SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni prijediplomski studij Računarstvo

ANTE ALJINOVIĆ

Z A V R Š N I R A D

WEB APLIKACIJA ZA VIZUALIZACIJU I PRETRAGU GRAFOVA KORISTEĆI NEO4J I D3.JS

Split, rujan 2024.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni prijediplomski studij Računarstvo

Predmet: Objektno programiranje

Z A V R Š N I R A D

Kandidat: Ante Aljinović

Naslov rada: Web aplikacija za vizualizaciju i pretragu grafova koristeći Neo4j i d3.js

Mentor: Ljiljana Despalatović, viši predavač

Split, rujan 2024.

Sadržaj

Sažetak

Ovaj rad opisuje razvoj web aplikacije za vizualizaciju i pretragu grafova koristeći tehnologije Neo4j i D3.js.

U radu su objašnjeni osnovni pojmovi vezani uz grafove, te određeni algoritmi korišteni za pretragu grafova. Također su predstavljeni Neo4j graf baza podataka specijalizirana za pohranu grafova te D3.js kao JavaScript biblioteka za izradu dinamičnih i prilagodljivih vizualizacija grafova. Navedene su funkcionalnosti aplikacije poput interakcije i izmjene elemenata grafa, te učitavanja i spremanja grafova. Također su objašnjeni korišteni algoritmi i opisan proces animacije algoritama. Uz Neo4j i D3.js, korištene su i dodatne tehnologije kao što su HTML, CSS i JavaScript za razvoj korisničkog sučelja.

Cilj rada je prikazati kako se kombinacijom ovih alata može stvoriti interaktivna aplikacija koja omogućava korisnicima stvaranje i pretragu grafova.

Ključne riječi: Algoritam, D3.js, graf, Neo4j, web aplikacija

Summary

Web application for graph visualization and search using Neo4j and d3.js

This paper describes the development of a web application for graph visualization and search using Neo4j and D3.js technologies.

It explains the basic concepts related to graphs and certain algorithms used for graph search. Additionally, it presents the Neo4j graph database specialized in graph storage and D3.js as a JavaScript library for creating dynamic and adaptable graph visualizations. The main functionalities of the application, such as interaction and modification of graph elements, as well as loading and saving graphs, are listed. The algorithms used are also explained, and the animation process of these algorithms is described. Alongside Neo4j and

d3.js, additional technologies such as HTML, CSS, and JavaScript were used for the development of the user interface.

The aim of this paper is to demonstrate how the combination of these tools can create an interactive application that allows users to create and search graphs.

Keywords: Algorithm, d3.js, graph, Neo4j, web application

1. Uvod

Vizualno razumijevanje algoritama može značajno poboljšati proces učenja. Algoritmi za pretragu i optimizaciju grafova često su kompleksni i zahtijevaju duboko razumijevanje kako bi se pravilno shvatili i primijenili. Motivacija za ovaj rad proizašla je iz potrebe za aplikacijom koja će korisnicima olakšati proces učenja i razumijevanja algoritama na jednostavan i intuitivan način.

Aplikacija omogućuje korisnicima da kreiraju vlastite grafove i zatim odaberu jedan od ponuđenih algoritama koji će biti animiran kako bi se prikazao proces izvršavanja algoritma na zadanom grafu. Vizualizacija algoritama ne samo da pomaže u razumijevanju koraka koje algoritam poduzima, već i pokazuje njegovu učinkovitost.

Cilj rada je razviti interaktivnu web aplikaciju koja će koristiti tehnologije Neo4j i D3.js za vizualizaciju grafova. Neo4j je graf baza podataka specijalizirana za pohranu i obradu grafova, dok je D3.js JavaScript biblioteka namijenjena izradi dinamičnih i prilagodljivih vizualizacija. Korištenjem ovih tehnologija, stvorena je interaktivna i korisnički prilagođena aplikacija koja omogućava korisnicima jednostavno upravljanje nad elementima grafa. Pored glavnih tehnologija, korišteni su i HTML, CSS te JavaScript za razvoj interaktivnog korisničkog sučelja i samih algoritama.

Rad je podijeljen u šest cjelina. Nakon uvoda, drugo poglavlje pruža uvod u osnovne pojmove vezane uz grafove. Treće poglavlje daje uvid u tehnologije korištene za razvoj ove aplikacije, uključujući Neo4j i d3.js, te dodatne alate poput HTML-a, CSS-a i JavaScripta. Četvrto poglavlje obuhvaća proces razvoja aplikacije, te opisuje njene glavne funkcionalnosti. U petom poglavlju objašnjena je implementacija algoritama koji se koriste u ovoj aplikaciji te njihov proces vizualizacije, dok je zaključak rada, koji sumira postignute ciljeve i razmatra mogućnosti budućeg razvoja te nadogradnje aplikacije, predstavljen u završnom, šestom poglavlju.

2. Osnovni pojmovi vezani uz teoriju grafova

Osnova ovog rada su grafovi, te je za potpuno razumijevanje rada funkcionalnosti ove aplikacije potrebno poznavati osnovne pojmove teorije grafova.

Graf je uređena trojka $G = (V, E, \varphi)$, skupa vrhova $V=V(G) \neq \emptyset$, skupa bridova $E=E(G)$ takvog da je $V \cap E = \emptyset$ i funkcije φ koja svakom bridu $e \in E$ pridružuje dva (ne nužno različita) vrha $u,v \in V$ [1]. Graf može biti neusmjeren ili usmjeren. Kod neusmjerenog grafa, brid $e \in E$ je neuređeni par $e = \{u, v\}$, gdje su $u, v \in V$. Vrhovi u i v su krajevi od e [2]. Kod usmjerenog grafa, usmjereni brid $e \in E$ je uređeni par $e = (u, v)$, gdje su $u, v \in V$. Vrh u je početni vrh, a v krajnji vrh usmjerenog brida e [2]. Graf može biti težinski ili bestežinski. Graf je jednostavan ako nema petlji ni višestrukih bridova. Ako je svakom bridu e grafa G pridijeljena težina, govorimo o težinskom grafu. Ova aplikacija radi isključivo sa grafovima bez petlji što znači da će gore spomenuta funkcija φ svakom bridu $e \in E$ pridruživati isključivo dva različita vrha. Također u aplikaciji je moguće imati isključivo jedan brid između dva vrha.

Incidenti brid za određeni vrh je brid koji je povezan s tim vrhom. Drugim riječima, incidenti bridovi su oni bridovi koji dodiruju vrh. Poznavanje incidentnih bridova pomaže u određivanju stupnja vrha i analizi lokalne strukture grafa.

Stupanj $d(v)$ **vrha** $v \in V(G)$ je broj bridova grafa G incidentnih s vrhom v [1].

Tri osnovne strukture podataka za reprezentaciju grafa su **matrica susjedstva**, **matrica incidencije** i **lista susjedstva** [2].

Matrica incidencije grafa G je matrica $M=M(G)=[m_{ij}]$, gdje je $m_{ij} \in \{0,1,2\}$ broj koliko puta su v_i i e_j incidentni [2]. To je način predstavljanja grafa, gdje su redovi matrice vrhovi, a stupci bridovi. Element na poziciji (i, j) označava koliko puta su incidentni vrh v_i i brid e_j. U ovoj aplikaciji nije moguće raditi petlje što znači da je $m_{ij} \in \{0,1\}$.

Matrica susjedstva grafa G matrica $A(G)=[a_{ij}]$, gdje je a_{ij} broj bridova koji spajaju vrhove v_i i v_j. Matrica susjedstva je simetrična matrica (tj. $a_{ij} = a_{ji}$), čiji su elementi nenegativni cijeli brojevi. To je način predstavljanja grafa, u kojem redovi i stupci predstavljaju vrhove grafa, a element na poziciji (i, j) označava postoji li brid između vrha v_i i vrha v_j. Ako je graf jednostavan, ova matrica sadrži samo nule i jedinice te nule na glavnoj dijagonali.

Lista susjedstva je način predstavljanja grafa gdje se za svaki vrh zapisuje lista susjednih vrhova, odnosno vrhova koji su direktno povezani bridom s tim vrhom.

Put u grafu je niz vrhova i bridova u kojem svaki brid povezuje susjedne vrhove iz tog niza. Put počinje u jednom vrhu, prolazi kroz jednog ili više bridova povezanih s tim vrhovima te završava u nekom drugom (ili istom) vrhu.

Eulerov put je put kroz graf koji prolazi kroz svaki brid točno jednom, a **Eulerov ciklus** je put kroz graf koji prolazi kroz svaki brid grafa točno jednom i vraća se u početni vrh. Odnosno, to je **Eulerov put** koji se vraća u početni vrh. Ako u grafu postoji **Eulerov ciklus**, tada je to **Eulerov graf**.

Komponenta povezanosti u grafu je podgraf u kojem su svi vrhovi međusobno povezani te nemaju veze s vrhovima izvan te komponente. Svaka komponenta povezanosti sadrži vrhove između kojih postoji put, dok ne postoji put prema vrhovima iz drugih komponenti.

Brid je **rezni** ako se njegovim izbacivanjem iz grafa, graf raspada na više komponenti povezanosti.

3. Korištene Tehnologije

3.1 Neo4j

Neo4j je graf baza podataka koja se koristi za pohranu, upravljanje i pretraživanje podataka koji su međusobno povezani na složene načine [3]. Umjesto tradicionalnog pristupa tabličnim strukturama kod bazi podataka, Neo4j koristi grafički model podataka, gdje su podaci predstavljeni čvorovima (engl. *nodes*) i vezama (engl. *relationships*), omogućujući prirodnije i učinkovitije modeliranje složenih mreža podataka. Neo4j koristi Cypher, deklarativni jezik koji omogućuje jednostavno izražavanje složenih upita nad grafičkom strukturom.

Zbog sposobnosti modeliranja i analize složenih odnosa među podacima, Neo4j je postao ključan alat u raznim industrijama za rješavanje problema koji zahtijevaju razumijevanje povezanosti u bazi podataka.

3.2 D3.js

D3.js (Data-Driven Documents) je JavaScript biblioteka koja omogućuje stvaranje dinamičnih i interaktivnih vizualizacija podataka unutar web preglednika [4]. Umjesto unaprijed definiranih grafika, D3.js pruža fleksibilne alate koji pružaju kontrolu nad načinom na koji će se podaci prikazati, koristeći standardne web tehnologije.

Pomoću D3.js podaci se izravno povezuju s elementima na stranici, omogućujući da sadržaj, poput grafova ili dijagrama, dinamički reagira na promjene koje se automatski prilagođavaju korisničkim akcijama ili novim informacijama. Animacije i prijelazi ugrađeni u D3.js omogućuju glatke promjene između različitih prikaza, čineći korisničko iskustvo boljim.

Sve ovo čini D3.js idealnim za projekte u raznim industrijama kao koristan alat za pretvaranje podataka u vizualno dojmljive i informativne prikaze, omogućujući korisnicima dublje razumijevanje složenih podataka poput grafova.

3.3 HTML

HTML (engl. *HyperText Markup Language*) je osnovni jezik koji se koristi za izradu i strukturiranje web stranica. Omogućava organizaciju različitih vrsta sadržaja, uključujući naslove, paragrafe, slike, tablice itd. Svaki element u HTML-u definiran je pomoću oznaka (engl. *tag*) koje određuju njegovu funkciju i poziciju unutar web stranice. Uglavnom se koristi zajedno s CSS-om i JavaScriptom. CSS služi za uređivanje vizualnog izgleda stranice, a JavaScript omogućuje interaktivnost te dinamično ponašanje stranica.

3.4 CSS

CSS (engl. *Cascading Style Sheets*) je stilski jezik koji se koristi za uređivanje vizualnog izgleda web stranica, uključujući aspekte poput fontova, boja i rasporeda elemenata. Dokumenti napisani koristeći HTML jezik koriste CSS kako bi definirali kako će HTML elementi biti prikazani. Korištenje CSS-a omogućava odvajanje dizajna stranice od njezina sadržaja te se dizajn stranice obično smješta u zasebni CSS dokument, odvojen od sadržaja stranice.

3.5 JavaScript

JavaScript je programski jezik koji se koristi za dodavanje interaktivnosti i dinamičkog ponašanja web stranicama. Za razliku od HTML-a, koji definira strukturu i sadržaj stranice i CSS-a, koji uređuje njen vizualni izgled, JavaScript omogućuje web stranicama da odgovaraju na korisničke akcije.

