Razvoj web aplikacije za otkrivanje plagijata u programskom kodu

Legović, Mateo

Sveučilište Jurja Dobrile

Fakultet informatike u Puli

Mateo Legović

Razvoj web aplikacije za otkrivanje plagijata u programskom kodu

Završni rad

Pula, rujan, 2024.

Sveučilište Jurja Dobrile

Fakultet informatike u Puli

Mateo Legović

Razvoj web aplikacije za otkrivanje plagijata u programskom kodu

Završni rad

JMBAG: 0303095056

Redovni student

Studijski smjer: Informatika

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Kolegij: Strukture podataka i algoritmi

Mentor: izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan, 2024.

Sažetak

Neprestanom inovacijom novih tehnologija poput umjetne inteligencije, kao jedna od posljedica toga je plagijarizam. Bilo to svjesno ili nesvjesno pogotovo u obrazovnim ustanovama, plagrizam će uvijek postojati. Ovaj rad je jedan korak naprijed pri rješavanju takvom problema. Rad se fokusira na razvoju web aplikacije za detekciju plagijata u programskom kodu, specifično u C++ jeziku. Temelji se na RKR-GST algoritmu i tokenizaciji. Aplikacija pruža preciznu usporedbu kodova te identificira potencijalne plagijate. Ovaj rad pruža alternativno rješenje postojećim alatima integrirajući njihove najbolje značajke s vlastitim prilagodbama. Budući razvoj aplikacije mogao bi uključivati naprednije metode poput umjetne inteligencije, čime bi se dodatno unaprijedila preciznost i efikasnost otkrivanja plagijata.

Ključne riječi

RKR-GST algoritam, Tokenizacija, Plagijarizam, Detekcija plagijata, Usporedba programskog koda, Web aplikacija

Abstract

With the constant innovation of new technologies such as artificial intelligence, plagiarism has become an increasing concern. Whether committed consciously or unconsciously, plagiarism is especially prevalent in educational institutions and will continue to persist. This bachelor's thesis takes a step forward in addressing this issue. It focuses on the development of a web application for detecting plagiarism in programming code, specifically in the C++ language. It is based on the RKR-GST algorithm and tokenization techniques. This bachelor's thesis offers precise code comparison and identifies potential instances of plagiarism. It presents an alternative solution to existing tools by integrating their best features with customizations. Future development could involve incorporating more advanced methods, such as artificial intelligence, which would further enhance the accuracy and efficiency of plagiarism detection.

Keywords

RKR-GST algorithm, Tokenization, Plagiarism, Plagiarism detection, Code comparison, Web application

1. Uvod

U današnje vrijeme u kojem se razne tehnologije brzo razvijaju, tako i nedavno umjetna inteligencija, dostupno je sve više sadržaja na internetu. U taj sadržaj spadaju i programska rješenja koja su intelektualno vlasništvo neke osobe. Takvi se isječci koda sve češće kopiraju. Često to može biti i nesvjesno, budući da se razna rješenja nalaze na GitHubu ili nekom forumu, a i rješenja koja UI generira se također mogu smatrati tuđim intelektualnim vlasništvom.

Navedeni problem svjesnog ili nesvjesnog plagijarizma izuzetno se ističe u obrazovnim ustanovama, gdje studenti ili učenici koriste tuđi kod kao svoje vlasništvo. Takvim se ponašanjem studenata ruši akademski ili profesionalni integritet u svijetu programiranja. Konkretno institucije i profesori imaju najveći problem s tim jer je njihova zadaća spriječiti ili u krajnjem slučaju detektirati radi li se o plagrijarizmu. Ručna detekcija plagijata je vremenski iscrpljujuća ako se radi o većem broju programskih zadataka ili projekata. Naravno, za to postoje adekvatna programska rješenja kojima se zapravo i koriste mnoge institucije. Neka od tih rješenja su MOSS, JPlag, Turnitin itd. Svaki od tih alata u suštini funkcionira na slične algoritme. Svo troje od gore navedenih alata koristi tokenizaciju, heshiranje, i neki algoritam za usporedbu tokena ili hash vrijednosti. Navedene se aplikacije razlikuju te svaka ima svoje prednosti i mane. Konkretno Turnitin i Copyleaks su moderniji alati, koji koriste umjetnu inteligenciju kao proširenje uz algoritam usporedbe tokeniziranog koda. Oni također pretražuju i potencijalne plagijate na internetu dok MOSS i JPlag, koji su stariji alati, uspoređuju samo ulazne programske kodove. Iako su zastarjeli alati, i danas se zbog svoje funkcionalnosti i dalje koriste. JPlag je najsličniji programskom rješenju ovog završnog rada jer je baziran na njemu i MOSS-u. Navedeno programsko rješenje koristi proces tokenizacije u kojemu su definirana naprednija pravila. Jedan token je jedna ključna riječ koda, varijabla, identifikator ili naziv metode. Nakon generiranih tokena, oni se šalju u RKR-GST algoritam kojemu je cilj pronalaženje najdužeg zajedničkog podstringa. On može primiti više dokumenata kao input, te će se tokenizirana verzija koda usporediti tako se svaki usporedio sa svakim. Na korisničkoj strani aplikacije prikazan je graf podudaranja koda gdje je 100% najveća razina podudaranja. Moguće je i pregledati uspoređeni kod jedan pored drugog i vidjeti označene dijelove plagijariziranog koda.

Ova aplikacija je alternativno, adekvatno i moderno rješenje detekcije plagijata među dokumenta. Korištenjem takvih alata je jedan korak dalje u očuvanju integriteta te olakšava profesorima nadzor nad plagijatima u svijetu gdje je tuđi kod jako lako raspoloživ.

2. Kritička analiza trenutačnih rješenja

2.1. MOSS (Measure of Software Similarity)

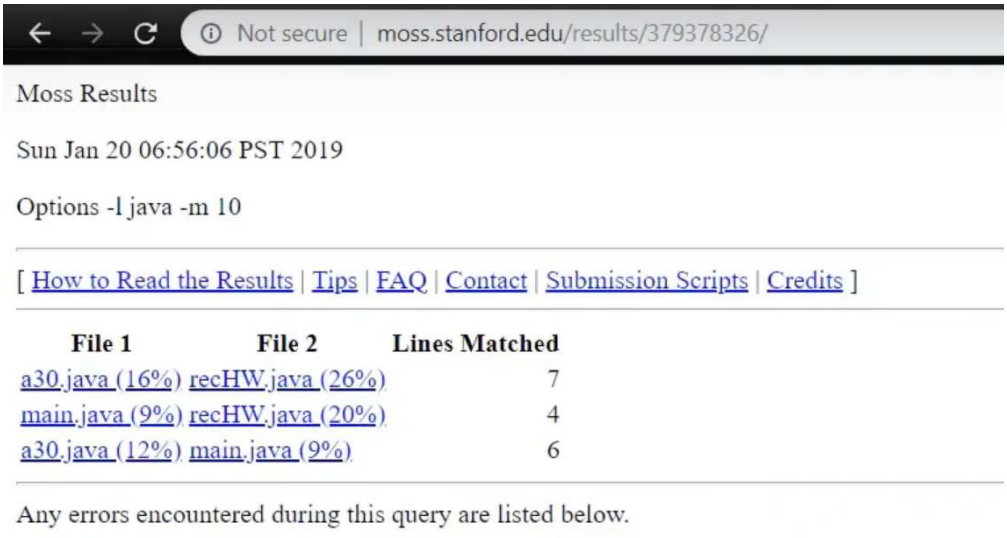
MOSS je alat koji se prvenstveno koristi za prepoznavanje sličnosti u kodu, s ciljem otkrivanja potencijalnog plagijata u akademskim okruženjima. Razvio ga je profesor Saul Schleimer na Stanford Sveučilištu, a danas se koristi širom svijeta za automatsku provjeru studentskih zadaća u programiranju.

Prvi korak u MOSS algoritmu je normalizacija koda. Algoritam mora prepoznati, te onda ignorirati beskorisnu sintaksu kao što je razmak. Također preimenovanje varijabla ne bi trebao biti problem. Brisanje komentara također spada u normalizaciju. Zatim se od procesiranog dokumenta generira sekvenca k-gram hash vrijednosti. Odabir k ovisi o specifičnoj tehnici. U trećem koraku odabire pod set hash vrijednosti koje se koriste kao otisak ili „fingerprint“ dokumenta. Oni otisci koji su odabrani u svakom dokumentu utjecati će na broj otkrivenih podudaranja. MOSS koristi algoritam koji se zove “winnowing”. Na visokoj razini, algoritam s veličinom prozora ili „windowa“ w klizi po popisu hash vrijednosti. U svakom koraku algoritam bilježi minimalnu vrijednost u prozoru (ako već nije zabilježena). Kada prozor dođe do kraja niza hash-ova, snimljeni skup se uzima kao otisak prsta dokumenta. U praksi MOSS koristi bolji „winnoving“ algoritam koji odabire manje otisaka. Dvije su glavne komponente MOSS-a. prvo je korisničko sučelje koji je specifično za jezik i priprema svaki dokument uklanjanjem razmaka, identifikatora i filtriranjem ključnih riječi za taj jezik. Dokument se zatim unosi u mehanizam za otiske. Otisci su označeni s pozicijom na izvorni kodu kako bi se korisniku prikazala podudaranja. MOSS uspoređuje svaki dokument sa svima te računa razinu podudaranja.

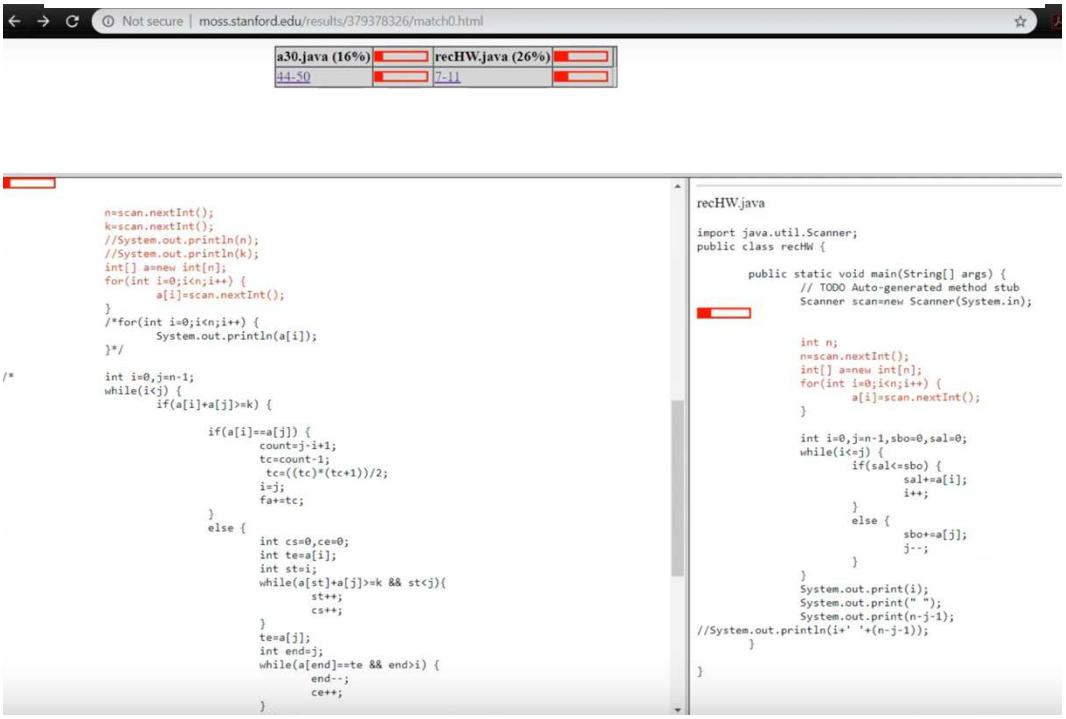
Jedna od mana MOSS-a je ta da nema korisničkog sučelja kao takvog. Korisnik koristi MOSS putem komandne linije gdje se kao ulaz stavljaju dokumenti koje želimo usporediti i parametri kao specifikacija programskog jezika. Nakon završetka, u komandnoj liniji postoji hiperveza koja nas usmjerava do HTML izvješća, kao što je prikazano u slici 1 (Kumar, 2019*)*. U detaljnijem izvješću, prikazani su uspoređeni kodovi jedan pored drugog i bojom su označeni slični dijelovi koda, prikazano u slici 2 (Kumar, 2019). MOSS je učinkovit u akademskim okruženjima, gdje se najviše i koristi. Zbog same metoda otiska koju MOSS koristi, mogu se javiti problemi kod naprednijeg restrukturiranja logičkih blokova ili kod razdvajanja izjava (Schleimer, Wilkerson, & Aiken, 2003).

Algoritam koji će kasnije u završnom radu biti detaljnije opisan također se sastoji od normalizacije i tokenizacije. Kod se normalizira na način da se brišu svi komentari, zajednički

deklarirane varijable se razdvajaju i pozicionirane su na samom početku funkcije u kojoj su deklarirane. Procesirani se kod zatim tokenizira prema preodređenim pravilima koja su pažljivo odabrana kako bi se suština koda pravilno tokenizirala (Đurić & Gašević, 2012).



Slika 1. Prikaz MOSS HTML izvješća (Kumar, 2019)

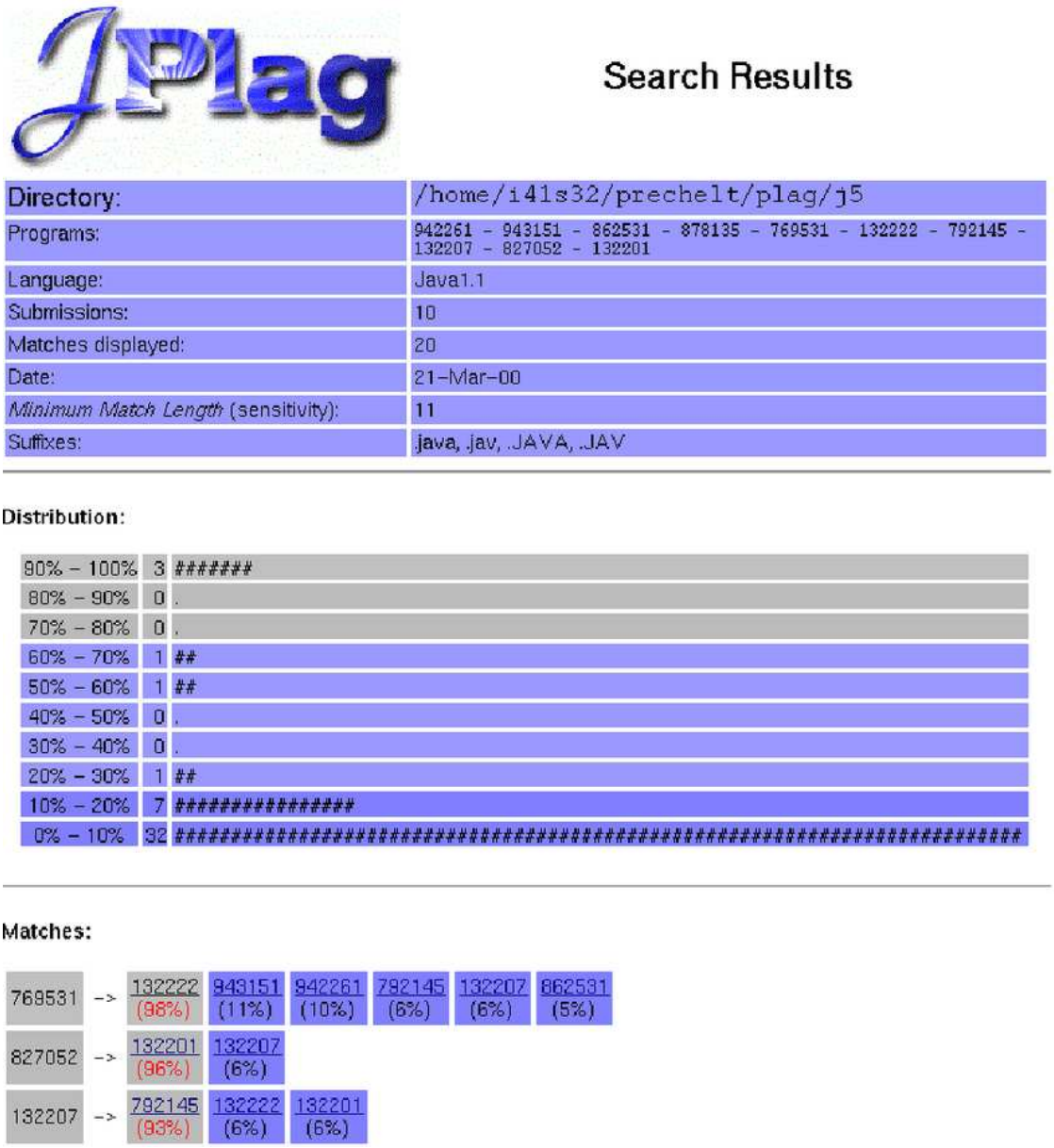


Slika 2. Prikaz podudarajućeg koda (Kumar, 2019)

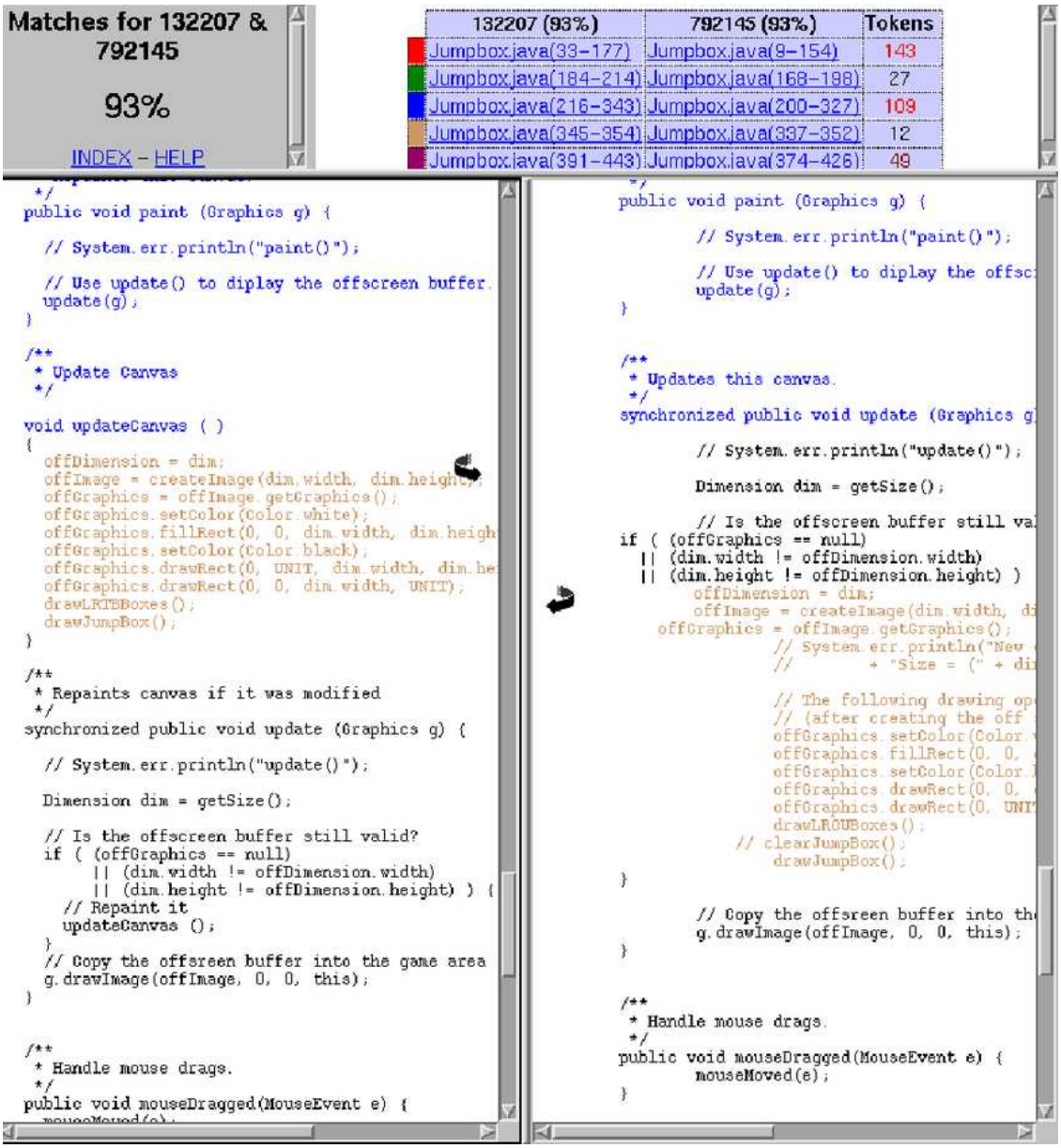
2.2. JPlag

JPlag je razvio Guido Malpohl s Tehničkog sveučilišta u Berlinu, s ciljem otkrivanja sličnosti u programskom kodu, a posebno za identifikaciju plagijata u studentskim zadaćama. JPlag koristi Greedy String Tiling (GST) algoritam za usporedbu više input kodova. Glavni cilj GST-a je pronalaženje najdužeg zajedničkog pod stringa između dva seta tokena. Proces tokenizacije omogućuje da se programski kod prikaže kao neki predodređeni znak ili token. Ali je prije tog procesa potrebna je normalizacija koda, što podrazumijeva brisanje komentara, razmaka, imena varijabla itd. Primjenjuje se GST algoritam koji traži maksimalni pod string koji se podudara između dva seta tokena. Nakon pronalaženja istih pod stringova (tokena), tada se oni moraju označiti i ne mogu više biti dio nekakvog drugog pod stringa, proces se ponavlja dok odgovarajući rezultati ne dođu do određene minimalne vrijednost podudaranja. Kompleksnost ovog algoritam, u najgorem slučaju je $O(n^3)$. Kako bi se smanjila složenost i optimizirao algoritam, uvedeno je heshiranje. To je pred provjera koja pomaže pri filtriranju potencijalnih podudaranja prije GST algoritma. Međutim, nakon heshiranja. JPlag se još uvijek oslanja na tradicionalno pronalaženje podudarajućih pod stringova kao što je i prije opisano. S ovim izmjenama, složenost u najgorem slučaju je još uvijek $O(n^3)$, budući da se svi pod nizovi moraju usporediti token po token. U praksi se obično opaža složenost manja od $O(n^2)$. JPlag može procesirati inpute na više programskih jezika (Java, C, C++, Scheme), što ga čini adekvatnim za akademsku upotrebu. Njegov prilagođeni proces tokenizacije omogućuje točnost detekcije za sve navedene programske jezike (Prechelt, Malpohl, & Philippsen, 2002).

