EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 94af9948-8540-4c7d-8446-749aa45f49e1

Aplikácia hľadajúca požadované záznamy v systémovom logovacom súbore operačného systému vytvorená v jazyku C

Bakalárska práca

2020 **Alexander Prokop**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Aplikácia hľadajúca požadované záznamy v systémovom logovacom súbore operačného systému vytvorená v jazyku C

Bakalárska práca

Študijný program: Hospodárska informatika

Študijný odbor: Informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2020 **Alexander Prokop**

Čestné prehlásenie

Čestne prehlasujem, že bakalársku prácu som vypracoval samostatne s použitím uvedenej literatúry.

V Bratislave dňa ..... Podpis .....

Poďakovanie

Chcel by som poďakovať svojmu vedúcemu práce Ing. Igorovi Košťálovi, PhD. za jeho pripomienky a postrehy, ktoré pozdvihli kvalitu práce aj vytvoreného programu.

Abstrakt

PROKOP, Alexander: *Aplikácia hľadajúca požadované záznamy v systémovom logovacom súbore operačného systému vytvorená v jazyku C*. – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Igor Košťál, PhD. – Bratislava: FHI, 2020, 27 strán

V práci je predstavená problematika vyhľadávania v logovacom súbore operačného systému Windows. Práca sa venuje teoretickej stránke vyhľadávania, objasňuje rozdelenia podľa štruktúry textu, vyhľadávacích vzorov, a predstavuje najpoužívanejšie reťazcové vyhľadávacie algoritmy. Praktické využitie týchto poznatkov je ukázané na reálnej implementácií programu v jazyku C, ktorý umožňuje filtrovanie a vyhľadávanie podľa zadaných kritérií z vyexportovaných záznamov logu systému Windows vo formáte CSV.

Kľúčové slová: vyhľadávanie, reťazce, vyhľadávacie algoritmy, regulárne výrazy, logy, Windows, jazyk C, CSV

Abstract

PROKOP, Alexander: *Application for searching through messages in the system log file of the operating system written in the C language*. – University of Economics in Bratislava. Faculty of Business Informatics; Department of Applied Informatics. – Thesis supervisor: Ing. Igor Košťál, PhD. – Bratislava: FHI, 2020, 27 pages

This work introduces the problematic of searching inside the log file of the Windows operating system. Theoretical basis of search is presented, and different input formats, search patterns, and well-known string-search algorithms are explained. Practical use of these ideas is demonstrated on a real program in written in C, which allows filtering and searching in CSV-exported Windows system log according to selected criteria.

Keywords: search, strings, string-search algorithms, regular expressions, logs, Windows, C language, CSV

Obsah

Úvod

Logovanie je dôležitou súčasťou každého operačného systému a aplikácie. Uľahčuje nielen diagnostiku a riešenie problémov, ale poskytuje aj informácie na ďalšie spracovanie a analýzu. Z logov je možné zistiť frekvenciu problémov, momentálne vyťaženie systému, a ďalšie informácie, ktoré sú hodnotné pre Business Intelligence.

Aby sme vedeli informácie z logov užitočným spôsobom využiť, musíme sa v nich vedieť zorientovať a vyhľadať záznamy, ktoré sú pre nás relevantné. V logovacom súbore operačného systému sa spravidla nachádzajú záznamy z všetkých častí systému a na ňom bežiacich aplikácií. Nefiltrované logy tak skôr užívateľa zaplavia svojím obrovským počtom.

Cieľom tejto práce je poskytnúť nástroj na filtrovanie a vyhľadávanie v logoch podľa zadaných kritérií a umožniť tak využitie ich potenciálu. Takýto nástroj nutne stavia na teoretických poznatkoch z odvetvia vyhľadávacích algoritmov a práca predstavuje tie najpoužívanejšie z nich.

1 Súčasný stav riešenej problematiky doma a v zahraničí

Vyhľadávanie – v čomkoľvek – nie je v počítačovej vede žiadnou novinkou. Už desaťročia sa objavujú nové vyhľadávacie algoritmy a postupnou evolúciou a vylepšeniami od viacerých autorov sa ich efektivita ešte zvyšuje. Vyhľadávacie algoritmy však neexistujú vo vákuu a výber toho správneho algoritmu je potrebné vidieť v kontexte podmienok na jeho konkrétne použitie.

Je potrebné jasne definovať parametre vyhľadávania: aká je štruktúra vstupov, ako je reprezentovaný vyhľadávaný reťazec, či je reťazcov viac, a koľkokrát ho treba vo vstupe nájsť. Ďalšími parametrami sú aj implementačná, časová a priestorová náročnosť. Podľa týchto parametrov je potom potrebné vyberať ten správny algoritmus.

Na nasledujúcich stranách budú postupne predstavené príklady existujúcich vstupných formátov, reprezentácií reťazcov, či vyhľadávacích algoritmov – vždy s popisom ich silných aj slabých stránok a zdôvodnenie ich možného použitia.

1.1 Štruktúra vstupu

Jedným zo základných parametrov je štruktúrovanosť vstupu, alebo jej absencia. Pri neštruktúrovaných logoch sú tieto zaznamenané iba ako obyčajný text. Naopak štruktúrované logy majú presne vymedzený formát polí a môžu byť ukladané do rozličných formátov – textových či binárnych. Predstavíme si niekoľko spôsobov záznamových štruktúr.

1.1.1 Bežný text

Toto je neštruktúrovaný formát, ktorý využíva napríklad tradičný Unixový logovací démon syslog. Ide o jednoduché textové záznamy vymedzené iba znakom nového riadku oddeľujúcim jednotlivé záznamy. Akákoľvek ďalšia štruktúra závisí na konkrétnom logovacom programe, ale väčšinou je formát nastaviteľný.

Vyhľadávanie v takto formátovaných logoch tak nutne zahŕňa použitie silných nástrojov na textové vyhľadávanie a patričnú dávku predpokladov vzhľadom na formát polí. Aj keď existujú snahy o štandardizáciu syslog protokolu (RFC 3164 a RFC 5424), tieto štandardizujú hlavičku správy, ale už nie obsah. Ten ostáva doménou každej jednej aplikácie, ktorá syslog využíva a naprieč nimi štandardný formát správ prakticky neexistuje.