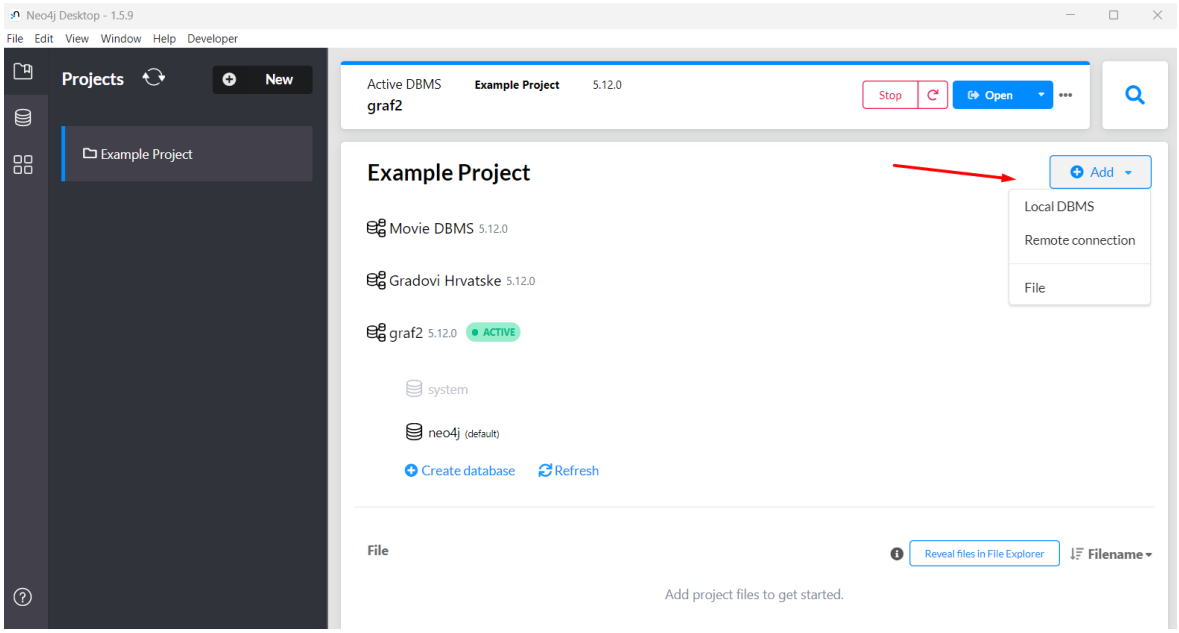
JavaScript se obično integrira unutar HTML dokumenta pomoću `<script>` oznake ili se poziva iz vanjskih JavaScript datoteka koje sadrže JavaScript kôd. Ovaj jezik može direktno upravljati HTML i CSS elementima na stranici putem Document Object Modela (DOM), omogućavajući dinamično mijenjanje sadržaja i stilova u stvarnom vremenu.

4. Razvoj aplikacije

4.1 Baza podataka

Za korištenje Neo4j graf baze podataka, koristi se aplikacija Neo4j Desktop.

Projekt se izradi klikom na dugme `Add` te potom na `Local DBMS` što je prikazano na slici 1. Nakon toga se upišu osnovni podaci poput naziva, zaporke itd.



Slika 1: Kreiranje projekta u aplikaciji Neo4j Desktop

U Neo4j Browseru, kojem se može pristupiti nakon što se napravi projekt, mogu se provjeriti osnovni podaci o samom projektu kao što su: naziv samog projekta, status servera, podaci unutar baze podataka itd.

Pošto je Neo4j graf baza podataka, ne koristi tablice kao relacijska baza podataka poput MySQL-a ili PostgreSQL-a te sličnih relacijskih bazi podataka. Umjesto toga, Neo4j organizira podatke kao čvorove (entitete), odnose (veze između čvorova) i svojstva (atributi čvorova i odnosa).

U Neo4j Browseru se također mogu izvršavati Cypher upiti. Primjer Cypher upita je dan u ispisu 1, a upit stvara čvor koji nosi naziv „A“.

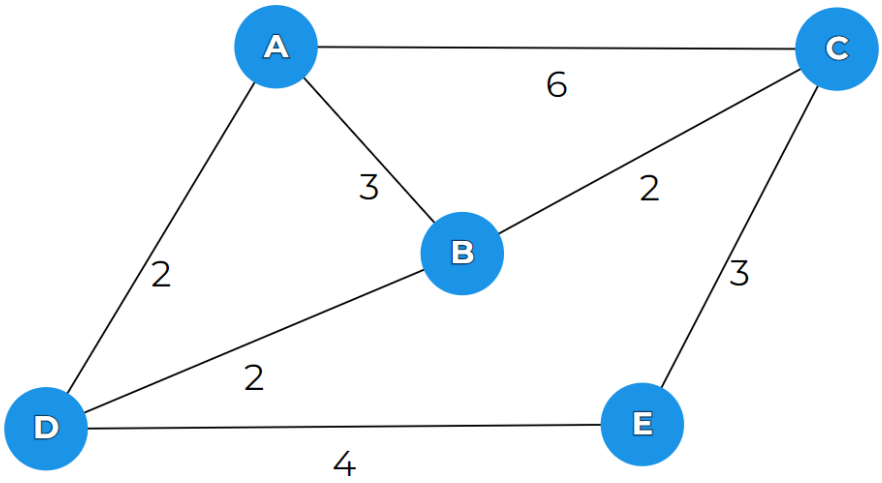
```
CREATE (n:Node {name: 'A'})
```

Ispis 1: Primjer stvaranja čvora u bazi podataka

```
CREATE (A)-[:CONNECTS {distance: 3}]->(B)
```

Ispis 2: Primjer stvaranja odnosa između dva čvora u bazi podataka

Upit u ispisu 2 stvara odnos između čvorova „A“ i „B“ sa svojstvom `distance` koje poprima vrijednost 3.



Slika 2: Primjer jednostavnog grafa

Svojstva čvorova „A“ i „B“ te svojstva njihove veze (brida) su prikazani na slici 3:

Slika 3: Svojstva elemenata grafa u Neo4j

Na slici se vidi da osim atributa `name` i `distance` koje je korisnik stvorio, Neo4j sam stvara dodatne atribute za čvor i vezu (odnos).

Kod spajanja aplikacije s Neo4j bazom, koristimo `fetch` metodu preko parametara koje smo odredili u aplikaciji Neo4j Desktop prilikom stvaranja same baze podataka. Primjer toga dan je u ispisu 3.

```
const neo4j_http_url = "http://localhost:7474/db/neo4j/tx"
const neo4jUsername = "neo4j"
const neo4jPassword = "admin123"

function submitQuery() {
    let nodeMap = {}
    let linkMap = {}
    let cypherString = "MATCH (n) OPTIONAL MATCH (n)-[r]->(m) RETURN n, r,
    m";

    fetch(neo4j_http_url, {
        method: 'POST',
        headers: {
            "Authorization": "Basic " +
            btoa(`${neo4jUsername}:${neo4jPassword}`),
            "Content-Type": "application/json",
            "Accept": "application/json;charset=UTF-8",
        },
        body: JSON.stringify({
            statements: [{
                statement: cypherString,
                resultDataContents: ["graph"]
            }] …
```

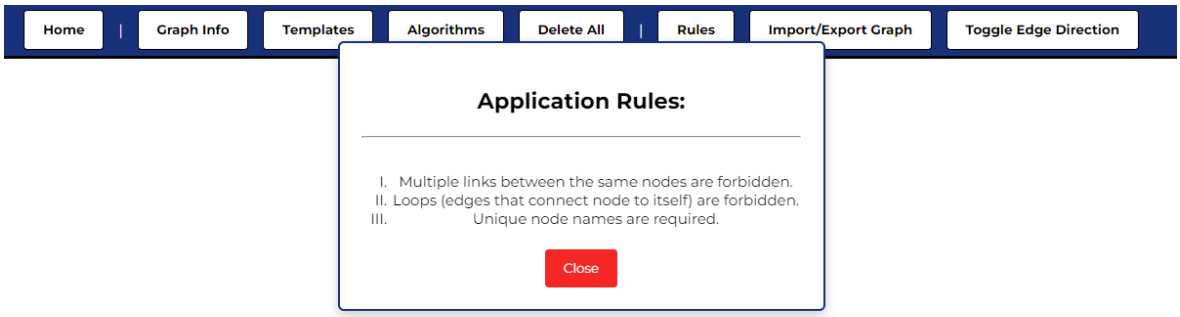
Ispis 3: Primjer dohvaćanja podataka iz baze

Ovaj kôd vraća odgovor u JSON (JavaScript Object Notation) formatu, a odgovor sadrži podatke o čvorovima i odnosima u bazi.

4.2 Početna stranica

Kod svakog posjeta ovoj web stranici, pojavit će se skočni prozor sa osnovnim pravilima ove aplikacije vezanim za grafove. To su tri pravila:

- Višestruki bridovi između dva vrha su zabranjeni.
- Petlje su zabranjene.
- Aplikacija zahtjeva jedinstveno ime za svaki vrh

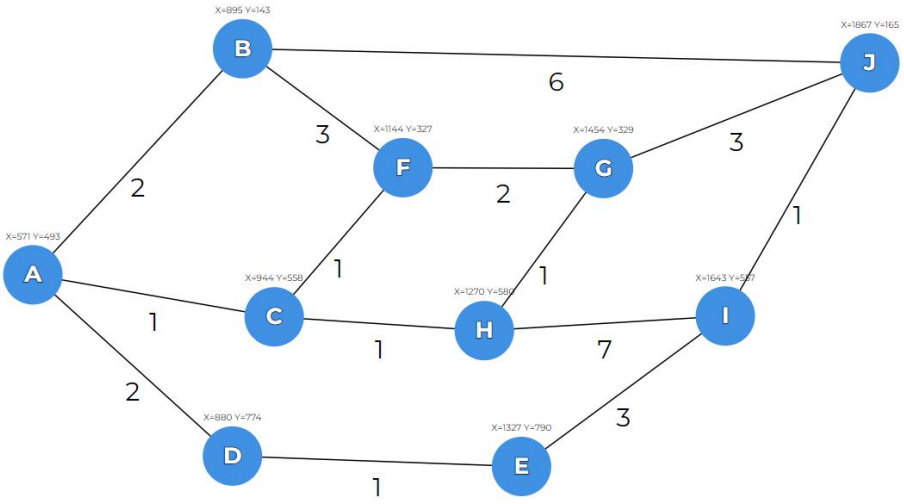


Slika 4: Korisničko sučelje

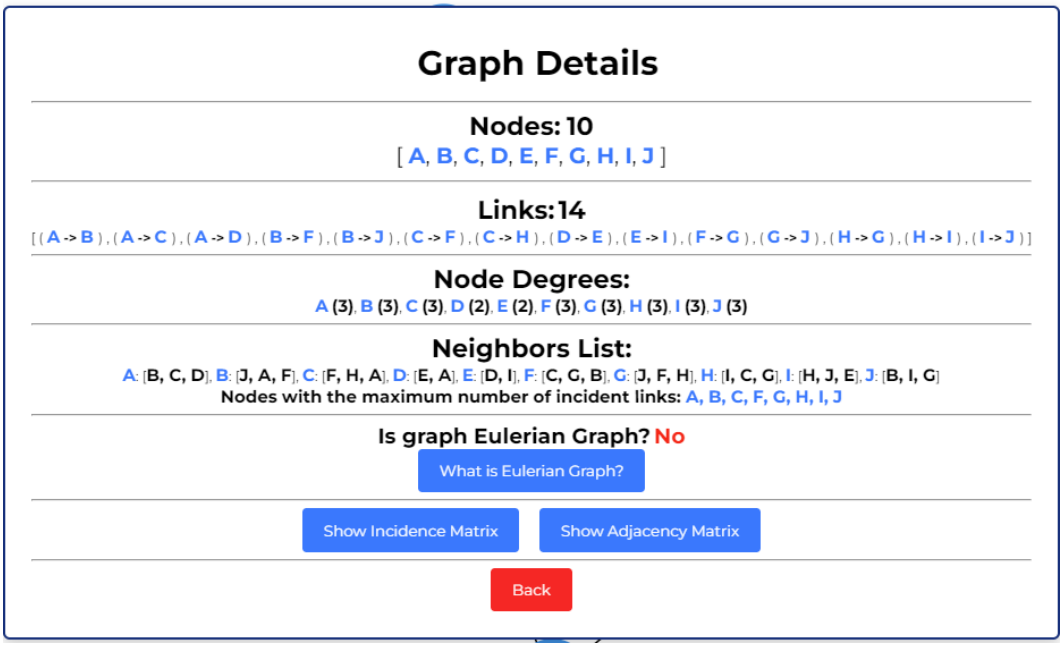
Također se na slici 4 vidi korisničko sučelje te navigacijska traka.

4.2.1 Graph Info

Klikom na dugme `Graph Info` mogu se vidjeti osnovni podaci o grafu: vrhovi, bridovi, stupanj svakog vrha, lista susjedstva, vrh(ovi) sa najviše incidentnih bridova, matrica incidencije, matrica susjedstva te provjera je li graf Eulerov graf. Sve informacije dobiju se obradom dva niza objekata: `nodes` i `links`. Primjerice, na slici 5 je prikazan jednostavan graf, a njegovi osnovni podaci su prikazani na slici 6.



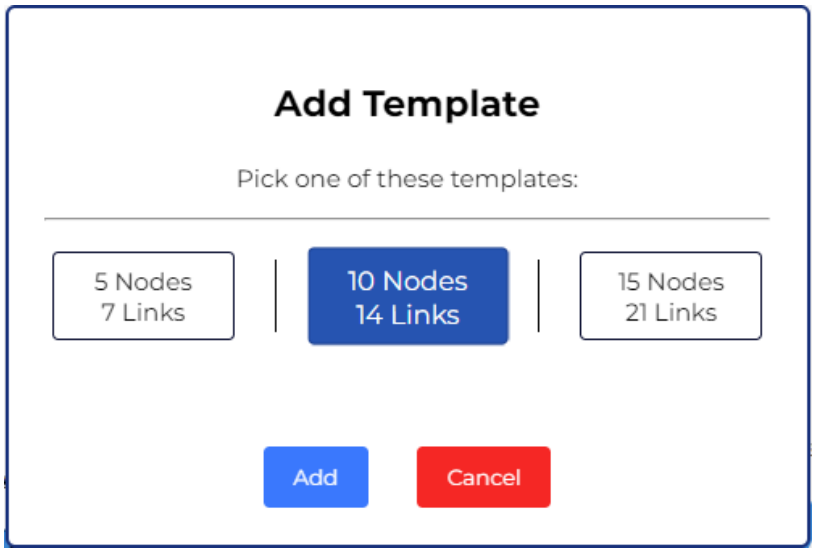
Slika 5: Primjer jednostavnog grafa



Slika 6: Primjer osnovnih podataka jednostavnog grafa

4.2.2 Predlošci

Klikom na dugme `Templates` otvara se izbornik s tri predloška kao na slici 7.



