Vizualizacija matematičkih tema koristeći react i D3

Đurić, Željko

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Željko Đurić

VIZUALIZACIJA MATEMATIČKIH TEMA KORISTEĆI REACT I D3

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Ivica Nakić

Zagreb, 2019

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

React je JavaScript biblioteka za kreiranje korisničkog sučelja. React omogućava kreiranje komponenata koje se mogu ponovno koristiti i zajedno spojiti kako bi se formiralo kompleksno korisničko sučelje. Koristeći React mogu se napraviti brze aplikacije kod kojih je minimizirano dupliciranje koda te je olakšano razvijanje kompleksnog korisničkog sučelja. React aplikacije je lakše razvijati i održavati nego što je to slučaj kod korištenja biblioteka koje su se do nedavno koristile. Primjer jedne od tih biblioteka je jQuery.

Današnji poslovni svijet se bazira na interpretaciji ogromnih količina podataka pa je tako vizualizacija podataka postala jedan od ključnih alata. Vizualizacija podataka se koristi kako bi se na jasan i jednostavan način iskomunicirali podaci. Vizualizacija podataka omogućava ljudima da lakše upiju informacije, da se na brži način interpretira velika količina podataka te kako bi se stečena saznanja mogla podijeliti s drugima na zanimljiv način. Postoji puno biblioteka koje se koriste za upravo tu svrhu, ali jedna se ističe. To je biblioteka D3.

Korištenje biblioteka React i D3 zajedno može biti jako korisno, ali još uvijek ne postoji najbolji način njihove integracije. U ovom diplomskom radu će se dati kratki uvodi kako koristiti navedene biblioteke samostalno. Nakon toga će se opisati problemi koji se javljaju u nastojanju korištenja ovih biblioteka te će se prikazati različiti pristupi kako ih integrirati.

Na kraju se opisuje aplikacija koja koristi React i D3 zajedno. Aplikacija prikazuje vizualizaciju algoritama na grafovima. Na početnoj stranici aplikacije može se odabrati algoritam čiji se koraci žele vizualizirati. Može se birati između Dijkstrinog i Kruskalovog algoritma. Aplikacija dopušta da se na interaktivan način dodaju čvorovi i bridovi, a svakom promjenom izgleda grafa, aplikacija računa nove korake za vizualizaciju.

Poglavlje 1

React

1.1 Uvod

React je JavaScript biblioteka za kreiranje korisničkog sučelja koja je razvijena u Facebook-u. Biblioteka je prvi put upotrebljena na Facebook-ovom newsfeed-u 2011. godine, a kod je postao otvoren 2013. godine. React aplikacija je građena od puno manjih cjelina koje se zovu komponente. Cijela aplikacija je jedna komponenta koja sadrži puno podkomponenti koje zajedno čine kompleksno korisničko sučelje. Komponente se mogu implementirati kao JavaScript klase ili funkcije koje unutar sebe sadrže opis kako trebaju izgledati. React u pozadini čuva efikasnu reprezentaciju DOM-a (Document Object Model) koja se naziva VDOM (Virtual DOM). VDOM koristi čisti JavaScript što omogućava jednostavniju i bržu manipulaciju nego što je to slučaj s čistim DOM-om. Komponente čuvaju informacije kroz svojstva i stanje. Mijenjajući stanje komponente, React ažurira VDOM nakon čega na efikasan način pokušava saznati koje su razlike između VDOM-a i DOM-a te zatim mijenja samo onaj dio DOM-a koji se razlikuje. Na taj način DOM je uvijek sinkroniziran s VDOM-om. Imajući u vidu način na koji React funkcionira, mogu se navesti neke njegove prednosti.

Prednosti React-a

- pregledniji kod
- jednom kreirane komponente se mogu kasnije ponovno koristiti
- lakša manipulacija DOM-a
- brža aplikacija (minimizira se mijenjanje DOM-a)

Zbog navedenih prednosti React je postao jedna od najpopularnijih JavaScript biblioteka. Za razumijevanje React-a potrebno je objasniti pojmove koji su usko vezani uz njega, poput Babel-a i JSX-a.

Babel

Babel je JavaScript kompajler koji prevodi markup ili neki programski jezik u JavaScript. Babel se bazira na plugin sistemu koji parsira moderni JavaScript u apstraktno sintaksno stablo i pretvara ga u oblik koji željeni internet preglednik može koristiti. Koristeći Babel, unutar aplikacije se može koristiti najnovija JavaScript sintaksa (ES6). React koristi Babel kako bi mogao koristiti JSX sintaksu.

JSX

Komponente sadrže opis koji određuje kako će se renderirati unutar DOM-a. Za taj opis koristi se JSX (JavaScript XML). JSX omogućava korištenje sintakse slične HTML-u unutar JavaScripta, koja se onda pomoću Babel-a pretvara u React objekte. Primjer:

```
<div name="ime">Hello</div>
```

se pomoću Babel-a pretvori u:

```
React.createElement( "div", { name: "ime" }, "Hello");
```

1.2 Komponente

Komponente su sastavni dio svake React aplikacije. Komponente čuvaju informacije u JavaScript objektima `props` (svojstva) i `state` (stanje). Svojstva su informacije koje su predane prilikom inicijalizacije komponente i ne mogu se mijenjati. Stanje sadrži informacije koje se odnose na samu komponentu. Komponenta unutar sebe može inicijalizirati, mijenjati i koristiti stanje. Komponente se mogu implementirati na dva načina.

1. Način: Komponenta kao JavaScript funkcija

```
function HelloFrom(props) {
   return <h3>Hello from {props.name}!</h3>;
}
```

Komponenta definrana pomoću funkcije ne može koristiti stanje pa se zato zove i 'komponenta bez stanja'. Za definiranje komponente dovoljno je napraviti funkciju koja prima argument `props` (kako bi bilo moguće pristupiti svojstvima) i vraća JSX izraz koji opisuje kako će se komponenta renderirati unutar DOM-a.

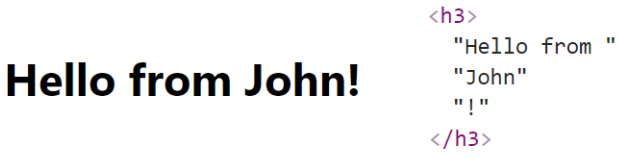
2. Način: Komponenta kao JavaScript klasa

```
class HelloFrom extends React.Component {
    render() {
        return <h3>Hello from {this.props.name}!</h3>;
    }
}
```

Komponenta definirana pomoću klase mora naslijeđivati klasu `React.Component` koja se nalazi unutar biblioteke React. Opis izgleda komponete je sadržan u funkciji `render()` koja se nalazi unutar klase. Komponenta definirana na ovaj način može koristiti i stanje.

Primjer inicijalizacije komponente

```
<HelloFrom name="John" />
```



Slika 1.1: Izgled komponente HelloFrom kojoj je predano svojstvo name.

Stanje

```
class ButtonExample extends Component {
  state = {
    numberOfClicks: 0
  }

  handleOnClick = () => {
    this.setState(prevState => {
      return { numberOfClicks: ++prevState.numberOfClicks };
    });
  }

  render() {
    return <button
        onClick={this.handleOnClick}>{this.state.numberOfClicks}</button>;
  }
}
```

Slika 1.2: Izgled komponente ButtonExample nakon tri klika.

Komponenta `ButtonExample` sadrži stanje koje govori koliko je puta kliknut gumb. Na svaki klik, korištenjem metode `setState()` se mijenja stanje komponente. Budući da je metoda `setState()` asinkrona i da novo stanje ovisi o prethodnom, potrebno je metodi kao argument predati callback funkciju. Ta funkcija kao argument prima prošlo stanje komponente i vraća novo. Nakon svake promjene stanja, ponovno će se renderirati komponenta.

