W języku nieformalnym, oznacza to, że wydarzenie a_2 musiało wystąpić w tej samej chwili lub w chwili późniejszej niż wydarzenie a_1. Analogiczną właściwość mają elementy z zbioru B, ponieważ mają takie samo źródło.

$$\forall_{a_1,a_2 \in A} \forall_{b_1,b_2 \in B} \begin{cases} t(a_1) \leq t(a_2) \\ t(b_1) \leq t(b_2) \end{cases} \tag{2.15}$$

Aby macierz M była Bitoniczną Macierzą Monga, spełniony musi być warunek 2.16. 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 2.3. 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 $d(a_1, b_2) + d(a_2, b_1)$.

$$\forall_{a_1,a_2 \in A} \forall_{b_1,b_2 \in B} \quad d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1) \tag{2.16}$$

Gdzie: $d(x, y) = |t(x) - t(y)|$



Rysunek 2.3: Geometryczna interpretacja równania 2.16

W równaniu 2.16, 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, szczególnie w omawianym kontekście. Problem ten rozwiązać można poprzez nałożenie dodatkowych ograniczeń na relację między elementami a oraz b. Wszystkie takie sześć wariantów opisano w równaniu 2.17. 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 2.15 oraz równaniu 2.16.

$$\begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_2) \\ t(a_1) \geq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \leq t(b_2) \\ t(a_1) \leq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \leq t(b_2) \\ t(a_2) \geq t(b_1) \\ t(a_1) \geq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \leq t(b_2) \\ t(a_2) \geq t(b_1) \\ t(a_1) \leq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \leq t(b_2) \\ t(a_2) \leq t(b_1) \end{cases} \vee \begin{cases} t(a_2) \geq t(b_2) \\ t(a_1) \geq t(b_2) \end{cases} \tag{2.17}$$

W sekcjach poniżej przeanalizowano wszystkie wyżej opiasne warianty w porządku chronologicznym. Każdy z wariantów reprezentuje inną relację wydarzeń: a_1, a_2, b_1 i b_2. W każdym z nich, dowiedzione jest że spełnione zostało równanie 2.18 zgodnie z warunkami danego wariantu. Dowodzi to, że prawdą jest równanie 2.18, co zarazem kończy dowód.

$$|t(a_2) - t(b_2)| + |t(a_1) - t(b_1)| \leq |t(a_2) - t(b_1)| + |t(a_1) - t(b_2)| \tag{2.18}$$

Wariant 1

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Jest Prawdą

Wariant 2

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Wariant 3

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Wariant 4

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Jest Prawdą

Wariant 5

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Jest Prawdą

Wariant 6

W tym wariancie przyjęto następującą kolejność wydarzeń, czyli linijek logów, która została opisana poniżej. Są to po prostu 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:

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

Korzystając z tych warunków, wykorzystać można równanie 2.18 do sprawdzenia czy jest ono prawdziwe w przyjętych warunkach. W ten sposób uzyskane zostały, a następnie uproszczone następujące wyrażenia:

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

Jest Prawdą

3. ZAPROPONOWANE ROZWIĄZANIE

Zaproponowany system ma za zadanie wyznaczyć, które linijki logu w pliku checked są dla niego unikalne (czyli podejrzane) i/lub, które linijki 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 linijek z obu plików logów. Teoria związana z wyciąganiem wzorców z logów opisana została dokładniej w sekcji 1.1.
- **Ekstrakcja i normalizacja czasu** - Po znalezieniu wzorca dla linijki, kolejnym krokiem jest ekstrakcja czasu jej wystąpienia. Problem ten jest jednak dość złożony i mało przebadany w innych artykułach naukowych, ze względnu na swoją nietypowość. **Do zrobienia:** *WNIOSEK: Jest to coś do przebadania w przyszłości !*
- **Minimalne dopasowanie** - Mając dwa pliki jako zbiory logów - każdy z czasem wystąpienia oraz unikalnym ID wzorca, wykonać można dopasowanie logów z obu plików. Dla logów z obu plików, o tym samym unikalnym ID wzorca, wykonać można takie dopasowanie względem relatywnego czasu wystąpienia. W ten sposób, wiadomo jaka jest różnica czasu między wydarzeniami w programie.

