

Cuprins

Introducere	3
1 Specificații Funcționale	5
1.1 Editorul de cod	6
1.2 Editorul de grafuri	7
1.3 Management-ul datelor	9
1.4 <i>Flow</i> -ul de lucru	11
2 Arhitectura Aplicației	13
2.1 Aplicația desktop.....	13
2.1.1 Meniul	13
2.1.2 Fereastra principală.....	14
2.1.3 Editoarele	15
2.2 API-ul REST	16
2.3 Aplicația web.....	17
2.4 Baza de date	18
3 Implementare și Testare	19
3.1 Tehnologii folosite	19
3.1.1 JavaFX	19
3.1.2 .NET Core	20
3.1.3 Angular	21
3.1.4 Alte biblioteci.....	21
3.2 Detalii tehnice	22
3.2.1 Compilarea algoritmului	22
3.2.2 Rularea Algoritmului	23
3.2.3 Colectarea comenzilor	24
3.2.4 Execuția comenzilor.....	25

3.2.5	Serializare si Deserializare.....	27
3.2.6	Generatorul de grafuri.....	28
3.2.7	Upload si Download	31
3.2.8	Management-ul bazei de date	32
3.2.9	Editorul de cod.....	33
3.2.10	Editorul de grafuri.....	34
4	Testare	37
5	Concluzii.....	38
	Bibliografie	39

Introducere

Cel mai ușor mod de a înțelege felul în care funcționează anumite rutine este prin exemplu și prin lucru practic. Consider că această aplicație susține nevoia de a experimenta, oferind și o reprezentare grafică ușor de urmărit, facilitând astfel buna înțelegere a unor probleme specifice materiei “Algoritmica Grafurilor”.

Ideea mea pentru acest proiect de licență a fost dezvoltarea unei aplicații desktop ce poate servi drept unealtă educativă în materia “Algoritmica Grafurilor”. Aplicația permite utilizatorului să scrie sau să încarce un algoritm specific și să îl ruleze pe un graf desenat anterior într-o altă fereastră. Aplicația oferă modalități de a controla cât mai intuitiv editarea codului și a grafului. De asemenea, la rularea algoritmului aplicația oferă utilizatorului controlul asupra vitezei la care algoritmul operează asupra grafului, pentru a asigura coerența animațiilor. Totodată, mi-am propus să dezvolt și un generator de grafuri pe baza de constrângeri, pentru a ajuta la testarea rapidă a algoritmilor. Fiecare graf și algoritm poate fi stocat local, însă aplicația oferă și posibilitatea de a le încărca într-o bază de date centralizată, de unde, după ce au fost acceptate de către un administrator, vor putea fi descărcate de către alți utilizatori.

Din punct de vedere teoretic, un graf este format dintr-o mulțime finită de noduri și o mulțime finită de muchii, fiecare muchie având rolul de a conecta o pereche de noduri. Există două tipuri de grafuri: grafuri orientate și grafuri neorientate. Deci, graful, ca structura de date, poate fi folosit foarte ușor pentru a descrie rețele și comportamente din viața reală, de la mulțimea străzilor dintr-un oraș la diferitele relații atașate unui grup de prieteni.

Modelarea grafurilor la probleme reale a dus la punerea accentului pe astfel de structuri de date. S-au dezvoltat o multitudine de algoritmi capabili să rezolve probleme specifice, de la parcurgeri și găsirea celor mai scurte căi la probleme de conectivitate și de găsire a fluxului maxim. Astfel, ideea aplicației mele aduce o reală îmbunătățire la aplicațiile deja existente, ajutând atât studenții, cât și alte persoane pasionate de domeniu în înțelegerea mult mai ușoară a unor algoritmi specifici grafurilor.

O aplicație asemănătoare ar fi VisuAlgo. Aceasta lucrează pe mai multe structuri de date: liste, grafuri, vectori. Fiind o aplicație web destul de cunoscută, VisuAlgo se folosește de niște animații extrem de bine puse la punct pentru a demonstra felul în care funcționează o serie de algoritmi. Totuși, există și câteva neajunsuri, VisuAlgo nu permite utilizatorului să scrie un nou algoritm și nici să modifice unul deja existent. Acest lucru împiedică procesul de experimentare, care, din punctul meu de vedere, este crucial în buna înțelegere a unor rutine.

O altă aplicație similară este Greenfoot, menită să ajute în învățarea programării în limbajul Java, oferind unelte necesare creării unor jocuri 2D. Având un scop mult mai extins, Greenfoot m-a ajutat să înțeleg în profunzime cât de important este un mod inteligent de derulare a evenimentelor reprezentate grafic.