JPlag se može pristupiti preko weba, bez komandne linije. Nakon unosa koda kojeg želimo usporediti, JPlag prikazuje općenitiji prikaz gdje su postotci, dokumenti koji se podudaraju i ostale informacije o unosu što je i moguće vidjeti na slici 3 (Prechelt, Malpohl, & Philippsen, 2002). U detaljnijem prikazu prikazani su i bojom označeni programski kodovi koji se podudaraju s izraženim postotcima, prikazano u slici 4 (Prechelt, Malpohl, & Philippsen, 2002). JPlag je široko korišten i dobro obavlja svoju funkciju, no modernizacijom korisničkog sučelja i implementacijom RKR-GST algoritma, umjesto samo modificiranog GST-a s heširanjem, može se postići bolja optimizacija algoritma i poboljšati izgled web stranice. RKR-GST algoritam (Wise, 1993) primijenjen je u aplikaciji ovog završnog rada. Integracijom heshiranja postiže se bolja optimalnost. JPlag je najsličniji po algoritmu i funkcionalnosti ali se razlikuju po naprednijoj tokenizaciji, optimiziranijim algoritmom i po naprednijem korisničkom sučelju.



Slika 3. JPlag općeniti prikaz (Prechelt, Malpohl, & Philippsen, 2002)



Slika 4. Prikaz podudarajućeg koda (Prechelt, Malpohl, & Philippsen, 2002)

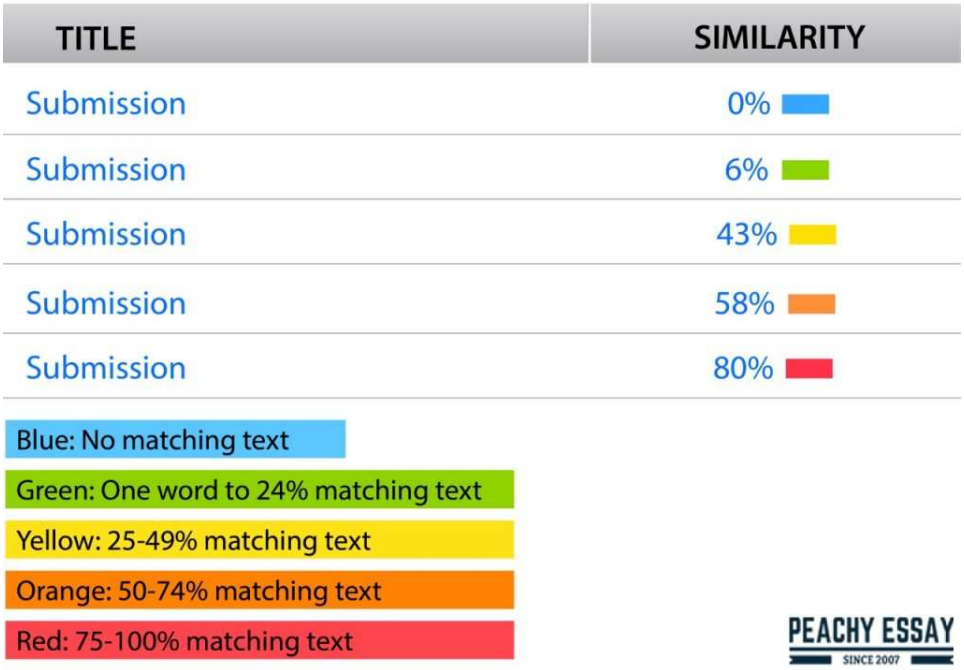
2.3. Turnitin

Turnitin je alat za detekciju plagijata koji se koristi u obrazovnim institucijama širom svijeta. Razvijen s ciljem očuvanja akademske čestitosti, Turnitin omogućuje profesorima i studentima provjeru originalnosti pisanih radova usporedbom s velikom bazom podataka.

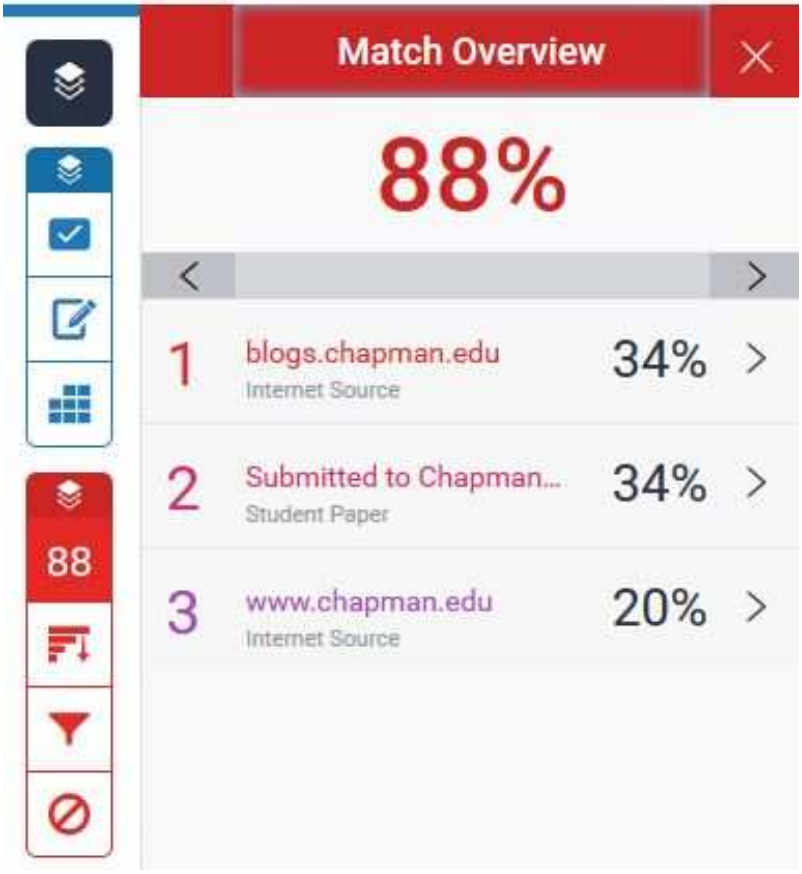
Turnitin, osim detekcije tekstualnog plagijarizma, ima i adaptaciju za detekciju plagijarizma u programskom kodu. Njegov algoritam se također koristi tokenizacijom i preprocesiranjem koda, tj. transformacijom koda u značajne dijelove, bez komentara i razmaka. Tokeni predstavljaju elemente koda kao što su imena varijabla, ključne riječi , operatori i kontrolne strukture. Cilj tokenizacije je izlučivanje esencijalne logike i strukture koda, ali tako da promjena samog koda , kao preimenovanje varijabla ili promjena redoslijeda koda, ne utječe

na funkcionalnost detekcije istog koda. Sljedeći korak je dodjeljivanje jedinstvenih digitalnih identifikatora ili „fingerprinta“ određenoj sekvenci tokena (k-grams). Grupa uzastopnih tokena (k-grams) je heshirana zajedno kako bi ta hash vrijednost predstavljala digitalni „fingerprint“. Turnitin koristi spomenute „k-grams“, gdje se uzima broj tokena opisan vrijednosti k, pa se uzastopno u listi tokena selektirani dio liste opisan s k, klizi na sljedeći niz tokena. Tako se za svaki podstring duljine k računa „fingerprint“. Ova tehnika omogućuje Turnitinu otkrivanje sličnosti u strukturi koda, čak i kada su napravljene manje promjene nad pojedinačnim tokenima. Svi programski kodovi ili tekstualni zapisi koji su uneseni u Turnitin, spremaju se u repozitorij, što omogućuje usporedbu s prijašnjim i sadašnjim podatcima. Turnitin ima pristup
Podatcima od „web-crawler-a“, što uključuje javno dostupne repozitorije koda. Također,
dostupni su mu mnogi akademski ili znanstveni radovi koji mogu biti sastavljeni od nekih dijelova koda. Turnitin prikazuje razinu podudaranja za tekstualni ili kodni input koja se računa usporedbom s svim mogućim izvorima spomenutima prije. U suštini , razina podudaranja predstavlja postotak usporedbe teksta ili koda s ostalim dostupnim izvorima. Kao što je i prikazano na slici 5 (Peachy Essay, 2021), različite boje (plava, zelena, žuta, narančasta i crvena) koriste se za označavanje razine sličnosti, pri čemu crvena predstavlja najveći postotak podudaranja (75-100%). U Turnitinu postoji općeniti prikaz podudaranja koji je prikazan na slici 6 (Chapman University, 2017). Uspoređivanje manjih dijelova koda može prouzročiti lažno pozitivne rezultate, pogotovo u slučajevima kada su petlje ili klasični algoritmi više puta upotrebljeni . Turnitin je komercijalan alat, a cijena može varirati za manje institucije ili za osobnu upotrebu. Također, adaptacija za detekciju plagijarizma u programskom kodu nije uključena u svim paketima (Turnitin, 2024). Turnitin, kao i MOSS, koristi “fingerprinting ”, što je solidan pristup detekcije plagijata u razini s RKR-GST algoritmom.

Iako Turnitin ne koristi napredniju tokenizaciju, nadoknađuje to sa širim pretraživanjem mogućih plagijata, za razliku od aplikacije bazirane na RKR-GST algoritmu koji samo uspoređuje dokumente međusobno. Također , Turnitin nije besplatan i potrebna je registracija i prijava korisnika, što je suprotno od aplikacije ovog završnog rada.



Slika 5. Turnitin prikaz podudaranja u bojama
(Peachy Essay, 2021)



Slika 6. Općeniti prikaz podudaranja
(Chapman University, 2017)

2.4. Copyleaks

Copyleaks je napredan alat za detekciju plagijata koji radi slično kao Turnitin, ali za razliku od Turnitina ima mogućnost prepoznavanja parafraziranog koda te koristi strojno učenje za unaprjeđivanje detekcije.

I on koristi tokenizaciju u koju spada normalizacija koda (brisanje komentara i razmaka), a ključne riječi, operatori i kontrolne strukture pretvaraju se u tokene. Nakon tokenizacije generiraju se digitalni otisci prsta ili „fingerprints“ koristeći n-grame (uzastopne sekvence tokena). Ti otisci su zapravo jedinstvene hash vrijednosti koje se koriste za usporedbu s ostalim „otiscima prsta“ koji su uneseni kao input, ili s javno dostupnim kodovima na akademskim repozitorijima, prijašnje unesenih studentskih kodova i ostalim web izvorima. Copyleaks koristi kombinaciju detekcije točnih podudaranja između „fingerprint-a“ i pomoću „function-level“ parafraziranja koje pronalazi gdje je promijenjena ili parafrazirana logika koda ali bez gubitka funkcionalnosti. Umjetna inteligencija igra ključnu ulogu u detekciji parafraziranog koda. Algoritmi strojnog učenja trenirani su da prepoznaju uzorke po kojima studenti pokušavaju prikriti kopirani kod npr. promjenom varijabla ili restrukturiranjem koda. Nakon detekcije parafraziranog koda, označuje parafrazirani dio žutom bojom od egzaktnih podudaranja na temelju „fingerprinta“ koja je označena crvenom, što je i vidljivo na slici 7 (Copyleaks, 2020). Copyleaks je efektivan alat, ali lažno pozitivni rezultati su i dalje mogući.

Kao i prijašnje spomenuti Turnitin, Copyleaks je također komercijalni alat koji dolazi sa svojom cijenom, što bi moglo predstaviti problem dostupnosti za osobne svrhe ili institucije (Copyleaks, n.d.). Boljom tokenizacijom koja je primijenjena u aplikaciji ovog završnog rada, postignut je sličan rezultat koji je Copyleaks postigao umjetnom inteligencijom. Na slici 7 (Copyleaks, n.d.), žuto označen bojom je označeni parafrazirani dio. Tokenizacija aplikacije ovog završnog rada nam omogućuje da preimenovanje varijabla ili mijenjanje redoslijeda koda ne utječe na efikasnost prepoznavanja plagijata.

Slika 7. Parafrazirani i identični kod
(Copyleaks, 2020)

3. Korištene tehnologije

Prilikom izrade web aplikacije korištene su razne tehnologije. Za pohranjivanje i praćenje razvoja koda korišten je GitHub, dok je za samo pisanje koda korišten Visual Studio Code. Korisničko sučelje aplikacije je izgrađeno uz pomoć Vue.js-a, dok se za backend koristio Node.js i Express.js. MongoDB je u ovoj web aplikaciji korišten kao baza podatka

3.1. Vue.js

Vue.js je moderna JavaScript biblioteka koja omogućuje brzo i jednostavno stvaranje korisničkih sučelja za web stranice ili aplikacije. Biblioteka se fokusira na deklarativni pristup u izgradnji korisničkog sučelja, što znači da se naglasak stavlja na opisivanje onoga što se treba prikazati, a ne na specifikaciju kako se to treba učiniti. U Vue.js-u promjene na korisničkom sučelju automatski ažuriraju na temelju promjena na podacima koje predstavljaju. Namijenjen je početnicima i iskusnim programerima koji žele kreirati responzivne i interaktivne web aplikacije (Vue.js, n.d.). Primjer Vue.js koda prikazan je na slici 8.

```
ur > {} "ComparisonView.vue" > template > div.comparison-container > div > div.comparison-item > div.com
  1   <template>
  2     <div class="comparison-container">
  3       <div v-if="filteredComparisons.length > 0">
  4         <div
  5           v-for="(result, index) in filteredComparisons"
  6           :key="index"
  7           class="comparison-item"
  8         >
  9           <h4>Comparison {{ index + 1 }}</h4>
 10           <h4>Similarity Score: {{ result.similarityScore }}</h4>
 11
 12           <div class="comparison-container">
 13             <div class="code-section">
 14               <p v-if="result[`text${index + 1}Filename`]">
 15                 {{ result[`text${index + 1}Filename`] }}
 16               </p>
 17               <h5>Text File Processed Code:</h5>
 18
 19               <pre
 20                 class="code-box"
 21                 v-if="textProcessedCodes[index]"
 22                 v-html="
 23                   getFinalHighlightedCode(
 24                     textProcessedCodes[index],
 25                     textFileMetadata[index],
 26                     result.similarityScore
 27                   )
 28                 "
 29               ></pre>
 30             </div>
 31
 32             <div class="code-section">
 33               <p v-if="result[`pattern${index + 1}Filename`]">
 34                 {{ result[`pattern${index + 1}Filename`] }}
 35               </p>
```

Slika 8. Prikaz Vue.js koda

3.2. Node.js

Node.js je radno okruženje koje omogućuje izvršavanje JavaScript koda na strani servera i kompatibilno je s različitim operativnim sustavima, uključujući Windows, Linux i macOS. Zbog svoje popularnosti, često se koristi za razvoj dinamičkih web stranica i aplikacija. Jedna od ključnih prednosti Node.js-a je njegova arhitektura koja omogućuje da se više zadataka obavlja istovremeno, čak i kada pojedini procesi nisu dovršeni. Ova značajka je posebno važna za servere koji moraju obraditi velik broj zahtjeva odjednom. Zahvaljujući npm-u, Node.js nudi pristup ogromnom broju biblioteka koje olakšavaju razvoj aplikacija. Arhitektura vođena događajima koristi "callback" funkcije, što omogućuje brzu i skalabilnu izradu mrežnih aplikacija. Sve ove značajke čine Node.js jednim od najpopularnijih izbora za izgradnju backend sustava (Kopecky, 2020). Primjer Node.js koda prikazan je u slici 9.

Slika 9. Prikaz Node.js koda

3.3. Flex

Flex je alat koji se koristi za generiranje leksičkih analizatora ili skenera dizajniranih da ulazni tekst razbiju na manje dijelove koji se zovu tokeni. Tokeni mogu biti riječi, brojevi, simboli ili bilo koji drugi elementi teksta. Flexu se definiraju pravila pomoću regularnih izraza koja određuju kako prepoznati određene dijelove teksta, što je i prikazano u slici 10. Na temelju tih pravila, Flex generira C kod koji može prepoznati te dijelove. Primjerice, možete definirati pravilo koje kaže da bilo koja sekvenca brojeva predstavlja broj, ili da niz slova predstavlja identifikator. Najčešće se koristi u razvoju kompajlera, gdje je važno prepoznati različite dijelove programskog koda, ali se može koristiti i za razne druge aplikacije koje uključuju obradu teksta ili podataka (Stanford University, n.d.).

```
%%

[\n]                    { line_num++; column_num = 1; }  // Increment line number on newline

[ \t]+                  { column_num += yyleng; /* Ignore whitespace */ }
"//".*                  { /* Ignore single-line comments */ }
\/\*([^*]|\*+[^*/])*\*\/ { /* Ignore multi-line comments */ }
^#.*  { /* Ignore preprocessor directives */ }

"private"   { column_num += yyleng; /* Ignore 'private' */ }
"protected" { column_num += yyleng; /* Ignore 'protected' */ }
"public"    { column_num += yyleng; /* Ignore 'public' */ }
"internal"  { column_num += yyleng; /* Ignore 'internal' */ }
"static"    { column_num += yyleng; /* Ignore 'static' */ }
"final"     { column_num += yyleng; /* Ignore 'final' */ }
"const"     { column_num += yyleng; /* Ignore 'const' */ }

":" { column_num += yyleng; /* Ignore ':' after access specifiers */ }

"namespace" { tokenStream << token_to_char[NAMESPACE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng;
"class"     { BEGIN(CLASS_DECL); tokenStream << token_to_char[CLASS] << ' ' << line_num << ' ' << column_num << '\n'; column_
"void" {
    tokenStream << token_to_char[VOID] << ' ' << line_num << ' ' << column_num << '\n';
    column_num += yyleng;  // Update column number after "void"

    // Handling any whitespace or newline immediately after "void"
    int c;
    while ((c = yyinput()) != EOF) {
        if (c == '\n') {
            line_num++;
            column_num = 1;  // Reset column number at the start of a new line
        } else if (isspace(c)) {
            column_num++;
        } else {
            unput(c);  // Put back the non-whitespace character
            break;
        }
    }

    BEGIN(FUNC_DECL);  // Switch to function declaration state
    return VOID;
}
```

Slika 10. Prikaz Flex-a

3.4. Express.js

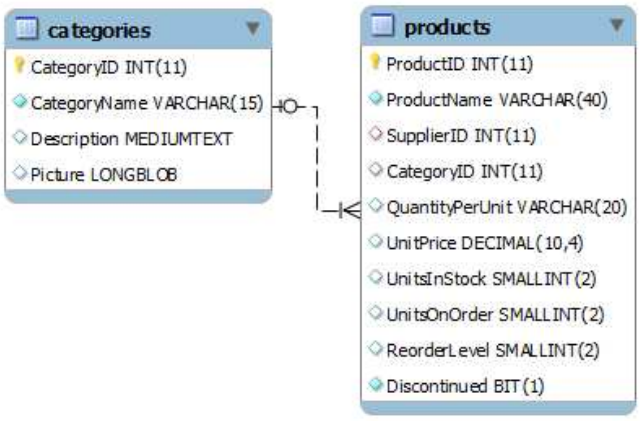
Express.js je popularni framework za Node.js koji se koristi za izradu API-ja i web stranica, posebno u backend razvoju. Slika 11 prikazuje glavnu prednost Express.js-a, on pojednostavljuje procese poput određivanja ruta i postavljanja portova, što omogućuje bržu i lakšu izradu web aplikacija. Express.js omogućuje jednostavno kreiranje HTTP zahtjeva za specifične rute, koje zatim izvršavaju unaprijed definirane operacije. Također, Express.js podržava middleware, što su funkcije koje se izvode prije glavne logike rute. Ove funkcije mogu poslužiti za autentifikaciju, obradu podataka ili zapisivanje logova, a pridonose čišćem i preglednijem kodu. Zbog svoje fleksibilnosti i lakoće upotrebe, Express.js je široko korišten za izradu web aplikacija (Codecademy, n.d.).

```
app.get("/processedCode/:userId", async (req, res) => {
  const { userId } = req.params;
  console.log(`Retrieving all processed code for user ID ${userId}`);

  try {
    const db = await connectToUserDB(userId);
    const filesCollection = db.collection("uploads.files");
    const chunksCollection = db.collection("uploads.chunks");
    const tokensCollection = db.collection("tokens");


    const comparisons = await tokensCollection.find().toArray();

    if (!comparisons || comparisons.length === 0) {
      return res
        .status(404)
        .json({ error: "No comparisons found for the user" });
    }
```

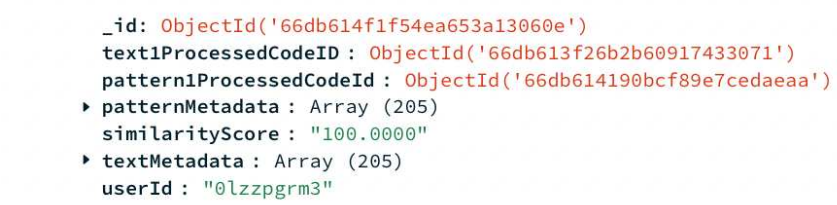
Slika 11. Prikaz Express.js koda

3.5. MongoDB

MongoDB je nerelacijska baza podataka koja ne pohranjuje podatke u tablicama, već koristi kolekcije i dokumente u JSON formatu. Za razliku od relacijskih baza podataka, kao što je prikazano u slici 12 (Zentut, n.d.), gdje tablice imaju međusobne odnose preko primarnih i stranih ključeva, MongoDB koristi nestrukturirane kolekcije prikazane u slici 13, što omogućuje veću fleksibilnost u organizaciji podataka. Jedna od glavnih prednosti MongoDB-a je brzina čitanja i izmjene podataka, budući da podaci nisu međusobno povezani na način kao u relacijskim bazama. MongoDB također podržava indeksiranje podataka, čime se omogućuje brže izvršavanje upita (MongoDB, n.d.).