Ako už bolo spomenuté, na vyhľadávanie v takýchto logoch sa v praxi využívajú programy špecializujúce sa na textové vyhľadávanie. Unix systémy ich už tradične obsahujú niekoľko a s veľmi vysokou kvalitou. Jedná sa o nástroje ako *grep*, doménové jazyky ako *sed* a *AWK*, či všeobecný programovací jazyk *Perl*.

1.1.2 CSV

Comma-Separated Values, v preklade *Čiarkou-oddelené hodnoty,* je *de facto* formát na ukladanie tabelárnych údajov. Je to formát „*de facto*“ z dôvodu veľkého množstva variantov a to na vzdory snahe o štandardizáciu podľa RFC 4180 a neskoršieho doplnenia v RFC 7111.

Ako je jasné z názvu, CSV oddeľuje jednotlivé polia čiarkou. Samotné záznamy sú oddelené znakom nového riadku. Vzniká tak prirodzený tabuľkový formát. V prvom riadku bývajú spravidla názvy polí. Ak niektorá z hodnôt obsahuje čiarku alebo má viac riadkov, je potrebné celé pole ohraničiť úvodzovkami (úvodzovky vo vnútri takéhoto poľa musia byť zdvojené).

Výhodou formátu CSV je univerzálnosť jeho podpory naprieč aplikáciami. Užívateľ môže vyexportovať databázovú tabuľku do CSV súboru, ten importovať do programu Microsoft Excel, kde môže údaje spracovávať spolu s ostatnými a generovať z nich reporty a grafy. Alebo naopak: užívateľ vyexportuje dáta z Excelu do CSV súboru, ktorý potom importuje do databázy ako tabuľku.

Nevýhodou formátu CSV je nejednotnosť jeho univerzálnej podpory naprieč aplikáciami. Popísaný štandard je síce dominantný, no ani z ďaleka nie jediný. V praxi sa stretneme s CSV formátmi s niekoľkými modifikáciami – napríklad vo viacerých variantoch je oddeľovačom bodkočiarka namiesto čiarky.

Čo sa týka logov, CSV nebýva väčšinou primárnym formátom na ich uchovávanie, aj keď v tejto kapacite by vedel poslúžiť – je to predsa štruktúrovaný formát s neobmedzeným počtom polí. Väčšinou sú však logy uložené v inom, viac flexibilnom formáte, ale dajú sa vyexportovať do CSV. To platí o logoch systému Windows, ktoré sú uchovávané v binárnom formáte, ale dajú sa vyexportovať to viacerých textových formátov, medzi inými aj práve CSV.

Na spracovanie CSV vstupu sa používa CSV parser. Štandardný formát je pomerne jednoduchý a teda aj parser je možné jednoducho implementovať. Komplikácie môžu nastať, ak má parser správne spracovať aj neštandardný vstup – tu je zväčša na programátorovi, s akým stupňom voľnosti vstupu bude počítať.

Príklad formátu CSV:

ID, MENO, ADRESA

1, Ján Novák, "Račianska 9, Bratislava"

2, Peter Kováč, "Námestie SNP 4, Bratislava"

1.1.3 XML

eXtensible Markup Language je populárny formátovací jazyk dokumentov, odvodený z veľmi podobných jazykov SGML a HTML, od ktorých sa líši oveľa striktnejšími pravidlami (XML 1.0). XML a príbuzné štandardy sú široko používané na reprezentáciu komplexných vnorených dátových štruktúr.

XML používa na opis štruktúry dát značky medzi znamienkami väčší a menší, ktoré môžu byť vnorené do seba a reprezentovať tak zložité stromové štruktúry. Okrem toho môže mať každá značka takzvané *atribúty*, ktoré popisujú jej vlastnosti. Tie sa uvádzajú vnútri značky za jej menom ako páry *kľúč=hodnota*.

XML štandardy však majú komplikované pravidlá a na zaručenie časovo prijateľného a správneho spracovania je odporúčané použiť niektorú z existujúcich knižníc. Pre softvérový projekt to znamená pridanie ďalšej závislosti na externom prvku.

Príklad formátu XML:

```
<Zamestnanci>
  <Zamestnanec id=1>
    <Meno meno="Ján" priezvisko="Novák"/>
    <Adresa>
      <Ulica>Račianska 9</Ulica>
      <Mesto>Bratislava</Ulica>
    </Adresa>
  </Zamestnanec>
</Zamestnanci>
```

1.2 Reprezentácia hľadaného reťazca

Ďalším významným parametrom je konkrétny druh reprezentácie hľadaného reťazca. Ten je možné reprezentovať doslovne, alebo pomocou špeciálnych *meta-znakov*, ktoré upravujú hľadaný reťazec a rozširujú možnosti hľadania.

1.2.1 Doslovné vyhľadávanie

V tomto prípade vyhľadávame všetky znaky reťazca presne tak, ako boli napísané. Je to ten najjednoduchší spôsob vyhľadávania reťazca v texte. Výhodou je minimálna náročnosť na komplexnosť kódu a samotného vyhľadávania. Nevýhodou je veľmi malá flexibilita – môžeme vyhľadať len presne to, čo napíšeme.

Tento druh vyhľadávania sa preto hodí na nenáročné účely, kde stačí vyhľadávať jednoduché reťazce, ale kde môžu byť vysoké nároky na rýchlosť vyhľadávania. Známou

implementáciou je program *fgrep*, ktorý je jedným z viacerých variantov nástroja *grep* na systémoch Unix.

1.2.2 Glob

Toto je pomenovanie pre vzory známe z príkazového rozhrania *Shell* (sh) systému Unix. Počas svojej existencie sa vyvíjali spolu s rôznymi verziami Shellu, a momentálne existuje viacero vzájomne nekompatibilných, alebo iba čiastočne kompatibilných, variantov. Kvôli popularite a všeobecnej prítomnosti Unix systémov sa tieto vzory dostali do všeobecného povedomia a boli použité aj v iných systémoch (napr. MS-DOS), ale aj v rámci vyhľadávania na webových stránkach.

Aj napriek veľkému množstvu nekompatibilných verzií sa implementácie väčšinou zhodujú na niekoľkých základných, univerzálnych meta-znakoch, ktoré reprezentujú určité textové vzory. Hviezdička „*“ označuje ľubovoľný (aj nulový) počet akýchkoľvek znakov. Otáznik „?“ určuje presne jeden ľubovoľný znak. Hranaté zátvorky „[ ]“ označujú presne jeden znak spomedzi znakov v nich obsiahnutých. Prípustné sú aj rozsahy znakov označené pomocou pomlčky medzi prvým a posledným znakom rozsahu. Pokiaľ je prvým znakom výkričník „!“, tak označujú akýkoľvek jeden znak *okrem* znakov v nich obsiahnutých.