3.1. Idea działania systemu detekcji ważnych wydarzeń

Zadaniem algorytmu jest analiza zależności pomiędzy linijkami pliku `checked` oraz pliku `baseline`. Oba pliki przez algorytm interpretowane są jako listy linijek tekstu - logów. Każda linijka ma możliwość ekstrakcji wzorca oraz znacznika czasowego. Algorytm, dla każdej linijki dokonuje obu ekstrakcji, następnie z wykorzystaniem algorytmu najlepszego dopasowania względem czasu wystąpienia, wybiera linijki, które *nie* znalazły dopasowania. Intuicyjnie, sytuacja taka oznaczać ma, że nie ma wydarzenia które wystąpiło w analogicznym pliku. Prawdopodobnie wydarzenie 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 tego paragrafu 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 linijki znajduje wzorzec do niej pasujący. W algorytmie uwzględniono samo unikalne ID wzorca, więc funkcja mapuje dowolną linijkę 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 linijki 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 linijek 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` - Optymalne struktury danych, odpowiednio: mapa/słownik, 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
```

Pseudokod 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*) [10] z wykorzystaniem którego złożoności tych algorytmów mogą być nawet $O(1)$. Złożoności te zostały oznaczone poniżej, w wyrażeniach 3.2. Dodatkowo, w celu uproszczenia wyrażeń, wykonanie obu ekstrakcji oznaczone zostało jako wspólny algorytm "pre-processingu" (PRE).

$$\begin{aligned} O_{TE} &\text{ - Złożoność algorytmu ekstrakcji czasu} \\ O_{PE} &\text{ - Złożoność algorytmu ekstrakcji wzorca} \\ O_{PRE} &\text{ - Złożoność preprocessingu, czyli } O_{TE} \text{ oraz } O_{PE} \end{aligned} \tag{3.2}$$

Z wykorzystaniem oznaczeń 3.2, dokonać można oszacowań złożoności poszczególnych komponentów O_{Alg}: X, Y i Z, tak jak przedstawiono to na poniższych równaniach 3.3. Dwa pierwsze, X i Y, są do siebie bardzo podobne. Każda linijka jest przetwarzana przez dwa algorytmy: ekstrakcji czasu oraz ekstrakcji wzorca. Następnie, zapisywane są do dalszego przetworzenia przez algorytm z wykorzystaniem optymalnych struktur danych. Teoretycznie możliwe jest wykorzystanie struktur o złożoności gorszej niż $O(1)$, jednak w tej pracy wariant ten jest pominęty. Operacja zapisu/odczytu jest więc pominięta przy tym szacunku.

$$\begin{aligned} X &= O(|C|) \cdot O_{TE}(C) \cdot O_{PE}(C) = O(|C|) \cdot O_{PRE}(C) \\ Y &= O(|B|) \cdot O_{TE}(B) \cdot O_{PE}(B) = O(|B|) \cdot O_{PRE}(B) \\ Z &= O(|C| + |B|) \cdot O_{BMM}(|C|, |B|) = O((|C| + |B|) \cdot min(|C|, |B|) \cdot \log(max(|C|, |B|))) \\ O_{BMM}(a, b) &= min(a, b) \cdot \log(max(a, b)) \end{aligned} \tag{3.3}$$

Równania 3.3 opisują poszczególne złożoności fragmentów pełnego algorytmu. Oznaczenia te następnie wykorzystano w równaniu 3.1 opisującym złożoność $O(\dots)$ pełnego algorytmu. W ten sposób uzyskano, a następnie rozpisano, poniższe wyrażenie 3.4, uzyskując w ten sposób pełną złożoność systemu. Złożoność ta opisana została jako funkcja względem dwóch danych wejściowych - zbiorów logów `checked` i `baseline`.