Slika 7: Izbornik predložaka

Ovaj dio aplikacije omogućuje korisniku brz i jednostavan početak rada, što može značajno uštedjeti vrijeme. Na primjer, ako korisnik želi stvoriti graf koji sadrži 12 vrhova i 16 bridova, može odabrati jedan od predložaka i prilagoditi ga prema svojim potrebama, umjesto da cijeli graf stvara ispočetka.

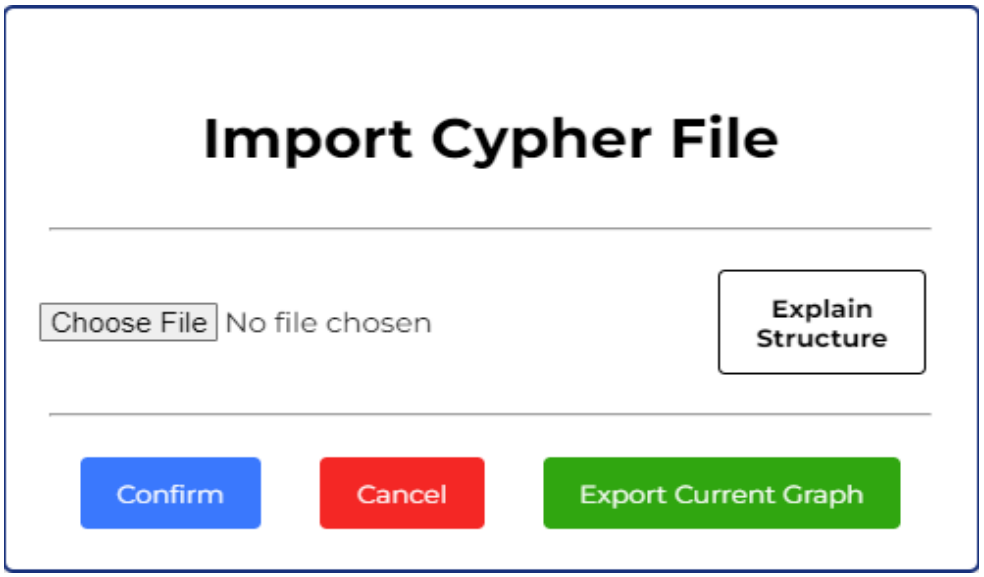
Kada se odabere i potvrdi dodavanje jednog od predložaka, postojeći graf se briše, a baza podataka se ažurira odabranim predloškom kao u primjeru kôda u ispisu 4.

```
async function addTemplate_5() {
    console.log("Executing addTemplate_5");

    let cypherString = `
        CREATE (A:Node {name: 'A'})
        CREATE (B:Node {name: 'B'})
        CREATE (C:Node {name: 'C'})
        CREATE (D:Node {name: 'D'})
        CREATE (E:Node {name: 'E'})

        CREATE (A)-[:CONNECTS {distance: 3}]->(B)
        CREATE (A)-[:CONNECTS {distance: 6}]->(C)
        CREATE (A)-[:CONNECTS {distance: 2}]->(D)
        CREATE (B)-[:CONNECTS {distance: 2}]->(C)
        CREATE (D)-[:CONNECTS {distance: 4}]->(E)
        CREATE (D)-[:CONNECTS {distance: 2}]->(B)
        CREATE (E)-[:CONNECTS {distance: 3}]->(C)` ;…
```

Ispis 4: Primjer unosa odabranog predloška u bazu

4.2.3 Uvoz/Izvoz grafa

Dugme `Import/Export graph` otvara prozor prikazan na slici 8.



Slika 8: Izbornik za uvoz/izvoz datoteke

Unutar prozora korisnik može napraviti uvoz ili izvoz Cypher datoteke koja sadrži informacije o vrhovima i bridovima grafa. Dugme `Explain Structure` daje uvid u „nacrt“ kako treba izgledati datoteka koja se učitava.

Primjer datoteke kojom se uvodi graf sa 3 vrha i 3 brida je dan u ispisu 5.

```
CREATE (A:Node {name: 'A'})
CREATE (B:Node {name: 'B'})
CREATE (C:Node {name: 'C'})

CREATE (A)-[:CONNECTS {distance: 3}]->(B)
CREATE (B)-[:CONNECTS {distance: 6}]->(C)
CREATE (C)-[:CONNECTS {distance: 2}]->(A)
```

Ispis 5: Primjer datoteke kojom se uvodi graf u bazu

Ako datoteka ne sadrži ispravno oblikovane naredbe ili se ne pridržava osnovnih pravila aplikacije (graf ne smije imati: petlje, više bridova između istih vrhova, vrhove s istim imenima), prikazat će se greška, a datoteka neće biti učitana.

Uvoz grafa:

Klikom na `Choose File` te odabirom željene datoteke koju korisnik želi učitati, pokreće se funkcija `importCypherFile` koja učitava datoteku, a potom se čita njen sadržaj. Ako sadržaj nije uspješno pročitan, korisnik će biti obaviješten o grešci, a ako je sadržaj uspješno pročitan, provjerava se je li zadovoljena zadana sintaksa naredbi i ostala pravila aplikacije. Ako su svi uvjeti ispunjeni, trenutni graf se briše, a novi graf se učitava u bazu podataka.

Izvoz grafa:

Kako bi se trenutni graf izveo kao datoteka, potrebno je pritisnuti na dugme `Export Current Graph`. Izvezena datoteka se zatim može lako ponovo učitati u bazu podataka, omogućujući nastavak rada.

Podaci o grafu se dohvaćaju iz baze podataka te nakon što se podaci dohvate, obrađuju se i izvoze koristeći `Blob` objekt, objekt koji omogućava rad s podacima kao što su tekstualne datoteke, slike, zvuk itd. Kada se podaci pretvore u `Blob`, može se stvoriti privremeni URL za njihovo preuzimanje, a preuzeta datoteka je u `.cypher` formatu. Primjer rada sa `Blob` objektom je pokazan u ispisu 6.

```
function downloadCypherCommands(cypherCommands) {
   const blob = new Blob([cypherCommands.join('\n')], {
   type:'text/plain' });
   const url = URL.createObjectURL(blob);
   const link = document.createElement('a');
   link.href = url;
   link.download = 'graph.cypher';
   document.body.appendChild(link);
   link.click();
   document.body.removeChild(link);
   URL.revokeObjectURL(url);
```

Ispis 6: Primjer rada sa `Blob` objektom

4.3 Vizualizacija grafa

Najvažniji dio aplikacije odvija se unutar funkcije `updateGraph`, koja upravlja vizualizacijom grafa i interakcijom s njegovim elementima. U ovoj funkciji grafički se

generiraju vrhovi i bridovi, te se prikazuju njihova svojstva poput naziva vrhova i težine bridova. Također se unutar funkcije upravlja i simulacijom fizikalnih sila između vrhova i bridova, omogućava dodavanje i brisanje elemenata, izmjenu težina bridova, promjenu naziva vrhova te cjelokupnu interakciju poput raspoređivanja elemenata grafa. Uz to, ovdje se animiraju rješenja algoritama radi lakšeg razumijevanja njihovog rada.

4.3.1 Postavljanje SVG elementa

Unutar `index.html` datoteke, kao u primjer ispisa 7, postavi se `graph` kontejner kojeg se potom poveže sa SVG (Scalable Vector Graphics) elementom, unutar `updateGraph` funkcije, pomoću D3.js biblioteke. Ovaj kontejner služi kao okvir unutar kojeg će se prikazivati vizualizacija grafa te je dan u ispisu 8.

```
<div class="graph-container">
            <div id="graph"></div>
</div>
```

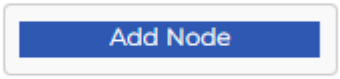
Ispis 7: Postavljanje `graph` kontejnera

```
const container = document.querySelector('#graph');
const width = container.clientWidth;
const height = container.clientHeight;

let svg = d3.select('#graph').select('svg');

if (svg.empty()) {
        svg = d3.select('#graph').append('svg')
            .attr('width', width)
            .attr('height', height);
}
svg.selectAll("*").remove();
```

Ispis 8: Povezivanje `graph` kontejnera sa d3.js

4.3.2 Kontekstualni izbornik pozadine

Desnim klikom na pozadinu, korisnik može odabrati dugme `Add Node` kojim dodaje novi vrh u graf kao na slici 9.



Slika 9: Dugme `Add Node`

Korisnik potom upisuje ime novog vrha, a ako već postoji vrh sa upisanim imenom, aplikacija javlja da je to ime zauzeto te korisnik mora odabrati drugo ime.

4.3.3 Primjena D3.js funkcionalnih sila između elemenata grafa

Nakon što se SVG element kreira ili pronađe unutar `graph` kontejnera, na njega se primjenjuju različite sile i pravila za pozicioniranje vrhova i bridova koje povezuju te vrhove.

Koristi se funkcija `d3.forceSimulation`, čiji je primjer dan u ispisu 9, koja simulira fizičke sile kako bi rasporedila vrhove (*nodes*) i bridove (*links*) unutar grafa:

```
const simulation = d3.forceSimulation(nodes)
    .force('link', d3.forceLink(links).id(d =>
     d.id).distance(300))
    .force('charge', d3.forceManyBody().strength(-1000))
    .force('center', d3.forceCenter(width / 2, height / 2));
```

Ispis 9: Primjer korištenja d3.js simulacijskih sila

- **forceLink** povezuje vrhove putem bridova i postavlja zadanu udaljenost od 300 piksela između njih
- **forceManyBody** primjenjuje odbojnu silu između vrhova, sprječavajući njihovo preklapanje
- **forceCenter** centrira cijeli graf u sredinu SVG elementa

Nakon što se simulacija fizičkih sila postavi, funkcija `drag` omogućava korisniku povlačenje odnosno raspoređivanje vrhova unutar grafa što omogućuje korisniku da interaktivno mijenja raspored vrhova unutar grafa radi jasnijeg prikaza strukture grafa.

4.3.4 Vizualizacija vrhova

Za vizualizaciju vrhova u grafu kreira se grupa `<g>` elemenata unutar SVG-a, gdje će se smjestiti svi vrhovi. Svakom vrhu iz niza podataka `nodes` dodjeljuje se vlastiti SVG element, koji se potom pozicionira i povezuje s funkcijom `drag`. Potom se svakom vrhu dodaje kružnica `circle`, čiji su atributi: polumjer, boja kruga te boja i debljina obruba. Ova kružnica predstavlja osnovni vizualni element svakog vrha, što pridonosi vizualnosti i interaktivnosti grafa.

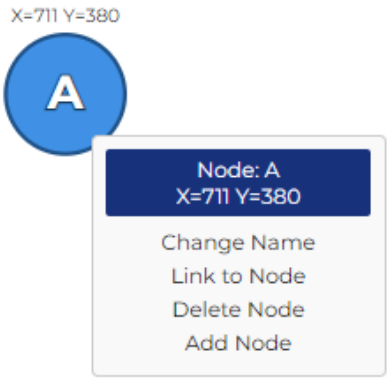
```
const node = svg.append('g')
    .attr('class', 'nodes')
    .selectAll('g.node')
    .data(nodes)
    .enter().append('g')
    .attr('class', 'node')
    .call(drag(simulation));

node.append('circle')
    .attr('r', circleSize)
    .attr('fill', '#1B93E6')
    .attr('stroke', '#06548a')
    .attr('stroke-width', 3);
```

Ispis 10: Primjer grafičkog stvaranja vrha

Nakon dodavanja same kružnice, kao u ispisu 10, svakom vrhu se dodaje tekstualni element `<text>` koji prikazuje ime vrha. Dodatno se iznad kružnice prikazuju X i Y koordinate vrha, kao na slici 10.

Slika 10: Primjer dva vrha u grafu

Desnim klikom na vrh, otvara se kontekstualni izbornik s opcijama prikazanim na slici 11.



Slika 11: Kontekstualni izbornik vrha

Preimenovanje vrha moguće je samo ako novo ime već nije zauzeto.

Ako se odabere opcija `Link to Node`, otvara se izbornik u kojem korisnik može odabrati jedan od dostupnih vrhova za povezivanje i unijeti težinu novog brida.

Prilikom brisanja vrha, automatski se brišu i svi bridovi povezani s tim vrhom.