Jednoduchosť tohto konceptu a nízke nároky na používateľa spravila z Globov univerzálnu súčasť takmer každého vyhľadávania. Táto jednoduchosť má však aj odvrátenú stránku a znamená, že nie je možné pomocou Globov špecifikovať pokročilé textové vzory – napríklad Globy nepoznajú alternatívy (t.j. vyhľadaj „jablká“ alebo „hrušky“). Pokročilé varianty pôvodného programu Shell, napríklad Bash alebo Korn Shell, túto situáciu riešia pomocou pridávaní ďalších meta-znakov s rôznymi funkciami. No aj napriek tomu musí používateľ v zložitejších prípadoch siahnuť po pokročilejšom nástroji.

1.2.3 Regulárne výrazy

Tým nástrojom sú práve Regulárne výrazy, tiež známe pod skratkou „regex“, ktorá je odvodená z anglického „regular expression“. Praktické využitie regulárnych výrazov má pôvod v textovom editore QED (Thompson 1968) a na ňom založenom editore Ed systému Unix. Vďaka svojej ohromnej flexibilite a využiteľnosti sa rýchlo rozšírili mimo svojho pôvodného hostiteľa a dnes ich vieme nájsť na najrôznejších miestach. Z Unixových systémov je známy textový vyhľadávací program *grep*, prúdový editor *sed*, či jazyk *AWK* (Kernighan 2020). Aj ďalší známy programovací jazyk – Perl – má svoju vlastnú implementáciu Regulárnych výrazov.

So širokým rozšírením ide ruka v ruke aj veľký počet variantov. Medzi tie najpoužívanejšie patria pôvodné *Základné* regulárne výrazy, neskoršie *Rozšírene* výrazy, a *Perl-kompatibilné regulárne výrazy* (známe pod skratkou PCRE).

Regulárne výrazy majú, na rozdiel od Globov, oveľa väčší počet meta-znakov a pokročilejšiu syntax. Tým dosahujú oveľa väčšie možnosti pri vytváraní textových vzorov. Predstavíme si syntax Rozšírených regulárnych výrazov, ktoré sa však vo väčšine prípadov zhodujú so vzormi Základnými.

Univerzálnym znakom je bodka „.“, ktorá označuje ľubovoľný znak. Hranaté zátvorky „[ ]“ zase označujú jeden ľubovoľný znak v nich uvedený, alebo znakové rozsahy pomocou pomlčky. Komplement znakov v hranatých zátvorkách je strieška „^“ ako prvý znak.

Ďalšou kategóriou meta-znakov sú takzvané *kvantifikátory*. Tie upravujú počet výskytu znaku pred nimi. Hviezdička „*“ označuje, že predchádzajúci znak sa môže vyskytnúť v ľubovoľnom počte (aj nula). Otáznik „?“ zase určuje, že predchádzajúci znak sa vyskytne buď vôbec alebo iba raz. Rozšírené regulárne výrazy poznajú aj kvantifikátory plus „+“, ktorý hovorí, že znak sa vyskytne aspoň raz, a zložené zátvorky „{ }“, v ktorých je presný počet výskytu, alebo čiarkou-oddelený interval.

Regulárne výrazy poznajú aj takzvané *ukotvenia*, čo sú meta-znaky „^“ a „$“ označujúce začiatok, respektíve koniec riadka a tým „ukotvujú“ vzor v riadku.

Rozšírené regulárne výrazy poznajú aj znaky pre alternatívu „|“ a zoskupenie „( )“. Zoskupenia sú brané ako jeden znak a platia pre nich kvantifikátory. Niektoré implementácie podporujú takzvané *spätné odkazy*, ktoré pomocou spätného lomena a čísla poradia zoskupenia zopakujú obsah konkrétneho zoskupenia na neskoršom mieste výrazu.

Ako je zrejmé, výhodou Regulárnych výrazov je ich vysoká flexibilita pri špecifikácií vyhľadávaných vzorov. Nároky kladené na užívateľa sú tiež vyššie, a preto nasadenie Regulárnych výrazov je skôr v odbornejšej sfére – tým sa myslia hlavne výkonné vyhľadávacie nástroje a rozličné programovacie jazyky.

Nevýhodou môžu byť vyššie nároky na implementáciu, hlavne ak implementácia podlieha istým výkonovým kritériám. Aj keď sa nazývajú Regulárne výrazy, implementácie nemusia mať regulárnu gramatiku. To je zapríčinené hlavne spätnými odkazmi, ktorých funkčnosť vyžaduje použitie bezkontextovej gramatiky. Tento fakt zvyšuje nároky a zložitosť parsera a vyhľadávacieho algoritmu.

1.3 Vyhľadávacie algoritmy

Samotné vyhľadávanie určitého vzoru v texte je realizované vyhľadávacími algoritmami. Tie sú rôzne, odlíšené svojimi vlastnosťami, ktoré ich predurčujú na konkrétne druhy použitia. Pri vyhľadávacích algoritmoch sledujeme, mimo iných, hlavne tieto veličiny: časová náročnosť, priestorová náročnosť, prípadne náročnosť predprípravy.

1.3.1 Naivné vyhľadávanie

Tiež známe ako „brute-force“, pri tomto vyhľadávaní postupne skúšame každý znak, až pokým nenarazíme na zhodu s hľadaným vzorom. Ako je jasné už z popisu, tento algoritmus je nevýhodný pre veľké reťazce. Zložitosť v najhoršom prípade sa pohybuje v medziach O(mn), čiže je kvadratická. Aj keď sa to môže zdať neefektívne, v praktických podmienkach textu písaného prirodzeným jazykom máva tento algoritmus oveľa lepšie vlastnosti. Boyer a Moore píšu, že naivné vyhľadávanie aplikované na anglický text je často prakticky lineárne, čiže blížiace sa zložitosti O(m+n).

Výhodou naivného vyhľadávania je jeho prakticky neexistujúca pamäťová zložitosť, a nepotrebnosť predprípravy. Hodí sa preto na účely, ktoré sú drasticky limitované pamäťou, neznesú latenciu spojenú s predprípravou a text má štruktúru priaznivú pre tento algoritmus.