În ceea ce privește aspectele tehnice, proiectul meu de licență este realizat în limbajul de programare Java. În cadrul dezvoltării aplicației, m-am folosit de JavaFX, o platforma software dedicată creării de aplicații-client moderne, care pot fi rulate pe o gama largă de sisteme de operare. Am mai utilizat RichTextFX, o bibliotecă specifică JavaFX, pentru a avea acces la evidențierea sintaxei, numerotarea liniilor și alte caracteristici necesare unui editor de cod modern. Majoritatea stilizărilor din aplicație au fost realizate programatic, prin metodele JavaFX, însă, pentru o mai bună organizare am folosit și mici fragmente de stilizare CSS.

Pentru compilarea și rularea codului la *runtime* m-am folosit de clasa `JavaCompiler`, oferită de către `System` și de `Reflection`, un API Java folosit pentru a modifica, a examina și a rula metode ale unor obiecte Java la *runtime*. Prin `Reflection` am avut posibilitatea de a instanția clasa fișierului rezultat în urma compilării codului scris de către utilizator. Formatarea codului respectiv a fost posibilă folosind API-ul `Google Java Format`, care formatează surse java după standardele cerute de către Google.

Baza de date centralizată, creată în Microsoft SQL Server, și `Data Storage`-ul pentru blob-uri de date au fost găzduite în Cloud-ul Microsoft Azure. Pentru management-ul bazei de date m-am folosit de un REST API implementat în framework-ul `.NET Core`, expunând și o interfață web destinată administratorilor realizată în Angular.

1 Specificații Funcționale

Scopul meu a fost dezvoltarea unei aplicații desktop capabilă să ofere utilizatorilor o platformă prietenoasă în ceea ce privește scrierea și reprezentarea pașilor unor algoritmi specifici grafurilor. Odată cu rularea unui algoritm aplicația trebuie să ofere utilizatorului posibilitatea de a controla, după bunul plac, modul și viteza cu care respectivul algoritm manipulează graful. Utilizatorul poate alege să suspende *flow*-ul algoritmului sau să avanseze manual sau automat prin pașii acestuia.

O altă funcționalitate majoră este generatorul de grafuri pe bază de constrângeri. Odată pornit, acesta interoghează utilizatorul cu privire la constrângerile dorite, ca, mai apoi, să încerce să genereze un graf ce se încadrează în limitele impuse de către utilizator. Aceste constrângeri ar putea viza numărul de noduri sau de muchii, conectivitatea grafului, valorile etichetelor componentelor și altele. Aplicația oferă și posibilitatea de a partaja grafuri și algoritmi, utilizatorul fiind capabil să încarce resurse care, după ce au fost acceptate de către un administrator, vor putea fi descărcate și folosite de către toți utilizatorii aplicației.