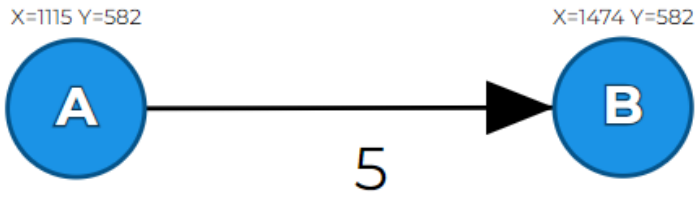
4.3.5 Vizualizacija bridova

Za vizualizaciju bridova u grafu kreira se grupa `<g>` unutar SVG-a, gdje će se smjestiti svi bridovi grafa. Svakom bridu iz niza podataka `links` dodjeljuje se vlastiti SVG element `<line>`, koji se koristi za vizualizaciju linije koja predstavlja brid između dva vrha. Atribut `stroke-width` određuje debljinu linije, dok `stroke` određuje boju linije. Korištenjem `marker-end` atributa dodaje se strelica na kraj bridova, što vizualno označava smjer brida. Taj atribut, strelica za smjer brida, se može vizualno „isključiti“ što može biti korisno kod algoritama koji ne mare za smjer brida, čime se izbjegava mogućnost zbunjivanja korisnika. Primjer kôda je dan u ispisu 11.

```
const linkGroup = svg.append('g')
        .attr('class', 'links')
        .selectAll('.link')
        .data(links)
        .enter().append('g')
        .attr('class', 'link');

linkGroup.append('line')
        .attr('stroke-width', 3)
        .style('stroke', '#000000')
        .attr('marker-end', 'url(#arrowhead)');
```

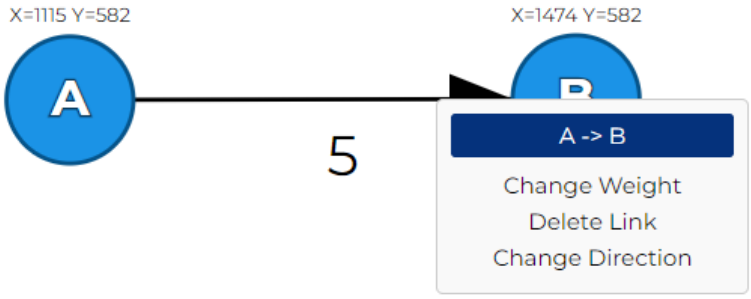
Ispis 11: Primjer grafičkog stvaranja bridova

Potom se svakom bridu dodaje tekstualni element `<text>` koji prikazuje težinu brida. Stilizacija teksta obuhvaća boju (engl. *fill*), veličinu fonta (engl. *font-size*), te pozicioniranje teksta. Konačni izgled brida dan je u slici 12.



Slika 12: Izgled veze između 2 vrha

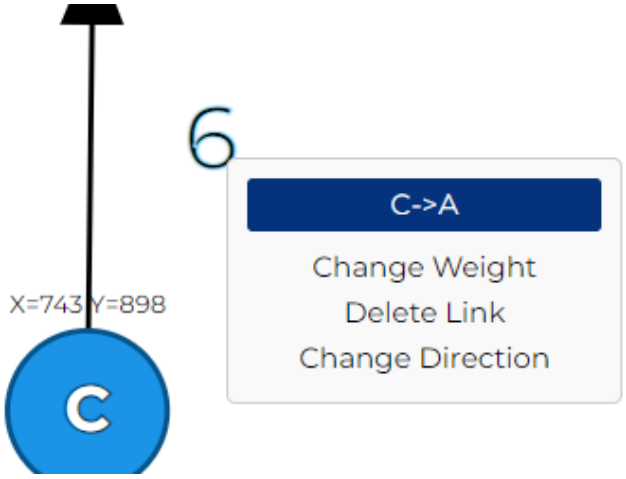
Desnim klikom na brid otvara se kontekstualni izbornik s opcijama kao na slici 13.



Slika 13: Kontekstualni izbornik brida

Promjena težine brida omogućava korisniku unos nove težine za odabrani brid. Ako unesena vrijednost nije ispravna tj. nije cijeli broj, aplikacija će prikazati upozorenje, a promjena neće biti izvršena. Brisanje brida uklanja brid iz grafa te baze podataka. Promjena smjera brida mijenja atribute postojećeg brida tako da ciljni vrh postane izvorišni, a izvorišni postane ciljni, uz zadržavanje atributa težine brida.

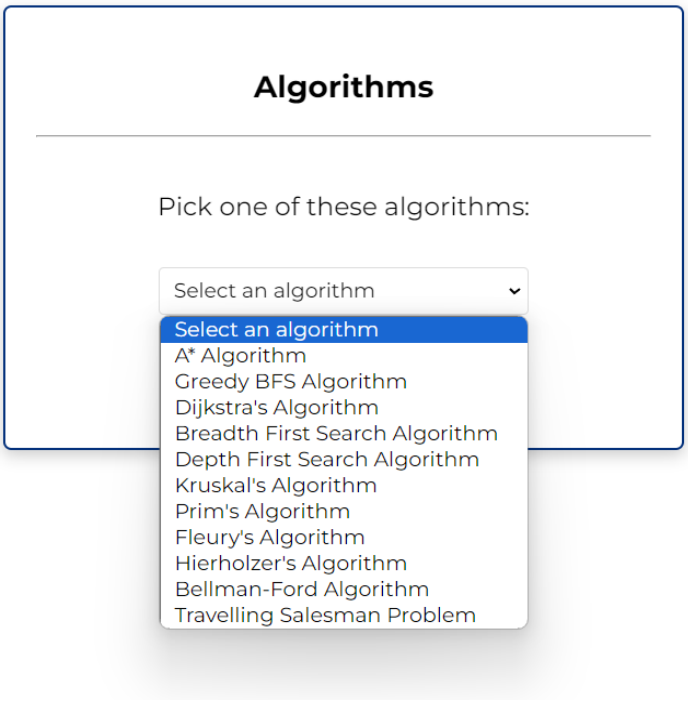
Ovaj kontekstualni izbornik za brid se također može otvoriti ako korisnik desnim klikom klikne na težinu brida (broj), kao na slici 14.



Slika 14: Kontekstualni izbornik brida otvoren preko težine brida

5. Implementacija algoritama

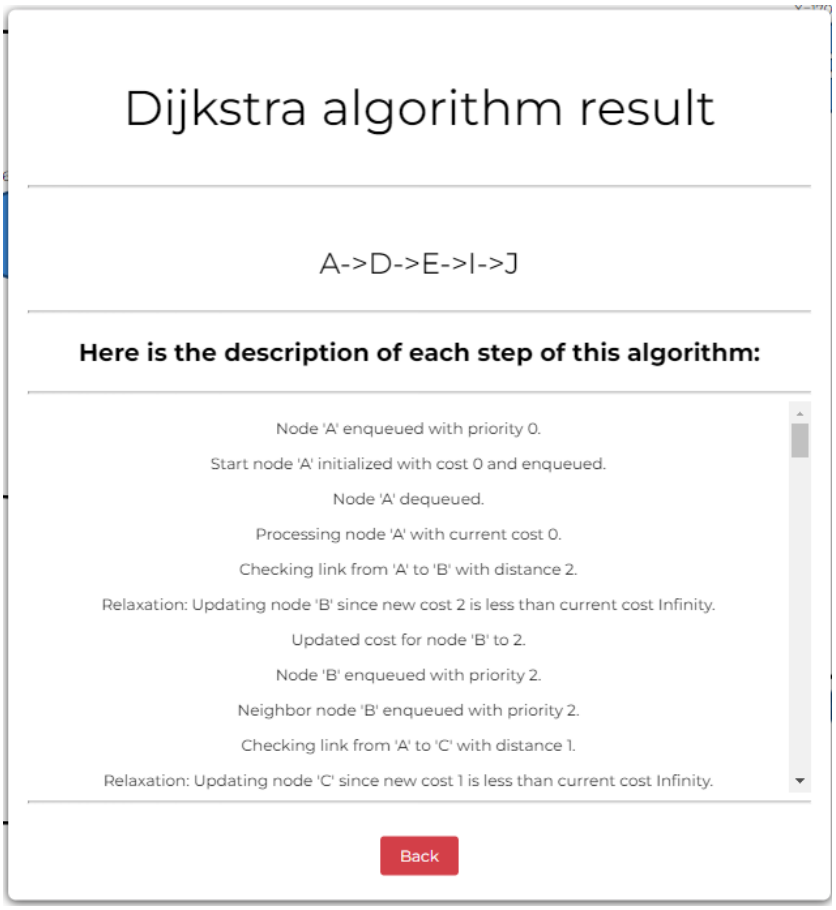
Klikom na dugme `Algorithms` koji se nalazi u navigacijskoj traci početne stranice, otvara se izbornik u kojem se može izabrati jedan od dostupnih algoritama. Izgled izbornika vidi se na slici 15.



Slika 15: Izbornik algoritama

Potom, ovisno o algoritmu, prikazuju se dodatne opcije, poput odabira početnog i ciljnog vrha, objašnjenja izabranog algoritma i odabir načina izvedbe (od početnog do ciljnog vrha ili od početnog do svih vrhova grafa).

Također, nakon svakog izvršavanja i animacije algoritma, korisnik će moći vidjeti tekstualni opis koraka, prikazano na slici 16, koje je aplikacija izvela, kako bi korisniku bilo lakše shvatiti što i kako se odvija u izabranom algoritmu. Ovaj tekstualni opis se može vidjeti klikom na dugme `Algorithm result` u kojem se nalazi rješenje algoritma te je zabilježen svaki korak izvršavanja.



Slika 16: Tekstualni opis koraka algoritma

5.1 Implementacija Breadth First Search algoritma

Breadth First Search algoritam (BFS), odnosno Algoritam pretraživanja u širinu, je osnovna metoda pretraživanja grafa koja istražuje sve vrhove jednog nivoa, tj. sve vrhove incidentne s promatranim vrhom, prije nego što pređe na vrhove sljedećeg nivoa.

BFS se koristi za pronalaženje najkraćeg puta u grafu ili za pretragu svih mogućih puteva od početnog vrha. Ovaj algoritam se primjenjuje u različitim područjima, poput mrežnih pretraga te rješavanja problema u igricama.

Koraci BFS algoritma:

1. Odabere se početni vrh te ga se stavi u red čekanja.
2. Uzme se vrh sa početka reda, istraže se njegovi susjedni vrhovi te ih se doda u red (osim ako su već bili istraženi).

3. Promatrani vrh se označi kao istražen te ga se ukloni iz reda.
4. Postupak se ponavlja za sljedeći vrh u redu.
5. Ponavljaju se koraci 2-4 dok red nije prazan ili se ne dođe do ciljnog vrha.

Implementacija algoritma u aplikaciji započinje inicijalizacijom strukture reda (engl. *queue*) za upravljanje vrhovima koje treba posjetiti. Red je struktura podataka, čiji je primjer dan u ispisu 12, koja omogućuje dodavanje elemenata na kraj reda i njihovo dohvaćanje s početka reda, čime se osigurava da se vrhovi posjećuju redoslijedom kojim su otkriveni.

```
class Queue {
      constructor() {
          this.nodes = [];
      }
      enqueue(value) {
          this.nodes.push(value);
      }
      dequeue() {
          const value = this.nodes.shift();
          return value;
      }
      isEmpty() {
          return this.nodes.length === 0;
      }
  }
```

Ispis 12: Primjer kôda `Queue` strukture

BFS algoritam se temelji na istraživanju svih susjednih vrhova trenutnog vrha. Algoritam počinje od zadanog početnog vrha, dodaje ga u red i označava kao posjećenog. Zatim, u svakoj iteraciji, algoritam uklanja vrh s početka reda, provjerava je li to ciljni vrh, a ako nije, dodaje sve neposjećene incidentne vrhove u red. Ako se stigne do ciljnog vrha, algoritam vraća put koji je obrađen i spreman za izvršavanje animacije. Ako red ostane prazan prije nego što se dođe do ciljnog vrha, algoritam vraća poruku da put do njega ne postoji. Primjer ovog dijela kôda je dan u ispisu 13.

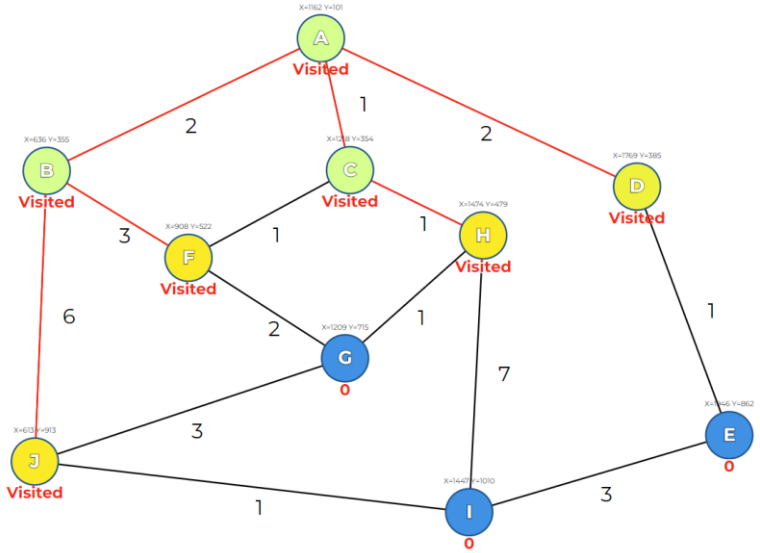
```
while (!queue.isEmpty()) {
            const currentNode = queue.dequeue();
            steps.push([currentNode.properties.name]);
            if (currentNode.properties.name ===
                endNode.properties.name) {
                return { path: reconstructPath(cameFrom,
                currentNode), steps };
            }
            links.filter(link => link.source.properties.name ===
```

```
currentNode.properties.name ||
link.target.properties.name ===
currentNode.properties.name)
    .forEach(link => {
        const neighbor = link.source.properties.name
        === currentNode.properties.name
        ? nodes.find(node => node.properties.name ===
        link.target.properties.name)
        : nodes.find(node => node.properties.name ===
        link.source.properties.name);

        if (!visited.has(neighbor.properties.name)) {
            visited.add(neighbor.properties.name);
            cameFrom.set(neighbor.properties.name, {
            node: currentNode, linkId: link.id });
            queue.enqueue(neighbor);
            steps.push([currentNode.properties.name,
            Number(link.id),
            neighbor.properties.name]);
```

Ispis 13: Dio kôda BFS algoritma

Tokom animiranja ovog algoritma, oni vrhovi koji su posjećeni i čiji susjedni vrhovi su dodani u red, su obojani zelenom bojom, a oni vrhovi koji su posjećeni ali ne i obrađeni (njihovi susjedni vrhovi nisu dodani u red) su obojani žutom bojom.