1.3.2 Boyer-Moore algoritmus

Tento algoritmus bol objavený v roku 1977 dvojicou amerických autorov. Stoja za ním Robert S. Boyer zo Stanfordského výskumného inštitútu a J Strother Moore z výskumného strediska Xerox PARC, ktorí ho publikovali v žurnále ACM (Boyer 1977).

Výhodou Boyer-Moore algoritmu je nízka časová náročnosť, ktorá plynie z jeho vlastnosti preskakovať veľké nezhodujúce sa časti textu. Základom je poznatok, že ak budeme porovnávať hľadaný reťazec z textom odzadu, vieme tak aplikovať viac informácií o reťazci a predísť zbytočnému porovnávaniu znakov. Spomínané informácie vieme zistiť priamo z reťazca počas predprípravy, ktorá vyprodukuje dve tabuľky na rýchle vyhľadávanie posunov. Tie sú realizované podľa dvoch heuristických pravidiel. Algoritmus tak dosahuje v najlepšom prípade zložitosť iba O(n/m) a priestorovú zložitosť O(k). Predpríprava má zložitosť O(m+k).

Prvou heuristickou metódou je pravidlo chybného znaku. Pokiaľ sa nezhoduje znak reťazca so znakom z textu, vyhľadá sa v reťazci najbližšia ľavá pozícia tohto znaku z textu. Ak sa v reťazci takýto znak nachádza, reťazec sa vzhľadom k textu posunie doprava o vzdialenosť k tomuto znaku (text aj reťazec tak budú mať na tomto mieste zhodný znak). Ak sa v reťazci takýto znak nenachádza, reťazec sa posunie doprava až za hranicu tohto

znaku v texte – bolo by zbytočné ďalšie porovnávanie znakov na mieste, kde vieme, že existuje znak ani sa nenachádzajúci v reťazci.

Druhou metódou je pravidlo správnej prípony. Ak sa po niekoľkých správnych znakoch nájde znak chybný, tieto správne znaky sa zoberú ako predpona, a tá sa vyhľadá v reťazci od konca smerom doľava. Ak sa nájde, celý reťazec sa posunie, aby bola vzhľadom na text zarovnaná. Ďalšou podmienkou je, aby znak nasledujúci po prípone bol iný, ako odhalený chybný znak z posledného porovnania. Zaujímavosťou je, že táto prípona môže „pretiecť“ cez ľavý okraj reťazca – t.j. pre reťazec ABC... je platná prípona YZAB, pričom zhodné sú znaky AB a znaky YZ „pretečú“ cez ľavý okraj.

Samotný algoritmus je možné popísať nasledovne. Vyhľadávaný reťazec za priloží k začiatku vyhľadávaného textu. Kurzor sa nastaví na posledný znak reťazca, ktorý sa porovná s príslušným znakom textu. Ak sa znaky nezhodujú, použije sa prvé pravidlo. Pokiaľ sa zhodujú, porovnávanie pokračuje ďalším znakom. Pokiaľ sa ten nezhoduje, použijú sa obe pravidla, pričom výsledný posun je maximom nimi vypočítaných posunov. Takto algoritmus pokračuje až do nájdenia reťazca, alebo pokiaľ nenarazí na koniec textu bez zhody.

Ako je zrejmé z popisu heuristických pravidiel, na ich výpočet stačia informácie obsiahnuté v samotnom reťazci, plus veľkosť možnej abecedy (pri vyhľadávaní v počítačovom texte spravidla 256 znakov, čo zodpovedá bežným 8-bitovým znakom). Kvôli tejto vlastnosti sa heuristika nerobí on-line počas porovnávania reťazca z textom, ale počas predprípravy sa predpočítajú statické tabuľky posunov – jedna pre každé pravidlo. Prvá tabuľka má veľkosť abecedy a obsahuje hodnotu posunu sprava po najbližší výskyt daného znaku reťazca (dĺžka reťazca, pokiaľ sa v ňom daný znak nenachádza). Druhá tabuľka má veľkosť dĺžky reťazca a obsahuje posuny jednotlivých prípon sprava k najbližšiemu výskytu tejto prípony. Takto predpočítané tabuľky zaisťujú, že počas behu algoritmu je výpočet posunu operácia v konštantnom čase – jednoduchá referencia daného indexu v tabuľke.

V pôvodnej práci Boyer a Moore nepopísali algoritmy na vytvorenie tabuliek posunov. Efektívne algoritmy na ich vytvorenie boli opísané až v neskoršej práci (Knuth 1977). V roku 1980 publikoval Wojciech Rytter nájdenú chybu a jej opravu v Knuthovom algoritme na generovanie druhej tabuľky (Rytter 1980).

1.3.3 Rabin-Karp algoritmus

Tento algoritmus vyniká úplne iným prístupom k problematike vyhľadávania. Namiesto porovnávania znakov používa takzvaný „rolling hash“ vypočítaný z textu, ktorý porovnáva s hashom hľadaného reťazca (Karp 1987). Kľúčovým poznatkom je právne rolling hash, ktorý umožňuje nenáročné vypočítanie nasledujúceho hashu pomocou toho predchádzajúceho, čím znižuje časovú náročnosť algoritmu.

Beh algoritmu možno popísať takto: Ako prvé sa vypočíta hash pre hľadaný reťazec (alebo viac reťazcov). Následne sa vypočíta hash pre prvú skupinu znakov z textu dĺžky hľadaného reťazca. Hashe sa porovnajú a pokiaľ sú rovnaké, tak sa porovnajú aj jednotlivé znaky reťazca a textu. Ak nie sú rovnaké, kurzor v texte sa posunie o jeden znak doprava a vypočíta sa nový hash na základe toho starého. Najprv sa vhodnou operáciou odpočíta hodnota predchádzajúceho prvého znaku a následne sa pripočíta hodnota nového posledného znaku. Táto operácia je prípustná vďaka špecifickému dizajnu rolling hash algoritmu. Následne zase dôjde k porovnaniu hashov a algoritmus takto pokračuje až do nájdenia hľadaného reťazca v texte, alebo po koniec textu bez nájdenia zhody.