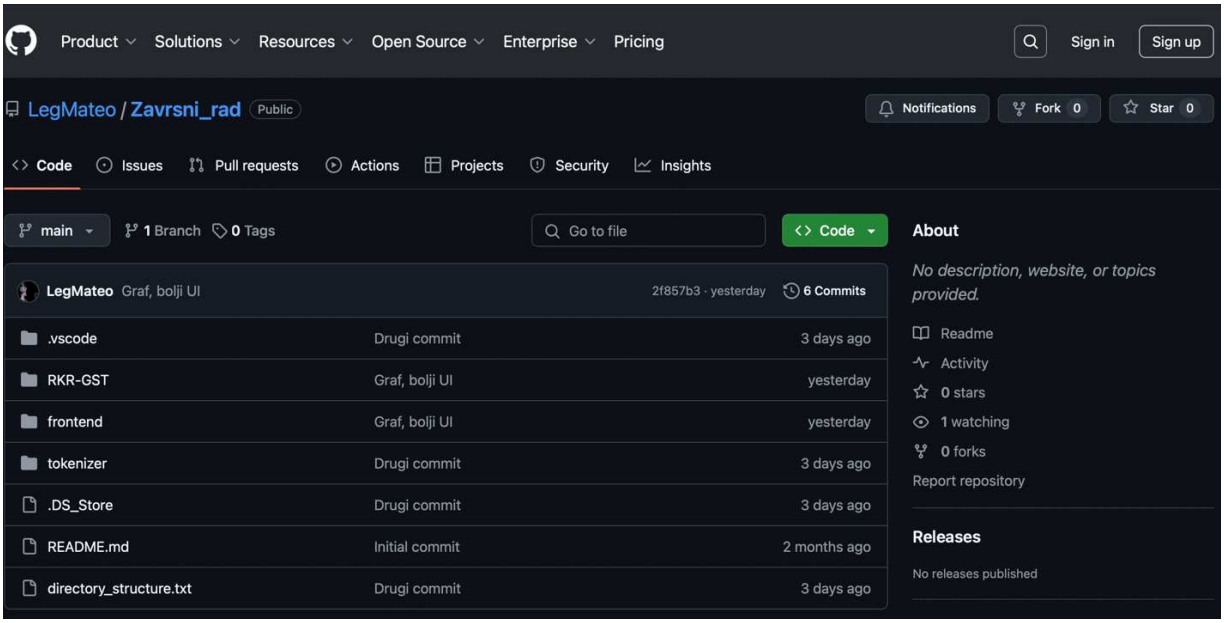
Slika 12. Prikaz relacijske baze podataka (Zentut, n.d.)



Slika 13. Prikaz nerelacijske baze podataka

3.6. GitHub

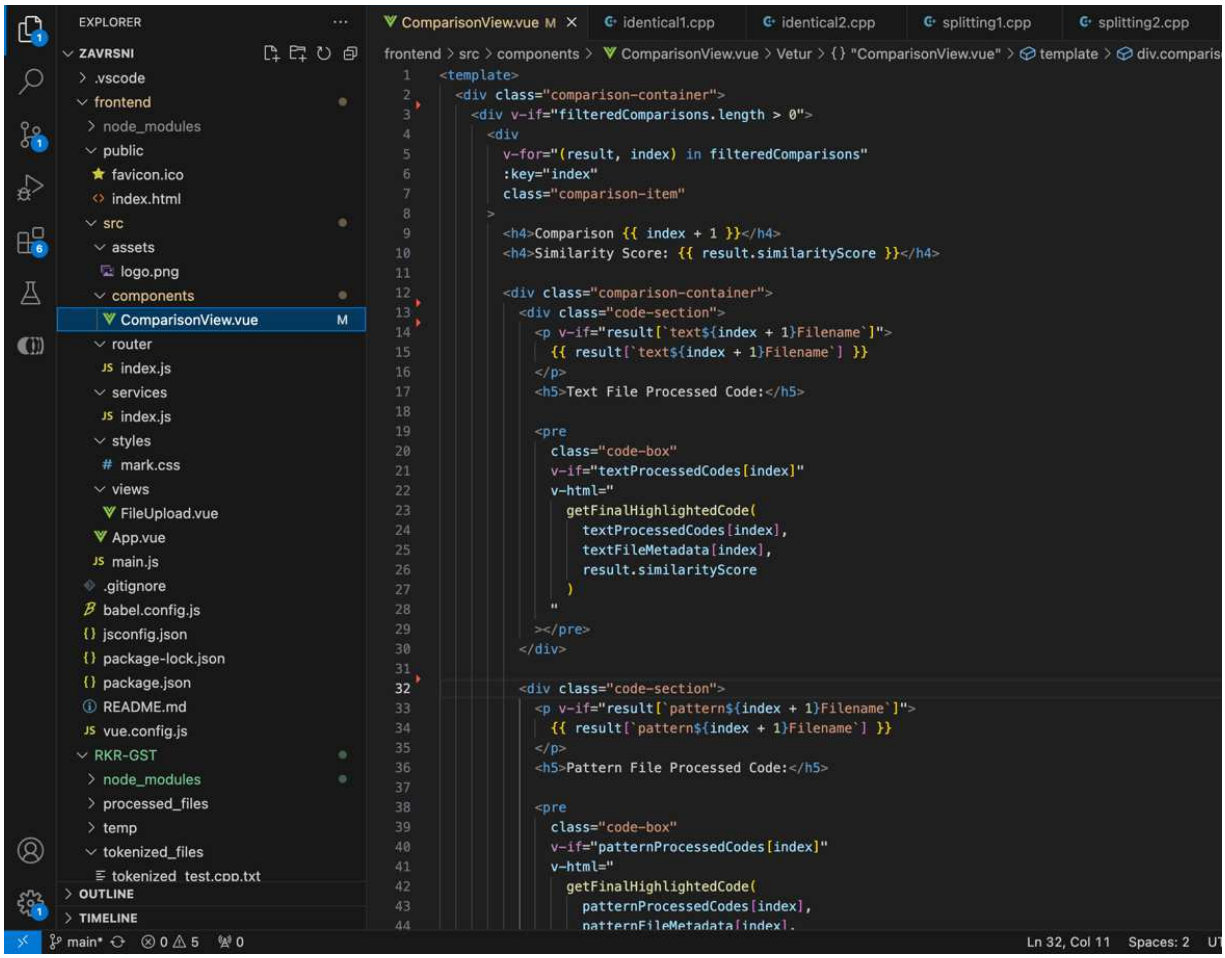
GitHub je web platforma prikazana u slici 14, on omogućuje kontrolu verzija i kolaboraciju na razvoju softvera, prvenstveno putem Git-a, alata za praćenje promjena u kodu. Kontrola verzija omogućava praćenje povijesti izmjena u projektu, što olakšava povratak na starije verzije koda ako dođe do grešaka ili problema. Zahvaljujući GitHub-ovim značajkama kao što su grane i spajanje (merge), više ljudi može istovremeno raditi na različitim dijelovima istog projekta. Ovo je posebno korisno za velike i kompleksne projekte jer omogućuje paralelan rad, preglednost izmjena i lakše rješavanje potencijalnih problema (GitHub, n.d.).



Slika 14. Prikaz GitHub repozitorija

3.7. Visual Studio Code

Visual Studio Code (VS Code), čije je sučelje prikazano na slici 15, je popularan uređivač koda koji podržava razne operativne sustave poput Windowsa, macOS-a i Linuxa. Dolazi s ugrađenom podrškom za više programskih jezika, što ga čini univerzalnim alatom za programere. VS Code omogućava pronalaženje i ispravljanje grešaka u kodu pomoću ugrađenih alata za debuggiranje, a visoko prilagodljivo korisničko sučelje omogućuje korisnicima da prilagode izgled i funkcionalnost prema svojim potrebama. VS Code podržava instalaciju ekstenzija koje olakšavaju i ubrzavaju rad, kao što su alati za formatiranje koda, autokompletiranje i integraciju s raznim razvojnim alatima. Također, nudi integraciju s GitHubom putem ekstenzija, što olakšava kontrolu verzija i spremanje koda izravno na GitHub repozitorije (Visual Studio Code, n.d.).

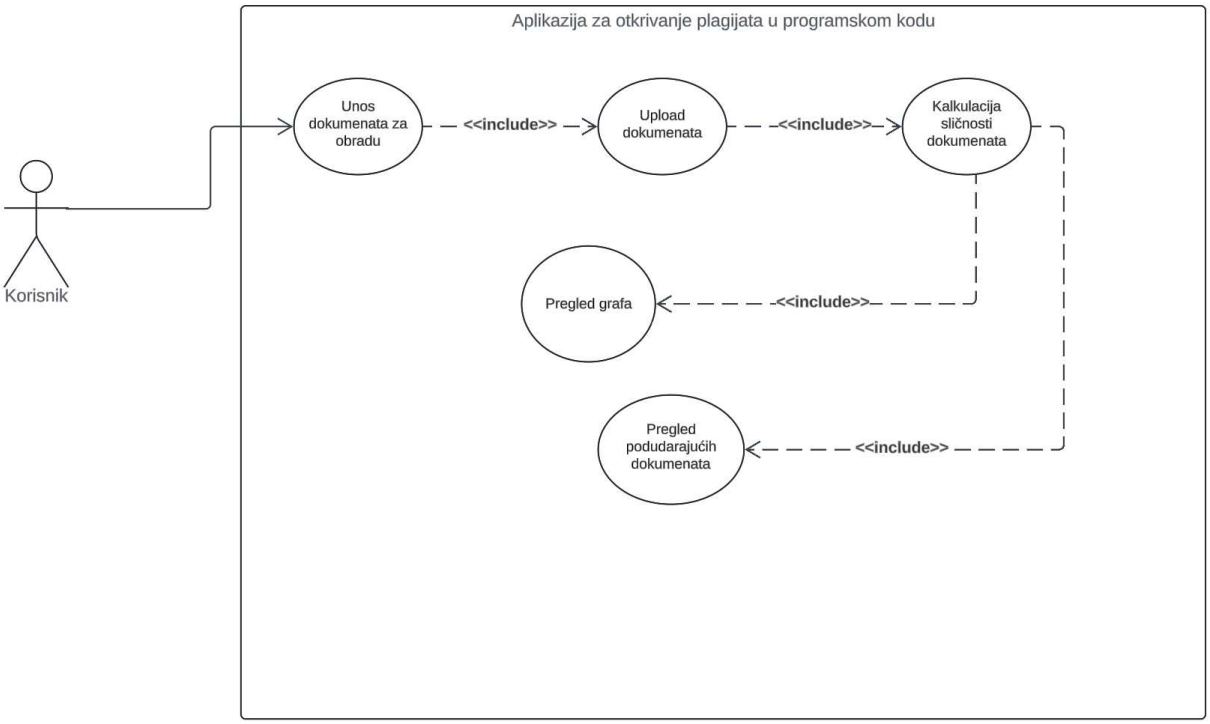


Slika 15. Prikaz VSC sučelja

4. Pregled dijagrama

4.1. Dijagram slučajeva korištenja

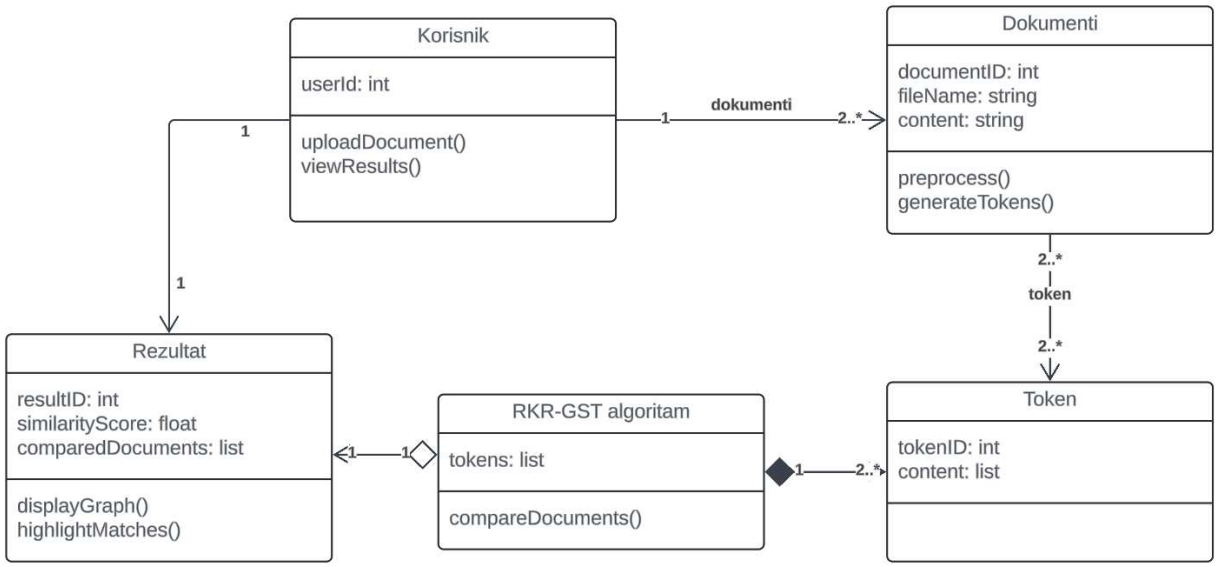
Dijagram slučajeva toka, prikazan na slici 16, predstavlja aplikaciju za detekciju plagijata u programskom kodu. Unutar pravokutnika definirane su sve funkcionalnosti (slučajevi korištenja). Postoji samo jedan akter ili korisnik koji ima interakciju sa sustavom. Korisnik mora unijeti ulazne dokumente kao prvi korak aplikacije, pa je to onda vizualizirano adekvatno vezom. Do kraja ovog dijagrama postoje samo „include“ veze pošto je takva priroda aplikacije te se svaka funkcionalnost dešava sekvencijalno. Nakon unosa podataka, oni se logikom backenda obrađuju te se zatim, prikazanom još jednom „include“ vezom, ti podatcipodaci šalju u kalkulaciju sličnosti. Nakon svih obrađenih procesa, moraju se prikazati vizualizacije kalkulacije koja uključuje graf i pregled podudarajućih dokumenata.



Slika 16. Dijagram slučajeva toka

4.2. Klasni dijagram

Kao što je i prikazano u klasnom dijagramu na slici 17, svaki korisnik ima svoj jedinstveni ID. Klasa Korisnik također sadrži metode uploadDocument() i viewResults(). Jedan korisnik može unijeti 2 ili više dokumenata, drugim riječima, najmanje 2 dokumenta mogu ući u algoritam. Nakon unosa, u klasi Dokumenti koja sadrži ID, ime i sadržaj dokumenata, kroz metode se procesira uneseni kod i generiraju se tokeni. Jedan dokument može generirati 2 ili više setova tokena. Svaki dokument je procesiran u set tokena koji će se kasnije koristiti za usporedbu. To je pokazano s kardinalnošću 2..* na obje strane veze, što znači da su 2 seta tokena generirana za 2 dokumenta. Klasa token sadrži ID i listu tokena za svaki dokument. Potrebna su dva ili više setova tokena da bi RKR-GST algoritam radio. To je predstavljeno vezom 2..* između klase token i RKR-GST algoritma. RKR-GST algoritam sadrži listu tokena iz klase token. Povezani su kompozicijom jer su tokeni potrebni za funkciju algoritma. Algoritam generira 1 rezultat, što je i pokazano vezom 1 na 1, zato jer je cijeli izlaz algoritma jedan rezultat u koji spadaju grafikon, razina podudaranja i pogled na uspoređene kodove. Tablica rezultata povezana je s korisnikom koji može vidjeti rezultate. Jedan korisnik može vidjeti jedan rezultat, jer se svakim unosom dokumenta kreira novi identifikacijski ključ za korisnika.



Slika 17. Klasni dijagram

5. Backend

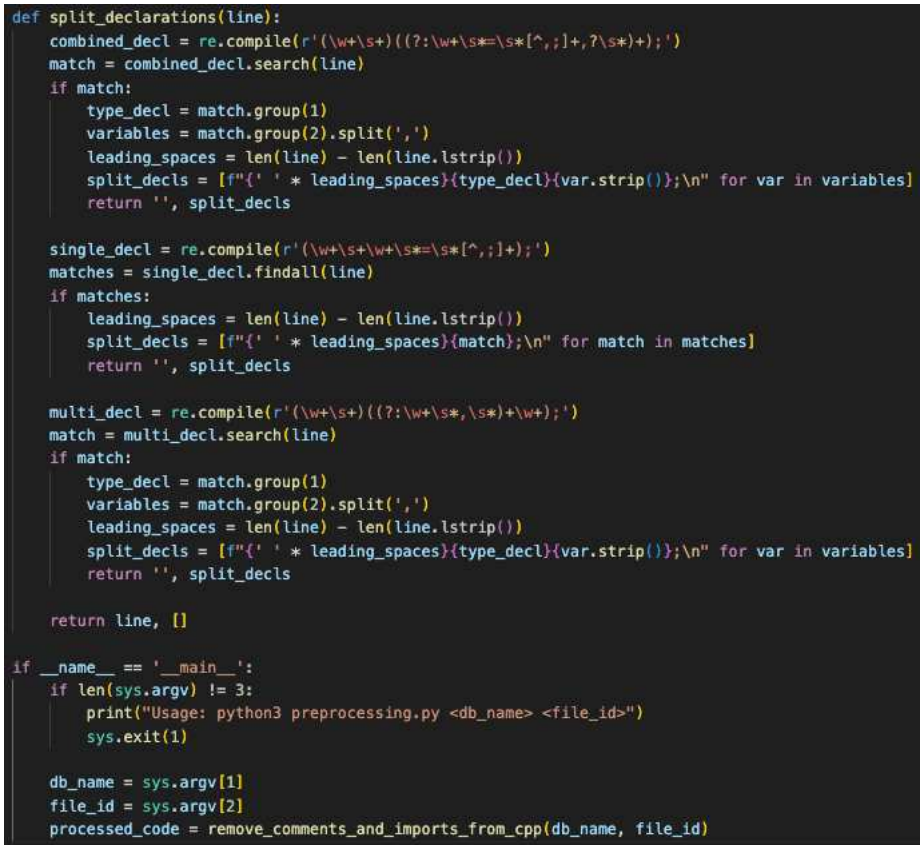
5.1 Preprocesiranje

Proces tokenizacije uključuje procesiranje ulaznog koda, a zatim generiranje tokena od tog procesiranog koda. Tokeni se generiraju prema određenim pravilima navedenim u dokumentu tokenizer.l. Za procesiranje koda postoji Python skripta koja procesira C++ kod spremljen u MongoDB GridFS bazi podataka. Skripta briše sve vrste komentara, #include import izjave, pojednostavljuje kvalificirano ime za neku klasu, npr. std::string (briše se std:),::), te upravlja deklaracijama varijabli. Kasnije se procesirani kod sprema natrag u bazu podataka.