1.3 React aplikacija

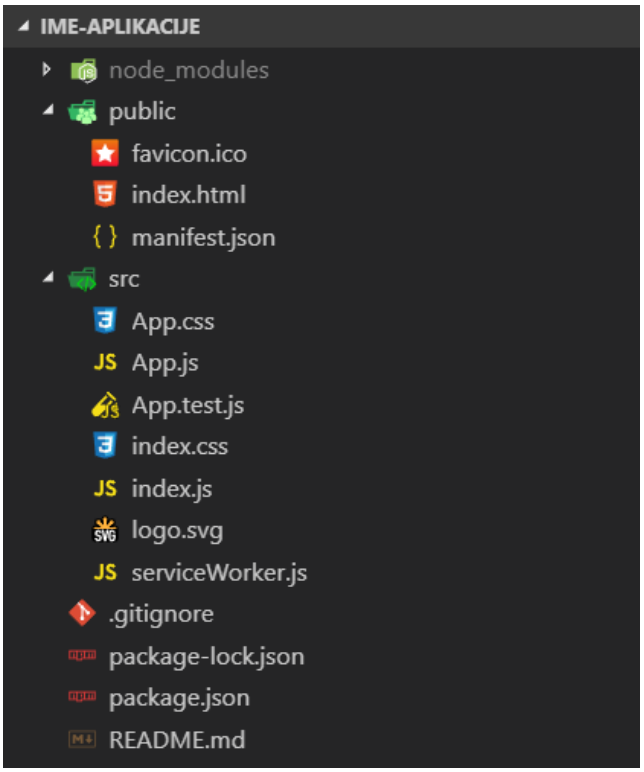
Najpopularniji način kreiranja React aplikacije je korištenjem Facebook-ovog alata koji se zove create-react-app. Koristeći create-react-app jednom naredbom se generira kostur React aplikacije koji sadrži sve alate potrebne za razvoj. Neki od alata su Babel, webpack te različite skripte koje olakšavaju razvoj React aplikacija. Za korištenje create-react-app skripte, na računalu trebaju biti instalirani Node (verzija ≥ 6) i npm (verzija ≥ 5.2).

Kreiranje aplikacije

Za kreiranje aplikacije, dovoljno je u terminalu izvršiti sljedeću naredbu:

```
npx create-react-app ime-aplikacije
```

Nakon izvršavanja te naredbe, generirat će se aplikacija koja će imati naziv 'ime-aplikacije'.



Slika 1.3: Struktura Aplikacije.

Struktura aplikacije

Ovdje će se opisati kako je strukturirana aplikacija i na koji način funkcionira. Prvo će se navesti ključne datoteke koje su generirane i koja je njihova svrha.

- package.json - Unutar ove datoteke se nalazi popis biblioteka o kojoj zavisi generirana aplikacija. Prilikom pokretanja aplikacije sadržaj tih biblioteka će se preuzeti s interneta i staviti u direktorij node_modules. Ako se želi koristiti biblioteka koja se zove xyz, dovoljno je izvršiti naredbu 'npm install xyz'.
- public/index.html - Prilikom pokretanja aplikacije se učita ova stranica. Ovo je jedina HTML datoteka u čitavoj aplikaciji budući da se za React aplikaciju obično koristi JSX.
- src/index.js - JavaScript datoteka koja se odnosi na gore opisanu datoteku index.html.
- src/App.js - U ovoj datoteci se nalazi komponenta App. Komponenta App je glavna komponenta u React aplikaciji unutar koje se nalaze sve ostale komponente.

Dvije datoteke su odgovorne za spajanje React-a s DOM-om. To su datoteke index.html koja se nalazi unutar mape 'public' i datoteka index.js koja se nalazi unutar mape 'src'. U index.html datoteci nalazi se sljedeća linija koda:

```
<div id="root"></div>
```

Unutar tog div elementa će se renderirati cijela React aplikacija. Unutar index.js datoteke može se pronaći sljedeći sadržaj:

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(<App />, document.getElementById('root'));
8
9  // If you want your app to work offline and load faster, you can change
10 // unregister() to register() below. Note this comes with some pitfalls.
11 // Learn more about service workers: http://bit.ly/CRA-PWA
12 serviceWorker.unregister();
```

U 7. liniji koda poziva se metoda `ReactDOM.render()` koja povezuje biblioteku React s DOM-om. Metoda `ReactDOM.render()` prima dva argumenta. Prvi argument sadrži

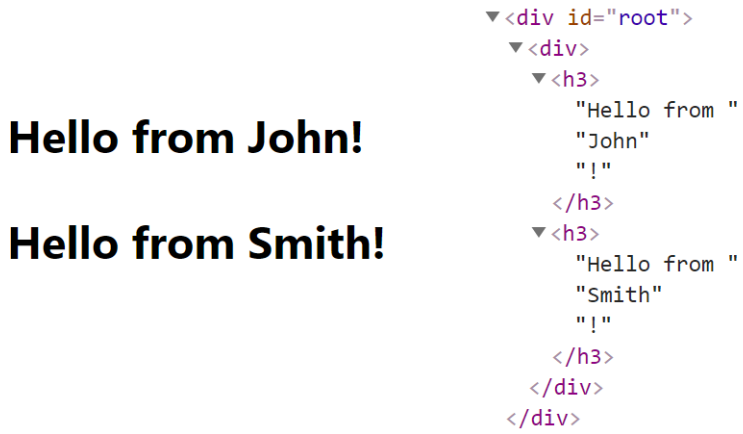
komponentu koju želimo renderirati dok drugi argument sadrži HTML element unutar kojeg će se renderirati komponenta. Preciznije, unutar gore navedenog div elementa će se nalaziti sadržaj koji je pomoću JSX izraza opisan u komponenti App.

Uobičajena praksa prije stvaranja komponenti je napraviti mapu u kojoj će se one nalaziti. Obično se ta mapa zove 'components' i nalazi se unutar mape 'src'. Kreiranjem datoteke HelloFrom.js unutar mape 'components' s ispod navedenim sadržajem i promjenom sadržaja datoteke App.js:

```
//HelloFrom.js
import React, { Component } from 'react';

class HelloFrom extends Component {
  render() {
    return <h3>Hello from {this.props.name}!</h3>;
  }
}

export default HelloFrom;
```

```
//App.js
import React, { Component } from 'react';
import ButtonExample from './components/ButtonExample';

class App extends Component {
  render() {
    return (
      <div>
        <HelloFrom name="John" />
        <HelloFrom name="Smith" />
      </div>
    );
  }
}

export default App;
```

se prilikom pokretanja aplikacije prikazuje sljedeći sadržaj:



Slika 1.4: Izgled gore navedene aplikacije.

Za pokretanje aplikacije u razvojnoj okolini potrebno je izvršiti sljedeću naredbu:

```
npm start
```

Ovdje je prikazano kako je koristeći React moguće napraviti jednu komponentu koja se kasnije može ponovno koristiti. Komponenta `HelloFrom` je jednostavna i ne pokazuje puni potencijal korištenja komponenti. Na ovaj način bi se primjerice mogla kreirati komponenta koja kao svojstvo prima podatke, a renderira HTML tablicu. Ako bi se kasnije trebalo prikazati više tablica ne bi se morao ispočetka pisati HTML kod koji je nepregledan i kojeg je teško održavati. Dovoljno je kreirati novu komponentu s drugim svojstvima. Također, kad bi se u gornjem primjeru htjelo prikazati umjesto 'Hello from John' i 'Hello from Smith' u 'John says hi!' i 'Smith says hi!' dovoljno bi bilo promijeniti sadržaj metode `render()`.

Poglavlje 2

D3

2.1 Uvod

D3 (Data-Driven Documents) je JavaScript biblioteka za manipulaciju dokumentima baziranim na podacima. D3 je nasljednik biblioteke Protovis, a razvio ju je Mike Bostock 2011. godine. D3 se koristi za izradu jednostavnih interaktivnih vizualizacija u web pregledniku. Da bi se mogao koristiti D3, potrebno je dobro poznavati sljedeće tehnologije:

- HTML
- CSS
- SVG
- JavaScript

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <script src="https://d3js.org/d3.v5.min.js"></script>
    </head>
    <body>
        <div class="imeKlase"></div>
        <script>
            //Ovdje ide d3 kod.
        </script>
    </body>
</html>
```

Jedan od načina korištenja D3 biblioteke je njeno uključivanje unutar taga 'head'. Potrebno je postaviti putanju na lokaciju D3 biblioteke. U nastavku diplomskog rada će se koristiti verzija 5. Budući da je D3 JavaScript biblioteka, sav D3 kod se stavlja unutar taga `<script>`. U nastavku će svi primjeri prikazivati samo dio koda koji se nalazi unutar taga `<body>`.