Ako je vidieť z popisu algoritmu, väčšina práce je odvedená práve hashovacím algoritmom. Preto treba dbať na vybratie toho správneho algoritmu, ktorý bude mať priaznivé vlastnosti pre nasedenie v textovom vyhľadávaní. Tiež treba dbať na obmedzenie plynúce zo špecifickej potreby rolling hashu, teda je nutné vedieť hash jednoducho prepočítať s novým prvkom. Populárnym hashovacím algoritmom používaným v implementáciách Rabin-Karp algoritmu je takzvaný *Rabinov odtlačok* (Rabin 1981).

Rabin-Karp algoritmus v porovnaní s inými vyhľadávacími algoritmami nie je výhodný pre vyhľadávanie jedného reťazca – jeho priemerná časová náročnosť je O(n+m), na rozdiel od Boyer-Moore algoritmu s O(n/m). Jeho silná stránka sa však ukáže, pokiaľ hľadáme reťazcov niekoľko. Potom na porovnanie stačí jednoduchá numerická komparácia hashu každého hľadaného reťazca, zatiaľ čo pri tradičných algoritmoch by sme museli porovnávať znaky všetkých reťazcov zvlášť.

2 Cieľ práce

Cieľom práce je implementovať teoretické poznatky z odvetvia textového vyhľadávania do podoby programu na vyhľadávanie záznamov v logovacom súbore operačného systému Windows.

Program má byť napísaný v jazyku C. Keďže ide o logy operačného systému Windows, prenositeľnosť na iné operačné systémy nie je nutná. Prenositeľnosť je však možné uľahčiť obmedzením používaných funkcií na štandardnú knižnicu C dostupnú všade.

Hlavnou prioritou je jednoduchosť implementácie. Táto často podceňovaná vlastnosť je veľmi dôležitá na udržanie prehľadnosti programu, čo by sa v konečnom dôsledku malo prejaviť na nižšom počte chýb. Jednoduchosť nie je absolútna a treba ju vyvažovať s ostatnými parametrami programovej architektúry, ako funkcie programu alebo výkonnosť. Systémový logovací súbor býva rozsiahli, preto priveľmi primitívne algoritmy by v konečnom dôsledku spôsobili značné spomalenie behu programu.

Vzhľadom na vyššie uvedené priority by vstupný formát mal byť jednoduchý na spracovanie. Windows umožňuje exportovať systémové logy do viacerých formátov, medzi nimi dva otvorené štandardy XML a CSV. Ako už bolo predstavené v teoretickej časti práce, XML je pomerne zložitý formát na spracovanie, pokiaľ nie sú použité externé knižnice, ktoré zase do projektu pridávajú zbytočné závislosti. Oproti tomu formát CSV má veľmi jednoduché pravidlá, čo značne zľahčuje jeho implementáciu. Formát vstupu by teda mal byť práve CSV.

Výber znakovej sady je priamočiary. Hoci Windows preferuje kódovanie UTF-16 a široké 16-bitové znakové typy, pri exporte do formátu CSV používa ušľachtilejšie kódovanie UTF-8, ktoré používa bežné osembitové znakové typy. Pre jednoduchosť použitia, a ako ústretový krok voči prípadnej prenositeľnosti, program bude pracovať v UTF-8. Nevýhodou je istá úroveň nekompatibility s príkazovým riadkom vo Windowse, keď dochádza k poškodeniu niektorých znakov na výstupe z programu. Toto však nie je problém pri zápise do súboru a preto je to len malá cena v porovnaní s výhodami.

Dôležitá je aj flexibilita rozhrania. Program by mal akceptovať vstup zo súboru, ale aj zo štandardného vstupu – podobne výstup by mal byť okrem súboru možný aj na štandardný výstup. Program tak môže fungovať aj ako filter v spojení s inými programami (napr. koncept Unixových potrubí – pipes). Funkcie programu by teda mali byť ovládateľné pomocou argumentov z príkazového riadku.

Ako ústretový krok voči menej skúseným používateľom by program mal obsahovať aj interaktívne menu, ktoré sa spustí pokiaľ nebol vyžiadaný dávkový mód (batch mode).

Menu nemusí byť grafické – stačí jednoduché textové menu ovládané príkazmi – no musia sa cez neho dať plnohodnotne nastaviť parametre programu.

3 Výsledky práce

Výsledkom práce je program *Logfind* na vyhľadávanie záznamov v logoch systému Windows. Program prijíma súbory vo formáte CSV, ktoré sa dajú v systéme Windows vyexportovať z logovacieho súboru. Kvôli jednoduchosti implementácie bola zvolená doslovná interpretácia hľadaného reťazca. Ako vyhľadávací algoritmus bol použitý Boyer-Moore, ktorého vysoká efektivita je podporená priemernou zložitosťou implementácie. Program pracuje v textovom rozhraní a ponúka interaktívne menu na nastavenie parametrov vyhľadávania, ako aj dávkový mód pre hromadné použitie z príkazového riadku.

Kód programu je voľne dostupný z git repozitáru hosťovaného službou Gitlab na adrese https://gitlab.com/alexprokop/logfind . Kód je pod ultra-permisívnou licenciou WTFPL, ktorá umožňuje bezpodmienečné a neobmedzené používanie, modifikáciu a šírenie – licencia je explicitne definovanou obdobou „verejnej oblasti“ (public domain) známej z angloamerického práva.

3.1 Implementácia programu

Program je napísaný v jazyku C a fyzicky aj logicky rozdelený na nasledujúce celky: Načítavanie záznamov z CSV do vnútornej reprezentácie programu je obsiahnuté v zdrojovom súbore *logfind.c*. Textové vyhľadávanie založené na algoritme Boyer-Moore je vyčlenené do súboru *boymo.c*. Funkcia *main()* obsluhujúca spracovanie argumentov, interaktívne menu, ako aj beh programu a integráciu jeho častí sa nachádza v súbore *main.c*.

3.1.1 main.c

Tu sa nachádza hlavná slučka programu ako aj potrebné „lepidlo“ na spojenie všetkých programových častí dohromady.

Zbežne sa dá algoritmus programu popísať nasledovne. Najprv program inicializuje svoje dátové štruktúry a premenné a alokuje dynamickú pamäť na halde. Nasleduje načítanie a spracovanie argumentov z príkazového riadku. Keďže Windows nepozná štandardnú funkciu *getopt()* na spracovanie argumentov, časť jej funkcionality musela byť implementovaná svojpomocne pomocou *for* cyklu a *switch*-u. Pokiaľ nebol špecifikovaný dávkový mód (batch mode), tak program vypíše textové menu, kde používateľ môže nastaviť parametre programu. Hodnoty z predtým spracovaných argumentov sú uvedené ako predvolené. Potom program otvorí uvedené vstupné a výstupné súbory do streamov (prípadne použije štandardný vstup a výstup ak nie sú uvedené). Pokiaľ bolo zvolené textové vyhľadávanie, sú na základe vyhľadávaného reťazca inicializované dve delta tabuľky (špecifické pre Boyer-Moore algoritmus). Napokon začne hlavná slučka programu.