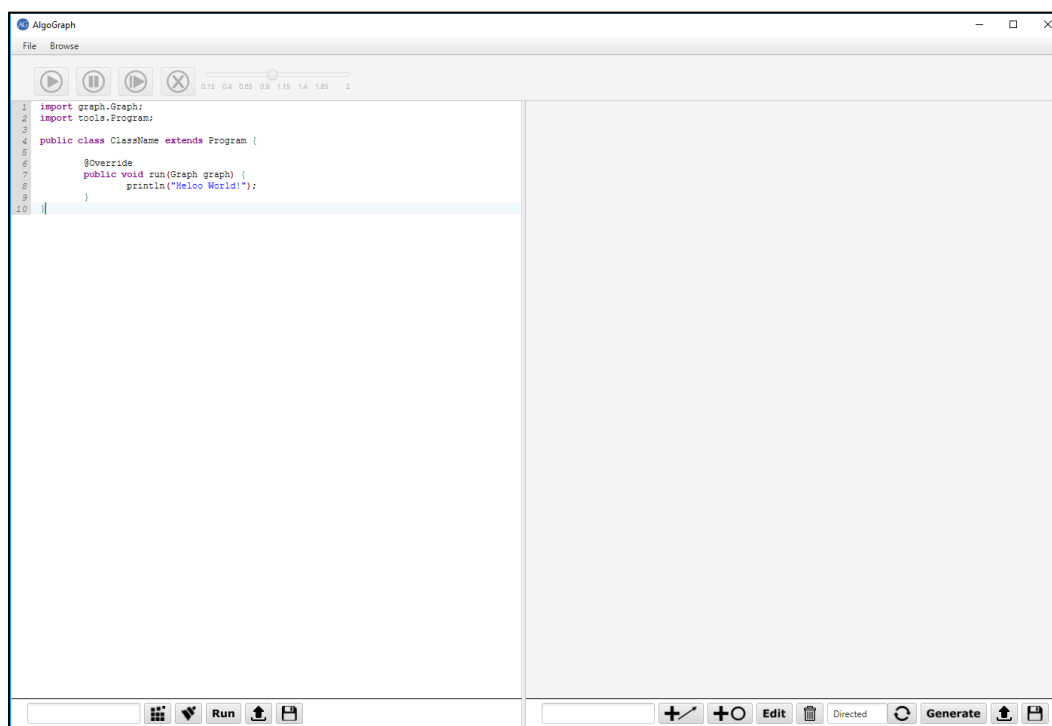


Figure 1- Interfața AlgoGraph

Am dedus că pentru a duce la sfârșit tot ceea ce mi-am propus voi avea nevoie să dezvolt 4 mari module: un editor de cod, un editor de grafuri, un sistem solid de management al resurselor utilizatorului și o infrastructură inteligentă care face legătură între cod și graf, rezultând o serie de animații coerente.

1.1 Editorul de cod

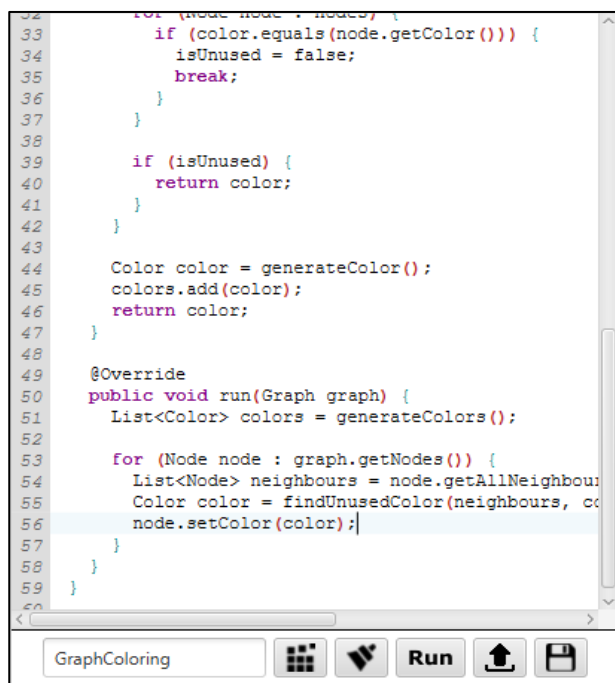


Figure 2 - Editorul de cod

Editorul de cod este o fereastră ce permite utilizatorului să își scrie propriul program ce va manipula un graf. Am ales limbajul de programare Java, ca limbaj folosit de utilizator pentru construcția algoritmilor, fiind modern, cunoscut și cu o sintaxa prietenoasă. Acesta permite toate operațiunile necesare unui editor de text: copiere, tăiere, lipire, selectare etc. Fiind dezvoltat cu ajutorul RichTextFX, acesta profită de anumite caracteristici importante, precum evidențierea sintaxei și numerotarea liniilor. Acest editor prezintă următoarele butoane și funcții:

Save: permite salvarea progresului într-un fișier ce poate fi ulterior deschis.

Compile: compilează codul sursă scris de către utilizator, fiind necesară apăsarea lui înainte de rulare. În cazul în care există erori de compilare, editorul de cod va evidenția acest lucru printr-un semn specific la linia cu pricina.

Format: preia codul sursă scris de către utilizator și îl formatează conform standardelor impuse de Google.

Run: se folosește de fișierul produs prin compilarea codului și, având acces la graful desenat în cealaltă fereastră, începe procesul de reprezentare grafică a algoritmului.

1.2 Editorul de grafuri

Editorul de grafuri este o funcționalitate extrem de importantă în aplicație. Pe lângă necesitatea evidentă de a permite utilizatorului să creeze grafuri după bunul plac, acesta reprezintă totodată și locul în care se desfășoară toate animațiile și manipulările survenite în urma rulării unui algoritm.

Există mai multe biblioteci care se ocupă cu reprezentarea grafică a grafurilor într-o aplicație Java. Una dintre acestea este JGraphX care, având la baza structura de date definită de biblioteca JGraphT, poate afișa cu ușurință grafurile necesare. Totuși, acestea nu au suport JavaFX, ci doar Swing. Deși există posibilitatea importului de componente Swing într-o aplicație JavaFX, am decis că din cauza acestui inconvenient, dar și a altor motive cum ar fi aspectul nesatisfăcător și lipsa controlului total asupra funcționalității grafurilor, îmi voi dezvolta propria modalitate de a reprezenta grafuri în JavaFX.

În ceea ce privește înfățișarea grafului am optat pentru o temă ușor de înțeles, punând pe primul loc coerența și vizibilitatea datelor. Ca paleta inițială de culori am ales alb și negru, sperând să scot în evidență animațiile și culorile alese de către utilizator.