Nakon deklariranja potrebnih import izjava i učitavanja MongoDB URI-a iz environment varijabla, deklarirana je funkcija remove_comments_and_imports_from_cpp prikazana na slici 18. Ona kao parametar prima id korisnika i ulazni kod. Nakon spajanja na monogo i inicijaliziranja GridFS objekta za pristup kolekciji uploads gdje su ulazni podatci spremljeni, dohvaća datoteku iz GridFS-a koristeći file_id. Pročita sadržaj datoteke, dekodira ga iz binarnog u UTF-8 i dijeli ga u pojedinačne linije radi lakše obrade. Regex nam služi za identificiranje dijelova koda kojeg kasnije brišemo. Regex prepoznaje komentare na jednoj liniji (//), više linija (/* neki komentar ovdje */), prepoznaje “#include” kvalificirana imena poput „std::string“ i uklonit će dio „std::“. Kod iterira kroz svaku liniju datoteke, te briše sve navedeno u Regex izrazima. Funkcija pronalazi početak funkcija pomoću vitičastih zagrada ({}) i sprema deklaracije varijabla u pomoćnu listu. Pozvana je funkcija split_declarations, prikazana na slici 19. Ona razdvaja zajedno deklarirane varijable. Takvo razdvajanje deklaracija varijabla je bitno jer će svi dokumenti koji se uspoređuju imati odvojene deklaracije, što rješava potencijalno prekrivanje plagijarizmima spajanjem ili razdvajanjem deklaracija. Razdvojene varijable pozicioniraju se na vrhu funkcije u kojoj su deklarirane. Nakon procesiranja, očišćeni kod ponovno se spaja i pohranjuje kao nova datoteka u GridFS s prefiksom p_ koji je dodan izvornom nazivu datoteke.

```python
def remove_comments_and_imports_from_cpp(db_name, file_id):
    try:
        db_name = f"user_{db_name}"

        mongo_uri = os.getenv('MONGODB_URI')
        client = pymongo.MongoClient(mongo_uri)
        db = client[db_name]
        fs = GridFS(db, collection='uploads')

        file = fs.get(ObjectId(file_id))
        code = file.read().decode('utf-8').splitlines()

        single_line_comments = re.compile(r'//.*')
        block_comments_start = re.compile(r'/\*')
        block_comments_end = re.compile(r'\*/')
        import_statements = re.compile(r'#include\s*<.*?>|#include\s*".*?"')
        fully_qualified_names = re.compile(r'(\w+::)+(\w+)')

        cleaned_code = []
        in_block_comment = False
        in_function = False
        function_vars = []
        function_start = None

        for line in code:
            if in_block_comment:
                if block_comments_end.search(line):
                    line = block_comments_end.sub('', line, 1)
                    in_block_comment = False
                else:
                    continue

            if block_comments_start.search(line) and not block_comments_end.search(line):
                line = block_comments_start.sub('', line)
                in_block_comment = True

            line = single_line_comments.sub('', line)

            if import_statements.search(line):
                continue

            if line.strip().endswith('{') and not in_function:
                in_function = True
                function_start = len(cleaned_code)
            elif line.strip().endswith('}') and in_function:
                in_function = False
                if function_vars:
                    cleaned_code = (
                        cleaned_code[:function_start+1]
                        + function_vars
                        + cleaned_code[function_start+1:]
                    )
                    function_vars = []

            line, declarations = split_declarations(line)
            if declarations:
                if in_function:
                    function_vars.extend(declarations)
                else:
                    cleaned_code.extend(declarations)
                continue

            line = re.sub(fully_qualified_names, lambda match: match.group(2), line)
            if line.strip():
                cleaned_code.append(line)

        processed_code = '\n'.join(cleaned_code)
        fs.put(processed_code.encode('utf-8'), filename=f'p_{file.filename}', contentType='text/plain')

        return processed_code

    except NoFile:
        print(f"No file found with ID: {file_id} in collection 'uploads'")
        return f"No file found with ID: {file_id}"

    except Exception as e:
        print(f"Unexpected error occurred: {str(e)}")
        return f"Unexpected error occurred: {str(e)}"
```

Slika 18. Glavna funkcija skripte



Slika 19. Funkcija za razdvajanje varijabli

5.2. Tokenizacija

Za tokenizaciju ulaznog koda koristimo Flex koji tipično služi za leksičku analizu. Flex koristi tokenizer.l dokument u kojemu su definirana pravila tokenizacije. Nakon toga u komandnoj liniji potrebno je generirati “lex.yy.c.” datoteku komandom “ flex tokenizer.l“. Flex pretvara ta pravila u C kod koji se zatim može kompajlirati u lexer. Generirani lex.yy.c sadrži svu logiku i pravila definirana u tokenizer.l. Komandom “g++ lex.yy.c -o tokenizer” kompajliramo lex.yy.c datoteku koristeći g++ kompajler. Oznaka -o specificira izlaz, a to je u ovom slučaju tokenizer koji je izvršni file kojeg kasnije pozivamo za izvršavanje tokenizacije.

Prvi dio u tokenizer.l datoteci je deklaracija koda unutar “(%{ ... %})“. Potreban je za lex.yy.c datoteku, kod je prikazan na slikama 20 i 21. Deklarirani su razni headeri za ulaz/izlaz, za rad s stringovima itd. Deklarirane su globalne varijable, simbolička tablica za praćenje imena funkcija i klasa. „std::map<int, char> token_to_char” mapira tipove tokena u char vrijednosti. To se koristi za spremanje tokena na kompaktniji način i za olakšavanje RKR-GST algoritmu usporedbu tokena. Definirana je varijabla za spremanje tokena s meta podatcima, te se prati trenutna linija i stupac koji se unose kao meta podatci. Enum TokenType definira sve moguće tokene, a initialize_token_to_char() mapira tipove tokena u char vrijednosti.

```
%{
#include <iostream>
#include <string>
#include <map>
#include <sstream>

using namespace std;

void yyerror(const char *s);
bool isNumericType(const string &str);
bool isStringType(const string &str);

std::map<std::string, std::string> symbol_table;
std::map<int, char> token_to_char;

std::ostringstream tokenStream;
int line_num = 1;
int column_num = 1;

enum TokenType {
    NAMESPACE = 256,
    CLASS,
    VOID,
    OUTPUT_STREAM,
    INPUT_STREAM,
    FUNCTION_CALL,
    METHOD_CALL,
    CLASS_NAME,
    FUNCTION_NAME,
    IDENTIFIER,
    NUMERIC_TYPE,
    STRING_TYPE,
    NUMERIC_VALUE,
    CHAR_VALUE,
    STRING_VALUE,
    PLUS,
    MINUS,
    TIMES,
    DIVIDE,
    EQUAL,
    LESS_THAN,
    GREATER_THAN,
    LEFT_SHIFT,
    RIGHT_SHIFT,
    OPEN_PARENTH,
    CLOSE_PARENTH,
    OPEN_CURLY,
    CLOSE_CURLY,
    OPEN_BRACKET,
    CLOSE_BRACKET,
    COMMA,
    PATTERN_ARROW,
    COLON,
    PERIOD,
    RETURN,
    VECTOR,
    MAP,
    SET,
    TRY,
    CATCH,
    THROW,
    AND,
    IF,
    ELSE,
    WHILE,
    DO,
    SWITCH,
    CASE,
    BREAK,
    CONTINUE,
    VIRTUAL,
    NOT,
    MODULO,
};
```

Slika 20. Prvi priakz %{

```
void initialize_token_to_char() {
    token_to_char[NAMESPACE] = 'A';
    token_to_char[CLASS] = 'B';
    token_to_char[VOID] = 'C';
    token_to_char[OUTPUT_STREAM] = 'D';
    token_to_char[INPUT_STREAM] = 'E';
    token_to_char[FUNCTION_CALL] = 'F';
    token_to_char[METHOD_CALL] = 'G';
    token_to_char[CLASS_NAME] = 'H';
    token_to_char[FUNCTION_NAME] = 'I';
    token_to_char[IDENTIFIER] = 'J';
    token_to_char[NUMERIC_TYPE] = 'K';
    token_to_char[STRING_TYPE] = 'L';
    token_to_char[NUMERIC_VALUE] = 'M';
    token_to_char[CHAR_VALUE] = 'N';
    token_to_char[STRING_VALUE] = 'O';
    token_to_char[PLUS] = 'P';
    token_to_char[MINUS] = 'Q';
    token_to_char[TIMES] = 'R';
    token_to_char[DIVIDE] = 'S';
    token_to_char[EQUAL] = 'T';
    token_to_char[LESS_THAN] = 'U';
    token_to_char[GREATER_THAN] = 'V';
    token_to_char[LEFT_SHIFT] = 'X';
    token_to_char[RIGHT_SHIFT] = 'Y';
    token_to_char[OPEN_PARENTH] = 'W';
    token_to_char[CLOSE_PARENTH] = 'X';
    token_to_char[OPEN_CURLY] = 'Y';
    token_to_char[CLOSE_CURLY] = 'Z';
    token_to_char[OPEN_BRACKET] = 'a';
    token_to_char[CLOSE_BRACKET] = 'b';
    token_to_char[COMMA] = 'c';
    token_to_char[PATTERN_ARROW] = 'd';
    token_to_char[COLON] = 'e';
    token_to_char[PERIOD] = 'f';
    token_to_char[RETURN] = 'g';
    token_to_char[VECTOR] = 'h';
    token_to_char[MAP] = 'i';
    token_to_char[SET] = 'j';
    token_to_char[TRY] = 'k';
    token_to_char[CATCH] = 'l';
    token_to_char[THROW] = 'm';
    token_to_char[AND] = 'n';
    token_to_char[IF] = 'o';
    token_to_char[ELSE] = 'p';
    token_to_char[WHILE] = 'q';
    token_to_char[DO] = 'r';
    token_to_char[SWITCH] = 's';
    token_to_char[CASE] = 't';
    token_to_char[BREAK] = 'u';
    token_to_char[CONTINUE] = 'v';
    token_to_char[VIRTUAL] = 'w';
    token_to_char[NOT] = 'z';
    token_to_char[MODULO] = 'x';
}
void add_class_name(const char* name) {
    symbol_table[name] = "class";
}

void add_function_name(const char* name) {
    symbol_table[name] = "function";
}

%}
```

Slika 21. Drugi priakz %}

U dijelu tokenize.l datoteke označene s „%%” definiramo leksička pravila gdje su neka prikazana na slici 22. Ovaj dio sadrži Regex izraze koji odgovaraju različitim uzorcima i koje radnje će se izvesti nad svakim uzorkom. Neka od ključnih pravila su ignoriranje komentara, kojih ne bi trebalo biti jer smo komentare uklonili procesiranjem. Nadalje postoje pravila za ključne riječi (namespace, class, void…), te se za tu ključnu riječ spremaju meta podatci i tip tokena pretvoren u char. Kao i za ključne riječi, postoje i pravila za operatore i simbole. Neke su ključne riječi kao private, public i const ignorirane za tokenizaciju jer ne utječu značajno na logiku ili funkcionalnost programa.

```
%%

[\n]                    { line_num++; column_num = 1; }

[ \t]+                  { column_num += yyleng; /* Ignore whitespace */ }
"//".*                  { /* Ignore single-line comments */ }
\/\*([^*]|\*+[^*/])*\*\/ { /* Ignore multi-line comments */ }
^#.*  { /* Ignore preprocessor directives */ }

"private"   { column_num += yyleng; /* Ignore 'private' */ }
"protected" { column_num += yyleng; /* Ignore 'protected' */ }
"public"    { column_num += yyleng; /* Ignore 'public' */ }
"internal"  { column_num += yyleng; /* Ignore 'internal' */ }
"static"    { column_num += yyleng; /* Ignore 'static' */ }
"final"     { column_num += yyleng; /* Ignore 'final' */ }
"const"     { column_num += yyleng; /* Ignore 'const' */ }

":" { column_num += yyleng; /* Ignore ':' after access specifiers */ }

"namespace" { tokenStream << token_to_char[NAMESPACE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return NAMESPACE; }
"class"     { BEGIN(CLASS_DECL); tokenStream << token_to_char[CLASS] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return CLASS;
"void" {
    tokenStream << token_to_char[VOID] << ' ' << line_num << ' ' << column_num << '\n';
    column_num += yyleng;  // Update column number after "void"

    int c;
    while ((c = yyinput()) != EOF) {
        if (c == '\n') {
            line_num++;
            column_num = 1;
        } else if (isspace(c)) {
            column_num++;
        } else {
            unput(c);
            break;
        }
    }

    BEGIN(FUNC_DECL);
    return VOID;
}
"virtual"   { tokenStream << token_to_char[VIRTUAL] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return VIRTUAL; }
"cout"      { tokenStream << token_to_char[OUTPUT_STREAM] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return OUTPUT_STREAM; }
"cin"       { tokenStream << token_to_char[INPUT_STREAM] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return INPUT_STREAM; }

"return"    { tokenStream << token_to_char[RETURN] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return RETURN; }
"vector"    { tokenStream << token_to_char[VECTOR] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return VECTOR; }
"map"       { tokenStream << token_to_char[MAP] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return MAP; }
"set"       { tokenStream << token_to_char[SET] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return SET; }

"try"       { tokenStream << token_to_char[TRY] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return TRY; }
"catch"     { tokenStream << token_to_char[CATCH] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return CATCH; }
"throw"     { tokenStream << token_to_char[THROW] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return THROW; }

"if"        { tokenStream << token_to_char[IF] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return IF; }
"else"      { tokenStream << token_to_char[ELSE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return ELSE; }
"while"     { tokenStream << token_to_char[WHILE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return WHILE; }
"do"        { tokenStream << token_to_char[DO] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return DO; }
"switch"    { tokenStream << token_to_char[SWITCH] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return SWITCH; }
"case"      { tokenStream << token_to_char[CASE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return CASE; }
"break"     { tokenStream << token_to_char[BREAK] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return BREAK; }
"continue"  { tokenStream << token_to_char[CONTINUE] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return CONTINUE; }

"&"         { tokenStream << token_to_char[AND] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return AND; }
"endl"      { column_num += yyleng; /* Ignore 'endl' */ }

"!"   { tokenStream << token_to_char[NOT] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return NOT; }
"%"   { tokenStream << token_to_char[MODULO] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return MODULO; }

"["         { tokenStream << token_to_char[OPEN_BRACKET] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return OPEN_BRACKET; }
"]"         { tokenStream << token_to_char[CLOSE_BRACKET] << ' ' << line_num << ' ' << column_num << '\n'; column_num += yyleng; return CLOSE_BRACKET; }
```

Slika 22. Prikaz pravila

Pravilo za provjeru imena klasa i funkcija provjerava simboličku tablicu ako postoji to ime klase ili funkcije, te ako postoji vraća odgovarajući token (CLASS_NAME ili FUNCTION_NAME). Postavljeno je pravilo da se sve numeričke vrijednosti (integer, double, float itd.) tretiraju isto. To znači da postoji jedan token za njih (NUMERIC_VALUE). Isto vrijedi za tipove stringa. Kod za izvršavanje tih pravila prikazan je na slici 23, a definicija pravila prikazana je na slici 24.

```
[a-zA-Z_][a-zA-Z0-9_]* {
    string id(yytext);
    if (symbol_table.find(id) != symbol_table.end()) {
        if (symbol_table[id] == "class") {
            tokenStream << token_to_char[CLASS_NAME] << ' ' << line_num << ' ' << column_num << '\n';
            column_num += yyleng;
            return CLASS_NAME;
        } else if (symbol_table[id] == "function") {
            tokenStream << token_to_char[FUNCTION_NAME] << ' ' << line_num << ' ' << column_num << '\n';
            column_num += yyleng;
            return FUNCTION_NAME;
        }
    } else if (isNumericType(id)) {
        tokenStream << token_to_char[NUMERIC_TYPE] << ' ' << line_num << ' ' << column_num << '\n';
        column_num += yyleng;
        return NUMERIC_TYPE;
    } else if (isStringType(id)) {
        tokenStream << token_to_char[STRING_TYPE] << ' ' << line_num << ' ' << column_num << '\n';
        column_num += yyleng;
        return STRING_TYPE;
    } else {
        tokenStream << token_to_char[IDENTIFIER] << ' ' << line_num << ' ' << column_num << '\n';
        column_num += yyleng;
        return IDENTIFIER;
    }
}
```

Slika 23. Prikaz pravila za numeričke i string tipove

```
bool isNumericType(const string &str)
{
    return str == "byte" || str == "short" || str == "int" || str == "long" ||
           str == "float" || str == "double" || str == "Byte" || str == "Short" ||
           str == "Integer" || str == "Long" || str == "Float" || str == "Double";
}

bool isStringType(const string &str)
{
    return str == "std::string" || str == "char" || str == "wchar_t" || str == "char16_t" || str == "char32_t";
}
```

Slika 24. Definicija pravila za numeričke i string tipove

Glavna funkcija prikazana na slici 25 postavlja lexer za čitanje ulazne datoteke, ako je to uneseno u komandu liniju ili čita kroz standardnog ulaza (stdin). Aplikacija koristi stdin. Inicijalizira se pretvaranje tokena u znakove (char), te se input procesira dok više nema tokena za pronaći.

```
int main(int argc, char **argv)
{
    if (argc > 1) {
        yyin = fopen(argv[1], "r");
        if (!yyin) {
            cerr << "Error: Could not open file " << argv[1] << endl;
            return 1;
        }
    } else {
        yyin = stdin;
    }

    initialize_token_to_char();

    while (yylex());

    std::cout << tokenStream.str();

    return 0;
}
```

Slika 25. Glavna funkcija

5.3. RKR-GST algoritam.

RKR-GST algoritam (Running Karp-Rabin - Greedy String Tiling) (Wise, 1993) mjeri sličnosti između dva stringa i pronalazi podnizove koji su isti. Kombinira metodu Greedy String Tiling (GST), koja pronalazi najdulje podnizove. Karp-Rabin algoritam heshiranja služi za optimiziraniju usporedbu podnizova. Algoritam funkcionira tako da prvo izračunava hash vrijednosti za podnizove oba niza te im onda uspoređuje hash vrijednosti. Jednom kada se hash vrijednosti podudaraju, provjeravaju stvarne vrijednosti znak po znak te se proširuje pretraživanje dok se znakovi ne podudaraju. Ta se podudaranja tada označuju kao pločice (eng. tiles) koje se isključuju iz daljnjih usporedbi. Proces se ponavlja iterativno gdje se postupno identificiraju kraća podudaranja. Pojmovi koji se koriste u opisu algoritma:

Text (T) – Tekst je prvi set tokena koji se uspoređuje s patternom i uvijek je duži od njega

Pattern (P) – Pattern je kraći set tokena, podudarajući podstringovi iz njega traže se u tekstu

Search length (s) – je duljina "prozora" odabrana kao duljina traženog stringa, inicijalno je fiksirana vrijednost ali se iteracijama algoritma smanjuje

Text_mark i **Pattern_mark** – Inicijalno to su liste s false boolean vrijednostima, Svaki false ili true odgovara jednom tokenu u textu ili patternu. Služe nam za onačavanje podudaranja.

Maksimalno podudaranje (**MaxMatch**) – najdulji podstring za P i T u toj iteraciji.