V slučke sa najprv získajú jednotlivé polia z aktuálneho riadku vstupného súboru pomocou funkcií z logfind.c. Následne sa vzhľadom na zvolené parametre porovnávajú jednotlivé polia s požadovanými hodnotami. Vo väčšine prípadov sú to jednoduché číselné alebo znakové/reťazcové porovnania. Výnimkou je porovnávanie dátumov, ktoré prebieha špeciálnou vlastnou funkciou, a pole „description“ (popis), kde sa vyhľadáva zadaný reťazec pomocou Boyer-Moore algoritmu v texte poľa. Kvôli šetreniu času skončí porovnávanie pri prvom nesprávnom výsledku. Pokiaľ záznam prejde všetkými filtrami, je vytlačený v ľudsky čitateľnom formáte do výstupného streamu a slučka sa začne opakovať až po vyčerpanie všetkých záznamov. Potom program skončí s návratovou hodnotou 0.

3.1.2 logfind.c

Súbor logfind.c obsahuje funkcie, ktoré načítavajú jednotlivé polia zo vstupného CSV súboru. Ďalej sa tu nachádzajú funkcie na spracovanie a porovnávanie dátumov. Keďže táto funkcionalita je oddelená, nie je vylúčené samostatné použitie v inom projekte.

Základom je funkcia *getfield()*, ktorá načíta jedno pole zo vstupného súboru. Funkcia obsahuje svoj vlastný CSV parser, ktorý je vo veľkej miere kompatibilný so štandardom RFC 4180. Na rozdiel od štandardu tento parser pripúšťa úvodzovky na ktorejkoľvek pozícií poľa a to z dôvodu zjednodušenia implementácie. Parser je implementovaný v rámci jednej *switch* štruktúry a pre kompaktnosť používa vedomé prepadávanie v niektorých *case*-och. Parser je volaný v slučke pre každý načítaný znak. Pokiaľ narazí na koniec poľa alebo riadku, funkcia skončí s návratovou hodnotou dĺžky poľa. Extrahované pole je zapísané do pamäte, na ktorú ukazuje smerník dodaný v argumentoch. Funkcia nikdy nenačíta viac znakov ako je veľkosť tohto pamäťového miesta, tiež dodaná ako argument funkcie. Ak funkcia narazí na koniec súboru, toto je signalizované zbytku programu pomocou globálnej premennej. Je to postačujúce riešenie, aj keď nie veľmi elegantné.

Nasledujúce funkcie slúžia na načítanie konkrétnych polí zo vstupu. Všetky používajú vyššie popísanú funkciu getfield() a takto získané pole upravujú do vhodnej reprezentácie. Číselne a znakové hodnoty sú vrátené priamo hodnotou, zatiaľ čo reťazce sú vrátené ako smerník na pamäťové miesto, ktoré ich obsahuje. Výnimkou je funkcia *getdesc()*, ktorá vracia dĺžku poľa, zatiaľ čo reťazec je zapísaný na pamäťové miesto za pomoci smerníka v argumentoch.

Posledným párom funkcií sú funkcie pracujúce s dátumami. Funkcia *strtodate()* transformuje znakový reťazec s dátumom do špeciálnej dátumovej štruktúry, ktorá obsahuje hodnoty roku, mesiaca, dňa, hodiny, minúty a sekundy. V momentálnej podobe dokáže

pracovať iba so vstupným formátom „D. M. RRRR HH:MM:SS“. Druhá funkcia *datecmp()* potom porovnáva dátum a čas medzi dvoma takýmito štruktúrami. Je to iba približné riešenie problematiky porovnávania dátumov – mimo iného ignoruje napríklad časové zóny. Pre potreby programu je však postačujúce, keďže ťažisko práce neleží v spletitých detailoch merania času. Funkcia postupne porovnáva jednotlivé polia dátových štruktúr od poľa s najväčším významom (rok) po najnižší (sekundy). Porovnanie skončí pri prvom páre polí, ktoré sa nezhodujú a návratová hodnota je 1 pokiaľ je vyšší dátum v prvej štruktúre a -1 pokiaľ je v druhej. Ak sa obe dátumové štruktúry zhodujú, vráti sa hodnota 0. Správanie funkcie tak tvorí paralelu k štandardnej funkcií na porovnávanie reťazcov *strcmp()*.

Pridružený je hlavičkový súbor *logfind.h*, ktorý definuje makrá, dátumovú štruktúru, deklaruje externú premennú na detekciu konca vstupu a obsahuje deklarácie funkcií v súbore logfind.c. Makrá slúžia na špecifikáciu maximálnej veľkosti jednotlivých polí načítavaných zo vstupu. Hlavičkový súbor je určený na *include*-nutie do programov, ktoré využívajú funkcie v logfind.c.

3.1.3 boymo.c

Tento súbor zahŕňa implementáciu vyhľadávania založenú na Boyer-Moore algoritme. V programe sa využíva na prehľadávanie poľa Popis.

Funkcia *d1_init()* slúži na vytvorenie prvej delta tabuľky, čo je v kontexte algoritmu súčasťou predprípravy. Funkcia najprv alokuje miesto veľkosti abecedy (v tomto prípade 256 znakov) na halde. V ďalšom kroku sa všetky prvky alokovanej tabuľky nastavia na dĺžku hľadaného reťazca – čo je podľa algoritmu predvolená hodnota, pokiaľ sa daný znak v reťazci nenachádza. V nasledujúcom cykle sa postupne nastavia pre všetky znaky v reťazci ich vzdialenosti od konca reťazca. Keďže cyklus postupuje reťazcom zľava doprava, pre viacnásobne sa vyskytujúce znaky sa vždy hodnota prepíše na najmenšiu vzdialenosť od konca reťazca. Návratová hodnota je smerník na novovytvorenú tabuľku.