Selekcije

Selekcije su jedan od dva najvažnija koncepta u D3-u. Bazira se na CSS selektorima te omogućava odabir jednog ili više elemenata u HTML dokumentu. Također dopušta modifikaciju, brisanje ili dodavanje novih elemenata u skladu s nekim skupom podataka.

Odabir elemenata

U biblioteci D3, moguće je odabrati elemente pomoću sljedeće dvije metode: `select()` i `selectAll()`. Metoda `select()` odabire prvi DOM element koji odgovara danom CSS selektoru, a ako ima više elemenata koji odgovaraju danom selektoru, vraća prvi odgovarajući. Metoda `selectAll()` funkcionira na sličan način. Jedina razlika je ta što vraća sve elemente koji odgovaraju danom CSS selektoru.

Primjer korištenja selekcija

```
1  <div class="imeKlase">
2     Prvi element.
3  </div>
4  <div class="imeKlase">
5     Drugi element.
6  </div>
7  <script>
8     d3.select('.imeKlase');
9     d3.selectAll('.imeKlase');
10 </script>
```

U 8. liniji koda je korištena metoda `select()` za odabir prvog elementa koji pripada klasi `imeKlase`, a u 9. liniji koda je korištena metoda `selectAll()` za dohvat svih elemenata koji pripadaju toj klasi.

Dodavanje elemenata

D3 pruža mogućnost da se odabranim elementima dodaju novi elementi. Za to se koristi metoda `append()` koja dodaje novi element na kraj odabranih elemanata. Primjer:

```
<div class="imeKlase"></div>
<script>
    d3.select('.imeKlase')
        .append('span');
</script>
```

Slika 2.1: Izgled HTML-a nakon dodavanja elementa.

Mijenjanje elemenata

Za mijenjanje odabranih elemenata koriste se metode `attr()`, `style()` i `text()`. Metoda `attr()` mijenja atribut odabranog elementa, metoda `style()` mijenja stil elementa dok se metoda `text()` koristi za postavljanje sadržaja odabranom elementu. Primjer:

```
<h2 id="id1">Prvi naslov!</h2>
<h2>Drugi naslov!</h2>
<script>
    d3.select('#id1')
        .style('color', 'red')
        .attr('title', 'Opis naslova.')
        .text('Novi naslov!');
</script>
```

Slika 2.2: Izgled HTML-a nakon korištenja metoda za mijenjanje elemenata.

2.2 Spajanje podataka

Što je spajanje podataka?

Spajanje podataka je drugi najvažniji koncept D3-a. Spajanje podataka sa selekcijom omogućava da se ubacuju, modificiraju i brišu elementi u ovisnosti o vrijednostima nekog skupa podataka. Kako se podaci mijenjaju, tako se i elementi koji odgovaraju tim podacima mogu mijenjati. Spajanje podataka sa selekcijom kreira usku vezu između skupa podataka i elemenata u dokumentu.

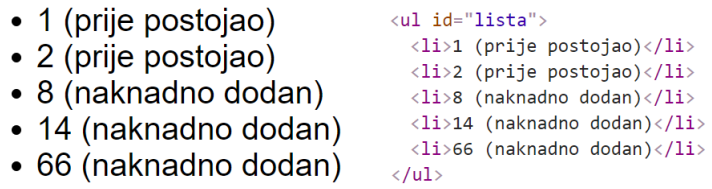
Kako funkcionira?

Glavni zadatak spajanja podataka sa selekcijom je mapiranje elemenata dokumenta s nekim skupom podataka. D3 kreira virtualnu reprezentaciju dokumenta u ovisnosti o zadanom skupu podataka i pruža metode koje omogućuju rad s virtualnom reprezentacijom. To su metode `data()`, `enter()` i `exit()`. Funkcija `data()` se koristi kako bi se spojila kolekcija elemenata iz HTML dokumenta sa skupom podataka. Metoda `data()` se koristi nakon selekcije HTML elemenata. Metoda `enter()` vraća skup podataka za koje trenutno ne postoje elementi u dokumentu. Metoda `exit()` vraća elemente za koje podaci više ne postoje, oni se mogu izbrisati korištenjem metode `remove()`. U sljedećim primjerima se prikazuje kako se koriste prethodno navedene metode.

Korištenje metode enter()

```
1  <ul id="lista">
2     <li></li>
3     <li></li>
4  </ul>
5
6  <script>
7     var dataSet = [1, 2, 8, 14, 66];
8     d3.select('#lista')
9        .selectAll('li')
10          .data(dataSet)
11             .text(function(d) { return d + ' (prije postojao)' })
12             .enter()
13                .append('li')
14                   .text(function(d) { return d + ' (naknadno dodan)' });
15 </script>
```

U 8. liniji koda dohvaća se lista iz koje se u 9. liniji koda dohvaćaju svi elementi s tagom <li>. U ovom primjeru to su samo dva elementa. Nakon spajanja elemenata s nizom definiranim u 7. liniji koda, dobije se virtualna reprezentacija dokumenta. U virtualnoj reprezentaciji se nalazi 5 elemenata. Prva dva elementa odgovaraju selekciji koja se vraća u 9. liniji koda. U 11. liniji koda mijenjaju se svi elementi koji su prije postojali. Pomoću metode `enter()` u 12. liniji koda se dohvaćaju svi podaci za koje ne postoji niti jedan element. U ovom primjeru, to su brojevi 8, 14 i 66. Za svaki dobiveni broj kreira se novi element liste, <li>. U 14. liniji koda se postavlja novi tekst tim elementima.



Slika 2.3: Primjer korištenja metode enter().

Korištenje metode exit()

Uz pretpostavku da je u prošlom primjeru na početku bilo šest <li> elemenata u listi, korištenjem prethodnog D3 koda rezultat izvršavanja bi bio sljedeći:

- 1 (prije postojao)
- 2 (prije postojao)
- 8 (prije postojao)
- 14 (prije postojao)
- 66 (prije postojao)
-

Slika 2.4: Izgled HTML-a bez korištenja metode exit()

D3 u ovom slučaju spoji prvih pet <li> elemenata s nizom, ali ostaje još jedan element za kojeg ne postoji podatak s kojim bi ga mogao spojiti. Za isti rezultat kao u prošlom primjeru, potrebno je obrisati svaki element dokumenta koji nema odgovarajući element u nizu. Za to se koriste metode `exit()` i `remove()`.

```
1  <ul id="lista">
2      <li><li>
3      <li><li>
4      <li><li>
5      <li><li>
6      <li><li>
7      <li><li>
8  </ul>
9  <script>
10   var dataSet = [1, 2, 8, 14, 66];
11   var lista = d3.select('#lista')
12      .selectAll('li')
13       .data(dataSet);
14   lista.text(function(d) { return d; });
15   lista
16      .exit()
17          .remove();
18 </script>
```

U 16. liniji koda dohvaćaju se svi elementi za koje ne postoji niti jedan podatak koji je vezan uz njih. U ovom primjeru postoji 6 elemenata, ali samo 5 podataka u nizu, zbog čega će metoda `exit()` vratiti 6. element liste koji se zatim u 17. liniji koda obriše.