$$\begin{aligned} O_{Alg}(C, B) &= O(|C|) \cdot O_{PRE}(C) + O(|B|) \cdot O_{PRE}(B) \\ &\quad + O((|C| + |B|) \cdot min(|C|, |B|) \cdot \log(max(|C|, |B|))) \\ &= O(|C|) \cdot O_{PRE}(C)\, O(min(|C|, |B|) \cdot \log(max(|C|, |B|))) \\ &\quad + O(|B|) \cdot O_{PRE}(B)\, O(min(|C|, |B|) \cdot \log(max(|C|, |B|))) \end{aligned} \tag{3.4}$$

Złożoność uzyskana w równaniu 3.4 jest dość skomplikowana. Z tego powodu, warto przyjąć dodatkowe założenie, które w przypadku logów z takiego samego systemu jest prawie zawsze prawdziwe. Linijki w pliku `checked` oraz `baseline` mają podobną strukturę. Oznacza to więc, że: $O_{PRE}(B) = O_{PRE}(C) = O_{PRE}(C, B)$. Wykorzystując to twierdzenie, uzyskano równanie 3.5. Tak więc, złożoność algorytmu opisuje równanie 3.6.

$$\begin{aligned} O_{Alg}(C, B) &= O(|C|) \cdot O_{PRE}(C, B)\, O(min(|C|, |B|) \cdot \log(max(|C|, |B|))) \\ &\quad + O(|B|) \cdot O_{PRE}(C, B)\, O(min(|C|, |B|) \cdot \log(max(|C|, |B|))) \\ &= O(|B| + |C|) \cdot O_{PRE}(C, B) \cdot O(min(|C|, |B|) \cdot \log(max(|C|, |B|))) \end{aligned} \tag{3.5}$$

$$O_{Alg}(C,B) \cdot O_{PRE}(C,B) = O_{PRE}(C,B) \cdot \begin{cases} O((|B|^2 + |B||C|)\log|C|) & |C| > |B| \\ O(|C|^2 \log|C|) & |C| = |B| \\ O((|C|^2 + |B||C|)\log|B|) & |C| < |B| \end{cases} \tag{3.6}$$

Złożoność 3.6 oznacza, że zaproponowany algorytm jest dość odporny na dysproporcje ($||C| - |B||$) w ilości linijek logów pomiędzy plikiem `baseline` oraz `checked`. Zgodnie z oczekiwaniami, przy zwiększaniu jednego z plików, głównym problemem pozostaną algorytmy: ekstrakcji wzorców oraz ekstrakcji czasów. Samo dopasowanie linijek, czyli operację najcięższą do pominięcia dzięki mechanizmom takim jak cache, ma bardzo "niegroźną" złożoność względem długości plików z logami.

Do zrobienia: *Opisać jaką złożoność ma zaimplementowany system (dict parser + naiwny timestamp parser)*

3.1.1. Przykładowy przebieg algorytmu

Do zrobienia: *Przykład jednego pełnego przebiegu algorytmu*

3.2. Dict Parser

Do zrobienia: *Jak zbudowany jest dict parser ? Jaka jest jego złożoność ?*

4. EKSPERYMENTY

Do zrobienia !

4.1. Implementacja

Do zrobienia !

4.2. Testy jakościowe

4.2.1. Sprefabrykowane przypadki

Do zrobienia: *Zrestawienie dla prostych sprefabrykowanych przypadków*

4.2.2. Logi prawdziwych systemów

Do zrobienia !

4.3. Testy wydajnościowe

Do zrobienia: *Zestawienie z innymi parserami*

4.4. Wnioski

Do zrobienia !

5. PODSUMOWANIE

Do zrobienia !

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] 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.

[6] 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.

[7] 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.

[8] 3Blue1Brown (Grant Sanderson). *What is backpropagation really doing? | Chapter 3, Deep learning*. 2017. URL: https://youtu.be/Ilg3gGewQ5U?si=ShgQ0c4Gii2qjkoU.

[9] 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/.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.