Funkcia *d2_init()* vytvára a inicializuje druhú delta tabuľku. Tiež ide o predprípravný krok a algoritmus ju neskôr bude využívať na zistenie vzdialenosti ďalšieho výskytu nájdenej správnej prípony v hľadanom vzore. Po alokácií miesta na halde veľkosti hľadaného reťazca začne algoritmus postupovať odzadu reťazca po možných príponách. Pre každú možnú príponu začne prehľadávať reťazec od konca a zisťovať na každej pozícií prítomnosť prípony pomocou pomocnej funkcie *is_suffix()*. Pokiaľ príponu nájde, alebo narazí na koniec reťazca, zapíše momentálnu vzdialenosť od konca reťazca do tabuľky. Ide teda v podstate o naivný algoritmus, ktorý ale postupuje pri porovnávaní znakov opačným

smerom. Je to z toho dôvodu, že podľa Boyer-Moore algoritmu môže prípona, či už čiastočne alebo celá, pretiecť cez ľavý okraj reťazca. Porovnávanie odzadu umožňuje túto vlastnosť jednoducho a prirodzene implementovať do porovnávacieho algoritmu – porovnávanie skončí so zhodou pokiaľ sa algoritmus dostane za ľavý koniec reťazca. Samozrejme, platia všetky nevýhody použitia naivného algoritmu, akou je vysoká časová náročnosť v najhoršom prípade (aj keď v priemere na bežnom texte sa správa výrazne lepšie). Oproti tomu stojí fakt, že vyhľadávané reťazce majú väčšinou obmedzenú dĺžku a tabuľka sa vytvára počas behu programu iba raz. Tieto faktory značne obmedzujú negatívne dopady použitia naivného algoritmu.

Poslednou funkciou je *bm_search()*, ktorá zabezpečuje už samotné vyhľadávanie. Na sledovanie polohy vo vyhľadávanom reťazci a texte používa funkcia dva indexy: *i_pat* a *i_txt*. Po zavolaní funkcie sa nastavia na rovnakú číselnú hodnotu – i_pat na index posledného znaku reťazca a i_txt na rovnaký index v texte. Následne sa začne hlavná slučka vyhľadávacieho algoritmu. Indexom určený znak reťazca sa porovná so znakom v texte a pokiaľ sa zhodujú, index i_pat sa posunie o jedno miesto doľava a cyklus sa zopakuje. Pokiaľ algoritmus v zhode dosiahne začiatočný znak reťazca, algoritmus skončí a vráti pozíciu prvého znaku nájdeného reťazca v texte. Ak sa znaky nezhodujú, z tabuľky *d1* sa načíta posun k najbližšiemu výskytu znaku z textu v reťazci od jeho konca. Podľa požiadaviek algoritmu sa ošetrí ešte jeden špeciálny prípad – ak by bol tento posun negatívny vzhľadom na momentálny index v reťazci, posun je veľkosti 1. Pokiaľ už boli nájdené nejaké zhodné znaky, algoritmus podľa druhého pravidla načíta aj druhý možný posun z tabuľky d2 – ide o posun k najbližšiemu ďalšiemu výskytu zhodnej prípony. Inak je hodnota druhého posunu 0. Nakoniec sa nájde maximum obidvoch možných posunov a toto určí výsledný posun, ktorý sa aplikuje. Index i_txt sa posunie doprava o výsledný posun a index i_pat sa znova nastaví na koniec reťazca a slučka sa zopakuje.

Pridružený hlavičkový súbor *boymo.h* obsahuje definície funkcií a jedno makro – *MAXALPHA* definujúce veľkosť abecedy používanej tabuľkou d1. Momentálna hodnota makra je 256, čo pokrýva všetky 8-bitové znaky a v spojení so znakovým kódovaním UTF-8 aj všetky možné znaky. Jediná predpokladaná zmena by bola spojená s úpravou algoritmu pre kódovanie založené na znakoch inej dĺžky, ako napríklad UTF-16 alebo staršie kódovania ázijských znakov. Takáto úprava by si však vyžiadala aj zmenu dátových typov pre znakové premenné a smerníky (napr. z char na w_char) vo všetkých funkciách v boymo.c.

3.1.4 Špecifické časti kódu

Na premenné s úlohou binárneho prepínača, ktoré sa vyskytujú vo väčšom množstve, bola použitá štruktúra s bitovými poliami. Bitové polia majú malú nevýhodu pretože štandard jazyka C nešpecifikuje ich presné poradie v štruktúre a rozhodnutie je ponechaná na danú implementáciu. Preto do štruktúry bitových polí nemožno spoľahlivo zapisovať alebo z nej čítať priamo – len do jednotlivých polí. Táto nevýhoda však nemá význam pre tento program a na oplátku značne sprehľadňuje nastavovanie jednotlivých premenných v porovnaní s alternatívou – bitové operácie na jednej premennej.

Porovnávanie načítaných polí s požadovanými hodnotami je v kóde riešené pomocou viacerých rozhodovacích štruktúr *if* za sebou (nie vnorených). Kvôli šetreniu času porovnávanie skončí pri prvom nesprávnom porovnaní. Toto je implementované pomocou príkazu *goto*, ktorý preskočí na vyznačené miesto na konci slučky, kde ešte pred opakovaním slučky treba uvoľniť alokovanú pamäť v niektorých smerníkoch. Aj keď v odbore existuje istá (oprávnená) nevraživosť proti príkazu goto, toto použitie je jedno z mála prípustných. Alternatívou by bola duplikácia kódu v každej rozhodovacej štruktúre alebo pridanie ďalších premenných a rozhodovacích štruktúr na dosiahnutie rovnakého výsledku. To by malo za následok menšiu prehľadnosť kódu a sťažilo by prípadné zmeny, pretože by sa museli realizovať na viacerých miestach v kóde.

Konvencia pri definícií premenných vo funkcii main() je definovať číselné a znakové premenné ako automatické premenné zásobníka a namiesto polí používať smerníky a alokáciu pamäte na halde. Dôvodom je hlavne obmedzená veľkosť zásobníka. Keďže veľkosť znakových reťazcov je definovaná pomocou makier z hlavičkových súborov a teda konfigurácia používateľom je prípustná, pre isté hodnoty by statické polia mohli presiahnuť veľkosť zásobníku a znefunkčniť program.