- 1 (prije postojao)
- 2 (prije postojao)
- 8 (prije postojao)
- 14 (prije postojao)
- 66 (prije postojao)

Slika 2.5: Izgled HTML-a nakon korištenja metode exit().

2.3 SVG

SVG (Scalable Vector Graphics) je XML jezik za prikazivanje dvodimenzionalne vektorske grafike. SVG se jako često koristi u kombinaciji s D3-em budući da dopušta da se niz podataka vizualizira crtežima.

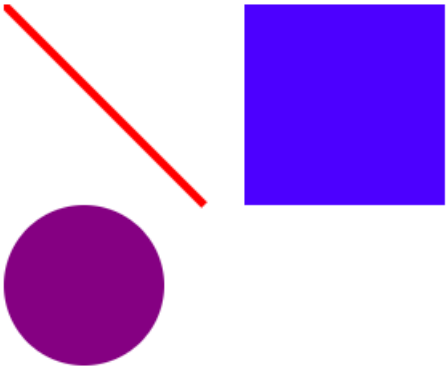
```
<svg width="300" height="300">
    <line x1="0" y1="0" x2="50" y2="50" stroke="red"
        stroke-width="2"></line>
    <rect x="60" y="0" width="50" height="50" fill="blue"></rect>
    <circle cx="20" cy="70" r="20" fill="purple"></circle>
</svg>
```

Za crtanje linije koristi se tag `<line>` koji kao atribute prima `x1`, `y1`, `x2`, `y2` koji označavaju koordinate početka i kraja linije.

Za crtanje pravokutnika koristi se tag `<rect>` koji kao atribute prima `x`, `y`, `width`, `height` koji označavaju koordinate gornjeg lijevog kuta pravokutnika te njegovu visinu i širinu.

Za crtanje kruga koristi se tag `<circle>` koji kao atribute prima `cx`, `cy`, `r` koji označavaju koordinate središta kruga i njegov radijus.

Za definiranje boje ruba oblika koristi se atribut `stroke`, a za definiranje boje ispune oblika koristi se atribut `fill`.



Slika 2.6: Rezultat izvršavanja gore navedenog SVG-ja.

2.4 Force layout

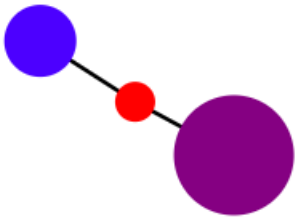
D3 pruža brojne mogućnosti za vizualizaciju podataka. Jedna od tih mogućnosti je force layout. D3-ev force layout pruža mogućnost da se postavljaju sile između elemenata. To je posebno korisno kod crtanja grafova.

Koristeći D3 na jednostavan način se mogu rasporediti elementi po SVG-u na pregledan način. Force layout funkcionira na način da se prvo postave elementi na kojima će se pokrenuti simulacija. Zatim se zadaju sile koje će se koristiti. Nakon što se pokrene, simulacija se izvršava iterativno. Nakon svake iteracije, D3 računa nove pozicije elementima nad kojima je pokrenuta simulacija. Nakon svake iteracije se mogu ažurirati pozicije elemenata.

Primjer korištenja force layouta

```
1  var width = 500, height = 500;
2
3  var nodes = [
4      { name: 'a', color: 'red', radius: 5 },
5      { name: 'b', color: 'blue', radius: 9 },
6      { name: 'c', color: 'purple', radius: 15 }
7  ]
8
9  var links = [
10     { source: 'a', target: 'b' },
11     { source: 'a', target: 'c' }
```

```
12 ]
13
14 var svg = d3.select('body')
15     .append('svg')
16         .attr('width', width)
17         .attr('height', height);
18
19 var circles = svg.selectAll("circle")
20     .data(nodes)
21         .enter()
22             .append("circle")
23                 .attr("fill", function(d) { return d.color; })
24                 .attr('r', function(d) { return d.radius; });
25
26 var lines = svg.selectAll("line")
27     .data(links)
28         .enter()
29             .append("line")
30                 .attr("stroke", "black");
31
32 var simulation = d3.forceSimulation(nodes)
33     .force('charge', d3.forceManyBody())
34     .force('x', d3.forceX().x(width / 2))
35     .force('y', d3.forceY().y(height / 2))
36     .force('link', d3.forceLink().links(links).id(function(d) { return
           d.name }))
37     .on('tick', ticked);
38
39 function ticked() {
40     lines
41         .attr("x1", function(d) { return d.source.x; })
42         .attr("y1", function(d) { return d.source.y; })
43         .attr("x2", function(d) { return d.target.x; })
44         .attr("y2", function(d) { return d.target.y; });
45
46     circles
47         .attr("cx", function(d) { return d.x; })
48         .attr("cy", function(d) { return d.y; });
49 }
```



Slika 2.7: D3-jev Force Layout.

Za korištenje Force Layout-a potrebno je inicijalizirati niz objekata koji će ga koristiti. U ovom primjeru je inicijalizirana simulacija koja koristi sljedeće sile: metoda `forceManyBody()` osigurava da se krugovi nacrtani pomoću SVG-a međusobno odbijaju. Metode `forceX()` i `forceY()` osiguravaju da se cijeli graf nalazi na pozicijama `width/2` i `height/2` koje u ovom slučaju predstavljaju sredinu SVG-a. Metoda `forceLink()` se koristi kako bi se ažurirale pozicije bridova. Metoda `forceLink()` vraća objekt koji sadrži metodu `id()` koja kao argument prima identifikator po kojem će bridovi znati kojim čvorovima pripadaju. U ovom primjeru koristit će se atribut name koji onda svaki čvor mora sadržavati. Kako bi se elementi ažurirali definira se metoda `ticked()` koja se poziva nakon svake iteracije simulacije. Kako će Force Layout nizovima `nodes` i `links` ažurirati pozicije, tako će se na SVG-u ažurirati pozicije krugova koji predstavljaju čvorove i pozicije linija koje predstavljaju bridove.

Poglavlje 3

React i D3 zajedno

3.1 Uvod

U prošla dva poglavlja je opisano kako koristiti biblioteke React i D3. Kad te dvije biblioteke treba koristiti zajedno, nastaje problem. Glavni izazov integracije D3-a s React-om je taj što obje biblioteke žele kontrolu nad DOM-om. Način na koji D3 mijenja DOM je potpuno različit od načina na koji to čini React. Za integraciju je potrebno prepustiti kontrolu DOM-a samo jednoj biblioteci. Postoji puno različitih pristupa koji rješavaju problem integracije. U ovom poglavlju će biti opisano par najpopularnijih te će se ukratko opisati njihove prednosti i mane.

3.2 Postavljanje projekta

Zajednička stvar pristupima koji će biti navedeni je postavljanje projekta. Veći dio postavljanja obavi skripta create-react-app koja je opisana u prvom poglavlju. Spomenuto je kako se u datoteci package.json nalazi popis biblioteka koje generirana aplikacija koristi. React aplikaciji je zadano koje biblioteke koristi, ali ako se želi koristiti biblioteka koja se ne nalazi u datoteci package.json, to se mora eksplicitno navesti. Izvršavanjem naredbe

```
npm install d3@5
```

unutar datoteke package.json će se pojaviti sljedeća linija koda:

```
"d3": "^5.9.0"
```

Aplikacija na ovaj način zna da treba koristiti biblioteku D3, i to najnoviju verziju 5. Preciznije, ako bi izašla verzija 5.9.1, aplikacija bi preuzela tu biblioteku s interneta, ali ako bi izašla verzija 6, prilikom pokretanja aplikacije ta verzija ne bi bila preuzeta.