Na slici 17 se može vidjeti kako algoritam radi na principu „nivo po nivo“, odnosno istražuje sve vrhove na trenutnom nivou prije nego što prijeđe na sljedeći nivo dubine u grafu.



Slika 17: Animiranje izvršavanja BFS algoritma

5.2 Implementacija Depth First Search algoritma

Depth First Search algoritam (DFS), odnosno Algoritam pretraživanja u dubinu, je osnovna metoda pretraživanja grafa koja istražuje što je moguće dalje od početnog vrha duž svake grane (brida) prije nego što se vrati nazad i istraži druge grane.

DFS se koristi za pretragu puteva u grafu, detekciju ciklusa, topološkom sortiranju te rješavanju zagonetki u igrama.

Koraci DFS algoritma:

1. Odabere se početni vrh te ga se označi kao posjećenog.
2. Početni vrh se stavi na stog (stog je struktura u kojoj se elementi dodaju i dohvaćaju s kraja, tj. posljednji dodani element je prvi koji se uklanja).
3. Dok stog nije prazan:
    - Uklanja se vrh s vrha stoga.
    - Svaki susjedni vrh tog uklonjenog vrha koji već nije posjećen se dodaje na vrh stoga i označava kao posjećenog.
4. Ponavlja se 3. korak dok stog nije prazan ili dok se ne dođe do ciljnog vrha.

Implementacija algoritma u aplikaciji započinje inicijalizacijom stoga (engl. *stack*), čiji je primjer dan u ispisu 14, za upravljanje vrhovima koje treba posjetiti. Stog je struktura podataka koja omogućuje dodavanje elemenata na vrh stoga i njihovo dohvaćanje s vrha, što omogućuje algoritmu da prati putanje duboko u graf prije nego što se vrati unatrag.

```
class Stack {
        constructor() {
            this.nodes = [];
        }
        push(value) {
            this.nodes.push(value);
        }
        pop() {
            const value = this.nodes.pop();
            return value;
        }
        isEmpty() {
            return this.nodes.length === 0;
        }
    }
```

Ispis 14: Primjer kôda stog strukture

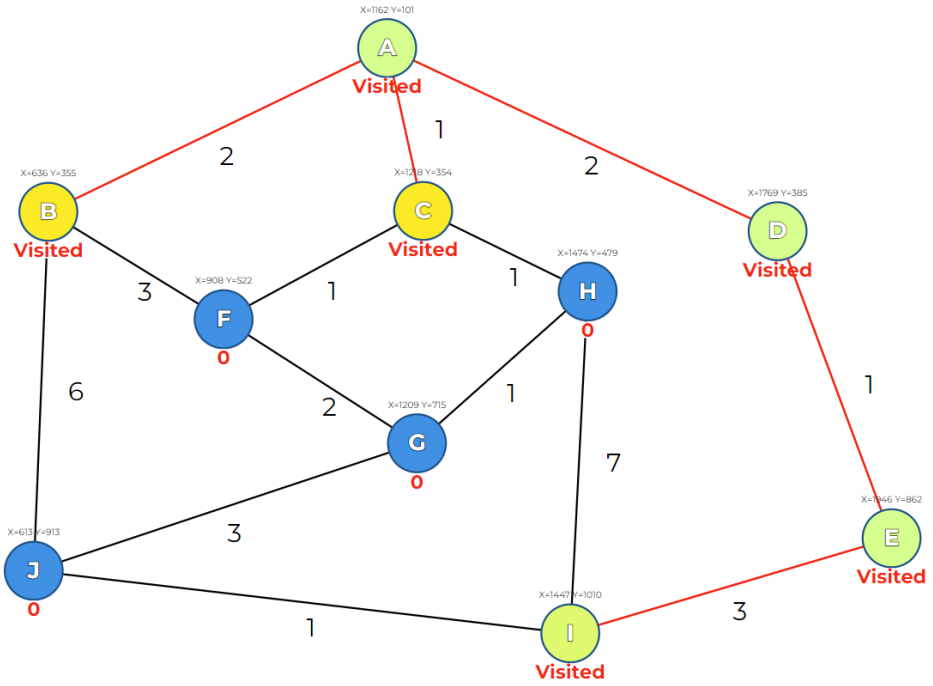
DFS algoritam započinje dodavanjem početnog vrha na stog i označavanjem kao posjećenog. Kroz svaku iteraciju, algoritam uklanja vrh s vrha stoga i provjerava je li to ciljni vrh. Ako je, obrađuje se put od početnog do ciljnog vrha te šalje u funkciju `updateGraph`. Ako nije, algoritam dodaje sve neposjećene susjedne vrhove tog vrha na stog i nastavlja pretragu u dubinu. Primjer ovog dijela kôda dan je u ispisu 15.

```
      while (!stack.isEmpty()) {
                  const currentNode = stack.pop();
                  steps.push([currentNode.properties.name]);
                  if (currentNode.properties.name ===
                  endNode.properties.name) {
                      return { path: reconstructPath(cameFrom,
                      currentNode), steps };
                  }
                  links.filter(link =>
                      link.source.properties.name ===
                      currentNode.properties.name ||
                      link.target.properties.name ===
                      currentNode.properties.name
                  ).forEach(link => {
                      const neighbor = link.source.properties.name ===
                          currentNode.properties.name
                          ? nodes.find(node => node.properties.name ===
                          link.target.properties.name)
                          : nodes.find(node => node.properties.name ===
                          link.source.properties.name);
                      if (!visited.has(neighbor.properties.name)) {
                          visited.add(neighbor.properties.name);
                          cameFrom.set(neighbor.properties.name, { node:
                          currentNode, linkId: link.id });
                          stack.push(neighbor);
                          steps.push([currentNode.properties.name,
                          Number(link.id), neighbor.properties.name]); …
```

Ispis 15: Dio kôda DFS algoritma

Slično kao kod BFS algoritma, tijekom animacije ovog algoritma, vrhovi koji su posjećeni i čiji su susjedni vrhovi dodani na stog obojani su zelenom bojom, dok su vrhovi koji su posjećeni, ali njihovi susjedni vrhovi još nisu obrađeni (nisu posjećeni), obojani žutom bojom.

Ova animacija, na slici 18, prikazuje kako DFS algoritam radi na principu "istraživanja u dubinu", gdje se prvo istražuju svi vrhovi duž jedne grane prije nego što se algoritam vrati unatrag i nastavi istraživati druge grane grafa.



Slika 18: Izvršavanje DFS algoritma

5.3 Implementacija pronalaska i vizualizacije minimalnog razapinjućeg stabla

Minimalno razapinjuće stablo poznato kao *Minimal Spanning Tree* (MST) je podskup bridova povezanog, neusmjerenog grafa koji povezuje sve vrhove zajedno, bez ciklusa i s najmanjom mogućom sumom težina bridova. MST se može koristiti za dizajn minimalnih troškova mreža, geografskih analiza itd.

Za pronalazak i vizualizaciju minimalnog razapinjućeg stabla se koriste Kruskalov i Primov algoritam.

5.3.1 Implementacija Kruskalovog algoritma

Kruskalov algoritam, nazvan po američkom matematičaru Josephu Kruskalu, koristi se za pronalaženje minimalnog razapinjućeg stabla u povezanim, neusmjerenim grafovima. Algoritam se temelji na dodavanju bridova u rastućem redoslijedu težine, uz osiguranje da se ne formiraju ciklusi.

Koraci Kruskalovog algoritma:

1. Sortiraju se svi bridovi po težini u rastućem redosljedu
2. Uzima se brid najmanje težine te se provjerava stvara li se ciklus dodavanjem tog brida u MST. Ako se ciklus ne stvara, dodaje se brid.
3. Ponavlja se 2. korak sve dok MST ne sadrži N-1 bridova (N je broj vrhova)

Implementacija algoritma u aplikaciji započinje korištenjem klase `UnionFind`, klase čija su osnova dvije funkcije, `union` i `find`. Funkcija `find` omogućuje provjeru pripadaju li dva vrha istom skupu, što pomaže u izbjegavanju ciklusa, a funkcija `union` spaja dva skupa vrhova kada se dodaje novi brid, osiguravajući izgradnju minimalnog razapinjućeg stabla. Kôd koji prikazuje strukturu klase `UnionFind` vidi se u ispisu 16.

```
class UnionFind {
        constructor(elements) {
            this.parent = {};
            elements.forEach(e => this.parent[e] = e);
        }

        find(id) {
            while (this.parent[id] !== id) {
                id = this.parent[id];
            }
            return id;
        }

        union(x, y) {
            let rootX = this.find(x);
            let rootY = this.find(y);
            if (rootX !== rootY) {
                this.parent[rootY] = rootX;
            }
        }
    }…
```

Ispis 16: Klasa `UnionFind`

Ključan dio pronalska minimalnog razapinjućeg stabla, koji je prikazan u ispisu 17, je obrada svakog brida u grafu, počevši od onog najmanje težine. Za svaki brid provjerava se pripadaju li izvorni i ciljni vrh tog brida istom skupu pomoću funkcije `find`. Ako nisu u istom skupu, brid se dodaje u minimalno razapinjuće stablo, a vrhovi se spajaju pomoću funkcije `union`, čime se izbjegava stvaranje ciklusa. Ako bi brid stvorio ciklus, preskače se.

```
edges.forEach(edge => {
            if (uf.find(edge.source) !== uf.find(edge.target)) {
                uf.union(edge.source, edge.target);
                mst.push({ id: edge.id, source: edge.source,
                target: edge.target });
            }
            }
        });
```

Ispis 17: Ključan dio pronalaska minimalnog razapinjućeg stabla

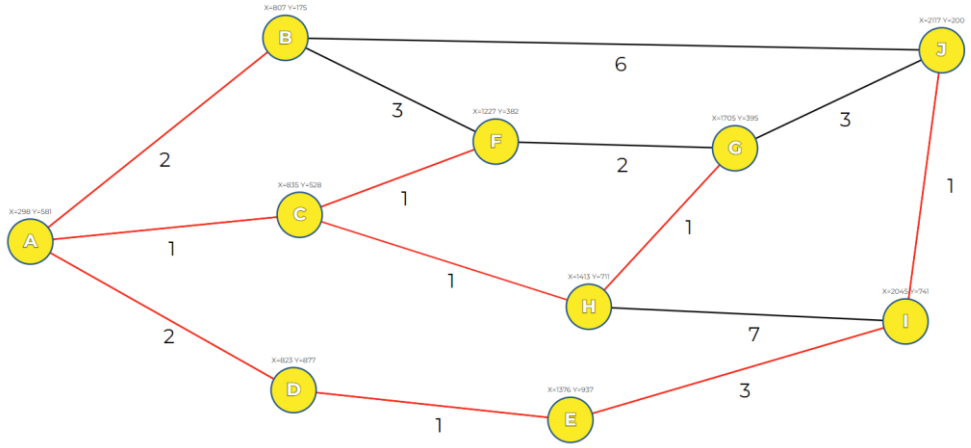
Nakon što se pronalazak završi, formatira se odgovor tako da se, kasnije u samoj animaciji algoritma, vrhovi ne animiraju više puta te se odgovor šalje u funkciju `UpdateGraph`.

Svi algoritmi pa tako i ovaj se animiraju na način da funkcija čita član po član iz liste te provjerava je li taj član vrh ili brid, a potom radi promjene nad elementima grafa kako bi se prikazao rad algoritma. Kod ovog algoritma, čiji je kôd prikazan u ispisu 18, uvijek će se prvo animirati brid pa potom dva vrha koja su povezana tim bridom, na način da će brid promijeniti boju iz crne u crvenu, a vrhovi iz plave u žutu. Izvedba animacije algoritma se može prekinuti klikom na dugme `Stop`, nakon čega se boje vrhova i bridova vrate u originalnu.

```
for (const item of pathSegment) {
                    if (animationInterrupted) {
                        revertColors();
                        break;
                    } else {
                        if (typeof item === 'string') {
                            svg.selectAll('g.node circle')
                                .filter(d => d.properties.name ===
                                item)
                                .transition()
                                .duration(750)
                                .attr('fill', '#ffea00');
                        } else {
                            const linkId = String(item);
                            console.log("Highlight link id:",
                            linkId);
                            svg.selectAll('.link line')
                                .filter(d => d.id === linkId)
                                .transition()
                                .duration(750)
                                .style('stroke', 'red')
                                .attr('stroke-width', 3)
                                .attr('marker-end',
                                'url(#arrowhead-red)');
```

Ispis 18: Primjer rada funkcije za animaciju algoritma

Nakon završetka animacije, korisniku je, kao na slici 19, prikazano minimalno razapinjuće stablo sa bridovima crvene boje.



Slika 19: Minimalno razapinjuće stablo – Kruskalov algoritam

5.3.2 Implementacija Primovog algoritma

Primov algoritam, nazvan po češkom matematičaru Vojtěchu Primu, također se koristi za pronalaženje minimalnog razapinjućeg stabla u neusmjerenom grafu. Algoritam se temelji na postupnom proširivanju stabla dodavanjem bridova s najmanjom težinom koji povezuju trenutno stablo s preostalim vrhovima.