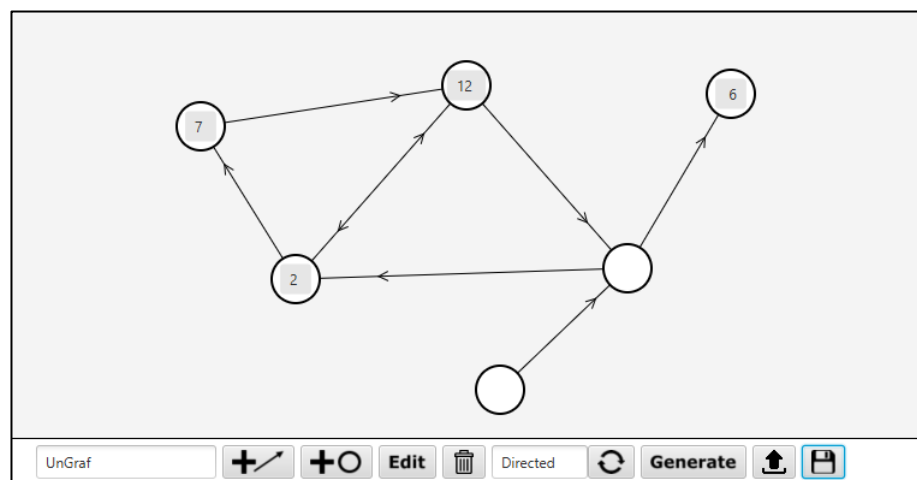


Figure 3 - Editorul de grafuri

Nodurile sunt reprezentate vizual de cercuri albe cu un contur negru. La dorința utilizatorului, nodurile pot conține o etichetă cu un nume sau o valoare numerică. Centrată în interiorul nodului, această etichetă este de culoare cenușie cu o ușoară transparență, pentru a nu obstrucționa prea mult reprezentarea nodului.

Muchiile, pe de altă parte, sunt linii negre ce conectează două noduri. La fel ca nodurile, muchiile pot avea sau nu etichete cu valori, care, de această dată, vor fi așezate la mijlocul liniei. În cazul în care avem de-a face cu un graf orientat, muchiile vor arată “cursul” sursă-destinație printr-o săgeata prezentă la unul dintre capete. În cazul în care muchia respectivă este una dublă, reprezentarea grafică a acesteia va înfățișa săgeți la ambele capete.

Graful rezultat, reprezentat în editor, este în totalitate *responsive*. Utilizatorul poate modifica poziția oricăror componente prin *drag and drop*, iar starea componentelor afectate se va schimba în mod corespunzător.

Editorul de grafuri prezintă următoarele butoane și funcții:

Save: similar editorului de cod, permite salvarea locală a progresului realizat în graf.

Add Edges: schimbă modul de lucru asupra grafului pentru a permite utilizatorului să adauge muchii. O muchie este adăugată atunci când două noduri sunt selectate consecutiv. Primul nod reprezintă sursa, iar cel de-al doilea destinația.

Add Nodes: Permite adăugarea de noduri, schimbând modul de lucru asupra grafului. Un nod este adăugat la apăsarea pe o zonă liberă din editor.

Delete Components: Permite ștergerea de muchii și de noduri, odată cu selectarea acestora. O apăsare dublă a acestui buton va conduce la ștergerea întregului graf.

Edit: Fiecare componentă a grafului are o etichetă cu valori care este ascunsă atunci când componenta nu are valoare. Acest buton facilitează apariția etichetelor pentru a putea fi editate.

Switch: Schimbă arhitectură grafului din orientat în neorientat și invers. Este atât o schimbare de reprezentare grafică, cât și una pe partea de “back-end”.

Generate: Pornește generatorul de grafuri pe baza de constrângeri.

1.3 Management-ul datelor

Atât algoritmi scriși, cât și grafurile desenate pot fi salvate local și încărcate la dorința utilizatorului. Acestea vor fi stocate sub formă de fișiere într-un folder vecin al proiectului, fiecare fișier având un nume unic selectat de utilizator. Algoritmi sunt salvați ca fișiere .java, făcând astfel mai ușoară compilarea și utilizarea lor. Având în vedere simplitatea resursei folosite de editorul de cod (algoritmul fiind un simplu *string*), aceasta a fost cea mai intuitivă variantă.

