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

SPIS TREŚCI

SPIS TREŚCI 4

1 WSTĘP 5

1.1 Struktura logów 6

2 PRZEGLĄD LITERATURY 8

2.1 Detekcja anomalii 8

2.2 Najlepsze dopasowanie 8

2.2.1 Algorytm Węgierski 9

2.2.2 Graf dwudzielny 9

3 ZAPROPONOWANE ROZWIĄZANIE 10

4 EKSPERYMENTY 11

5 PODSUMOWANIE 12

SPIS RYSUNKÓW 13

SPIS TABEL 14

BIBLIOGRAFIA 15

1. WSTĘP

Współczesne systemy informatyczne są złożone i często rozproszone pomiędzy różne niezależne komponenty, które działają asynchronicznie. Zrozumienie ich zachowań i interakcji w tych systemach staje się dużym wyzwaniem, zwłaszcza podczas diagnozowania problemów. Biorąc pod uwagę ograniczenia tradycyjnych narzędzi do rozwiązywania problemów, takich jak GDB, dla tak skomplikowanych środowisk, inżynierowie często uciekają się do bardziej dostępnej, a jednak skutecznej techniki: przechwytywania i zapisywania śladów aplikacji jako plików tekstowych, wzbogaconych 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 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, powagę informacji, lokalizację itp. Problem jednak polega na tym, że nie ma ujednoliconego sposobu strukturyzowania takich plików, więc w praktyce większość aplikacji ma zupełnie różne struktury plików logów[4], m. in. w zależności od:

- Języka programowania
- Bibliotek cyfrowych
- Systemu operacyjnego
- Dostępnych zasobów

Logi stanowią nieocenione źródło danych dla większości nowoczesnych aplikacji, oferując głęboki wgląd w działanie aplikacji bez generowania znaczących dodatkowych kosztów. Ich wszechstronność sprawia, że znajdują zastosowanie w różnych rozwiązaniach technologicznych, od zintegrowanych systemów wbudowanych po zaawansowane rozproszone serwisy oraz usługi internetowe. Ta uniwersalność i bogactwo zawartych informacji sprawiają, że logi są przedmiotem intensywnych badań, zwłaszcza w kontekście ekstrakcji istotnych danych, które mogą pomóc w usprawnieniu procesów diagnostycznych, monitoringu wydajności oraz wykrywaniu potencjalnych problemów.

Analiza logów stanowi kluczowy element utrzymania i diagnozowania współczesnych systemów informatycznych. Logi te służą jako cenne źródło informacji dla programistów, administratorów systemów i zespołów wsparcia, umożliwiając im zyskanie wglądu w zachowanie aplikacji i skuteczne identyfikowanie problemów, bez potrzeby ponownego uruchamiania systemu. Niestety, pomimo powszechności logów we współczesnym oprogramowaniu, brak ustandaryzowanego formatu stwarza interesujące wyzwania w tej dziedzinie.

Do zrobienia: Wraz z rozwojem pracy, aktualizuj ten fragment W tej pracy zaproponowano przykładowy system, 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 pliku *checked*.

- *Checked* - Ten plik zawierać powinien logi z przebiegu aplikacji, która zachowała się w nieporządany bądź niespodziewany sposób. Każda linijka w tym logu, w wyniku działania systemu, będzie posiadała odpowiadającą sobie linijkę w pliku *baseline* (o ile zostanie znaleziona) oraz heurystykę jak istotna jest dana linijka.

Dodatkowo użytkownik ma możliwość dostrojenia dodatkowych parametrów, w celu dalszego usprawnienia skuteczności systemu. Umożliwia to wykorzystanie wiedzy domenowej, kiedy użytkownik ma taką możliwość.

Postawionym zadaniem systemu jest **asysta** użytkownika, tak aby znalezienie istotnych infomacji była ułatwiona w stosunku do innych, bardziej manualnych metod.

Do zrobienia: TEZA

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 wspólną strukturę. Struktura ta opisuje to jak wygląda linijka oraz jakie informacje muszą w niej się znajdywać (pomijając samą informację związaną z logiem). 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 powyższej struktury. Sprawia to więc, że zadanie polegające na jej automatycznym zrozumieniu przez program komputerowy jest nietrywialne. Zilustrować to można na dość prostym przykładzie, na którego potrzeby sfabrykowano przykładową linijkę 1.1, oznaczoną dalej jako l' 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 w tym miejscu zastanowić się, jaki wzór (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: 1.1, 1.2 i 1.3

$$l' = \texttt{'01.01.2024 INFO Client connected'} \tag{1.1}$$

$$p', x' = \mathbb{P}(l') = \texttt{'<timestamp> <severity> <*> connected'}, [\texttt{'01.01.2024'}, \texttt{'INFO'}, \texttt{'Client'}] \tag{1.2}$$

$$l' = p'(x') \tag{1.3}$$

Łatwo wyobrazić sobie linijkę logu l^*, która przekazuje identyczną informację co linijka l', jednak ma zupełnie inną strukturę. Dla przykładu 1.1, wymyślić można następujące logicznie identyczne linijki 1.2, różniące się tylko metadanymi, notacją lub kolejnością.

Do zrobienia: Dokończyć przykład

```
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

2. PRZEGLĄD LITERATURY

2.1. Detekcja anomalii

Problem poruszany w tej pracy, zaliczyć można do działu informatyki nazywanego detekcją anomali (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 inne metadane.

Pierwszym typem algorytmów wykorzystywanych w tego typu systemach, są algorytmy automatycznie odnajdujące strukturę logów (patrz. 1.1) [7][5]. 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** - Jest to najbardziej oczywiste kryterium, jednak również dość ciężkie to ewaluacji. Istnieją systemy oceniające algorytmy pod tym względem [7], jednak kryterium to jest dość subiektywne i może różnić się w zależności od potrzeb.
- **Ilość hiperparametrów** - **Do zrobienia:**
- **Złożoność obliczeniowa** - **Do zrobienia:**

Do przykładów takich algorytmów należy algorytm Drain [6], 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, natomiast zmiejszanie - przeciwnie.

Do zrobienia: ST

Do zrobienia: Brain

2.2. Najlepsze dopasowanie

Plik logu traktować można jak uporządkowany zbiór linijek l. W przypadku problemu postawionego w pracy, jednym z podproblemów 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

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

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

Do zrobienia:

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 najbardziej 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 2.3

$$\min \sum_{(i',j') \in S} Q[i', j'] \tag{2.3}$$

Algorytm więgierski, umie znaleźć najlepsze takie dopasowanie z 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 równanie 2.4

$$Q[i, j] = d(A_i, B_j) \tag{2.4}$$

Do zrobienia:

2.2.2. Graf dwudzielny

Łatwo zauważyć, że relację taką zapisać można w postaci grafu dwudzielnego $G(V, E)$. W grafie tym, każda krawędź reprezentuje odległość pomiędzy pomiędzy elementami z odpowiednio zbioru A oraz B. W takiej sytuacji, problemem staje znalezienie grafu $G'(V, E')$, takiego że **Do zrobienia:**

3. ZAPROPONOWANE ROZWIĄZANIE

Do zrobienia:

4. EKSPERYMENTY

Do zrobienia:

5. PODSUMOWANIE

Do zrobienia:

SPIS RYSUNKÓW

1.1 Przykładowa linijka logu 6
1.2 Przykładowa linijka logu - inne warianty 7

SPIS TABLIC

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] 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.

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

[6] 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.

[7] 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)*. 2019, s. 121–130.