Koraci Primovog algoritma:

1. Odabere se početni vrh te ga se označi kao dio MST-a.
2. Odabere se brid najmanje težine koji povezuje trenutno uključene vrhove MST-a s nekim od preostalih vrhova u grafu. Taj brid i vrh kojeg povezuje s ostatkom MST-a se dodaju u MST.
3. Ponavlja se 2. korak sve dok svi vrhovi nisu uključeni u MST.

Implementacija algoritma u aplikaciji započinje inicijalizacijom prioritetnog reda za upravljanje rubovima po njihovoj težini. `PriorityQueue` je struktura podataka, čiji je kôd prikazan u ispisu 19, koja omogućuje dodavanje elemenata i njihovo dohvaćanje prema prioritetu, u ovom slučaju bridova prema najmanjoj težini. Algoritam kreće od proizvoljno

odabranog početnog vrha, dodaje sve njegove bridove u red te potom bira bridove s najmanjom težinom.

```
class PriorityQueue {
        constructor() {
            this.queue = [];
        }
        enqueue(element) {
            this.queue.push(element);
            this.queue.sort((a, b) => a.weight - b.weight);
        }
        dequeue() {
            const element = this.queue.shift();
            return element;
        }
        isEmpty() {
            return this.queue.length === 0;
        }
    }
```

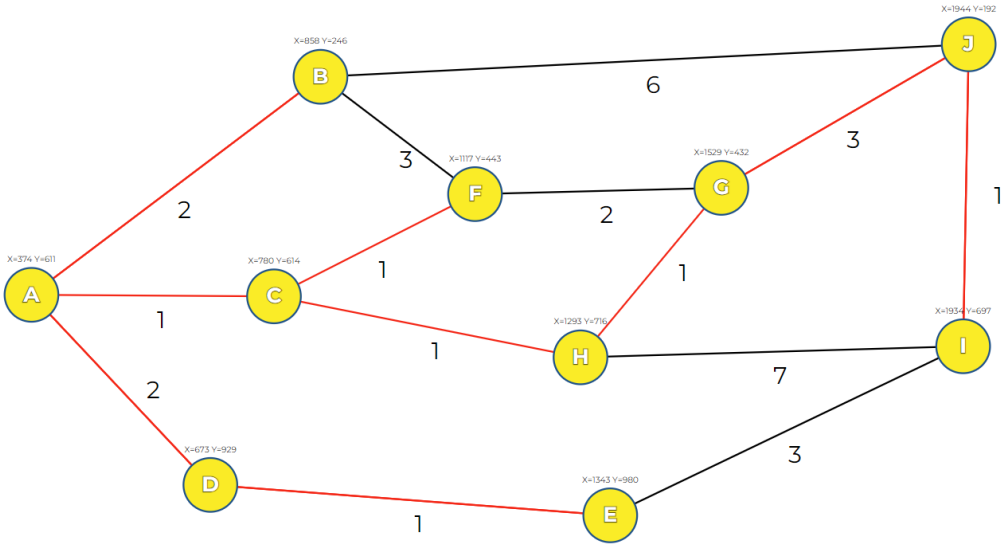
Ispis 19: Klasa `PriorityQueue`

Ključan dio Primovog algoritma je održavanje skupa posjećenih vrhova kako bi se izbjeglo stvaranje ciklusa. Kada se obradi brid najmanje težine, provjerava se je li ciljni vrh već posjećen. Ako nije, brid se dodaje u minimalno razapinjuće stablo, a novi bridovi iz ciljnog vrha dodaju se u prioritetni red. Ako bi dodavanje tog brida stvorilo ciklus, on se preskače. Kôd za ovaj dio algoritma je dan u primjeru ispisa 20.

```
while (!pq.isEmpty() && mst.length < nodes.length - 1) {
            const edge = pq.dequeue();
            if (!visited.has(edge.target)) {
                visited.add(edge.target);
                mst.push({ id: edge.id, source: edge.source,
                target: edge.target });
                adjList.get(edge.target).forEach(nextEdge => {
                    if (!visited.has(nextEdge.target)) {
                        pq.enqueue(nextEdge);
                    }
                });
            }
        }
```

Ispis 20: Izvršavanje Primovog algoritma

Nakon toga se odgovor formatira da bi se na ispravan način animirao algoritam, a sama animacija, na slici 20, radi na isti način kao kod Kruskalovog algoritma.



Slika 20: Minimalno razapinjuće stablo – Primov algoritam

5.4 Implementacija rješenja Problema trgovačkog putnika

Problem trgovačkog putnika poznat kao *Travelling Salesman Problem* (TSP) je jedan od najpoznatijih problema u kombinatornoj optimizaciji. Cilj je pronaći najkraći put koji posjećuje svaki grad jednom i vraća se u početni grad. Pitanje je kojim redoslijedom bi putnik trebao obilaziti gradove kako bi duljina puta bila minimalna. U ovom radu koriste se dva algoritma za rješenje i vizualizaciju problema trgovačkog putnika:

5.4.1 Implementacija algoritma najbližeg susjeda

Algoritam najbližeg susjeda (engl. *Nearest Neighbor algorithm*) je algoritam koji koristi jednostavnu strategiju u kojoj se uvijek odabire najbliži neposjećeni susjed, što minimizira ukupnu udaljenost puta.

Koraci algoritma:

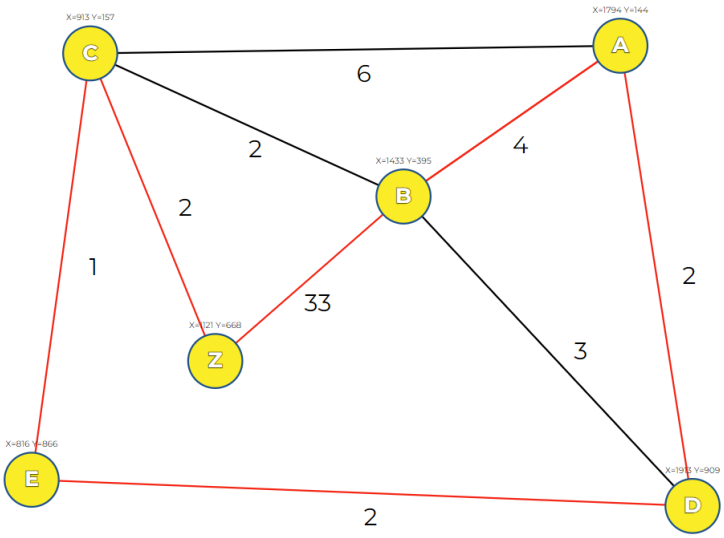
1. Odabere se početni vrh i označi kao posjećen.
2. Odabere se najbliži neposjećeni susjed i označi kao posjećen.
3. Ponavlja se 2. korak sve dok svi vrhovi ne budu posjećeni, a potom se vraća u početni vrh.

Implementacija algoritma u aplikaciji započinje postavljanjem početnog vrha kao trenutno aktivnog, pri čemu se stvara skup (engl. *set*) posjećenih vrhova kako bi se osiguralo da se svaki vrh posjeti samo jednom. Putanja i ukupna prijeđena udaljenost također se inicijaliziraju na početku. Tijekom svakog koraka algoritma, prolazi se kroz sve bridove grafa kako bi se pronašao najbliži, neposjećeni vrh. Ako se pronađe takav vrh, on se dodaje u putanju i označava kao posjećen. Ovaj proces, dan primjerom u ispisu 21, se ponavlja sve dok se ne obiđu svi vrhovi u grafu.

```
    while (visited.size < globalNodes.length) {
            let nearest = findNearestNode(currentNode, globalNodes,
                                          globalLinks, visited);
            if (!nearest.node) {
                return;
            }
            visited.add(nearest.node);
            path.push(Number(nearest.linkId), nearest.node);
            nodeNamesPath.push(nearest.node);
            totalPathDistance.distance += nearest.distance;
            currentNode = nearest.node;
     }
```

Ispis 21: Dio koda algoritma najbližeg susjeda

Nakon što su svi vrhovi posjećeni, algoritam pokušava pronaći brid koji povezuje trenutni vrh natrag s početnim vrhom, čime se ciklus zatvara. Primjer izvedbe algoritma u kojem je početni vrh, vrh „A“, je prikazan na slici 21. Redom posjećeni vrhovi su: A, D, E, C, Z, B i ponovo A.



Slika 21: TSP - Algoritam najbližeg susjeda

5.4.2 Implementacija algoritma sortiranih bridova

Algoritam sortiranih bridova sortira bridove po težini i dodaje ih u graf ako ne stvaraju vrh sa stupnjem većim od dva i ne dovode do formiranja ciklusa, osim kada taj ciklus obuhvaća sve vrhove grafa (Hamiltonov ciklus). Proces se ponavlja dok svi vrhovi ne budu povezani u jedan Hamiltonov ciklus.

Koraci algoritma:

1. Sortiraju se bridovi po težini u rastućem redoslijedu
2. Iz sortirane liste dodavaju se bridovi u graf ako se dodavanjem pojedinog brida ne stvara vrh sa stupnjom većim od dva i ne dovodi do formiranja ciklusa osim ako je taj ciklus Hamiltonov.
3. Ponavlja se 2. korak sve dok svi vrhovi ne budu povezani u jedan Hamiltonov ciklus.

Implementacija algoritma u aplikaciji započinje sortiranjem svih bridova grafa po udaljenosti od najmanje do najveće. Za upravljanje spajanjem vrhova bez stvaranja ciklusa, koristi se `UnionFind` struktura, slično kao kod Kruskalovog algoritma. Ova struktura omogućuje praćenje koji su vrhovi već povezani kako bi se izbjeglo stvaranje ciklusa prije nego što su svi vrhovi povezani u jednu komponentu. Osim toga, koristi se mapa (engl. *map*) za praćenje stupnja svakog vrha (broj incidentnih bridova), kako bi se osiguralo da nijedan vrh nema više od dva susjeda.

Algoritam obrađuje svaki brid grafa, počevši od onog najmanje težine. Za svaki brid, provjerava se zadovoljava li uvjete: oba vrha moraju imati stupanj manji od dva, a dodavanje brida ne smije stvoriti ciklus osim ako nije posljednji korak spajanja svih vrhova. Primjer kôda za ovaj algoritam je dan u ispisu 22.

```
for (let link of sortedLinks) {
        const {source, target, id} = link;
        const sourceName = source.properties.name;
        const targetName = target.properties.name;
        if (degrees.get(sourceName) < 2 &&
            degrees.get(targetName) < 2)
            {
            if (uf.union(sourceName, targetName) || new
            Set(globalNodes.map(node =>
            uf.find(node.properties.name))).size === 1)
                {
                path.push(sourceName, Number(id), targetName);
```

```
                    nodeNamesPath.push(sourceName, targetName);
                    degrees.set(sourceName, degrees.get(sourceName) +
                    1);
                    degrees.set(targetName, degrees.get(targetName) +
                    1);
                } …
            }
        }
```

Ispis 22: Primjer kôda za Algoritam sortiranih bridova

Nakon što su svi bridovi obrađeni, algoritam provjerava jesu li svi vrhovi povezani u jednu komponentu i imaju li stupanj točno dva. Ako su oba uvjeta zadovoljena, algoritam je uspješno konstruirao Hamiltonov ciklus, te se koraci algoritma animiraju.

5.5 Implementacija pronalaska i vizualizacije Eulerovog ciklusa i puta

Za pronalazak i vizualizaciju Eulerovog puta i Eulerovog ciklusa se koriste Flueryjev i Hierholzerov algoritam. Oni mogu biti značajni u optimizaciji ruta, analizi grafova i razvoju sustava u mnogim područjima.

5.5.1 Implementacija Flueryjevog algoritma

Flueryjev algoritam, nazvan po francuskom inženjeru i matematičaru Paulu-Victoru Fleuryju, se koristi za otkrivanje Eulerovih ciklusa i puteva u grafovima.

Koraci Flueryjevog algoritma:

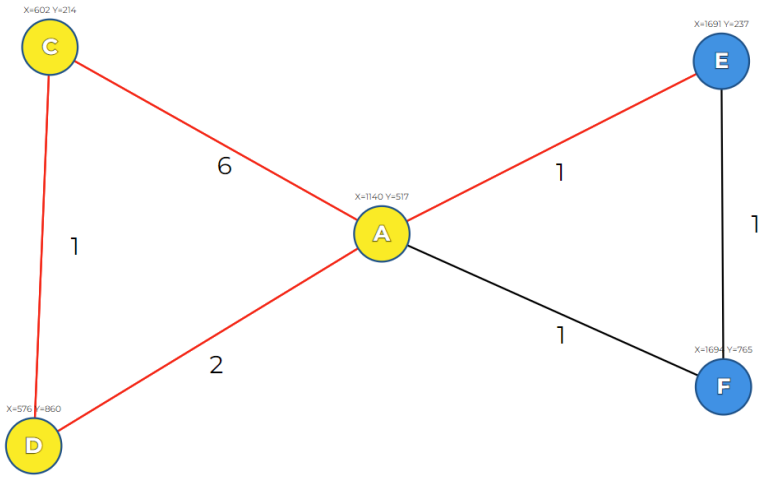
1. Provjerava se da li graf ima 0 ili 2 vrha s neparnim stupnjem.
2. Ako graf nema vrhova s neparnim stupnjem, počinje se s bilo kojim vrhom. Ako graf ima 2 vrha s neparnim stupnjem, počinje se s jednim od njih.
3. Od početnog vrha, slijede se bridovi jedan po jedan tako da se ne koriste već iskorišteni bridovi. Nikad se ne bira rezni brid, osim ako nema drugog izbora.
4. Ponavlja se 3. korak sve dok ima dostupnih bridova u grafu.