3.2 Práca s programom

Program vyžaduje vstup vo formáte CSV, buď ako samostatný súbor alebo vie čítať zo štandardného vstupu. Toto umožňuje jednoduchú integráciu s inými nástrojmi a postupmi. Logy systému Windows môže používateľ jednoducho vyexportovať z modulu *Event Viewer* dostupného v Správe počítača. V časti *Windows logs, System* stačí uložiť všetky záznamy do CSV formátu.

Program umožňuje takto získaný súbor filtrovať pomocou niekoľkých kategórií. Dostupné je filtrovanie podľa úrovne záznamu (informácia, upozornenie, chyba), zdroja, ktorý záznam vygeneroval, číselného typu záznamu, čísla kategórie úlohy a rozsahu dátumov a časov. Ďalej je možné textové vyhľadávanie v popise záznamu.

Parametre programu sa dajú nastaviť dvoma spôsobmi. Pre použitie v dávkovom móde je možné použiť argumenty z príkazového riadku. Napríklad pre vyhľadanie výrazu „Bluetooth“ vo všetkých záznamoch od 5. 7. 2020 z vyexportovaného súboru logy.csv by sme použili nasledujúci príkaz: *logfind.exe -B -m Bluetooth -d "5. 1. 2020 00:00:00" logy.csv*

Druhá možnosť je použitie interaktívneho menu, ktoré je východzím správaním programu. Program potom vypíše momentálnu konfiguráciu a umožní užívateľovi meniť jednotlivé položky podľa ich čísel. Argumenty z príkazového riadku sú stále spracovávané a v menu sa použijú ako východzie hodnoty. Príklad interaktívneho menu:

```
> logfind.exe logy.csv

============================================================
L O G F I N D

--- Input and Output ---
1 Input File: logy.csv
2 Output File:
--- Filters ---
3 Level:
4 Source:
5 Event ID: 0
6 Task Category: 0
7 Date Range:  -
8 Description:
--- Action ---
9 Go!
0 Exit
---
=> _
```

Na zmenu ľubovoľného nastavenia potom stačí zadať jeho číslo a stlačiť klávesu Enter. Program sa opýta na novú hodnotu poľa, ktorá sa rovnako aplikuje po stlačení Enteru. Príklad (užívaľov vstup je uvedený *kurzívou*):

```
...
=> 2
```

Specify Output File: *out.txt*

==========

L O G F I N D

--- Input and Output ---

1 Input File: logy.csv

2 Output File: out.txt

...

Akokoľvek už užívateľ nastaví program, po spustení prejde vstupným súborom a vypíše v ľudsky zrozumiteľnom formáte zhodujúce sa záznamy. Príklad výstupného formátu:

LEVEL: I

DATE: 6. 1. 2020 20:06:01

SOURCE: Microsoft-Windows-Kernel-Boot

EVENT: 32

TASK: 58

DESCRIPTION:

The bootmgr spent 0 ms waiting for user input.

3.3 Ďalšia práca a vylepšenia

Žiaden program nie je bezchybný, a je tomu tak aj v tomto prípade. Napriek snahe existujú stále mnohé vylepšenia, ktoré by na programe mohli byť realizované v budúcnosti.

Kód by mal prejsť úpravou a upratovaním, aby sa zosúladil systém argumentov a návratových hodnôt a volania funkcií – pravdepodobne podľa modelu funkcie getdesc(). Dynamická alokácia pamäte by tiež mala prejsť revíziou, aby sa zjednotili konvencie pri jej alokácií. Vhodnými optimalizáciami bude možné znížiť pamäťové nároky programu. Naivný algoritmus generovania d2 tabuľky je možné vymeniť za efektívnejší Knuthov algoritmus s Rytterovou opravou. Interaktívne menu je veľmi rudimentárne a dá sa vylepšiť, napríklad o zrušenie nastaveného filtra poľa.

Čo sa týka funkcionality, program by mohol ponúkať výstup do viacerých formátov. Vyhľadávanie a filtrovanie by mohlo akceptovať viac ako jednu hodnotu pre každé pole. Je možné pridať jednoduché grafické rozhranie – pravdepodobne za pomoci Tk.

Záver

V práci boli predstavené teoretické základy textového vyhľadávania. Boli popísané rôzne druhy vstupných formátov, možnosti reprezentácie hľadaného reťazca a predstavené vyhľadávacie algoritmy na rozličné použitie.

Praktickým výstupom práce je program na hľadanie záznamov v systémovom logovacom súbore operačného systému Windows. Opísaná bola implementácia programu a spôsob jeho použitia. Nakoniec boli popísané ďalšie možnosti vylepšenia programu, ktoré môžu byť realizované v budúcnosti.

Zoznam použitej literatúry

BOYER, Robert S. – MOORE, J Strother. A Fast String Searching Algorithm. In Communications of the ACM. Association for Computing Machinery, Inc., 1977, October, Vol. 20, No. 10, s. 762-772.

THOMPSON, Ken. Programming Techniques: Regular Expression Search Algorithm. In Communications of the ACM. Association for Computing Machinery, Inc., 1968, June, Vol. 11, No. 6, s. 419-422.

KNUTH, Donald E. – MORRIS JR., James H. – PRATT, Vaughan R. Fast Pattern Matching in Strings. In SIAM Journal on Computing. Society for Industrial and Applied Mathematics, 1977, June, Vol. 6, No. 2, s. 323-349.

RYTTER, Wojciech. A Correct Preprocessing Algorithm for Boyer-Moore String-Searching. In SIAM Journal on Computing. Society for Industrial and Applied Mathematics, 1980, August, Vol. 9, No. 3, s. 509-512.

KARP, Richard M. – RABIN, Michael O. Efficient randomized pattern-matching algorithms. In IBM Journal of Research and Development. International Business Machines Corporation, 1987, March, Vol. 31, No. 2, s. 249-260.

RABIN, Michael O. Fingerprinting by Random Polynomials. In Technical Report TR-15-81. 1981. 12 s.

KERNIGHAN, Brian W. UNIX: A History and a Memoir. Kindle Direct Publishing, 2020. ISBN 978-169597855-3

RFC 3164: 2001: The BSD syslog Protocol.

RFC 5424: 2009: The Syslog Protocol.

RFC 4180: 2005: Common Format and MIME Type for Comma-Separated Values (CSV) Files.

RFC 7111: 2014: URI Fragment Identifiers for the text/csv Media Type.

XML 1.0: 2008: Extensible Markup Language (XML) 1.0 (Fifth Edition)