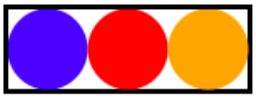
3.3 Korištenje biblioteka React i D3 zajedno

Prvi pristup: D3 renderiranje unutar biblioteke React

Ovo je najpopularniji način korištenja D3-a s React-om. U ovom pristupu React se koristi kako bi se napravila osnovna struktura aplikacije dok se D3-u prepusti kontrola nad elementima koji su zaslužni za vizualizaciju podataka. Obično je to SVG. Ovaj pristup napravi 'crnu kutiju' u kojoj se nalazi čisti D3, gdje se onda mogu koristiti D3 metode za manipuliranje DOM-om. Glavna prednost ovog pristupa je ta što za većinu D3 vizualizacija postoji već neki primjer koji se može koristiti kao polazna točka. Ovim pristupom svaki D3 kod se može na jednostavan način implementirati unutar React aplikacije. Mana ovog pristupa je ta što se ne koristi React-ov VDOM koji na efikasan način računa što bi se trebalo promijeniti u DOM-u.

```
1  import React, { Component } from 'react';
2  import * as d3 from 'd3';
3
4  class ThreeCircles extends Component {
5
6    createVisualization = () => {
7      const data = [
8        { position: 0, color: 'blue' },
9        { position: 1, color: 'red' },
10       { position: 2, color: 'orange' }
11     ];
12
13     d3.select(this.refs.svg)
14       .selectAll('circle')
15       .data(data)
16       .enter()
17         .append('circle')
18         .attr('cx', d => d.position * 20 + 10)
19         .attr('cy', 10)
20         .attr('r', 10)
21         .attr('fill', d => d.color);
22   }
23
24   componentDidMount(){
25     this.createVisualization();
26   }
27
28   componentDidUpdate(){
```

```
29      this.createVisualization();
30    }
31
32    render() {
33      return <svg ref="svg" width={62} height={22} style={{border: '1px
            solid black'}}/>;
34    }
35  }
36
37  export default ThreeCircles;
```



Slika 3.1: Prvi pristup.

Da bi ovaj pristup bio moguć, koriste se metode koje su prisutne u svakoj React komponenti. Metoda `componentDidMount()` se poziva nakon prvog renderiranja komponente dok se metoda `componentDidUpdate()` poziva kad komponenta primi nova svojstva. D3 je u ovom primjeru dohvatio SVG koristeći referencu. Unutar React-a svakoj komponenti kao svojstvo se može predati `ref`. To svojstvo označava referencu. Ako se kasnije želi pristupiti komponenti koja kao svojstvo `ref` ima vrijednost `'svg'`, to se može učiniti dohvaćajući `this.refs.svg`. Ako bi se u ovom primjeru proslijedilo svojstvo

```
ref = 'svgKomponenta'
```

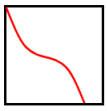
komponenta bi se dohvatila na sljedeći način: `this.refs.svgKomponenta`.

Drugi pristup: D3 za kalkulaciju, React za DOM

Koristeći ovaj pristup, pokušava se što manje koristiti D3. D3 se koristi samo za računanje toga što treba prikazati. Ovaj pristup je moguć zato što D3 sadrži mnogo modula koji nisu vezani za DOM. Primjer je već spomenuti force layout.

Prednost ovog pristupa je ta što je u duhu React-a i može se u potpunosti iskoristiti VDOM što ubrzava aplikaciju. Iako se u ovom pristupu pokušava smanjiti korištenje D3-a, zahtijeva puno znanja o D3-u. Neke D3 funkcionalnosti se moraju ponovno implementirati, što je u pravilu dosta velik posao.

```
import React, { Component } from 'react';
import * as d3 from 'd3';

class PathUsingD3 extends Component {
  state = {
    path: null
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    if (!nextProps.data) return null;

    var lineFunction = d3.line()
      .x(d => d.x)
      .y(d => d.y)
      .curve(d3.curveMonotoneX);

    return { path: lineFunction(nextProps.data) };
  }

  render() {
    return (
      <svg width={this.props.width} height={this.props.height}
          style={{border: '1px solid black'}}>
        <path d={this.state.path} fill="none" stroke="red"/>
      </svg>
    );
  }
}

export default PathUsingD3;
```



```
<svg width="50" height="50" style="border: 1px solid black;">
  <path d=
  "M0,0C3.3333333333333335,8.333333333333334,6.666666666666666,16.66
  6666666666668,10,20C16.666666666666668,26.666666666666668,23.33333
  3333333332,23.333333333333332,30,30C33.333333333333336,33.33333333
  3333336,36.666666666666664,41.666666666666664,40,50" fill="none"
  stroke="red"></path>
</svg>
```

Slika 3.2: Izgled linije koju je D3 izračunao a React renderirao.

Ovdje se pojavljuje statička metoda `getDerivedStateFromProps()`. To je metoda koju svaka React komponenta sadrži. Metoda `getDerivedStateFromProps()` pokreće se svaki put kad se promjene svojstva. Metoda kao argumente prima nova svojstva i prošlo stanje komponente. Metoda vraća novo stanje komponente nakon koje će se komponenta ponovno renderirati.

U ovom primjeru se prvi put pojavljuju i neke nove funkcionalnosti biblioteke D3. Metoda `line()` vraća funkciju koja prima niz podataka i vraća `d` atribut od SVG elementa `path`. Da bi funkcija `line()` znala koje vrijednosti u nizu predstavljaju x i y vrijednosti, potrebno je predati anonimne funkcije koje govore koji dio elemenata treba uzeti u obzir. Na kraju je još potrebno reći kako treba izgledati linija. D3 daje mogućnost izbora između puno različitih načina. Ovdje se koristi `curveMonotoneX`, koji stvara liniju koja je neprekidna po zadanim x koordinatama.

Svaki put kad se promijene svojstva komponente, npr. svojstvo `data`, D3 će izračunati kako bi trebalo popuniti atribut `d` da bi dobili liniju koja ide po zadanim koordinatama sa željenim izgledom. Nakon toga se vraća objekt koji će ažurirati stanje React komponente.

Gore navedena komponenta se unutar neke druge komponente poziva na sljedeći način. Prikazuje se samo dio unutar `render()` metode.

```
const data = [
    { "x": 0, "y": 0},
    { "x": 10, "y": 20},
    { "x": 30, "y": 30},
    { "x": 40, "y": 50},
];

return <PathUsingD3 width={50} height={50} data={data} />
```

Može se vidjeti da koristeći modul od D3-a za crtanje linija, na jednostavan način možemo nacrtati liniju koju bi inače bilo jako teško nacrtati.

Treći pristup: Korištenje biblioteka koje koriste React i D3

Budući da je jedna od prednosti React-a ta da se komponente mogu ponovno koristiti, nije čudo da postoji puno različitih biblioteka koje koriste React i D3 zajedno. Najveća mana navedenih biblioteka je ta što se ne mogu proširiti funkcionalnosti. Možemo koristiti jedino implementirane komponente. U nastavku se navode neke od tih biblioteka:

- Victory
- Recharts
- Nivo
- VX
- Britecharts React

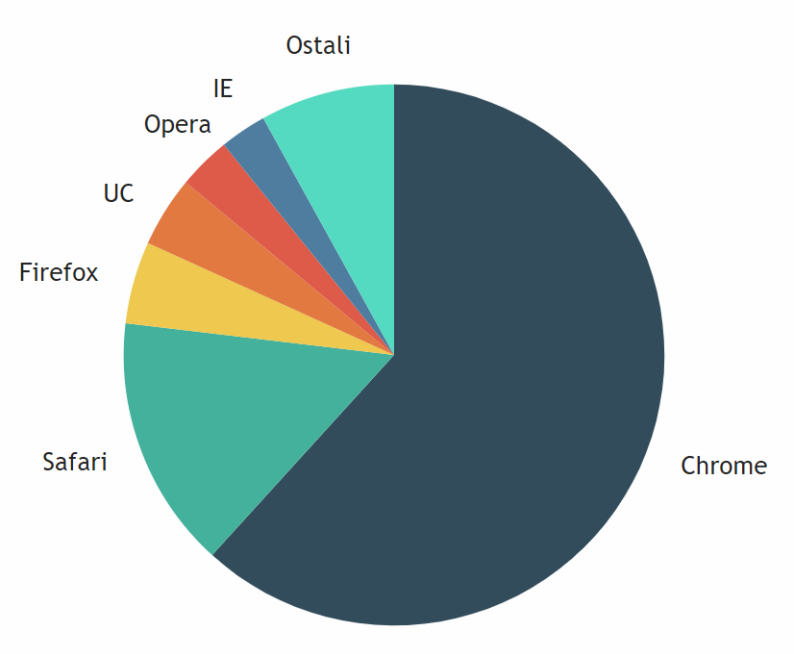
Sada će se opisati kako koristiti jednu od navedenih biblioteka. Kao primjer će se koristiti biblioteka Victory. Slično kao i kod korištenja D3-a, potrebno je navesti da će aplikacija koristiti biblioteku Victory. To se učini izvršavanjem naredbe:

```
npm install victory
```

Nakon toga se može napraviti komponenta sa sljedećim sadržajem:

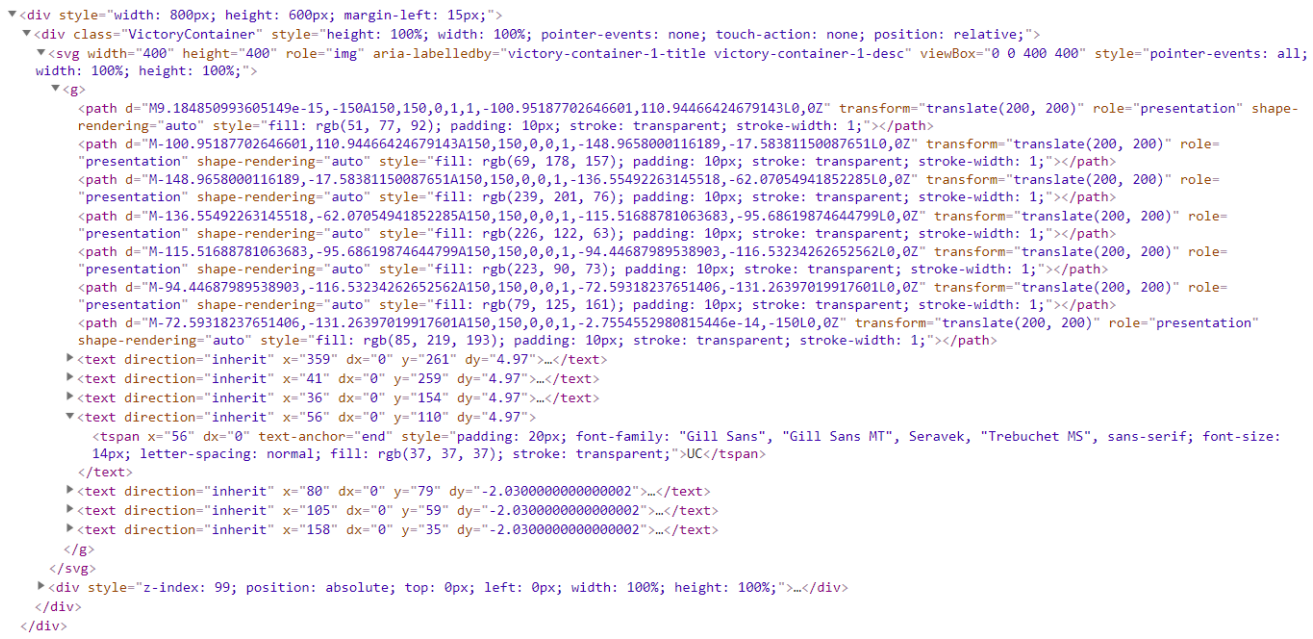
```
1  import React, { Component } from 'react';
2  import { VictoryPie } from "victory";
3
4  class BrowserUsagePieChart extends Component {
5
6    render() {
7      const pieChartData = [
8        { x: "Chrome",          y: 61.75 },
9        { x: "Safari",          y: 15.12 },
10       { x: "Firefox",         y: 4.92 },
11       { x: "UC",              y: 4.22 },
12       { x: "Opera",           y: 3.15 },
13       { x: "IE",              y: 2.8 },
14       { x: "Ostali",          y: 8.04 }
15     ];
16
```

```
17      return (
18        <div style={{ width: 800, height: 600, marginLeft: 15}}>
19          <VictoryPie data={pieChartData} colorScale="qualitative" />
20        </div>
21      );
22    }
23  }
24
25  export default BrowserUsagePieChart;
```



Slika 3.3: Izgled komponente BrowserUsagePieChart.

Ova komponenta opisuje distribuciju korištenja internet preglednika u 2018. godini. U 2. liniji koda se uveze komponenta `VictoryPie` iz biblioteke Victory koja se koristi za crtanje kružnih grafova. Kako bi se nacrtao kružni graf dovoljno je predati dva svojstva. Svojstvo `data` opisuje podatke koje treba prikazati dok svojstvo `colorScale` govori koje boje koristiti. U pozadini je biblioteka generirala sljedeći HTML kod:



Slika 3.4: Generirani HTML kod komponente BrowserUsagePieChart.

Po HTML kodu može se zaključiti da crtanje kružnih grafova nije jednostavan zadatak. Korištenjem komponente biblioteke Victory bitno se olakšava izrada kružnog grafa. Ako bi se kasnije podaci trebali promijeniti, dovoljno bi bilo promijeniti niz `pieChartData` definiran u 6. liniji koda.

3.4 Kako koristiti React i D3?

Korištenje React-a i D3-a je moguće, ali nije uvijek najjasnije koja je vrsta integracije najbolja. Biranje pristupa na koji će se koristiti D3 i React ovisi o više varijabli. Ako postoji biblioteka koja radi točno ono što se želi implementirati, logično je da će tada najbolji pristup biti korištenje točno te biblioteke. Biranje između prvog i drugog navedenog pristupa ovisi o iskustvu ljudi koji će raditi komponente. Ako osoba ima iskustva s D3-em, njoj će biti logičniji izbor prvi pristup, ali tada se gubi prednost koju React donosi kod manipulacije DOM-a. Ako osoba jako dobro poznaje React i D3, možda je bolje koristiti drugi pristup, ali se onda neke funkcionalnosti koje D3 sadrži moraju ponovno implementirati.

Prva dva pristupa imaju zajedničku prednost, a to je da kreiraju komponentu koju kasnije možemo na jednostavan način ponovno koristiti.

Poglavlje 4

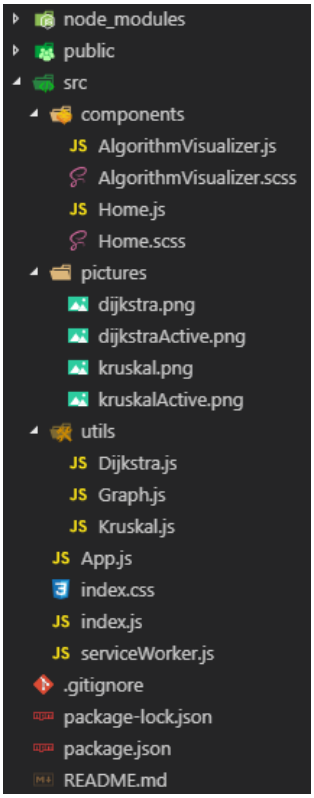
Aplikacija za vizualizaciju algoritama

4.1 Uvod

U ovom poglavlju će biti opisana implementacija složenije aplikacije koja koristi React i D3. Napravit će se aplikacija za vizualizaciju algoritama na grafovima. Za izradu korisničkog sučelja će se koristiti React, a za vizualizaciju D3. Preciznije, koristit će se prvi pristup iz prethodnog poglavlja. Na početnoj stranici aplikacije bira se algoritam koji se želi vizualizirati. Može se birati između Dijkstrinog i Kruskalovog algoritma. Aplikacija dopušta interaktivno dodavanje čvorova i bridova te na svaku promjenu grafa računa nove korake algoritma. U aplikaciji je moguće promatrati izvođenje algoritama korak po korak.