Implementacija algoritma u aplikaciji započinje provjerom je li graf povezan te imaju li svi vrhovi paran stupanj. Povezanost se provjerava koristeći DFS algoritam koji provjerava je li svaki vrh posjećen. Potom se odabire početni vrh, a zatim se prolazi kroz bridove grafa. U svakom koraku algoritma, bira se brid koji nije rezni, osim ako je to jedini preostali brid koji može biti odabran. Provjera je li brid rezni brid se također obavlja koristeći DFS algoritam tako da maknemo taj brid iz grafa, te ponovo provjerimo možemo li posjetiti sve vrhove kao i prije micanja tog brida. Izvedba ovog dijela kôda je dana u ispisu 23.

```
      while (Object.keys(this.links).length > 0) {
                  const edges = this.adjacencyList[currentNode];
                  let nextEdge = null;
                  for (const edge of edges) {
                            if (!this.isBridge(currentNode, edge.node,
                            edge.linkId)) {
                                nextEdge = edge;
                                break;
                            }
                       }
                       if (!nextEdge) {
                            nextEdge = edges[0];
                       }
                  cycle.push(this.nodes[currentNode].properties.name);
                  cycle.push(Number(nextEdge.linkId));
                  this.removeEdge(currentNode, nextEdge.node,
                  nextEdge.linkId);
                  delete this.links[nextEdge.linkId];
                  currentNode = nextEdge.node;
            }
```

Ispis 23: Dio Flueryjevog algoritma

Ako se uspješno prođe kroz sve bridove i vrati u početni vrh, algoritam je pronašao Eulerov ciklus. Nakon završetka algoritma, rješenje se animira kao na slici 22, prikazujući slijed vrhova i bridova (A -> C -> D -> A -> E -> F -> A).



Slika 22: Prikaz tokom animacije Flueryjevog algoritma

5.5.2 Implementacija Hierholzerovog algoritma

Hierholzerov algoritam, nazvan po njemačkom matematičaru Carlu Hierholzeru, se također koristi za otkrivanje Eulerovih ciklusa i puteva u grafovima.

Koraci algoritma:

1. Provjerava se ima li graf 0 ili 2 vrha s neparnim stupnjem. Graf nema Eulerov ciklus ako ima bilo koji vrh s neparnim stupnjem. Ako ima 2 vrha s neparnim stupnjem, ima Eulerov put.
2. Ako nema vrhova s neparnim stupnjem, počinje se s bilo kojim vrhom. Ako ima 2 vrha s neparnim stupnjem, počinje se s jednim od njih.
3. Slijede se bridovi od početnog vrha, formira se ciklus i uklanjaju se iskorišteni bridovi.
4. Ako ostanu neiskorišteni bridovi, odabire se vrh iz postojećeg ciklusa s neiskorištenim bridovima. Formira se novi ciklus i spaja s postojećim.
5. Postupak se ponavlja dok svi bridovi nisu iskorišteni.

Implementacija algortima u aplikaciji započinje provjerom imaju li svi vrhovi paran stupanj i je li graf povezan, koristeći DFS algoritam za provjeru povezanosti. Ako graf ne ispunjava ove uvjete algoritam obavještava korisnika da nije moguće pronaći Eulerov ciklus.

Bridovi grafa se označavaju kao neiskorišteni, a zatim se algoritam pokreće od nekog početnog vrha. Kroz svaki korak, algoritam koristi stog (engl. *stack*) za praćenje trenutne

putanje. Dok prolazi kroz graf, algoritam bira neiskorištene bridove i označava ih kao korištene. Ako se za trenutni vrh ne mogu naći neiskorišteni bridovi, taj vrh se uklanja iz stoga i dodaje u konačni ciklus. Primjer ovog dijela kôda je dan u ispisu 24.

```
      while (stack.length > 0) {
              const currentNode = stack[stack.length - 1];
              let unusedEdges = findUnusedEdges(currentNode);
              if (unusedEdges.length > 0) {
                  const edge = unusedEdges[0];
                  markEdgeAsUsed(edge);
                  const nextNode = edge.source === currentNode ?
                  edge.target : edge.source;
                  stack.push(nextNode);
                  pathEdges.push({ from: currentNode, to: nextNode, id:
                  edge.id });
              } else {
                  eulerianCycle.push(stack.pop()); …
```

Ispis 24: Dio kôda Hierholzerovog algoritma

Ako su svi bridovi iskorišteni, algoritam je uspješno konstruirao Eulerov ciklus. Rješenje se, baš kao kod Flueryjevog algoritma, šalje u funkciju `updateGraph` i animira.

5.6 Implementacija pronalaska i vizualizacije (najkraćeg) puta između vrhova

Pronalazak (najkraćeg) puta između vrhova u grafu je jedan od ključnih problema u teoriji grafova, navigacijskim sustavima, analizi mreža i mnogim drugim područjima. Postoji mnogo algoritama za pronalaženje najkraćeg puta, a njihova vizualizacija može značajno pomoći u razumijevanju njihovog načina rada i učinkovitosti.

5.6.1 Implementacija Dijkstrinog algoritma

Dijkstrin algoritam, kojeg je izumio nizozemski računalni znanstvenik Edsger W. Dijkstra, se koristi za pronalazak najkraćeg puta između dva vrha ili između početnog vrha do svih ostalih vrhova grafa, a svi bridovi imaju težine (udaljenosti).

Koraci Dijkstrinog algoritma:

1. Inicijalizira se udaljenost svih vrhova na beskonačnost osim izvorišnog vrha, koji se postavlja na 0. Prioritetni red (red u kojem su elementi poredani uzlazno po udaljenosti) se započinje s izvorišnim vrhom.

2. Odabire se vrh s najmanjom udaljenošću iz prioritetnog reda i označava se kao posjećen.
3. Za svaki neposjećeni susjed trenutnog vrha izračunava se nova udaljenost i ažurira ako je manja. Susjed se dodaje u prioritetni red s ažuriranom udaljenošću.
4. Koraci 2 i 3 se ponavljaju dok svi vrhovi ne budu posjećeni ili dok se ne pronađe najkraći put do svih vrhova.

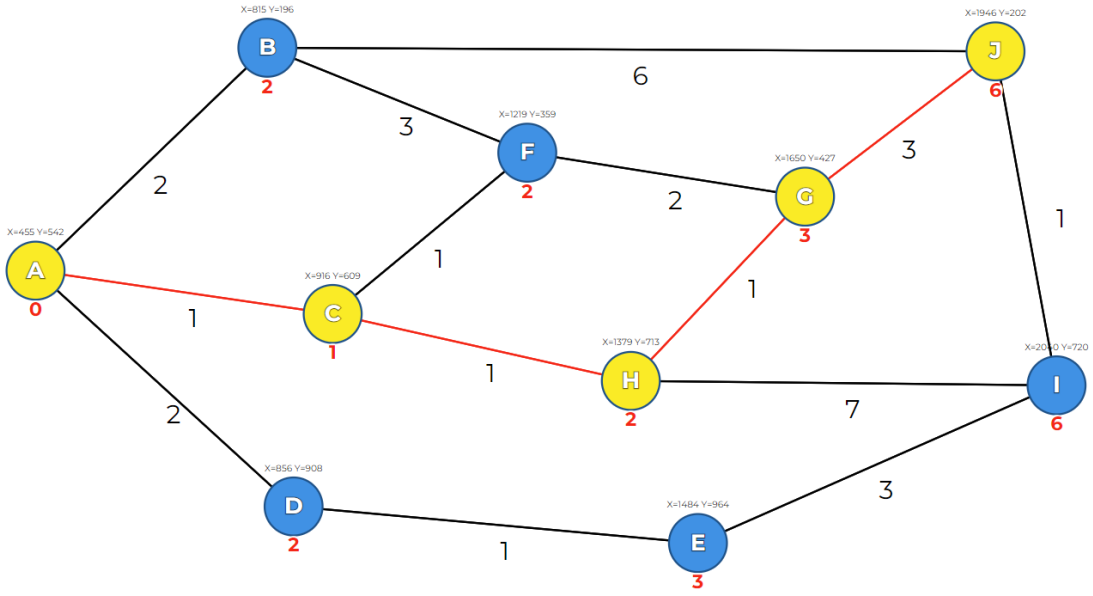
Implementacija algoritma u aplikaciji započinje inicijalizacijom prioritetnog reda (engl. *priority queue*) koji će poredati vrhove koje treba posjetiti na temelju najmanje trenutno poznate udaljenosti od početnog vrha.

Prvi korak algoritma je postavljanje početnog vrha u prioritetni red s udaljenošću nula. Algoritam zatim ponavlja proces izvlačenja vrha s najmanjom udaljenošću iz prioritetnog reda i provjerava sve njegove susjedne vrhove. Za svakog susjeda, izračunava se potencijalna nova udaljenost do tog vrha preko trenutnog vrha. Ako je nova udaljenost manja od trenutno poznate udaljenosti, ažurira se i susjed se ponovno dodaje u prioritetni red s novim prioritetom odnosno udaljenošću, kao što je prikazano u ispisu 25.

```
if (probableCost < cost.get(neighbor.properties.name)) {
      console.log(`Updating Node: ${neighbor.properties.name},
                  New Cost: ${probableCost}`);
      cameFrom.set(neighbor.properties.name, {
      node: currentNode, linkId: link.id });
      cost.set(neighbor.properties.name, probableCost);
      openSet.enqueue(probableCost, neighbor);
      console.log(`Neighbor Enqueued:${neighbor.properties.name},
                  Priority: ${probableCost}`);
                } else {
                    console.log(`No Update for Node:
                    ${neighbor.properties.name}, Current Cost:
                    ${cost.get(neighbor.properties.name)}`);
                }
```

Ispis 25: Dio kôda Dijkstrinog algoritma

Ako algoritam dođe do ciljnog vrha, rekonstruira se put od početnog do ciljnog vrha te se rezultat šalje u funkciju `updateGraph` gdje se animira. Tokom animacije, konstantno se ažuriraju trenutne udaljenosti svakog vrha od početnog, a zapisane su ispod svakog vrha crvenom bojom. Rezultat animacije ovog algoritma prikazan je na slici 23.



Slika 23: Rezultat izvedbe Dijkstrinog algoritma

Također, za ovaj algoritam, može se odabrati opcija da algoritam pronađe najkraći put od određenog vrha do svih ostalih vrhova grafa. Pretraga se ne zaustavlja kada se dođe do određenog vrha, već se nastavlja dok se ne pronađe najkraći put do svakog vrha u grafu.

5.6.2 Implementacija Greedy Best First Search algoritma

Greedy Best-First Search (GBFS) algoritam je algoritam koji se koristi za pretraživanje grafa tako da se pokušava pronaći put do cilja koristeći pristup „pohlepe“ odnosno koristi heurističku funkciju koja procjenjuje koliko je svaki vrh blizu cilja i prema tome odlučuje koji će se vrh sljedeći istražiti.

Koraci algoritma:

1. Postavlja se početni vrh u prioritetni red s odgovarajućom heurističkom vrijednošću.
2. Odabire se vrh s najmanjom heurističkom vrijednošću iz prioritetnog reda i postavlja kao trenutni vrh.
3. Ako je trenutni vrh ciljni vrh, algoritam se završava. Ako nije, istražuju se neposjećeni susjedi i dodaju u prioritetni red s izračunatom heurističkom vrijednošću.