5.3.1. Scanpattern

Scanpattern.js, prikazan na slici 26, dio je algoritma koji kao rezultat vraća najdulji podstring podudaranja. Scanpattern funkcija, iz glavne funkcije (main.js), kao parametre prima tokene koji su u dvostruko povezanoj listi za text i pattern, search length s (inicijalno 20), text_mark i pattern_mark. U while petlji iteriramo tekstom s duljinom s. To znači da će se u varijablu text spremiti svaki podstring teksta duljine s. Zatim se svaki taj podstring pomoću rabinKarpHash funkcije heshira i hash vrijednosti se spremaju u hash tablicu. Kao što možemo vidjeti na slici 27, prije iteracije postoji provjera (checkNextTokens) je li trenutni token, ili neki od sljedećih, već označen. Drugim riječima ako je sljedećih s tokena nije označeno, nastavlja se normalno s algoritmom. Ako postoji neki token koji je u toj duljini s već označen (u nekoj prijašnjoj iteraciji), onda se preskače do prvog neoznačenog tokena.

```
function scanpattern(T, P, s, text_mark, pattern_mark)
  console.log(`Scanning pattern with s = ${s}`);
  let currentT = T.head;
  let index = 0;
  let hashTable = {};
  let longestMatch = 0;

  const list = new DoublyLinkedList();

  while (currentT != null) {
    let text = "";
    let tempT = currentT;

    let result = checkNextTokens(
    currentT, text_mark, s, index);

    currentT = result.currentTP;
    index = result.index;

    if (currentT == null) {
      break;
    }
    tempT = currentT;

    for (let i = 0; i < s && tempT != null; i++) {
      text += tempT.data;
      tempT = tempT.next;
    }

    if (text.length == s) {
      let hashValue = rabinKarpHash(text, s);
      hashTable[index] = {
        hashValue: hashValue,
        startPoint: index,
      };
      index++;
    }

    currentT = currentT.next;
```

Slika 26. Iteriranje i heshiranje tekesta

```
unction checkNextTokens(currentTP, markArray, s, currentIndex) {
console.log(`Checking next tokens with s = ${s}`);

let index = currentIndex;
let tempT = currentTP;

let isMarked = false;
for (let i = 0; i < s && tempT != null; i++) {
  if (markArray[index + i]) {
    isMarked = true;
    console.log(`Token at index ${index + i} is marked`);
    break;
  }
  tempT = tempT.next;
}

if (isMarked) {
  let nextUnmarkedIndex = index + s;
  while (
    nextUnmarkedIndex < markArray.length &&
    markArray[nextUnmarkedIndex]
  ) {
    nextUnmarkedIndex++;
  }
  console.log(`Skipping from index ${index} to index ${nextUnmarkedIndex}`);
  if (nextUnmarkedIndex - index <= s) {
    for (let i = index; i < nextUnmarkedIndex && currentTP != null; i++) {
      currentTP = currentTP.next;
    }
    index = nextUnmarkedIndex;
    console.log(`Returning to index ${index} with currentT after skipping`);
  } else if (nextUnmarkedIndex >= markArray.length) {
    currentTP = null;
    index = markArray.length;
  } else {
    for (let i = index; i < nextUnmarkedIndex && currentTP != null; i++) {
      currentTP = currentTP.next;
    }
    index = nextUnmarkedIndex;
  }
}

console.log(
  `No marked tokens found from index ${index} to index ${index + s - 1}`
);

return { currentTP, index };
```

Slika 27. Porvjera za sljdeće tokene

Nakon kreiranja hash tablice za tekst, kao što je i prikazano na slici 29, kreće iteracija za prvi podstring patterna. Prije iteracije, naravno provjeravamo sljedeće tokene s checkNextTokens. Nakon valjane provjere heshira se prvi podstring patterna, te odmah slijedi provjera te hash vrijednost s hash tablicom. Ako postoji isti hash u tablici, imamo potencijalno podudaranje. Tada moramo opet iterirati kroz tekst, da bi dohvatili vrijednosti koje su heshirane, te tokene spremamo u varijablu za tekst. Među korak je provjera ako je to potencijalno podudaranje već bilo označeno u nekoj prijašnjoj iteraciji. Pozivamo funkciju isMarked, prikazanoj na slici 28, posebno za pattern i string, te se u listi boolean vrijednosti provjerava jesu li već označeni. Ova funkcija drugačija je od prijašnje provjere checkNextTokens, isMarked provjerava samo podstring kojeg smatramo kao moguće podudaranje. Ako je bilo koji token u tom podstringu već označen heshira se sljedeći podstring patterna te se uspoređuje s hash tablicom. Ako nema označenih vrijednosti u podstringu program se dalje izvodi, provjerava stvarne vrijednosti teksta i patterna.

```
function isMarked(index, markArray, s) {
  if (index < 0 || index + s > markArray.length) {
    console.log(`Index: ${index}, Out of bounds`);
    return false;
  }

  for (let i = index; i < index + s; i++) {
    if (markArray[i]) {
      console.log(`Index: ${i}, Value: ${markArray[i]}`);
      return true;
    }
  }

  console.log(`Index: ${index} to ${index + s - 1}, All values are false`);
  return false;
}
```

Slika 28. Provjera podstringa

```
while (currentP != null) {
  let pattern = "";
  let tempP = currentP;

  let result = checkNextTokens(currentP, pattern_mark, s, patternIndex);
  currentP = result.currentTP;
  patternIndex = result.index;

  if (currentP == null) {
    break;
  }
  tempP = currentP;

  for (let i = 0; i < s && tempP != null; i++) {
    pattern += tempP.data;
    tempP = tempP.next;
  }

  console.log(pattern);
  console.log(pattern.length);

  if (pattern.length == s) {
    let patternHash = rabinKarpHash(pattern, s);

    for (let key in hashTable) {
      if (hashTable[key].hashValue === patternHash) {
        let matchStart = hashTable[key].startPoint;
        let currentNode = T.head;
        for (let i = 0; i < matchStart; i++) {
          currentNode = currentNode.next;
        }

        let matchedText = "";
        for (let i = 0; i < s; i++) {
          matchedText += currentNode.data;
          currentNode = currentNode.next;
        }

        console.log(
          `Checking match at text index ${matchStart} with pattern index ${patternIndex}`
        );

        if (
          isMarked(matchStart, text_mark, s) ||
          isMarked(patternIndex, pattern_mark, s)
        ) {
          console.log(
            `Tokens are marked, skipping match at text index ${matchStart} and pattern index ${patternIndex}`
          );
          continue;
        }

        if (matchedText === pattern) {
```

Slika 29. Hashiranje patterna

If provjerava jesu li text i pattern stvarno isti. Ako jesu pokušava se istražiti podudaraju li se i izvan podstringa. Postavljamo pokazivače na kraju podstringa P i T, i sve dok su vrijednosti na koje pokazivači pokazuju iste, povećava se k i pokazivači se sele na sljedeći token sve dok T i P nisu različiti ili dok ne dođu do kraja vezane liste, što je i prikazano na slici 30.

```
if (matchedText === pattern) {
  let k = s;
  let tempNode = currentNode;
  let tempPattern = tempP;

  while (
    tempNode != null &&
    tempPattern != null &&
    tempNode.data === tempPattern.data
  ) {
    k++;
    tempNode = tempNode.next;
    tempPattern = tempPattern.next;

    if (tempNode == null || tempPattern == null) {
      break;
    }
  }
```

Slika 30. Produljenje trenutnog podudaranja

Ako je produljenje veliko, tj. ako je k > 2 * s, znači da je inicijalni s bio pre mali. Onda se ne isplati označiti novopronađeno podudaranje, jer tada postoji veća šansa da postoje i ostali podstringovi koji će se tim produljenjem tek pronaći, što bi značilo da će se puno tokena provjeravati jedan po jedan za pronalazak tog produljenja. Više se isplati rekurzivno pozvati scanpattern s postavljenim k kao nova duljina s kao što je i prikazano na slici 31. Ako se ne zadovolji uvjet rekurzivnog poziva, podatci o podudaranju spremaju se u dvostruko vezanu listu redova, na početku liste je red s maksimalnim podudaranjem (najveći s). U red se sprema s, početni indeks podudaranja u P i u T. Scanpattern vraća duljinu najduljeg pronađenog podudaranja i listu redova.

```
            if (k > 2 * s) {
              console.log(`Recursively calling scanpattern with k = ${k}`);
              return scanpattern(T, P, k, text_mark, pattern_mark);
            } else {
              console.log(
                `Exact match found at text index ${matchStart} and pattern index ${patternIndex} ${matchedText} === ${pattern}`
              );

              if (k > longestMatch) {
                longestMatch = k;
                console.log(`New maximal match recorded: ${k}`);
              }
              const node = list.find(k);
              if (node) {
                node.matches.push({ p: patternIndex, t: matchStart, s: k });
              } else {
                list.insert(k, { p: patternIndex, t: matchStart, s: k });
              }
            }
          } else {
            console.log(
              `Hash collision at text index ${matchStart} and pattern index ${patternIndex}`
            );
          }
        }
      }
    }

    currentP = currentP.next;
    patternIndex++;
  }

  console.log(`Hash table: ${JSON.stringify(hashTable)}`);
  list.printList();
  console.log(`Longest match: ${longestMatch}`);

  return { longestMatch, list };
```

Slika 31. Rekurzivni poziv i spremanje u listu redova

5.3.2. Hash funkcija

Za heshiranje podstringa,koristimo funkciju rabinKarpHash prikazanu na slici 32. Nakon inicijalizacija varijabla, računamo vrijednost od h kao $d^{(s-1)}$ % q. To će nam kasnije koristiti za „rolanje“ po hash vrijednostima. Sljedeća petlja računa hash vrijednost prvog podstringa duljine s. Za svaki znak u podsringu pretvaramo znak u broj koristeći “str.charCodeAt(i) - "A".charCodeAt(0) + 1.”. To nam daje brojeve počevši od 1 za A, 2 za B itd. Hash množimo s d(31), dodajemo vrijednost znaka i uzimamo modul q od 101da bi spriječili oveflow. Na primjer, ako je podniz "ABC", a s = 3, onda A postaje 1, B postaje 2, C postaje 3. Hash se izračunava kao:

tHash = (31 * 0 + 1) % 101 = 1

tHash = (31 * 1 + 2) % 101 = 33

tHash = (31 * 33 + 3) % 101 = 26

Hash za ABC je 26. Umjesto da ponovno računamo hash vrijednost za svaki podstring, implementiramo „rolling hash” tako da brišemo stari znak i dodajemo novi. Na primjer, iz podstringa ABC do BCD. Briše se A (1), dodaje se D (4), pa bi onda hash bio:

tHash = (31 * (26 - 1 * h) + 4) % 101

```
function rabinKarpHash(str, s) {
  const d = 31;
  const q = 101;
  const n = str.length;
  let h = 1;
  let tHash = 0;
  let hashValues = [];

  for (let i = 0; i < s - 1; i++) {
    h = (h * d) % q;
  }

  for (let i = 0; i < s; i++) {
    tHash = (d * tHash + (str.charCodeAt(i) - "A".charCodeAt(0) + 1)) % q;
  }

  hashValues.push(tHash);

  for (let i = 1; i <= n - s; i++) {
    tHash =
      (d * (tHash - (str.charCodeAt(i - 1) - "A".charCodeAt(0) + 1) * h) +
        (str.charCodeAt(i + s - 1) - "A".charCodeAt(0) + 1)) %
      q;

    if (tHash < 0) {
      tHash = tHash + q;
    }

    hashValues.push(tHash);
  }

  return hashValues.join();
}
```

Slika 32. Hash funkcija

5.3.3. Mararrays

Funkcija markarrays, prikazana na slici 33, služi za označavanje podudarenih tokena. Kao parametre prima dvostruko vezanu listu redova, s, pattern_mark i text_mark. Uzimaju se podati iz liste redova, u occluded funkciji prikazanoj na slici 34, ulazi prvo podudaranje (match), pattern_mark i text_mark. Ta funkcija provjerava jesu li neki tokeni iz danog ulaza već označeni (ako za trenutni match postoje neki tokeni označeni kao dio prijašnjeg match-a). Ako se desilo preklapanje, vraća koliko je tokena već označeno. Ako nema preklapanja, funkcija označuje tokene. Dalje ako je došlo do preklapanja i velika količina tokena je ostala neoznačena, specifično ako je neoznačeni dio barem pola od s, tad se neoznačeni dio postavi u dvostruko vezanu listu redova te se kasnije označava.

```
function markarrays(s, doublyLinkedList, pattern_mark, text_mark) {
  let match = null;

  while (!doublyLinkedList.isEmpty()) {
    let currentQueue = doublyLinkedList.head;

    while (currentQueue && currentQueue.isEmpty()) {
      doublyLinkedList.shift();
      currentQueue = doublyLinkedList.head;
    }

    if (currentQueue) {
      match = currentQueue.dequeue();
    }

    let occlusionResult = occluded(match, pattern_mark, text_mark);
    console.log(
      `Ocluded tokens: ${occlusionResult.occludedTokens}, IsOcluded: ${occlusionResult.isOccluded}`
    );

    if (!occlusionResult.isOccluded) {
      console.log("Marking tokens: " + match.p, match.t, match.s);
      mark_tokens(match, pattern_mark, text_mark);
    } else if (match.s - occlusionResult.occludedTokens >= s / 2) {
      let unmarkedPortion = {
        length: match.s - occlusionResult.occludedTokens,
        matches: [
          {
            p: match.p,
            t: match.t,
            s: match.s - occlusionResult.occludedTokens,
          },
        ],
      };
      doublyLinkedList.insert(
        unmarkedPortion.length,
        unmarkedPortion.matches[0]
      );
      console.log(
        `Reinserting unmarked portion: length = ${unmarkedPortion.length},
         p = ${unmarkedPortion.matches[0].p},  t = ${unmarkedPortion.matches[0].t},
         s = ${unmarkedPortion.matches[0].s}`
      );
    }
  }
```

Slika 33. Markarrays funkcija

```
function occluded(match, pattern_mark, text_mark) {
  let occludedTokens = 0;

  console.log(
    `Checking occlusion for match: p=${match.p}, t=${match.t}, s=${match.s}`
  );

  for (let i = 0; i < match.s; i++) {
    const isPatternOccluded = pattern_mark[match.p + i];
    const isTextOccluded = text_mark[match.t + i];

    console.log(
      `Index ${i}: pattern_mark[${
        match.p + i
      }] = ${isPatternOccluded}, text_mark[${match.t + i}] = ${isTextOccluded}`
    );

    if (isPatternOccluded || isTextOccluded) {
      occludedTokens++;
      console.log(
        `Token at index ${i} is occluded. Total occluded so far: ${occludedTokens}`
      );
    }
  }

  const isOccluded = occludedTokens > 0;
  console.log(
    `Total occluded tokens: ${occludedTokens}. Is occluded: ${isOccluded}`
  );

  return {
    occludedTokens: occludedTokens,
    isOccluded: isOccluded,
  };
}
```

Slika 34. Occluded funkcija, RKR.js

Funkcija za oznaku teksta i patterna prikazana na slici 35, vrlo je jednostavna. For petljama se iterira kroz text_mark i pattern_mark, a iterira se samo onom duljinom pattern_mark-a ili text_mark-a opisanim s početnim pozicijama match.p ili match.t.

```
function mark_tokens(match, pattern_mark, text_mark) {
  for (let i = match.p; i < match.p + match.s; i++) {
    pattern_mark[i] = true;
  }

  for (let i = match.t; i < match.t + match.s; i++) {
    text_mark[i] = true;
  }
}
```

Slika 35. Mark_tokens funkcija, RKR.js

5.3.4. Main.js

Glavna funkcija u main.js dokumentu je processUserTokens. Kad se uspostavi veza prema bazi za određenog korisnika, dohvaćamo kolekciju tokens i uploads.files. Iz kolekcije tokens izvlače se tokeni i meta podatci, a iz kolekcije uploads.files id i naziv procesiranog koda. For petlja iterira tokenima ali i zamjenjuje se redoslijed pozicije patterna i teksta ako je pattern veći. Pattern i string spremaju se u svoje varijable i inicijaliziraju se text_mark i pattern_mark s početnim false vrijednostima, što je i prikazano na slici 36. Kao što je moguće vidjeti na slici 37, definira se početni search length s, postavljamo vrijednost varijable "minimumMatchLength", što je zapravo najmanja moguća vrijednost za search length s. Count broji broj iteracija. U while petlji pozivamo scanpattern s potrebnim parametrima. Rezultat scanpatterna sprema se u objekt Lmax koji sadrži duljinu najduljeg podudaranja (MaxMatch) i dvostruko vezanu listu redova. Ako je MaxMatch veći od 2*s, s se postavlja na vrijednost MaxMatch-a te se scanpattern ponovno izvodi s novim s. Ako se ne poziva ponovno scanpattern, program nastavlja s označavanjem. Nakon prvog poziva i označavanja podudaranja dobivenih prvom iteracijom (ili ako se rekurzivno pozvao scanpattern drugom), tad se pronalaze manja podudaranja. Ako je s veći od 2* minimumMatchLength, s će se prepolovit na pola ili ako je s veći od minimumMatchLength, u s postavljamo vrijednost minimumMatchLength-a. Novi smanjeni s će ići u scanpattern i u markarrays sve dok je s veći ili jednak minimumMatchLength-u. Nakon svih završenih iteracija text_mark i pattern_mark sadržavat će true vrijednosti za podudarajuće tokene.

```
export async function processUserTokens(userId) {
  try {
    const db = await connectToUserDB(userId);
    const tokensCollection = db.collection("tokens");
    const filesCollection = db.collection("uploads.files");

    const tokenDocuments = await tokensCollection.find({ userId }).toArray();

    if (tokenDocuments.length === 0) {
      console.log("No tokens data found in the database.");
      return { tokensData: [], comparisons: [] };
    }

    const comparisons = [];
    let comparisonCount = 1;

    const fileIdToFilenameMap = {};
    const processedCodeIds = tokenDocuments.map(
      (doc) => doc.text1ProcessedCodeID
    );

    const filesData = await filesCollection
      .find({
        _id: {
          $in: processedCodeIds.map((id) => new ObjectId(id)),
        },
      })
      .toArray();

    filesData.forEach((file) => {
      fileIdToFilenameMap[file._id.toString()] = file.filename;
    });

    for (let i = 0; i < tokenDocuments.length - 1; i++) {
      for (let j = i + 1; j < tokenDocuments.length; j++) {
        let textDoc = tokenDocuments[i];
        let patternDoc = tokenDocuments[j];

        if (patternDoc.tokens.length > textDoc.tokens.length) {
          [textDoc, patternDoc] = [patternDoc, textDoc];
        }

        const textTokens = textDoc.tokens;
        const patternTokens = patternDoc.tokens;

        const textTokenList = convertArrayToList(textTokens);
        const patternTokenList = convertArrayToList(patternTokens);

        const text_mark = Array(textTokens.length).fill(false);
        const pattern_mark = Array(patternTokens.length).fill(false);
```

Slika 36. Prvi dio glave funkcije

```
let s = 20;
const minimumMatchLength = 5;
let count = 1;

let stop = false;

while (!stop) {
  let Lmax = scanpattern(
    textTokenList,
    patternTokenList,
    s,
    text_mark,
    pattern_mark
  );
  console.log(`Lmax = ${Lmax.longestMatch} `);
  console.log("S is: " + s);
  if (Lmax.longestMatch > 2 * s) {
    console.log(
      `IN MAIN Recursively calling scanpattern with k = ${Lmax.longestMatch}`
    );
    s = Lmax.longestMatch;
  } else {
    markarrays(s, Lmax.list, pattern_mark, text_mark);
    if (s > 2 * minimumMatchLength) {
      s = Math.floor(s / 2);
      count++;
      console.log(`Iteration number: ${count}`);
    } else if (s > minimumMatchLength) {
      s = minimumMatchLength;
      count++;
      console.log(`Iteration number: ${count}`);
    } else {
      stop = true;
    }
  }
}
```

Slika 37. While petlja za pokretanje algoritma

Meta podatci za tekst i pattern koji dolaze iz baze su meta podaci za sve tokene. Funkcijom filterMetadataByMark prikazanoj na slici 38, filtriramo meta podatke po text_mark i pattern_mark. Rezultat su meta podatci samo za označene tokene koji su spremljeni u svoju varijablu kao što je prikazano na slici 39.

```
function filterMetadataByMark(markArray, metadataArray) {
  return metadataArray
    .filter((_, index) => markArray[index])
    .map(({ line, column }) => ({ line, column }));
}
```

Slika 38. Poziv funkcije za filtriranje

```
const filteredTextMetadata = filterMetadataByMark(
  text_mark,
  textDoc.metadata
);
const filteredPatternMetadata = filterMetadataByMark(
  pattern_mark,
  patternDoc.metadata
);
```

Slika 39. Funkcija za filtriranje

Za izračun postotka sličnosti koji je prikazan na slici 40, u varijablu totalMatchingLength sprema se količina označenih tokena u tekstu ili u patternu (P i T imaju isti broj označenih tokena). Postotak sličnosti računa se kao 2 * totalMatchingLength / textTokens.length + patternTokens.length, gdje su textTokens i patternTokens ukupan broj tokena za P i T.

```
let totalMatchingLength = text_mark.filter(Boolean).length;

const similarityScore = (
  ((2 * totalMatchingLength) /
    (textTokens.length + patternTokens.length)) *
  100
).toFixed(4);
```

Slika 40. Izračun postotka sličnosti

5.4. Server.js

Servjer.js skripta koristeći Node.js i Express.ja upravlja svim prijašnje opisanim procesima. Server poziva skriptu za procesiranje ulaznog koda, tokenizaciju, spremanje podataka u bazu i pozivanje algoritma.