4.2 Struktura projekta

Glavna logika aplikacije se nalazi u datotekama AlgorithmVisualizer.js, Graph.js, Dijkstra.js i Kruskal.js. Ključna komponenta ove aplikacije je `AlgorithmVisualizer`. Ona se koristi dva puta, prvi put za vizualizaciju Dijkstrinog, a drugi put za vizualizaciju Kruskalovog algoritma. Na ovaj način se izbjegava dupliciranje koda. Komponenti je dovoljno predati svojstvo `Algorithm` i ona će znati kako iskoristiti to svojstvo za vizualizaciju algoritma. U klasi Graph.js je implementirana klasa koja predstavlja graf. U datotekama Dijkstra.js i Kruskal.js su redom implementirane klase `Dijkstra` i `Kruskal` koje generiraju korake algoritama. Budući da Dijkstrin i Kruskalov algoritam rade na grafovima, algoritmi su implementirani kao klase koje kao argument konstruktora primaju instancu klase `Graph`.



Slika 4.1: Struktura aplikacije.

Graph.js

```
export default class Graph {
  constructor(nodes = [], links = []) { ...
  }

  addNode(node) { ...
  }

  addLink(link) { ...
  }

  isCyclic() { ...
  }
}
```

Klasa Graph kao argumente konstruktora prima čvorove i bridove. Klasa Graph također sadrži metode kojima možemo dodavati nove čvorove i bridove te provjeravati postoji li u grafu ciklus.

Klase Kruskal i Dijkstra

Klase `Kruskal` i `Dijkstra` zadužene su za generiranje koraka algoritama koje kasnije komponenta `AlgorithmVisualizer` koristi da bi ažurirala izgled DOM-a. Klase sadržavaju metode `getStep()` i `getNumberOfSteps()` koje komponenta `AlgorithmVisualizer` koristi za vizualizaciju. Metoda `getStep()` vraća objekt koji sadrži dva niza, `nodes` i `links`. Objekt niza `nodes` sadržava identifikator čvora kojeg treba mijenjati i objekt `props` koji opisuje novi izgled čvora. Objekt niza `links` sadržava identifikatore dva čvora koji određuju brid i objekt `props` koji opisuje novi izgled brida. U slučaju da metoda `getStep()` vrati sljedeći objekt:

```
{
  nodes: [{
    id: "0",
    props: {
      color: "red"
    }
  }, {
    id: "2",
    props: {
      color: "orange"
    }
  }],
  links: [{
    source: "0",
    target: "2",
    props: {
      weight: 18
    }
  }]
}
```

Gore prikazani objekt komponenti `AlgorithmVisualizer` proslijedi informaciju da treba promijeniti boju čvorovima sa oznakama 0 i 2 i težinu brida koji je određen čvorovima sa oznakama 0 i 2.

AlgorithmVisualizer

Komponenta `AlgorithmVisualizer` koristi force layout koji je opisan u 2. poglavlju. Na komponentu su vezani nizovi `nodes` i `links` koji predstavljaju čvorove i bridove koje treba prikazati. Komponenta sadrži metodu `restart()` koja koristi navedene nizove kao argumente za pokretanje force layout-a. Komponenta omogućava dodavanje novih čvorova i bridova. Svaki put kad dođe do izmjene navedenih nizova ponovno se poziva metoda `restart()` koja ažurira SVG elemente. U nastavku slijedi opis svojstava koje komponenta prima i kako svojstva utječu na izgled komponente:

- algorithmName - String koji se prikazuje u gornjem lijevom kutu. U ovoj aplikaciji su se predali stringovi 'Kruskal' i 'Dijkstra' te je na taj način osigurano da se zna koji se algoritam vizualizira.

- Algorithm - Klasa koja sadrži metode getNumberOfSteps() i getSteps(). Koristi se za prikazivanje svakog koraka algoritma.

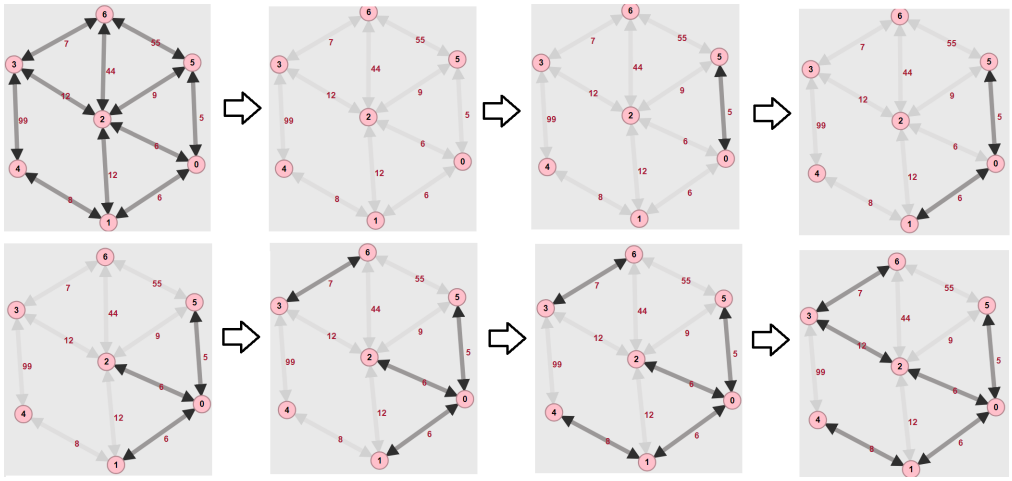
- nodeColor - Funkcija koja prima jedan argument i vraća boju u ovisnosti o tom argumentu. Funkcija nodeColor opisuje kojom će bojom biti opisani čvorovi u grafu koji se koristi za vizualizaciju.

- undirected - Boolean koji određuje da li će graf biti usmjeren ili neusmjeren.

- showDistance - Boolean koji određuje hoće li se iznad svakog čvora prikazivati udaljenost.

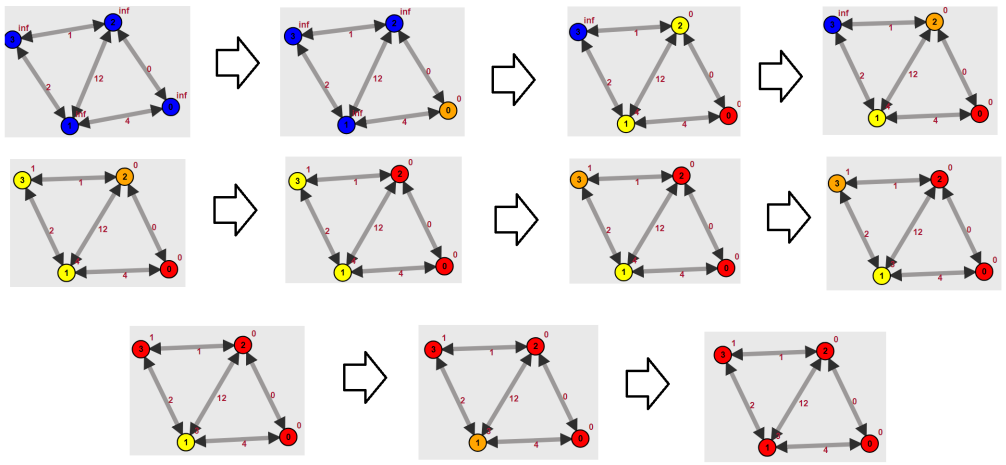
Primjeri korištenja

```
<AlgorithmVisualizer algorithmName="Kruskal" Algorithm={Kruskal}
    nodeColor={() => 'pink' } undirected={true} showDistance={false}/>

<AlgorithmVisualizer algorithmName="Dijkstra" Algorithm={Dijkstra}
    nodeColor={d => d.color} undirected={true} showDistance={true}/>
```



Slika 4.2: Vizualizacija Kruskalovog algoritma.