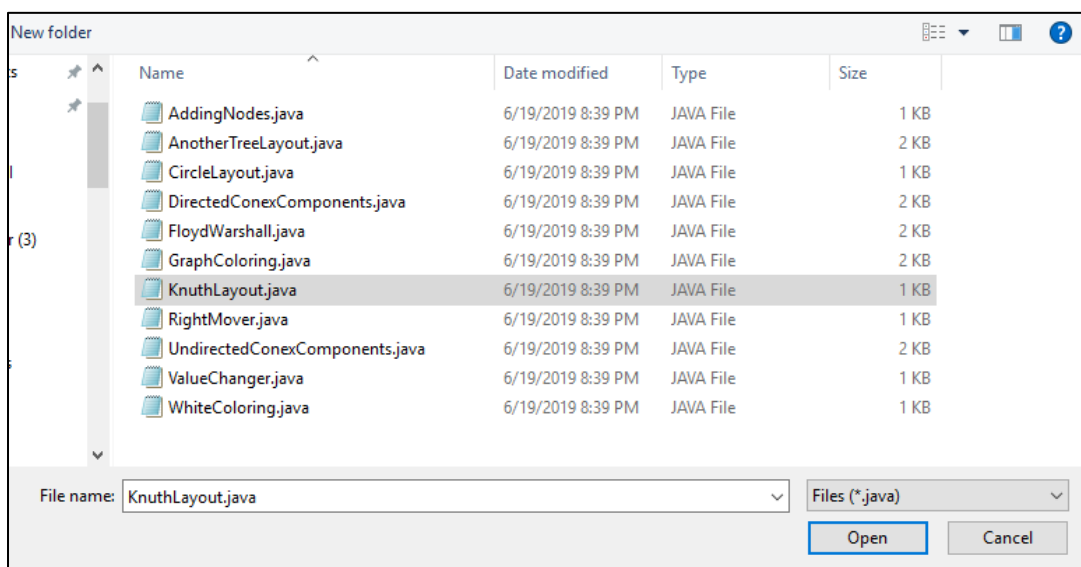


Figure 4 - Fereastra ce permite selectarea unui algoritm

Pe de altă parte, în cazul grafurilor nu am avut o soluție așa de ușoară. Am început prin a le serializa binar și stoca în fișiere. Această variantă a fost foarte ușor de implementat, serializarea obiectelor în java este ușoară și bine documentată. Totuși, există câteva neajunsuri. În cazul aplicației mele, clasa grafului este una în continuă schimbare, în sensul în care comportamentul acesteia este îmbogățit regulat. Acest lucru nu este suportat de către serializarea binară, întrucât aceasta descrie un model anterior al clasei, așadar la fiecare schimbare a clasei Graph, toate obiectele serializate anterior vor fi pierdute.

```

<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <key attr.name="NodeValue" for="node" id="d0" />
  <key attr.name="NodeX" for="node" id="d1" />
  <key attr.name="NodeY" for="node" id="d2" />
  <key attr.name="NodeColor" for="node" id="d3" />
  <key attr.name="EdgeValue" for="edge" id="d4" />
  <graph edgedefault="directed" id="G">
    <node id="199627477">
      <data key="d0">12</data>
      <data key="d1">332.0</data>
      <data key="d2">272.0</data>
      <data key="d3">
        <red>1.0</red>
        <green>1.0</green>
        <blue>1.0</blue>
      </data>
    </node>
    <node id="680246726">
      <data key="d0">7</data>
      <data key="d1">317.0</data>
      <data key="d2">437.0</data>
      <data key="d3">
        <red>1.0</red>
        <green>1.0</green>
        <blue>1.0</blue>
      </data>
    </node>
    <edge doubleEdged="false"
      id="1522328361"
      source="199627477"
      target="680246726">
      <data key="d4" />
    </edge>
  </graph>
</graphml>

```

Figure 5 - Serializarea GraphML a unui graf

Căutând o soluție adecvată, am descoperit formatul de fișier GraphML. Acesta este un format intuitiv și ușor de folosit pentru grafuri. Are suport pentru grafuri orientate și neorientate și este ușor de extins pentru a include diferite attribute nodurilor și muchiilor. Am căutat o bibliotecă capabilă să *parseze* un graf și să îmi ofere serializarea GraphML a acestuia, însă nu am găsit unul potrivit, fiecare fiind specializat pe grafuri descrise de biblioteci ca JGraphX sau, în cazul celor care puteau lucra cu un graf *custom*, adăugau prea multe attribute nefolosite de modelul meu de graf, rezultând o serializare greu de urmărit.

Așadar, mi-am făcut propriul *parser* de grafuri, capabil să transforme modelul meu de graf într-un fișier GraphML și viceversa. GraphML este un format bazat pe XML, deci nu are o sintaxă strictă și urmând setul de reguli propus de documentație am reușit să construiesc un *parser* satisfăcător. Cu ajutorul acestuia, am rezolvat problema schimbării modelului de graf, atât timp cât nu modific definiția grafului în sine (o mulțime de noduri și o mulțime de muchii ce conectează perechi de noduri), pot adăuga și modifica funcționalități. Astfel, serializarea GraphML va descrie corect modelul de graf. Odată cu acest avantaj mai apare încă unul, și anume standardul GraphML este destul de folosit și în alte aplicații de construcție de grafuri, ceea ce înseamnă că este foarte ușor să exportăm un graf din AlgoGraph și să îl folosim în respectivele aplicații și invers. Un utilizator poate prelua un graf interesant și îl poate încarca în AlgoGraph, atât timp cât serializarea acestuia respectă standardul GraphML, lucru imposibil de realizat folosind serializarea binară.