Svakim unosom podataka i bazu kreira se novi korisnik. Stariji korisnici nam više nisu potrebni pa je zato postavljen corn job koji starije korisnike kao što je i vidljivo na slici 41. Sama funkcija nalazi se u cleanup.js datoteci prikazanoj na slici 42, te ona briše korisnike po njihomom ID-u

```
cron.schedule("0 0 * * *", () => {
  console.log("Running daily user cleanup...");
  cleanupOldUsers();
});
```

Slika 41. Pozivanje funkcije za brisanje korisnika

```
import { MongoClient } from "mongodb";
import dotenv from "dotenv";

dotenv.config();

const uri = process.env.MONGODB_URI;
const client = new MongoClient(uri);

Codiumate: Options | Test this function
export async function cleanupOldUsers() {
  try {
    await client.connect();
    const adminDb = client.db().admin();

    const { databases } = await adminDb.listDatabases();

    let expirationDate = new Date();
    expirationDate.setDate(expirationDate.getDate() - 1);
    console.log("Expiration date for deletion (UTC):", expirationDate);

    for (const dbInfo of databases) {
      const dbName = dbInfo.name;

      if (dbName.startsWith("user_")) {
        console.log(`Checking user database: ${dbName}`);

        const db = client.db(dbName);

        const oldFiles = await db
          .collection("uploads.files")
          .find({
            uploadDate: { $lt: expirationDate },
          })
          .toArray();

        if (oldFiles.length > 0) {
          console.log(`Dropping database for user: ${dbName}`);
          await db.dropDatabase();
          console.log(`Database ${dbName} has been dropped.`);
        } else {
          console.log(`No old files found for user: ${dbName}`);
        }
      }
    }

    console.log("User cleanup completed.");
  } catch (error) {
    console.error("Error cleaning up old users:", error);
  } finally {
    await client.close();
  }
}
```

Slika 42. Funkcija za brisanje korisnika

Funkcija runPreprocessingScript prikazana na slici 43, radi asinkrono. Deklariran je put do preprocessing.py skripte i komanda za pokretanje skripte. Exec funkcija pokreće preprocessing skriptu. Ako ne dođe do greške izlaz (stdout) sprema se taj izlaz koji je procesirani kod. Kod se dalje normalizira, tako da postavlja /n za novi red. Put do tokenizer-a je kreiran, exec funkcija pokreće proces i čeka input. Izlaz od tokenizera je podijeljen na više linija (token, linija, stupac), pa se onda tokeni i meta podatci spremaju u zasebnu varijablu. Sljedeće šaljemo input u tokenizer putem stdin-a. Nakon što su procesiranje i tokenizacija završili funkcija vraća objekt koji sadrži procesirani kod, tokene i meta podatke.

Slika 43. Funkcija za izvršavanje procesiranja i tokenizacije

POST ruta /upload prikazana na slici 44, koristi multer za unos podataka u aplikaciju. Ako korisnik nije niša unio, dogovora s HTTP 400 i odgovarajućom porukom. Kreira je jedinstveni identifikacijski ključ za korisnika nakon svakog ulaza. Ulaz se sprema u MongoDB koristeći GridFS za spremanje većih podataka. Zatim se poziva runPreprocessingScript() funkcija. Nakon dobivenog odgovora funkcije provjerava se postoji li već taj procesirani kod u bazi, ako ne postoji sprema ga u bazu. Kao što je i prikazano na slici 45, tokeni i meta podatci iz funkcije se također spremaju u kolekciju tokens. U for petlji iteriramo ostalim ulaznim kodovima da bi se i oni procesirali i tokenizirali. Jednom kad je sve izvršeno server odgovara s 201 s JSON odgovorom prikazanim na slici 46.

```
app.post("/upload", upload.array("files"), async (req, res) => {
  console.log("Files received:", req.files);
  if (!req.files || req.files.length === 0) {
    return res.status(400).json({ message: "No files uploaded" });
  }

  const userId = generateUserId();
  console.log(`Generated user ID: ${userId}`);
  const db = await connectToUserDB(userId);
  console.log(`Database connected for user ID: ${userId}`);

  let filesUploaded = [];
  let processingResults = [];
  let errors = [];

  const processFile = async (file) => {
    try {
      const readableStream = new Readable();
      readableStream.push(file.buffer);
      readableStream.push(null);

      const bucket = new GridFSBucket(db, { bucketName: "uploads" });
      const uploadStream = bucket.openUploadStream(file.originalname, {
        contentType: file.mimetype,
      });

      const fileId = await new Promise((resolve, reject) => {
        readableStream.pipe(uploadStream);
        uploadStream.on("finish", () => resolve(uploadStream.id));
        uploadStream.on("error", reject);
      });

      console.log(
        `Original file uploaded: ${file.originalname}, ID: ${fileId}`
      );
      filesUploaded.push({ filename: file.originalname, id: fileId });

      const { processedCode, tokens, metadata } = await runPreprocessingScript(
        fileId.toString(),
        userId
      );

      const processedFilename = `p_${file.originalname}`;
      const existingProcessedFile = await checkIfProcessedFileExists(
        userId,
        processedFilename
      );
```

Slika 44. Početak rute /upload

Slika 45. Kraj rute /upload

```
{
    "userId": "z5svv6okb",
    "files": [
        {
            "filename": "identical1.cpp",
            "id": "66e2203bed00d9eb4437c104"
        },
        {
            "filename": "plagiarised1.cpp",
            "id": "66e2203eed00d9eb4437c107"
        },
        {
            "filename": "reordering1.cpp",
            "id": "66e22041ed00d9eb4437c10a"
        },
        {
            "filename": "slightly_plagirised1.cpp",
            "id": "66e22044ed00d9eb4437c10d"
        },
        {
            "filename": "splitting1.cpp",
            "id": "66e22047ed00d9eb4437c110"
        }
    ],
    "processingResults": [
        {
            "id": "66e2203bed00d9eb4437c104",
            "processedCodeId": "66e2203d5e8a273c725b6ad9"
        },
        {
            "id": "66e2203eed00d9eb4437c107",
            "processedCodeId": "66e22040d71fe3eba2a2fe35"
        },
        {
            "id": "66e22041ed00d9eb4437c10a",
            "processedCodeId": "66e22042100b750af7d7b4b0"
        },
        {
            "id": "66e22044ed00d9eb4437c10d",
            "processedCodeId": "66e22046549a71b23ebf6e6d"
        },
        {
            "id": "66e22047ed00d9eb4437c110",
            "processedCodeId": "66e220495265ff485c4114a5"
        }
    ],
    "errors": []
}
```


Slika 46. JSON odgovor

GET ruta /calculate/:userId prikazana na slici 48, poziva se za određenog korisnika unesenog kao parametar. Otvara se konekcija na bazu gdje tražimo tog korisnika, zatim se poziva funkcija processUserTokens(userId) (u main.js-u) koja je odgovorna za usporedbu tokena. Funkcija vraća listu komparacija za tog korisnika. Const processedCodeIds = new Set() kreira set za pohranu jedinstvenih ID-a procesiranog koda. Zatim iterira kroz results.comparisons i dodaje text1ProcessedCodeID i pattern1ProcessedCodeId iz svake komparacije u processedCodeIds set. U sljedećem koraku cilj je dohvat imena datoteka iz baze te ih točno mapirati uz ID procesiranog koda. Kod zatim prolazi kroz svaku usporedbu i formatira ju. Dohvaća imena datoteka koje su uspoređene iz filenameMap koristeći ID datoteke. Za svaku usporedbu izrađuje se objekt koji uključuje postotak sličnosti, ID-ove datoteka, nazive datoteka i meta podatke za tekst i pattern. Ako nešto pođe po krivom, server vraća status kod 500, inače šalje se JSON odgovor prikazan na slici 47.

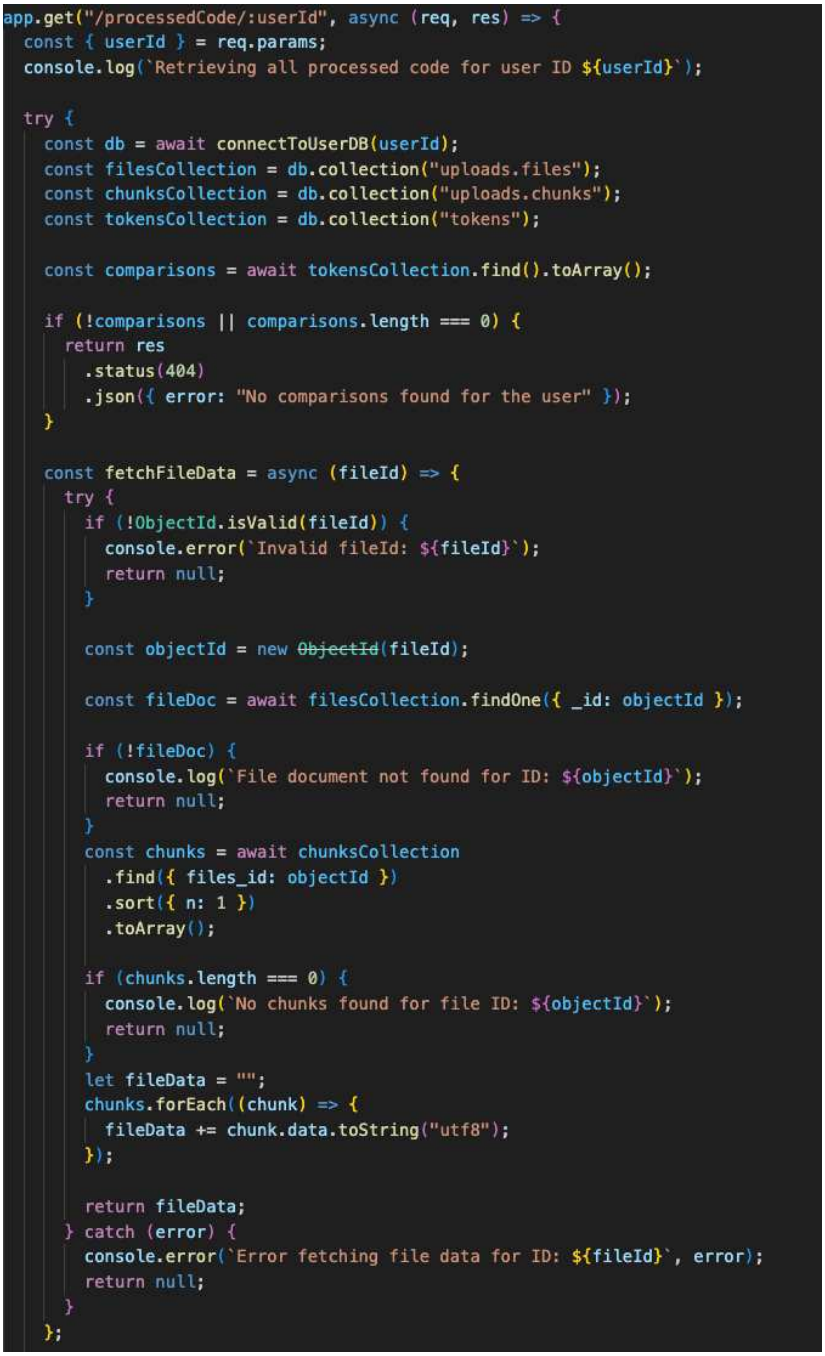
Slika 47. JSON odgovor rutae /calculate/:userId,

```
app.get("/calculate/:userId", async (req, res) => {
  const { userId } = req.params;
  try {
    const db = await connectToUserDB(userId);
    const filesCollection = db.collection("uploads.files");

    const results = await processUserTokens(userId);

    const processedCodeIds = new Set();

    results.comparisons.forEach((comparison) => {
      processedCodeIds.add(comparison.text1ProcessedCodeID);
      processedCodeIds.add(comparison.pattern1ProcessedCodeId);
    });

    const filesData = await filesCollection
      .find({
        _id: {
          $in: Array.from(processedCodeIds).map((id) => new ObjectId(id)),
        },
      })
      .toArray();

    const filenameMap = {};
    filesData.forEach((file) => {
      filenameMap[file._id.toString()] = file.filename;
    });

    const formattedComparisons = results.comparisons.map(
      (comparison, index) => {
        const textFilename =
          filenameMap[comparison.text1ProcessedCodeID] || "Unknown Filename";
        const patternFilename =
          filenameMap[comparison.pattern1ProcessedCodeId] || "Unknown Filename";

        return {
          similarityScore: comparison.similarityScore,
          text1ProcessedCodeID: comparison.text1ProcessedCodeID,
          pattern1ProcessedCodeId: comparison.pattern1ProcessedCodeId,
          [`text${index + 1}Filename`]: textFilename,
          [`pattern${index + 1}Filename`]: patternFilename,
          textMetadata: comparison.textMetadata || [],
          patternMetadata: comparison.patternMetadata || [],
        };
      }
    );

    res.status(200).json({ userId, comparisons: formattedComparisons });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Slika 48. Prikaz rute calculate

Prikazana GET ruta na slici 49, dohvaća procesirani kod za određenog korisnika na temelju njegovog userId-a. Procesirani kod je pohranjen u MongoDB bazi podataka i dohvaća se pomoću text1ProcessedCodeID i pattern1ProcessedCodeId. Nakon pronađenih svih kolekcija dohvaćaju se svi dokumenti (procesirani kod) za tog korisnika ili se javlja 404 greška. Funkcija fetchFileData odgovorna je za dohvat podataka iz baze, te onda rekonstruira chunk iz baze u

čitljivi string. Server iterira svakom usporedbom u comparisons listi. Za svaku usporedbu izvlači se text1ProcessedCodeID i pattern1ProcessedCodeId sa odgovarajućim kodom fetchFileData(textFileId) i fetchFileData(patternFileId), prikazano na slici 50. Dohvaćeni podatci se spremaju i šalju na frontend u JSON formatu prikazanim na slici 51.

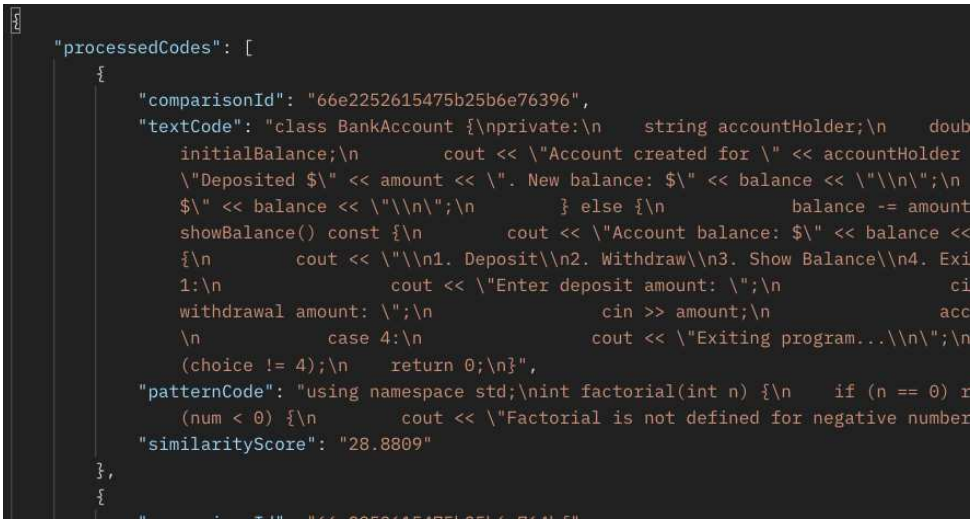


```
app.get("/processedCode/:userId", async (req, res) => {
  const { userId } = req.params;
  console.log(`Retrieving all processed code for user ID ${userId}`);

  try {
    const db = await connectToUserDB(userId);
    const filesCollection = db.collection("uploads.files");
    const chunksCollection = db.collection("uploads.chunks");
    const tokensCollection = db.collection("tokens");

    const comparisons = await tokensCollection.find().toArray();

    if (!comparisons || comparisons.length === 0) {
      return res
        .status(404)
        .json({ error: "No comparisons found for the user" });
    }

    const fetchFileData = async (fileId) => {
      try {
        if (!ObjectId.isValid(fileId)) {
          console.error(`Invalid fileId: ${fileId}`);
          return null;
        }

        const objectId = new ObjectId(fileId);

        const fileDoc = await filesCollection.findOne({ _id: objectId });

        if (!fileDoc) {
          console.log(`File document not found for ID: ${objectId}`);
          return null;
        }
        const chunks = await chunksCollection
          .find({ files_id: objectId })
          .sort({ n: 1 })
          .toArray();

        if (chunks.length === 0) {
          console.log(`No chunks found for file ID: ${objectId}`);
          return null;
        }
        let fileData = "";
        chunks.forEach((chunk) => {
          fileData += chunk.data.toString("utf8");
        });

        return fileData;
      } catch (error) {
        console.error(`Error fetching file data for ID: ${fileId}`, error);
        return null;
      }
    };
```

Slika 49. Početak GET processedCode rute

```
    for (const comparison of comparisons) {
      const textFileId = comparison.text1ProcessedCodeID;
      const patternFileId = comparison.pattern1ProcessedCodeId;

      const textCode = await fetchFileData(textFileId);
      const patternCode = await fetchFileData(patternFileId);

      processedCodes.push({
        comparisonId: comparison._id,
        textCode: textCode || "Text code not found",
        patternCode: patternCode || "Pattern code not found",
        similarityScore: comparison.similarityScore,
      });
    }

    res.status(200).json({ processedCodes });
  } catch (error) {
    console.error("Error retrieving processed codes:", error);
    res.status(500).json({ error: "Internal server error" });
  }
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Slika 50 .Kraj GET processedCode rute



Slika 51. JSON odgovor calclulate/userId

6. Frontend

Frontend se odnosi na dio aplikacije koji je korisniku vidljiv. Sastoji se od korisničkog sučelja (User Interface), koje uključuje izgled i dizajn vizualnih elemenata koje korisnik koristi. Uz predstavljanja informacija korisniku, frontend je također zadužen za komunikaciju s backend serverom, kao što je slanje zahtjeva ili dobivanje odgovora od strane backend servera.

6.1. Fetch API

Fetch API služi za komunikaciju s backendom, on je ugrađen u internet preglednik te šalje HTTP zahtjeve (GET, POST, itd.) prema serveru i obrađuje odgovore asinkrono. Za komunikaciju s backendom putem Fetch API-a služi nam services/index.js skripta.
Kao što je vidljivo na slici 52, prvi poziv prema backendu definiran je u export funkciji uploadFiles koja pomoću new FormData(), u kojem se uzlani dokumenti šalje POST prema backendu. Slanje je umotano u try i cath, ako dođe do greške prikazat će se adekvatna poruka, ako sve prođe i zahtjev je poslan JSON odgovor backenda šaljemo u return.

```
export async function uploadFiles(files) {
  const formData = new FormData();
  for (let i = 0; i < files.length; i++) {
    formData.append("files", files[i]);
  }

  try {
    const response = await fetch("http://localhost:3000/upload", {
      method: "POST",
      body: formData,
    });

    if (!response.ok) {
      throw new Error(`Error: ${response.statusText}`);
    }

    const data = await response.json();
    return data;
  } catch (error) {
    throw new Error(`Error uploading files: ${error.message}`);
  }
}
```

Slika 52. Funkcija uploadFiles

Prikazano na slici 53, sljedeći poziv prema backendu umotan je u funkciju calculateTokens koja kao parametar prima ID korisnika. U varijablu response sprema se odgovor GET zahtjeva, te se kao i u prijašnjem pozivu prikazuju greške te se podatci iz backenda vraćaju pomoću data.comparisons. Ovaj poziv služi sluzi za to da bi se ulazni podatci za tog korisnika izračunali na kraju u backenu kroz RKR-GST algoritam.

```
export async function calculateTokens(userId) {
  try {
    const response = await fetch(`http://localhost:3000/calculate/${userId}`, {
      method: "GET",
    });

    if (!response.ok) {
      throw new Error(`Error: ${response.statusText}`);
    }

    const data = await response.json();
    if (!data.comparisons) {
      throw new Error("Comparisons data is missing");
    }
    return data.comparisons;
  } catch (error) {
    throw new Error(`Error calculating tokens: ${error.message}`);
  }
}
```

Slika 53. Funkcija calculateTokens

Zadnji poziv koji radimo služi za prikaz procesiranog koda na frontendu. Funkcija fetchProcessedCodes prikazana na slici 54, koja kao parametar prima userId, šalje zahtjev za tog specifičnog korisnika putom GET metode dohvaćaju procesirani kodovi. I ovdje je također zahtjev (fetch), kao i prikaz grešaka, umotan u try i catch.

```
export async function fetchProcessedCodes(userId) {
  try {
    const response = await fetch(
      `http://localhost:3000/processedCode/${userId}`,
      {
        method: "GET",
      }
    );

    if (!response.ok) {
      throw new Error(`Error: ${response.statusText}`);
    }

    const data = await response.json();
    if (!data.processedCodes) {
      throw new Error("Processed codes data is missing");
    }
    return data.processedCodes;
  } catch (error) {
    throw new Error(`Error fetching comparison results: ${error.message}`);
  }
}
```

Slika 54. Funkcija fetchProcessedCodes

6.2. Ruter

Ruter u Vue.js aplikaciji odgovoran je za definiranje i upravljanje rutama (URL-ovima) do kojih korisnik može navigirati. Ruter je prikazan na slici 55, te povezuje različite komponente na određene rute, omogućujući navigaciju unutar aplikacije bez potrebe za ponovnim učitavanjem stranice.