U ovom primjeru Kruskalov algoritam sadrži 8 koraka. Svaki korak opisuje koje bridove treba sakriti a koje prikazati. Kad komponenta `AlgorithmVisualizer` dobije koje bridove treba sakriti, ona im poveća prozirnost. Na taj način u svakom koraku možemo vidjeti koje bridove algoritam bira. Kad bi za svojstvo `nodeColor` bila postavljena sljedeća anonimna funkcija `() => 'white'`, algoritam bi za vizualizaciju koristio čvorove bijele boje. Budući da je proslijeđeno svojstvo `showDistance` koje je postavljeno na vrijednost `false`, iznad čvorova se ne prikazuju udaljenosti.



Slika 4.3: Vizualizacija Dijkstrinog algoritma.

U ovom primjeru Dijkstrin algoritam sadrži 11 koraka. Svaki korak opisuje u koju boju se čvorovi trebaju obojati te se ažuriraju udaljenosti do čvorova koji se ovog puta prikazuju jer je proslijeđeno svojstvo `showDistance` koje je postavljeno na vrijednost `true`. U ovoj komponenti je svojstvo `nodeColor` definirano na drugačiji način. Poznato je da je svaki čvor predstavljen jednim objektom niza `nodes`. Proslijeđena funkcija preda uputu komponenti da za svaki čvor nađe objekt koji ga predstavlja u nizu `nodes` te iz njega pročita vrijednost svojstva `color` i na kraju oboja čvor tom bojom.

Navigacija

Kod opisivanja strukture React aplikacije, spomenuto je kako postoji samo jedna HTML datoteka. React je SPA (Single Page Application). To znači da se cijela aplikacija učita odmah, a kasnije se koristeći JavaScript dobije dojam kako koristimo klasičnu web aplikaciju. React to radi koristeći biblioteku 'react-router-dom'. Kao i prije, aplikaciji treba reći da se želi koristiti biblioteka 'react-router-dom'. To se radi izvršavanjem naredbe:

```
npm install react-router-dom
```

U datoteteci App.js se može pronaći sljedeći sadržaj:

```
1  import React, { Component } from 'react';
2  import { BrowserRouter, Route } from "react-router-dom";
3  import Dijkstra from './utils/Dijkstra';
4  import Kruskal from './utils/Kruskal';
5  import AlgorithmVisualizer from './components/AlgorithmVisualizer';
6  import Home from './components/Home';
7
8  class App extends Component {
9    render() {
10     return (
11       <Router>
12         <>
13           <Route exact path="/" component={Home} />
14           <Route exact path="/kruskalVisualization" component={() =>
15             <AlgorithmVisualizer algorithmName="Kruskal"
                   Algorithm={Kruskal} nodeColor={() => 'pink' }
                   undirected={true} showDistance={false}/>
16           } />
17           <Route exact path="/dijkstraVisualization" component={() =>
18             <AlgorithmVisualizer algorithmName="Dijkstra"
                   Algorithm={Dijkstra} nodeColor={d => d.color }
                   undirected={true} showDistance={true}/>
19           } />
```

```
20          </>
21        </Router>
22      );
23    }
24  }
25
26  export default App;
```

U 2. liniji koda se iz biblioteke 'react-router-dom' uvezu komponente `BrowserRouter` i `Route`. Unutar `BrowserRouter` komponente proslijede se komponente `Route` koje govore kad se koja komponenta treba renderirati. U ovoj aplikaciji komponentama `Route` smo proslijedili dva svojstva. Svojstvo `path` kaže na kojem URL-u treba prikazati komponentu. Svojstvo `component` se može definirati na više načina. U 13. liniji koda svojstvo je definirano kao komponenta koju treba prikazati. Ako se želi prikazati komponenta sa svojstvima, kao svojstvo se treba predati funkcija koja vraća komponentu.

U komponenti App.js može se vidjeti kako se komponenta `Home` prikazuje na početnoj stranici. Komponenta Home.js je definirana na sljedeći način:

```
1  import React from 'react';
2  import { Link } from "react-router-dom";
3  import "./Home.scss";
4
5  const Home = () => (
6    <>
7      <div id="visualizations">Algoritmi: </div>
8      <div className="nav-link">
9        <Link to={"/kruskalVisualization"}><div id="kruskal"
             className="nav-link-item"></div></Link>
10       <Link to={"/dijkstraVisualization"}><div id="dijkstra"
             className="nav-link-item"></div></Link>
11     </div>
12   </>
13 );
14
15 export default Home;
```

U React aplikaciji se linkovi definiraju na drugačiji način nego što je to slučaj kod klasične web aplikacije. Za dodavanje linka, koristi se komponenta `Link` iz biblioteke 'react-router-dom'. Dovoljno je predati svojstvo `to`. Klikom na komponentu unutar komponente `Link` se u web pregledniku promjeni URL koji odgovara svojstvu `to`.

4.3 Zaključak

U ovom poglavlju je napravljena aplikacija koja na jednostavan način vizualizira algoritme na grafovima. Koristeći React i D3, problem vizualizacije algoritama na grafovima je sveden na problem kreiranja klase koja sadrži metode `getStep()` i `getSteps()` koje opisuju kako taj algoritam funkcionira. Iako je korištenjem komponente `AlgorithmVisualizer` izbjegnuto dupliciranje koda, osoba koja nije upoznata s načinom na koji komponenta funkcionira, neće moći na jednostavan način proširivati njene funkcionalnosti.

Bibliografija

[1] A. Banks, *Learning React: Functional Web Development with React and Redux*, O'REILLY, 2017.

[2] E. Meeks, *D3.js in Action: Data visualization with JavaScript*, Manning Publications Co, 2017.

[3] www.formidable.com/open-source/victory/docs/victory-pie/ (01.02.2019.).

[4] www.github.com/d3/d3/wiki (01.02.2019.).

[5] www.reactjs.org (01.02.2019.).

[6] www.smashingmagazine.com/2018/02/react-d3-ecosystem/ (01.02.2019.).

Sažetak

U ovom diplomskom radu je prikazano korištenje JavaScript biblioteka React i D3. U prvom poglavlju je fokus bio na biblioteci React. U drugom poglavlju je fokus bio na biblioteci D3. U trećem poglavlju se prikazuju različiti pristupi integracije biblioteka React i D3. U zadnjem poglavlju je ukratko opisan kompleksniji primjer korištenja React-a s D3-em.

Summary

In this diploma thesis, the use of JavaScript libraries React and D3 is shown. In the first chapter, the focus is on the React library. The second chapter is focused on the D3 library. In the third chapter, different approaches to integrating React and D3 libraries are shown. The last chapter briefly describes a more complex example of integrating React with D3.

Životopis

Željko Đurić je rođen 16.11.1993. u Zagrebu gdje je pohađao osnovnu školu i gimnaziju. Nakon završetka gimnazije, 2012. godine upisuje preddiplomski studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Nakon završetka, 2017. godine upisuje diplomski studij Računarstva i matematike na istom fakultetu. Tokom studiranja, zaposlio se kao softverski inženjer u firmi CROZ gdje je radio preko dvije godine.