1.4 Flow-ul de lucru

Ce se întâmplă cu adevărat la apăsarea butonului “Run”?

Pentru început se verifică existența fișierului produs în urma compilării codului scris de către utilizator. Apoi, folosindu-ne de Reflection API creăm o instanță a clasei descrise de acel fișier. Un lucru foarte important de menționat este că obiectul rezultat trebuie să fie o instanță a clasei Program, astfel încât putem fi siguri de implementarea unor metode cruciale în rularea algoritmului, și anume, a metodei “run”. Dacă aceste condiții sunt îndeplinite vom avansa la următoarea etapă.

Aplicația își propune reprezentarea manipulării grafului într-un mod controlat de către utilizator. Acest lucru presupune derularea manuală sau automată, prin setarea un interval, a comenzilor asupra grafului. Deci nu putem permite algoritmului să lucreze direct pe graful din editor, întrucât ar însemna că modificările asupra grafului vor avea loc odată cu rularea codului, mult prea rapid pentru a putea fi urmărite de către utilizator. Această problemă poate fi rezolvată în două moduri:

Modul 1: Putem “coopera” cu utilizatorul, prin instruirea acestuia în a folosi apeluri de sleep ale sistemului, pentru a crea iluzia de parcurgere secvențială a comenzilor asupra grafului. Totuși, această abordare nu permite derularea manuală și nici pauză, însă cel mai mare neajuns este nevoia de a restricționa utilizatorul din a scrie cod curat și ușor de urmărit.

Modul 2: Putem reduce graful reprezentat “grafic” la o structura de date “dummy” mai simplă. Astfel, atât graful cât și toate componentele sale sunt reduse la obiecte ce implementează comportamentul unui graf, și care oferă posibilitatea adăugării de funcționalitate. Folosindu-ne de aceasta, am putea modifica obiectele “dummy” în mici istoricuri, capabile să memoreze toate evenimentele produse de către utilizator. Astfel, algoritmul utilizatorului este rulat pe un graf dummy ce joacă rolul de istoric, iar la final putem culege toate evenimentele, ca mai apoi să fie rulate pe graful autentic, după bunul plac.

Evident, am ales a doua variantă. Astfel, preluăm graful din editor și ne folosim de el pentru a crea un graf *dummy*. În continuare, algoritmul utilizatorului este rulat prin Reflection API,

oferindu-i graful *dummy* ca parametru. Rezultatul rulării este o listă de comenzi, care, puse în ordine, descriu perfect evenimentele ce trebuie parcurse pentru a respecta algoritmul utilizatorului. Această listă de comenzi este pasată mai departe unei clase `CommandsRunner` care, folosindu-se de input-ul utilizatorului, rulează comenzile în modul dorit. Vom discuta amănunțit această funcționalitate mai jos.