Datoteka za ruter (index.js) postavlja Vue Router. Import FileUpload nam omogućuje navigaciju pogled FileUpload, koji nam je glavni pogled. U const rutes listi definirana je jedna ruta /upload koja kao komponentu sadrži prijašnji import FileUpload, te će se navigiranjem na tu rutu otvoriti taj pogled. Ruta (/:pathMatch(.*)*) je posebna ruta koja registrira bilo koju pogrešno ručno unesenu rutu te onda preusmjerava na početnu I jedinu rutu /uploads.

```
import { createRouter, createWebHistory } from "vue-router";
import FileUpload from "../views/FileUpload.vue";

const routes = [
  {
    path: "/upload",
    name: "file-upload",
    component: FileUpload,
  },
  {
    path: "/:pathMatch(.*)*",
    redirect: "/upload",
  },
];

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

Slika 55. Ruter

6.3. Glavni pogled

Pogled FileUpload.vue odgovoran je za unos datoteka u aplikaciji te prikazan je na slici 56. Pruža sučelje i funkcionalnosti koje omogućuju korisnicima odabir datoteka sa svog uređaja i njihovo učitavanje na server.

U template dijelu Postoje više funkcionalnosti za prikaz korisničkog sučelja. Nakon naslova stranice definiran je prostor u kojem se prikazani svi gumbovi koji su nam potrebni. Input type="file" poziva nam handleFileUpload, a to je funkcija koja poziva POST metodu prema backendu. Nakon toga imamo gumbove Upload Files i Calculate Tokens, gdje klikom na njih pozivaju svoju metodu uploadFilesToServer ili calculateTokensForUser. Gumbovi su postavljeni tako da se calculate ne može pritisnuti prije uploada. Klikom na jedan ili drugi gumb, dok se procesi ne završe, prikazan je loading spinner. Ispod je prikazan grafikon i poziva se komponenta za prikaz podudarenih kodova. U komponentu šalje se comparisonResults, koji se dobivaju nakon poziva calculateTokens prema backend serveru i šalje se userId.

```
<template>
  <div class="container">
    <h2>Upload Files</h2>

    <div class="upload-actions">
      <input type="file" multiple @change="handleFileUpload" />

      <div class="buttons-container">
        <button
          @click="uploadFilesToServer"
          :disabled="uploadLoading || calculateLoading"
        >
          Upload Files
        </button>

        <button
          @click="calculateTokensForUser"
          :disabled="!userId || uploadLoading || calculateLoading"
        >
          Calculate Tokens
        </button>

        <div
          class="loading-spinner"
          :class="{ visible: uploadLoading || calculateLoading }"
        ></div>
      </div>
    </div>

    <canvas id="similarityChart" width="400" height="200"></canvas>

    <ComparisonView
      v-if="comparisonResults.length"
      :comparisons="comparisonResults"
      :userId="userId"
    />
  </div>
</template>
```

Slika 56. Template fileUpload.vue

U skriptnom dijelu glavnog pogleda prikazanom na slici 57, nalaze se razne metode. HandleFileUpload, prije spomenut u template dijelu, služi san za spremanje odabranog ulaza u selectedFiles: []. UploadFilesToServer, pozvan u gumbu Upload Files, prikazuje alert ako je uneseno manje od 2 dokumenta. Metoda ima funkcionalnost prepoznavanja koji su trenutni dokumenti i koji su zadnje uneseni. Ako su zadnje uneseni dokumenti isti kao i trenutni dokumenti, prikazuje se alert. To se implementirano da korisnik ne može slati više zahtjeva ako su u prijašnjom slanju ti isti podatci već bili poslani. Glavni dio te metode je umotan u try i catch u kojem se poziva uploadFiles funkcija koja se nalazi u sercices.index.js i šalje POST zahtjev prema backendu.

```
data() {
  return {
    selectedFiles: [],
    lastUploadedFiles: [],
    userId: null,
    comparisonResults: [],
    lastProcessedUserId: null,
    uploadLoading: false,
    calculateLoading: false,
    chartInstance: null,
  };
},
methods: {
  handleFileUpload(event) {
    this.selectedFiles = Array.from(event.target.files);
  },
  async uploadFilesToServer() {
    if (this.selectedFiles.length < 2) {
      alert("Please select at least 2 files.");
      return;
    }

    const currentFileNames = this.selectedFiles
      .map((file) => file.name)
      .sort()
      .join(",");
    const lastFileNames = this.lastUploadedFiles
      .map((file) => file.name)
      .sort()
      .join(",");

    if (currentFileNames === lastFileNames) {
      alert("These files have already been uploaded.");
      return;
    }

    try {
      this.uploadLoading = true;
      const uploadData = await uploadFiles(this.selectedFiles);
      this.userId = uploadData.userId;
      this.lastUploadedFiles = [...this.selectedFiles];
      console.log("Files uploaded successfully. User ID:", this.userId);
    } catch (error) {
      console.error("Error uploading files:", error);
    } finally {
      this.uploadLoading = false;
    }
  },
```

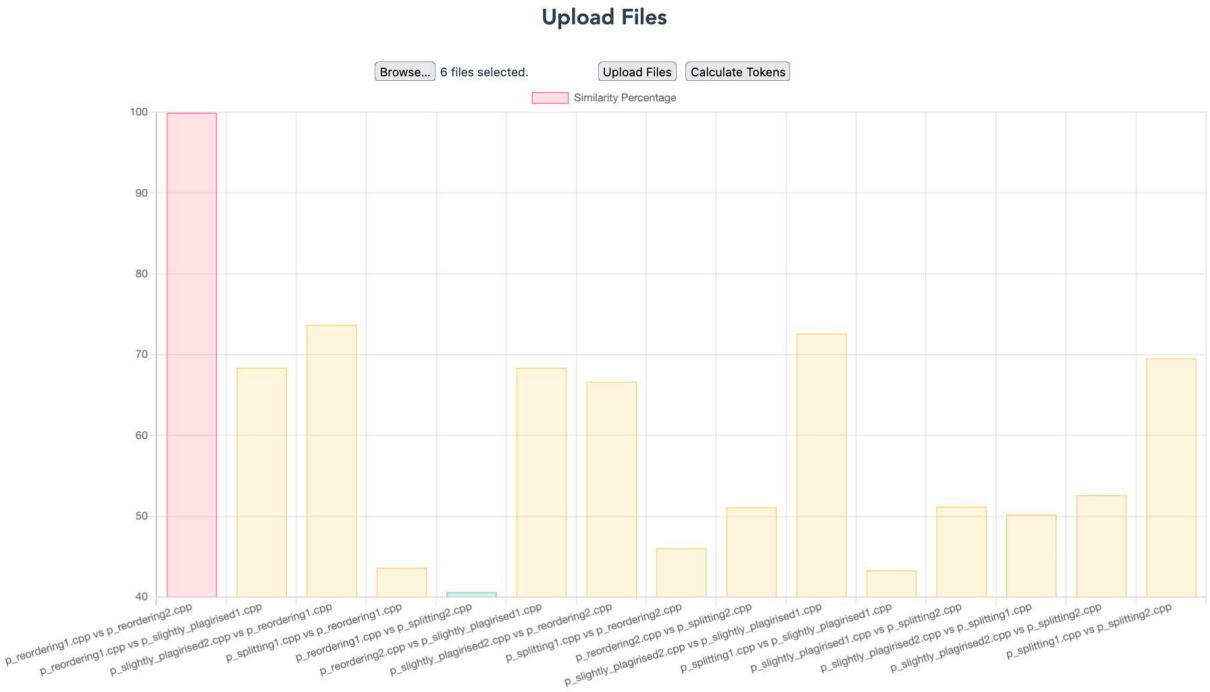
Slika 57. Prikaz funkcija handleFileUpload i UploadFilesToServer

Funkcija calculateTokensForUser poziva se klikom na gumb Calculate Tokens te je prikazana slikom 58. Funkcija provjerava postoji li korisnik (userId), koji se vraća kao dio rezultata uploadFiles. Ako postoji korisnik i ako je njegov ID isti kao prijašnji ID korisnika, to znači da opet pozvana funkcija calculateTokensForUser, tj. dvaput je kliknut gumb Calculate Tokens. To funkcionira za to jer sa svakim izvršenjem uploadFilesToServer se generira ID novog korisnika. Nadalje, ova funkcija pozva calculateTokens, koji šalje puti prema backendu s userId parametrom, te zatim se rezultati spremaju u comparisonResults. Nakon uspješno dobivenih rezultata od strane backenda, poziva se kreiranje grafikona.

```
async calculateTokensForUser() {
  try {
    if (!this.userId) {
      throw new Error("User ID is not available. Upload files first.");
    }

    if (this.userId === this.lastProcessedUserId) {
      alert("Calculation already completed for current data.");
      return;
    }

    this.calculateLoading = true;
    const comparisonResults = await calculateTokens(this.userId);
    this.comparisonResults = comparisonResults;

    this.lastProcessedUserId = this.userId;
    this.createBarChart();
    console.log("Token calculation completed.");
  } catch (error) {
    console.error("Error calculating tokens:", error);
  } finally {
    this.calculateLoading = false;
  }
},
```

Slika 58. Funkcija calculateTokensForUser

Grafikon prikazuje rezultate iz this.comparisonResults koji sadrži list podataka o komparaciji. Za svaki rezultat usporedbe, bit će prikazana 2 dokumenta koja su bila uspoređena (textFilename vs patternFilename), ako ime dokumenta ne postoji prikazat će se "Unknown File", što je i prikazano slikom 60. U varijablu similarityData spremaju se postotci podudaranja iz liste comparisonResults. Ti će se postotci prikazati grafičkim prikazom, kao što je i vidljivo na slici 59. Poziva se funkcija za prikaz boja s postotkom podudaranja kao argumentom jer različite razine podudaranja označene su različitim bojama, što je prikazano na slici 61. New Chart kreira grafički prikaz s definiranim argumentima.



Slika 59. Grafički prikaz korisničkog sučelja

```
createBarChart() {
  console.log("Comparison Results:", this.comparisonResults);
  if (!this.comparisonResults || this.comparisonResults.length === 0) {
    console.error("No comparison results available to display.");
    return;
  }

  const ctx = document.getElementById("similarityChart").getContext("2d");

  if (this.chartInstance) {
    this.chartInstance.destroy();
  }

  const labels = this.comparisonResults.map((result, index) => {
    const textFilename =
      result[`text${index + 1}Filename`] || "Unknown File";
    const patternFilename =
      result[`pattern${index + 1}Filename`] || "Unknown File";
    return `${textFilename} vs ${patternFilename}`;
  });

  const similarityData = this.comparisonResults.map((result) =>
    parseFloat(result.similarityScore)
  );

  const backgroundColors = this.generateBarColors(similarityData);

  this.chartInstance = new Chart(ctx, {
    type: "bar",
    data: {
      labels: labels,
      datasets: [
        {
          label: "Similarity Percentage",
          data: similarityData,
          backgroundColor: backgroundColors,
          borderColor: backgroundColors.map((color) =>
            color.replace("0.2", "1")
          ),
          borderWidth: 1,
        },
      ],
    },
    options: {
      scales: {
        y: {
          beginAtZero: true,
          title: {
            display: true,
            text: "Similarity Percentage",
          },
        },
        x: {
          title: {
            display: true,
            text: "Comparisons (File vs File)",
          },
        },
      },
    },
  });
},
```

Slika 60. Prikaz funkcije za generiranje grafikona

```
    generateBarColors(data) {
      return data.map((value) => {
        if (value >= 76) {
          return "rgba(255, 99, 132, 0.2)";
        } else if (value >= 41 && value <= 75) {
          return "rgba(255, 205, 86, 0.2)";
        } else {
          return "rgba(75, 192, 192, 0.2)";
        }
      });
    },
  },
};
</script>
```

Slika 61. Prikaz funkcije za odabir boja grafikona

6.4 Komponenta

ComparisonView komponenta odgovorna je za prikaz rezultata između 2 dokumenta, jedan pored drugog. Za svaki par dokumenta, komponenta će se jednom prikazati. Unutar komponente implementirana je logika označavanja plagijariziranog koda koristeći meta podatke koji su nastali procesom tokenizacije.

Kao što je vidljivo na slici 62, unutar vanjskog div-a omotani su svi rezultati komparacije. Unutar tog kontejnera, komponenta se prikazuje samo ako postoje rezultati komparacije, ako ne postoje prikazuje se "No plagiarism detected". Ako postoje rezultati komparacije, komponenta iterira kroz filteredComparisons listu koristeći v-for. Za svaku komparaciju prikazuje header (npr. "Comparison 1") i „Similarity score" uz koji se prikazuje postotak sličnosti. Što se tiče prikaza koda, 2 su koda prikazana, text i pattern dokumenti. Također prikazana su i imena dokumenata, a ispod toga, umotan u <pre>, prikazan je procesirani kod koristeći v-html direktivu. V-html nam je bitan zbog označavanja koda.

```
<template>
  <div class="comparison-container">
    <div v-if="filteredComparisons.length > 0">
      <div
        v-for="(result, index) in filteredComparisons"
        :key="index"
        class="comparison-item"
      >
        <h4>Comparison {{ index + 1 }}</h4>
        <h4>Similarity Score: {{ result.similarityScore }} %</h4>

        <div class="comparison-container">
          <div class="code-section">
            <p v-if="result[`text${index + 1}Filename`]">
              {{ result[`text${index + 1}Filename`] }}
            </p>
            <h5>Text File Processed Code:</h5>

            <pre
              class="code-box"
              v-if="textProcessedCodes[index]"
              v-html="
                getFinalHighlightedCode(
                  textProcessedCodes[index],
                  textFileMetadata[index],
                  result.similarityScore
                )
              "
            ></pre>
          </div>

          <div class="code-section">
            <p v-if="result[`pattern${index + 1}Filename`]">
              {{ result[`pattern${index + 1}Filename`] }}
            </p>
            <h5>Pattern File Processed Code:</h5>

            <pre
              class="code-box"
              v-if="patternProcessedCodes[index]"
              v-html="
                getFinalHighlightedCode(
                  patternProcessedCodes[index],
                  patternFileMetadata[index],
                  result.similarityScore
                )
              "
            ></pre>
          </div>
        </div>
      </div>
    </div>

    <div v-else>
      <p>No plagiarism detected.</p>
    </div>
  </div>
</template>
```

Slika 62. Template komponente

U skriptnom dijelu komponente prikazanim na slici 63, postoje potrebni „props“ comparisons i userId. Unutar “data” definirana su svojstva za spremanje potrebnih podataka. FilteredComparisons filtrira lokalne usporedbe kako bi se vratile samo one usporedbe s validnim postotkom sličnosti. Ovo izračunato svojstvo osigurava da se korisniku prikazuju samo relevantne usporedbe. Watcher sluša promjene u comparisons prop-u i odmah ažurira lokalnu kopiju localComparisons i pokreće loadProcessedCodes metodu svaki put kad je prop ažuriran. U metodama, funkcija loadProcessedCodes() prikazana na slici 64, briše podatke ako postoje u varijablama za spremanje teksta, patterna i meta podataka. Zatim, koristeći funkciju za upit pri backendu (fetchProcessedCodes (userId)), za tog određenog korisnika, prikuplja kod iz baze podataka kako bi se mogao prikazati na korisničkom sučelju. Dobivene kodove postavlja u liste (textProcessedCodes, patternProcessedCodes, textFileMetadata, patternFileMetadata).

```
<script>
import {
  fetchProcessedCodes,
  calculateTokens,
  uploadFiles,
} from "../services/index.js";
import Prism from "prismjs";
import "prismjs/themes/prism.css";
import "prismjs/components/prism-c";
import "prismjs/components/prism-cpp";

export default {
  props: ["comparisons", "userId"],
  data() {
    return {
      localComparisons: [],
      textProcessedCodes: [],
      patternProcessedCodes: [],
      textFileMetadata: [],
      patternFileMetadata: [],
      lastProcessedUserId: null,
      loading: false,
    };
  },
  computed: {
    filteredComparisons() {
      return this.localComparisons.filter((result) => result.similarityScore);
    },
  },
  watch: {
    comparisons: {
      immediate: true,
      handler(newComparisons) {
        this.localComparisons = [...newComparisons];
        this.loadProcessedCodes();
      },
    },
  },
```

Slika 63. Prikaz skript dijela komponente

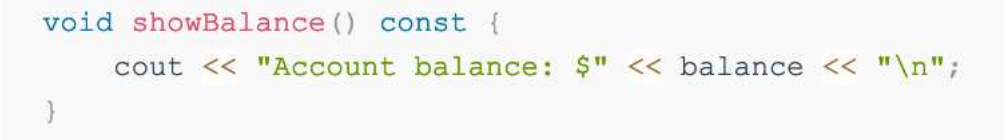
```
methods: {
  async loadProcessedCodes() {
    this.loading = true;
    try {
      this.textProcessedCodes = [];
      this.patternProcessedCodes = [];
      this.textFileMetadata = [];
      this.patternFileMetadata = [];

      const processedCodes = await fetchProcessedCodes(this.userId);

      processedCodes.forEach((processedCode, index) => {
        const comparison = this.localComparisons[index];

        if (comparison) {
          const textCode = processedCode.textCode || "";
          const patternCode = processedCode.patternCode || "";

          this.textProcessedCodes.push(textCode);
          this.patternProcessedCodes.push(patternCode);

          this.textFileMetadata.push(comparison.textMetadata || []);
          this.patternFileMetadata.push(comparison.patternMetadata || []);
        } else {
          console.error(`No comparison found for index: ${index}`);
        }
      });
    } catch (error) {
      console.error("Error loading processed codes:", error);
    } finally {
      this.loading = false;
    }
  },
```

Slika 64. Metoda loadProcessedCodes

Funkcija applySyntaxHighlighting(code) prikazana na slici 65, koristi Prism.js za označavanje sintakse danog koda. Što je drugačije od označavanja koda putem meta podataka. Prism.js postavlja kod u , te u taj span postavlja definiranu klasu kako bi bojom označio sintaksu kako je i prikazano na slici 66.

```
applySyntaxHighlighting(code) {
  return Prism.highlight(code, Prism.languages.cpp, "cpp");
},
```

Slika 65. Prism označavanje koda



```
void showBalance() const {
    cout << "Account balance: $" << balance << "\n";
}
```

Slika 66. Kod označen samo sa Prism

Funkcija wrapUnrecognizedTokens vidljiva na slici 67, postavlja one Prism.js tokene koje nisu prepoznati od Prism.js-a u . To nam je kasnije potrebno radi označavanja koda po meta podatcima. Svaki dio koda za koji postoje meta podatci, mora biti u . Prism.js to već radi za nas ali neke dijelove koda ne stavlja u .

Funkcija počinje tako što kreira privremeni div element (tempDiv) kako bi sigurno mogli manipulirati html stringom. Taj string sadrži kod koji je označio Prism.js. Koristeći innerHTM, pretvaramo html string u "DOM čvorove", pa ih možemo oblizati. Definirana je pomoćna funkcija koja ide kroz DOM čvorove unutar tempDiv-a. U toj funkciji ako je čvor tekstualnog formata (Node.TEXT_NODE) i ako sadržaj toga nije omotan u , onda ga omota s klasom "token unrecognized". Ako je trenutni čvor (Node.ELEMENT_NODE), tada funkcija rekurzivno putuje svom djecom tog čvora tako što poziva wrapNodes za svako dijete. Tako funkcija može istražiti cijelu DOM strukturu i primijeniti omotavanje u .
Nakon što su svi čvorovi obrađeni, tempDiv sada sadrži modificiranu HTML strukturu gdje je neprepoznati tekst omotan u . Funkcija pretvara DOM strukturu natrag u HTML niz i vraća je.

```
wrapUnrecognizedTokens(html) {
  const tempDiv = document.createElement("div");
  tempDiv.innerHTML = html;

  const wrapNodes = (node) => {
    if (node.nodeType === Node.TEXT_NODE) {
      let text = node.textContent;
      if (text.trim() && !node.parentNode.classList.contains("token")) {
        const span = document.createElement("span");
        span.className = "token unrecognized";
        span.textContent = text;
        node.replaceWith(span);
      }
    } else if (node.nodeType === Node.ELEMENT_NODE) {
      let child = node.firstChild;
      while (child) {
        const nextSibling = child.nextSibling;
        wrapNodes(child);
        child = nextSibling;
      }
    }
  };

  wrapNodes(tempDiv);
  return tempDiv.innerHTML;
},
```

Slika 67. Funkcija wrapUnrecognisedTokens

HighlightCodeWithMetadataAndScore funkcija prikazana na slici 68, dizajnirana je za označavanje koda putem meta podataka iz backenda. U meta podatcima nalaze se linija i stupac, te se po njima označuje kod s adekvatnim bojama. Funkcija počinje provjerom postoji li html, meta podaci ili postotak sličnosti. Ako bilo koji od njih nedostaje, funkcija vraća izvorni html. Kao što je i prije objašnjeno, pretvaramo HTML u DOM čvorove, a zatim funkcija dodjeljuje klase po određenom postotku. Crvena >76%, žuta između (41% i 75%) i zelena 40% ili niže. Funkcija iterira meta podatcima i rekurzivno putuje DOM strukturom, tražeći tekst čvorove, prati trenutnu liniju i stupac da bi se pronašao točan dio koda kojeg treba označiti. Nakon što su svi čvorovi obrađeni, funkcija pretvara modificirani DOM (tempDiv) natrag u HTML string.