4. Koraci 2 i 3 se ponavljaju dok se ne pronađe ciljni vrh ili dok se ne iscrpe svi vrhovi koje je moguće istražiti.

Algoritam koristi prioritetni red za sortiranje svih vrhova prema njihovoj procijenjenoj udaljenosti od ciljnog vrha, a ta se procjena temelji na heuristici. Ona se izračunava korištenjem pozicije vrha, odnosno x i y koordinata, a primjer kôda je dan u ispisu 26.

```
function heuristic(nodeA, nodeB) {
        const distance = Math.abs(nodeA.x - nodeB.x) +
        Math.abs(nodeA.y - nodeB.y);
        return Math.round(distance);}
```

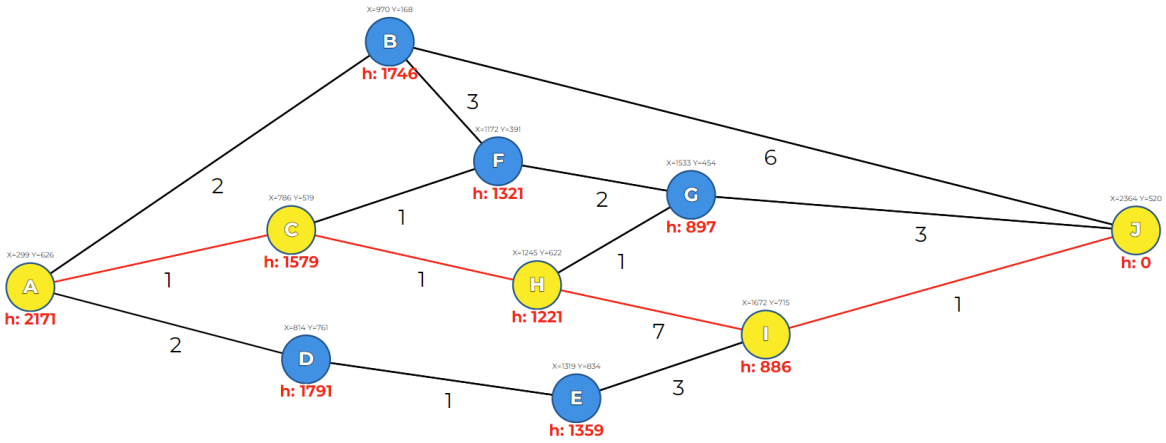
Ispis 26: Računanje udaljenosti dva vrha

Implementacija algoritma u aplikaciji započinje dodavanjem početnog vrha u prioritetni red po heurističkoj procjeni udaljenosti do ciljnog vrha. Potom, sve dok prioritetni red nije prazan, algoritam izvlači vrh s najvišim prioritetom iz reda i provjerava njegove susjedne vrhove. Za svakog susjeda, algoritam izračunava heurističku vrijednost (udaljenost do ciljnog vrha) i dodaje susjeda u prioritetni red, pod uvjetom da nije već posjećen, kao što je prikazano u ispisu 27.

```
if (neighbor && !visited.has(neighbor.properties.name)) {
                    const priority = heuristic(neighbor,
                    endNode);
                    cameFrom.set(neighbor.properties.name, {
                        predecessor: currentNode,
                        linkId: link.id
                    });
                    openSet.enqueue(priority, neighbor);
                    heuristicCost.set(neighbor.properties.name,
                    priority);
                    currentStep.push(Number(link.id));
                    currentStep.push(neighbor.properties.name);
                }
```

Ispis 27: Dio kôda GBFS algoritma

Ako algoritam dođe do ciljnog vrha, rezultat se šalje u funkciju `updateGraph` gdje se animiraju koraci algoritma. Za graf u kojem se traži najkraći put od vrha „A“ do vrha „J“, konačan rezultat animacije je prikazan na slici 24. Oznaka „`h: x`“ ispod svakog vrha, označava udaljenost tog vrha od ciljnog vrha (za ovaj primjer vrha „J“) i to u pikselima (engl. *pixel*).



Slika 24: Rezultat animacije GBFS algoritma

5.6.3 Implementacija A* algoritma

A* (A-star) algoritam koristi se za pronalazak najkraćeg puta između dva vrha u grafu, kombinirajući Greedy Best-First Search i Dijkstrin algoritam. A* koristi heuristiku kako bi procijenio udaljenost do cilja, ali također uzima u obzir stvarne težine odnosno udaljenosti puta od početnog do trenutnog vrha.

Koraci algoritma:

1. Postavlja se početni vrh u prioritetni red s ukupnom udaljenošću nula. Ostali vrhovi imaju beskonačne udaljenosti.
2. Odabire se vrh iz prioritetnog reda s najmanjom vrijednošću funkcije (f(n) = g(n) + h(n)), gdje je g(n) trošak (udaljenost) puta od izvorišnog vrha do trenutnog vrha, a h(n) heuristička procjena preostale udaljenosti od trenutnog vrha do ciljnog vrha.
3. Za svakog neposjećenog susjeda izračunava se nova udaljenost i ažurira ako je manja od trenutne. Dodaje se u prioritetni red s novom vrijednošću funkcije f(n).
4. Koraci 2 i 3 se ponavljaju dok se ne pronađe ciljni vrh ili dok se ne iscrpe svi vrhovi koje je moguće istražiti.

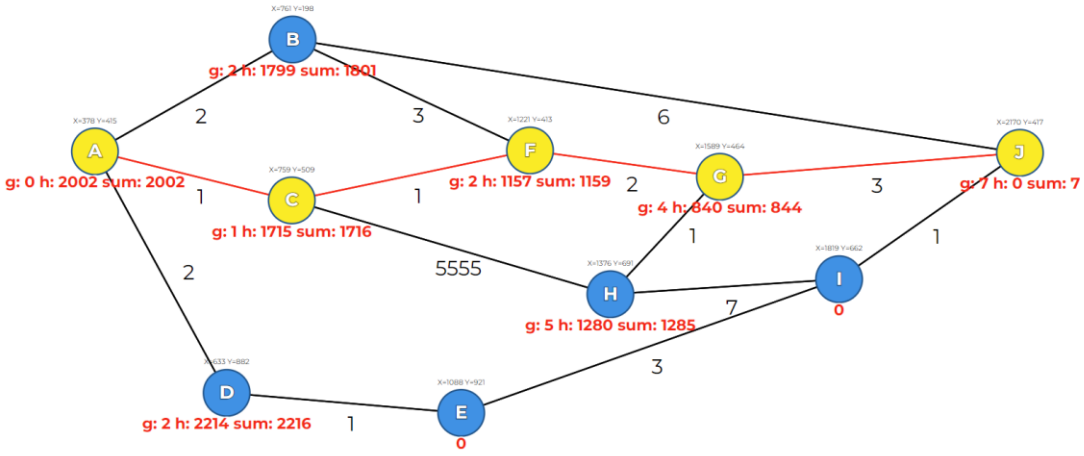
Implementacija algoritma u aplikaciji započinje inicijalizacijom prioritetnog reda, koji sortira vrhove na temelju njihove ukupne procijenjene udaljenosti do ciljnog vrha.

Prvi korak algoritma je postavljanje početnog vrha u prioritetni red s početnom udaljenošću od nula i procjenom heurističke udaljenosti do ciljnog vrha. Algoritam zatim ponavlja proces izvlačenja vrha s najnižim prioritetom iz reda i provjerava sve njegove susjedne vrhove te se za svakog susjeda izračunava potencijalni novi trošak (udaljenost) puta do tog vrha preko trenutnog vrha. Ako je ovaj novi trošak manji od trenutnog, trošak se ažurira, a susjed se ponovno dodaje u prioritetni red s ažuriranom ukupnom procjenom. Primjer kôda je dan u ispisu 28.

```
      if (tentativeStartToCurrent <
                startToCurrent.get(neighbor.properties.name).actual)
                {
                      cameFrom.set(neighbor.properties.name, { node:
                      currentNode, linkId: link.id });
                      startToCurrent.set(neighbor.properties.name,
                           {actual: tentativeStartToCurrent, heuristic:
                           heuristic(neighbor, endNode) });
                      totalCost.set(neighbor.properties.name,
                                    tentativeStartToCurrent
                                    + heuristic(neighbor, endNode));
                      if (!openSet.nodes.some(element =>
                          element.value.properties.name ===
                          neighbor.properties.name)) {
                              openSet.enqueue(
                                totalCost.get(neighbor.properties.name),
                                              neighbor);
                …
```

Ispis 28: Dio kôda A* algoritma

Rezultat se potom šalje u funkciju `updateGraph` gdje se animira. Tokom animacije, ažuriraju se oznake „`g: x`“ (težina brida), „`h: x`“ (heuristička udaljenost) te „`sum: x`“ (ukupna udaljenost) ispod svakog vrha. Rezultat animacije prikazan je na slici 25.



Slika 25: Rezultat animacije A* algoritma

5.6.4 Implementacija Bellman-Ford algoritma

Bellman-Ford algoritam, kojeg su razvila dva američka matematičara, Richard Bellman i Lester R. Ford, koristi se za pronalazak najkraćeg puta od jednog vrha do svih ostalih vrhova u grafu, čak i ako grafovi sadrže negativne težine bridova. Algoritam može otkriti i negativne cikluse u grafu. Negativan ciklus je ciklus čiji je zbroj težina bridova negativan broj.

Koraci algoritma:

1. Postavlja se udaljenost izvorišnog vrha na nulu, a sve ostale na beskonačnost.
2. Za svaki brid u grafu, provjerava se može li se udaljenost od izvorišnog vrha do krajnjeg vrha preko tog brida smanjiti u odnosu na trenutno poznatu udaljenost. Ako može, ta se udaljenost ažurira. Ovaj proces se ponavlja ukupno N−1 puta, gdje je N broj vrhova u grafu.
3. Nakon što se izvrši relaksacija svih bridova N−1 puta, prolazi se još jednom kroz sve bridove. Ako je u tom dodatnom prolasku moguće još smanjiti udaljenosti, to ukazuje na postojanje negativnog ciklusa u grafu. Negativan ciklus omogućava beskonačno smanjenje ukupne težine (udaljenosti) puta prolaskom kroz taj ciklus, što nije moguće u stvarnosti.

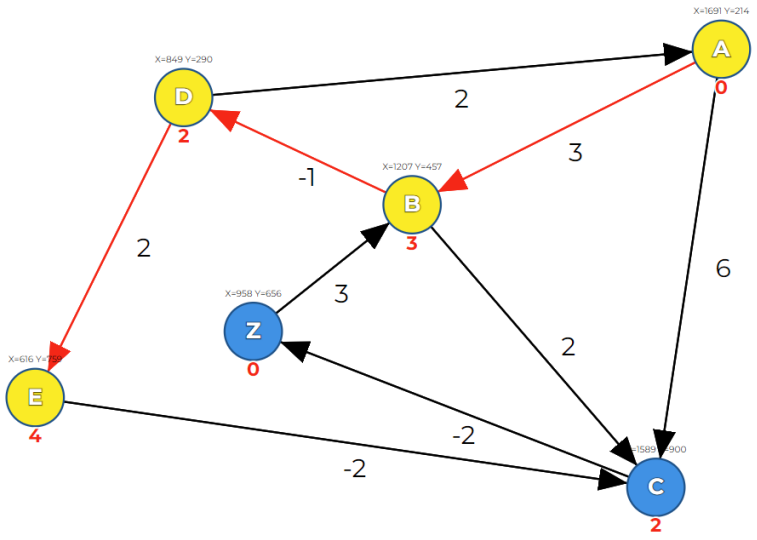
Implementacija algoritma u aplikaciji započinje postavljanjem početnog vrha s udaljenošću nula, dok su svi ostali vrhovi inicijalizirani s beskonačnom udaljenošću. Zatim algoritam prolazi kroz sve bridove u grafu i "relaksira" ih, što znači da provjerava može li se trenutna udaljenost do susjednog vrha smanjiti prelaskom preko trenutnog brida, što je prikazano u ispisu 29. Ovaj proces se ponavlja za sve bridove N-1 puta, gdje je N broj vrhova grafa.

```
      if (newCost < cost.get(targetNode.properties.name)) {
                    cost.set(targetNode.properties.name, newCost);
                    cameFrom.set(targetNode.properties.name, {
                                  node: sourceNode, linkId: link.id });
                    currentSources.push([sourceNode.properties.name,
                                         targetNode.properties.name]);
                    updated = true; }
```

Ispis 29: Dio kôda Bellman-Ford algoritma

Ako nakon k iteracija (gdje k < N−1) više ne dolazi do promjena u udaljenostima između vrhova, algoritam može završiti prijevremeno jer su svi najkraći putevi već pronađeni. Nakon toga, algoritam vrši još jednu iteraciju u kojoj provjerava postoji li negativni ciklus u grafu. Ako se pronađe negativni ciklus, najkraći put nije moguće odrediti jer bi prelazak preko ciklusa stalno smanjivao udaljenost, što bi vodilo u beskonačnost.

Nakon što algoritam pronađe rješenje, šalje ga u funkciju `updateGraph` gdje se rješenje animira i to baš na način N-1 iteracija u kojima se udaljenost vrhova postepeno mijenja. Rezultat animacije ovog algoritma je dan na slici 26.



Slika 26: Rezultat animacije Bellman-Ford algoritma

6. Zaključak

U ovom završnom radu napravljena je interaktivna aplikacija s posebnim naglaskom na vizualizaciju koraka koje algoritam poduzima tokom rješavanja zadanog problema. Aplikacija je osmišljena kao pomoćno sredstvo za korisnike tokom istraživanja i učenja algoritama za grafove, koji su često prisutni u okviru ovog studija.

Za izradu aplikacije korištene su tehnologije Neo4j i D3.js, koje su se pokazale izuzetno korisnima u procesu razvoja i implementacije vizualizacija grafova na zadovoljavajućoj razini.

Aplikaciju je moguće dodatno unaprijediti tako da podržava još više algoritama, uključujući i one koji trenutno rade isključivo na neusmjerenim grafovima, kako bi se omogućilo rješavanje problema i na usmjerenim grafovima. Također, postoji potencijal za implementaciju dodatnih funkcionalnosti koje bi omogućile personalizaciju iskustva korisnika, pa tako i integraciju naprednih analitičkih alata koji bi korisnicima pružili bolji, dublji uvid u rad i statistiku pojedinog algoritma. Sva ova poboljšanja i inovacije bi učinile aplikaciju još boljim alatom za obrazovanje i istraživanje u području računalne znanosti, s posebnim naglaskom na kolegije ovog studija koji u svom kurikulumu obuhvaćaju teme vezane uz teoriju grafova i njihovu primjenu.

Literatura

[1] Baras, I.: „DISKRETNA MATEMATIKA za studente Informacijske tehnologije“ https://moodle.oss.unist.hr/pluginfile.php/44120/mod_folder/content/0/DISKRETNA%20MATEMATIKA%202021%20-%202.dio.pdf?forcedownload=1 (posjećeno 3.8.2024).

[2] Despalatović, L. : „Graf algoritmi“

https://moodle.oss.unist.hr/mod/resource/view.php?id=49189 (posjećeno 3.8.2024)

[3] Neo4j, „Neo4j Graph Database“ https://neo4j.com/product/neo4j-graph-database/ (posjećeno 5.8)

[4] D3JS, „What is D3?“ https://d3js.org/what-is-d3 (posjećeno 5.8)