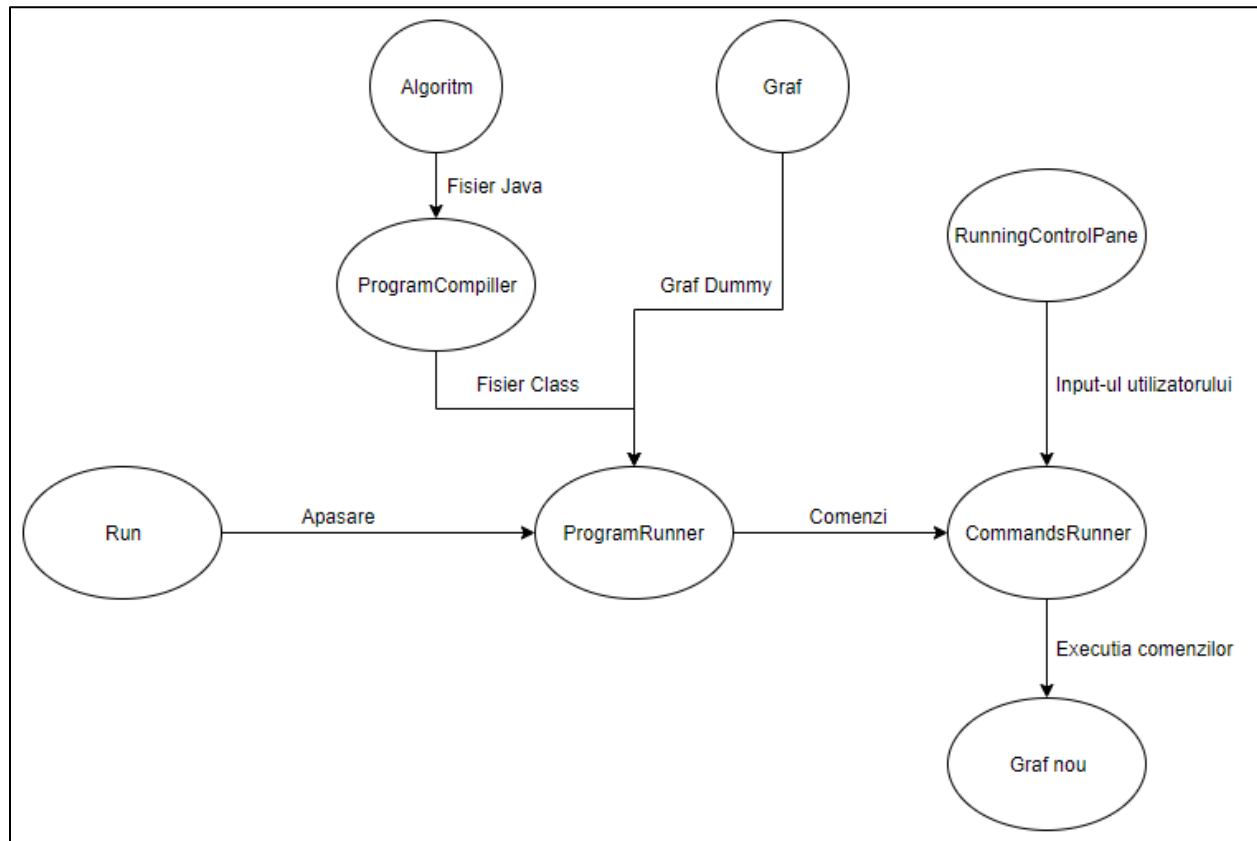


Figure 6 - Schema flow-ului principal de lucru

2 Arhitectura Aplicatiei

2.1 Aplicatia desktop

Aplicația realizată în JavaFX este formată din mai multe module ierarhizate inteligent. Fiecare componentă grafică este reprezentată printr-o clasă ce extinde un obiect JavaFX, în funcție de schema dorită. Pentru a așeza diferite view-uri într-o stivă orizontală, am creat o clasă recipient ce extinde HBox, pentru o așezare verticală am extins VBox, pentru o așezare liberă bazată pe coordonate am extins Pane și așa mai departe.

2.1.1 Meniul

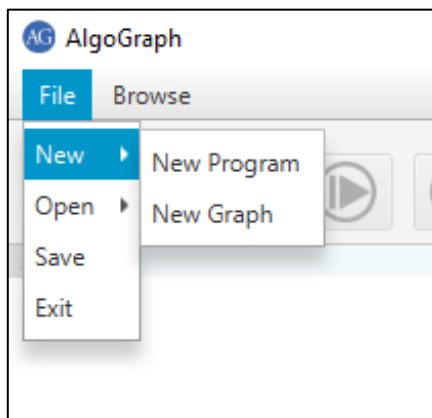


Figure 7 - Meniul AlgoGraph

Meniul aplicației, situat în partea de sus a ferestrei, este realizat prin extinderea MenuBar. În JavaFX un meniu este format din articole de tipul Menu și articole de tipul MenuItem. Un Menu este o clasă care oferă posibilitatea de a îngloba alte articole, astfel, cu ajutorul ei, putem crea meniuri ierarhizate.

Un MenuItem este un “capăt de linie” care, odată accesat, realizează o acțiune. Fiecare astfel de MenuItem permite adăugarea unui *listener* care permite execuția unor metode atunci când are loc un anumit eveniment.

Am creat un MenuBar format din două obiecte Menu, “File” și “Browse”, fiecare având mai multe noduri copil. Meniul “File” reprezintă diferite operații ce pot fi realizate asupra unui graf sau algoritmul, precum încărcarea, salvarea sau crearea unei resurse. Pe de altă parte meniul “Browse” reprezintă locul din care putem initializa fereastra de browsing a grafurilor sau a algoritmilor. Fiecare MenuItem din aceste meniuri este “ascultat” printr-un *listener*, astfel încât fiecare apăsare de buton execută o metodă potrivită.