```
highlightCodeWithMetadataAndScore(html, metadata, similarityScore) {
  if (!html || !metadata || !Array.isArray(metadata)) {
    return html;
  }

  const tempDiv = document.createElement("div");
  tempDiv.innerHTML = html;

  let highlightClass = "";
  if (similarityScore >= 76) {
    highlightClass = "highlight-high";
  } else if (similarityScore >= 41 && similarityScore <= 75) {
    highlightClass = "highlight-moderate";
  } else {
    highlightClass = "highlight-low";
  }

  metadata.forEach(({ line, column }) => {
    let found = false;
    let currentLine = 1;
    let position = 0;

    const walkNodes = (node) => {
      if (node.nodeType === Node.TEXT_NODE) {
        let text = node.textContent;

        for (let i = 0; i < text.length; i++) {
          if (currentLine === line && position === column - 1) {
            let tokenParent = node.parentNode;
            while (
              tokenParent &&
              !tokenParent.classList.contains("token")
            ) {
              tokenParent = tokenParent.parentNode;
            }

            if (
              tokenParent &&
              !tokenParent.classList.contains("highlight")
            ) {
              console.log(
                `Applying ${highlightClass} to: "${tokenParent.textContent.trim()}" at line ${currentLine}, column ${
                  position + 1
                }`
              );
              tokenParent.classList.add("highlight", highlightClass);
              found = true;
            }
            return;
          }
          position++;
          if (text[i] === "\n") {
            currentLine++;
            position = 0;
          }
        }
      } else if (node.nodeType === Node.ELEMENT_NODE) {
        for (
          let child = node.firstChild;
          child;
          child = child.nextSibling
        ) {
          walkNodes(child);
          if (found) break;
        }
      }
    };

    walkNodes(tempDiv);
  });

  return tempDiv.innerHTML;
},
```

Slika 68. Funkcija highlightCodeWithMetadataAndScore

GetFinalHighlightedCode je “top level” funkcija koja se poziva unutar <template> s ulaznim kodom, meta podatcima i postotkom sličnosti kao ulaznim podatcima. Te se onda kroz navedenu funkciju pozivaju prijašnje objašnjene funkcije što je i prikazano na slici 69.

```
getFinalHighlightedCode(code, metadata, similarityScore) {
  const highlightedCode = this.applySyntaxHighlighting(code);

  const wrappedCode = this.wrapUnrecognizedTokens(highlightedCode);

  return this.highlightCodeWithMetadataAndScore(
    wrappedCode,
    metadata,
    similarityScore
  );
},
```

Slika 69. Funkcija getFinalHighlightedCode

7. Demonstracija funkcionalnosti

7.1. Preimenovanje varijabla

Na slici 70 vidljiv je pokušaj plagijarizacije tako što su promijenjena imena varijabla, sadržaj cout-a i ime funkcije. Vidljivo je da se takvim prikrivanjem plagijarizacije algoritam ne može prevariti, čemu je zapravo zaslužan proces tokenizacije.

p_plagiarised1.cpp

Text File Processed Code:

```
using namespace std;
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    if (num < 0) {
        cout << "Factorial is not defined for negative numbers.\n";
    } else {
        cout << "Factorial of " << num << " is " << factorial(num) <<
endl;
    }
    return 0;
}
```

p_plagiarised2.cpp

Pattern File Processed Code:

```
using namespace std;
int findFactorial(int number) {
    if (number == 0) return 1;
    return number * findFactorial(number - 1);
}
int main() {
    int inputNumber;
    cout << "Please input a number: ";
    cin >> inputNumber;
    if (inputNumber < 0) {
        cout << "Negative numbers don't have a factorial.\n";
    } else {
        cout << "The factorial of " << inputNumber << " is " <<
findFactorial(inputNumber) << endl;
    }
    return 0;
}
```

Slika 70. Prikaz plagijarizacije s mijenjanjem imena varijabla

7.2. Promjena redoslijeda

Na prikazanoj slici 71, vidljivo je da promjena redoslijeda nije bitna. Zamijenjena je main funkcija s funkcijama factorial i square.

p_reordering1.cpp

Text File Processed Code:

```
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Square of " << num << " is " << square(num) << endl;
    cout << "Factorial of " << num << " is " << factorial(num) <<
endl;
    return 0;
}
int square(int x) {
    return x * x;
}
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

p_reordering2.cpp

Pattern File Processed Code:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
int square(int x) {
    return x * x;
}
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Factorial of " << num << " is " << factorial(num) <<
endl;
    cout << "Square of " << num << " is " << square(num) << endl;
    return 0;
}
```

Slika 71. Prikaz plagijarizacije mijenjanjem redoslijeda

7.3. Refaktoriranje funkcija

U ovom se primjeru prikazanom na slikama 72 i 73, koriste malo napredniji način prekrivanja plagijarizma. U dokumentu tekst, vidljivo je da postoje dvije funkcije, jedna za računanje faktorijela koji postoji i u patternu i jedna za zbrajanje. Vidljivo je da je da su prepoznate funkcije za izračun faktorijela kao plagijarizam. Nadalje, na prvu se možda može smatrati da su ostali dijelovi koda krivo označeni, ali uzimajući u kontekst opisani rad algoritma, ali i malo boljom obzervacijom može se zaključiti da je označeni kod točan. Gledajući sekvencu koda u patternu (specifično main funkcija), uzimajući u obzir preimenovanje varijabla i cout-a, označeni su kodovi isti. I u jednom i u drugom postoji sljedeća sekvenca: cout string vrijednosti, cin broja, if, uvjet u if-u, cout string vrijednosti, else, cout string vrijednosti, return.

p_slightly_plagirised2.cpp

Text File Processed Code:

```
using namespace std;
int addition(int a, int b) {
    return a + b;
}
int computeFactorial(int x) {
    if (x == 0) return 1;
    return x * computeFactorial(x - 1);
}
int main() {
    int num1;

    int num2;

    int inputNumber;

    cout << "Enter two numbers to add: ";
    cin >> num1 >> num2;
    cout << "The sum of " << num1 << " and " << num2 << " is " <<
addition(num1, num2) << endl;
    cout << "Please input a number for factorial: ";
    cin >> inputNumber;
    if (inputNumber < 0) {
        cout << "Negative numbers don't have a factorial.\n";
    } else {
        cout << "The factorial of " << inputNumber << " is " <<
computeFactorial(inputNumber) << endl;
    }
    return 0;
}
```

Slika 72. Prikaz pokušaja plagijarizacije (tekst)

p_slightly_plagirised1.cpp

Pattern File Processed Code:

```
using namespace std;
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    if (num < 0) {
        cout << "Factorial is not defined for negative numbers.\n";
    } else {
        cout << "Factorial of " << num << " is " << factorial(num) <<
endl;
    }
    return 0;
}
```

Slika 73. Prikaz pokušaja plagijarizacije (pattern)

7.4. Razdvajanje funkcija

Na slici 74 koja je slična prijašnjem primjeru, prikazan je pokušaj plagijarizacije razdvajanjem u funkcije. Dokument u patternu razdvojen je u više funkcija prikazanih u teksu. Analizirajući označeni kod, vidljivo je da je prikaz sličnosti točan u kontekstu algoritma. Ovakvo parafraziranje koda, alat Copyleaks rješava umjetnom inteligencijom. Iako se kod raščlanio na funkcije u tekstu, ipak se određeni dio prepoznao kao plagijarizam. Same deklaracije funkcije nisu označene jer ne postoje u patternu ali ključni dijelovi funkcije (int length, cout, cin length) prepoznali su se kao isti.

Slika 74. Prikaz plagijarizacije razdvajanjem funkcija

7.5. Umetanje mrtvog koda

Umetanjem mrtvog koda između plagijariziranog, kao što je prikazano na slici 75, ne može prevariti algoritam. Iako su uočljive male „greške" gdje je iznad main funkcije, u tekstu označena varijabla „b" i „}" umjesto varijable „radius" i „}" u funkciju calculatePerimeter. RKR-GST algoritam prioritizira označavanje najduljih podstringova.Taj je dio koda pronašao kao najdulji podstring. Kad se pronašao drugi podstring (od using namespace do * radius), došlo je do preklapanje (occlusion) jer su „radius" i „}" u patternu već označen u prijašnjoj iteraciji.

p_deadcode1.cpp

Text File Processed Code:

```
using namespace std;
double calculateArea(double radius) {
    return M_PI * radius * radius;
}
double calculatePerimeter(double radius) {
    return 2 * M_PI * radius;
}
void printHelloWorld() {
    cout << "Hello World!" << endl;
}
int addTwoNumbers(int a, int b) {
    return a + b;
}
int main() {
    double radius;
    cout << "Enter the radius of the circle: ";
    cin >> radius;
    cout << "Area: " << calculateArea(radius) << endl;
    cout << "Perimeter: " << calculatePerimeter(radius) <<
endl;
    printHelloWorld();
    return 0;
}
```

p_deadcode2.cpp

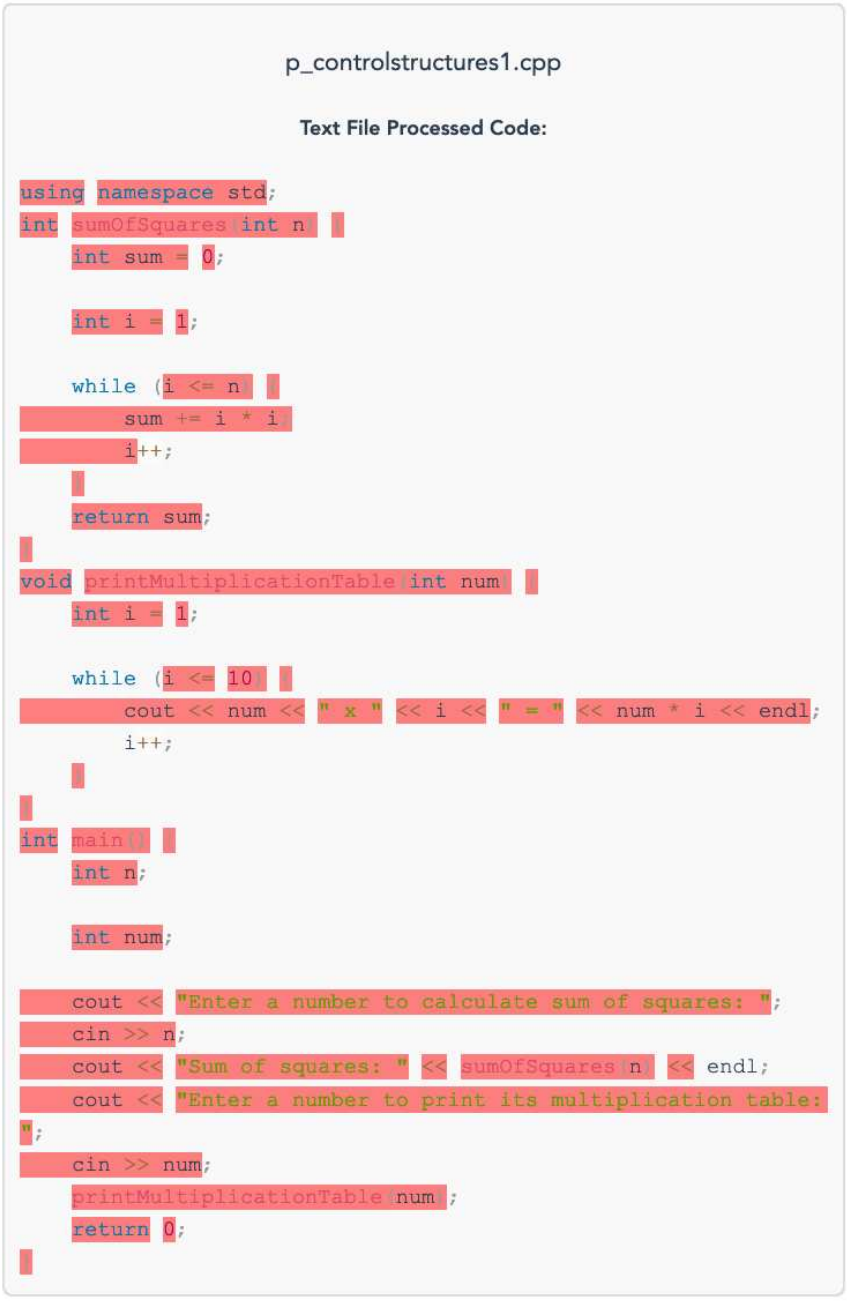
Pattern File Processed Code:

```
using namespace std;
double calculateArea(double radius) {
    return M_PI * radius * radius;
}
double calculatePerimeter(double radius) {
    return 2 * M_PI * radius;
}
int main() {
    double radius;
    cout << "Enter the radius of the circle: ";
    cin >> radius;
    cout << "Area: " << calculateArea(radius) << endl;
    cout << "Perimeter: " << calculatePerimeter(radius) <<
endl;
    return 0;
}
```

Slika 75. Prikaz plagijarizacije mrtvim kodom

7.6. Mijenjanje kontrolnih struktura

Postotak sličnosti veći je od 90%, kao što je prikazano na slikama 76 i 77. Ulazni kodovi su prepoznati kao plagijarizam ali ključne riječi while i for smatraju se kao dva različita tokena, te zato nisu označeni.



p_controlstructures1.cpp

Text File Processed Code:

```
using namespace std;
int sumOfSquares(int n) {
    int sum = 0;

    int i = 1;

    while (i <= n) {
        sum += i * i;
        i++;
    }
    return sum;
}
void printMultiplicationTable(int num) {
    int i = 1;

    while (i <= 10) {
        cout << num << " x " << i << " = " << num * i << endl;
        i++;
    }
}
int main() {
    int n;

    int num;

    cout << "Enter a number to calculate sum of squares: ";
    cin >> n;
    cout << "Sum of squares: " << sumOfSquares(n) << endl;
    cout << "Enter a number to print its multiplication table:
";
    cin >> num;
    printMultiplicationTable(num);
    return 0;
}
```

Slika 76. Prikaz plagijarizacije mijenjanjem kontrolih struktura (tekst)

p_controlstructures2.cpp

Pattern File Processed Code:

```
using namespace std;
int sumOfSquares(int n) {
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        sum += i * i;
    }
    return sum;
}
void printMultiplicationTable(int num) {
    for (int i = 1; i <= 10; i++) {
        cout << num << " x " << i << " = " << num * i << endl;
    }
}
int main() {
    int n;

    int num;

    cout << "Enter a number to calculate sum of squares: ";
    cin >> n;
    cout << "Sum of squares: " << sumOfSquares(n) << endl;
    cout << "Enter a number to print its multiplication table:
";
    cin >> num;
    printMultiplicationTable(num);
    return 0;
}
```

Slika 77. Prikaz plagijarizacije mijenjanjem kontrolih struktura (pattern)

8. Zaključak

Cilj ovog rada bio je razviti web aplikaciju za detekciju plagijata C++ koda. Aplikacija korisnicima omogućuje unos koda, te se metodom tokenizacije tokenizira kod pa se zatim šalje u RKR-GST algoritam koji uspoređuje kod i pronalazi potencijalni plagijarizam. U korisničkom sučelju aplikacije rezultati su prikazani grafičkim prikazom i označena je točna sličnost unesenog koda. Korištene su razne tehnologije za izradu aplikacije od Vue.js-a za korisničko sučelje, Express i Node.js za backend, Flex za tokenizaciju, a MongoDB, kao nerelacijska baza podataka, služio je kao baza za pohranu rezultata usporedbe.

U završnom radu je dokumentirana cijela aplikacija, kako se koristi i za što služe koji dijelovi koda. Detaljno je opisano kako radi RKR-GST algoritam, na koji način funkcionira RKR heshiranje, što se točno heshira i koja je prednost toga, koji su uvjeti za rekurziju i zašto te sam proces označavanja podudarenih tokena. Opisan je proces tokenizacije i pravila po kojima će se tokeni izraditi, koje se ključne riječi ignoriraju, zbog čega se svi tipovi podataka tokeniziraju kao numeričke ili srting vrijednosti, te je opisana početna skripta za preprocesiranje koda. Iako aplikacija služi svojoj svrsi te se može koristiti umjesto klasičnih ili komercijalnih alata, i ona ima svoja ograničenja. Jedno ograničenja može biti vremenska složenost, koja u najgorem slučaju može dostići $O(n^3)$. U najgorem slučaju algoritam mora usporediti sva moguća maksimalna podudaranja. Ova složenost nastaje kada algoritam naiđe na scenarije u kojima je moguće više maksimalnih podudaranje, što dovodi do puno usporedba podstringova. Točnije, kada postoji puno mogućih maksimalnih podudaranja različitih duljina, algoritam može završiti izvođenjem velikog broja usporedbi. To se obično događa kada postoji gusto preklapanje podudarenih podstringova u dva seta tokena koji se uspoređuju, posebno kada su podudaranja fragmentirana ili neravnomjerno raspoređena po stringovima. Međutim, u praksi, algoritam radi mnogo bliže linearnom vremenu ili $O(n^2)$ zbog optimizacija poput heshiranja i greedy metode. Neke od prednosti aplikacije ovog završnog rada su modernije i preglednije sučelje što omogućuje lakše korištenje te bolje optimiziran algoritam od ostalih rješenja. U budućnosti bi aplikacija mogla biti nadograđena na nekoliko ključnih područja. Integracija umjetne inteligencije mogla bi omogućiti napredniju detekciju plagijata, kao što je prepoznavanje parafraziranog koda ili dublje analize strukture koda. Također, implementacijom UI bit će moguće pretraživanja plagijata u javnim kodnim repozitorijima, te bi onda bila sposobna prepoznati sličnosti s kodovima koji su javno dostupni na internetu.

Unatoč određenim ograničenjima, aplikacija predstavlja moderno rješenje koje doprinosi bržem i jednostavnijem prepoznavanju plagijata u C++ kodu, s mogućnošću daljnjih poboljšanja i proširenja u budućnosti.

GitHub poveznica: https://github.com/LegMateo/RKR-GST

Literatura

1. Chapman University. (2017, October 25). *Turnitin: Beyond plagiarism detection*. Chapman University. https://blogs.chapman.edu/academics/2017/10/25/turnitin-3/

2. Codecademy. (n.d.). *What is Express.js?* Codecademy. Retrieved September 8, 2024, from https://www.codecademy.com/article/what-is-express-js

3. Copyleaks. (n.d.). *Is Copyleaks able to check if source code has been plagiarized?* Copyleaks Help Center. Retrieved September 13, 2024, from https://help.copyleaks.com/hc/en-us/articles/23768919877133-Is-Copyleaks-able-to-check-if-source-code-has-been-plagiarized

4. Copyleaks. (n.d.). *Is Copyleaks able to check if source code has been plagiarized?* Retrieved September 8, 2024, from https://help.copyleaks.com/hc/en-us/articles/23768919877133-Is-Copyleaks-able-to-check-if-source-code-has-been-plagiarized

5. Durić, Z., & Gašević, D. (2012). *A source code similarity system for plagiarism detection*. The Computer Journal, Advance Access. https://doi.org/10.1093/comjnl/bxs01

6. GitHub. (n.d.). *Get started with GitHub documentation*. GitHub Documentation. Retrieved September 8, 2024, from https://docs.github.com/en/get-started

7. Kopecky, C. (2020, September 30). *What is Node.js? A beginner's introduction to JavaScript runtime*. Educative. https://www.educative.io/blog/what-is-nodejs#what-is

8. Kumar, M. (2019, January 24). [2021] *MOSS code plagiarism checker | Complete and easiest way within 6 minutes using CMD* [Video]. YouTube. https://www.youtube.com/watch?v=5CyOTf2DHSg&t=322s

9. MongoDB. (n.d.). *Get started with MongoDB*. MongoDB Documentation. Retrieved September 8, 2024, from https://www.mongodb.com/basics/get-started

10. Peachy Essay. (2021, April 27). *Turnitin plagiarism checker: What you need to know*. Peachy Essay. https://peachyessay.com/blogs/turnitin-plagiarism-checker/

11. Prechelt, L., Malpohl, G., & Philippsen, M. (2002). *Finding plagiarisms among a set of programs with JPlag*. Journal of Universal Computer Science, 8(11), 1016-1038. https://doi.org/10.3217/jucs-008-11-1016

12. Prechelt, L., Malpohl, G., & Philippsen, M. (2002). *An example part of a JPlag results display page for one pair of programs*. ResearchGate. Retrieved September 11, 2024, from https://www.researchgate.net/figure/An-example-part-of-a-JPlag-results-display-page-for-one-pair-of-programs_fig2_36451198

13. Prechelt, L., Malpohl, G., & Philippsen, M. (2002). *The top of an example JPlag results overview page*. ResearchGate. Retrieved September 11, 2024, from https://www.researchgate.net/figure/The-top-of-an-example-JPlag-results-overview-page_fig1_36451198

14. Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). *Winnowing: Local algorithms for document fingerprinting*. Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 76-85. https://doi.org/10.1145/872757.872770

15. Stanford University. (n.d.). *Flex in a nutshell*. CS 143: Compilers [Course handout]. Retrieved September 8, 2024, from https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf

16. Turnitin. (2024, July 15). *Welcome to Turnitin Guides*. Turnitin. https://guides.turnitin.com/hc/en-us/articles/24008452116749-Welcome-to-Turnitin-Guides

17. Visual Studio Code. (n.d.). *Why Visual Studio Code*. Visual Studio Code Documentation. Retrieved September 8, 2024, from https://code.visualstudio.com/Docs/editor/whyvscode

18. Vue.js. (n.d.). *Introduction Vue.js*. Retrieved September 8, 2024, from https://vuejs.org/guide/introduction.html

19. Wise, M. J. (1993). *String similarity via greedy string tiling and running Karp-Rabin matching*. Unpublished manuscript, Basser Department of Computer Science, University of Sydney, Australia.

Popis slika