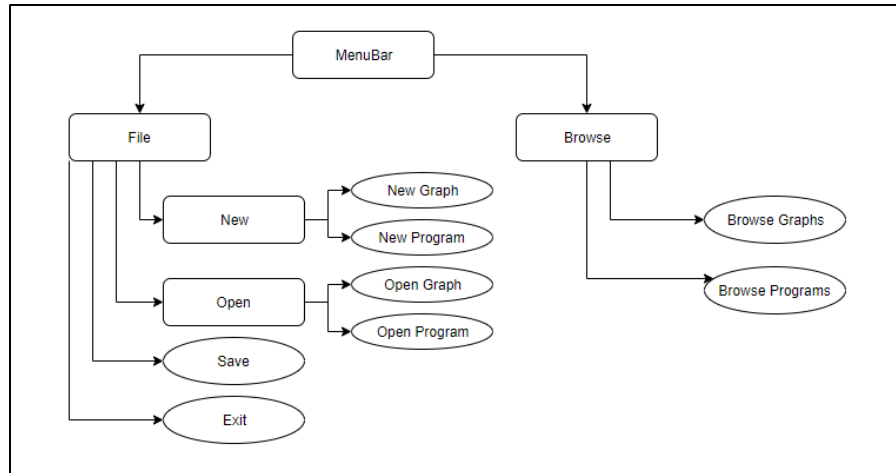
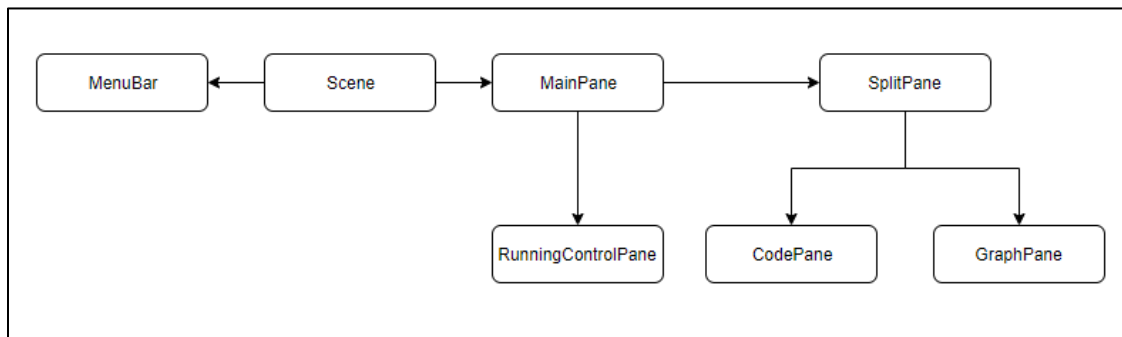


Figure 8 - Schema meniului

2.1.2 Fereastra principală

Interfața principală a aplicației este formată dintr-o scenă JavaFX, container-ul principal al aplicației. Aceasta conține atât meniul principal cât și un spațiu destinat corpului aplicației. Pentru a modulariza și simplifica crearea aplicației am optat la folosirea mai multor panouri, fiecare având un număr mic de funcții. Stilizarea CSS aplicată asupra acestor panouri este posibilă prin crearea unei legături între fișierele CSS și scena principală.



Corpul aplicației este format din două panouri. RunningControlPane, un mic panou așezat sub meniu, conține butoanele de control folosite de către utilizator în derularea comenzilor survenite în urmă rulării unui algoritm. Celălalt panou este o instanța a clasei SplitPane. Această clasă JavaFX permite reprezentarea mai multor surse de conținut într-o fereastră împărțită de bare

de divizare. Fiecare astfel de bară poate fi mutată cu mouse-ul pentru a acorda mai mult spațiu unei surse.

```
public MainPane() {  
    super();  
  
    codePane = new CodePane();  
    graphPane = new GraphPane();  
  
    display = new SplitPane();  
    display.getItems().addAll(codePane, graphPane);  
  
    HBox controls = new HBox();  
    runningControlPane = new RunningControlPane();  
    controls.getChildren().addAll(runningControlPane);  
  
    this.getChildren().addAll(controls, display);  
}
```

Figure 9 - Crearea panoului principal

M-am folosit de SplitPane pentru a realiza o fereastră ce permite desenarea de grafuri și scrierea de cod concomitentă. Pentru mai multă flexibilitate am implementat o funcționalitate ce permite folosirea întregului panou pentru un singur editor, având mai mult spațiu de desfășurare. Această funcționalitate poate fi activată prin triplu-click asupra editorului dorit.

2.1.3 Editoarele

Pentru o mai bună modularizare și structurare am ales să îmbin fiecare editor într-o componentă grafică EditorPane. Ambele panouri de editare au o structura similară, având un spațiu pentru vizualizare grafică și un spațiu inferior în care sunt așezate toate butoanele. Așadar, un EditorPane reprezintă atât o interfață grafică cât și una funcțională, deoarece acționează precum un element de legătură între exterior și structura de date a resursei de editat.

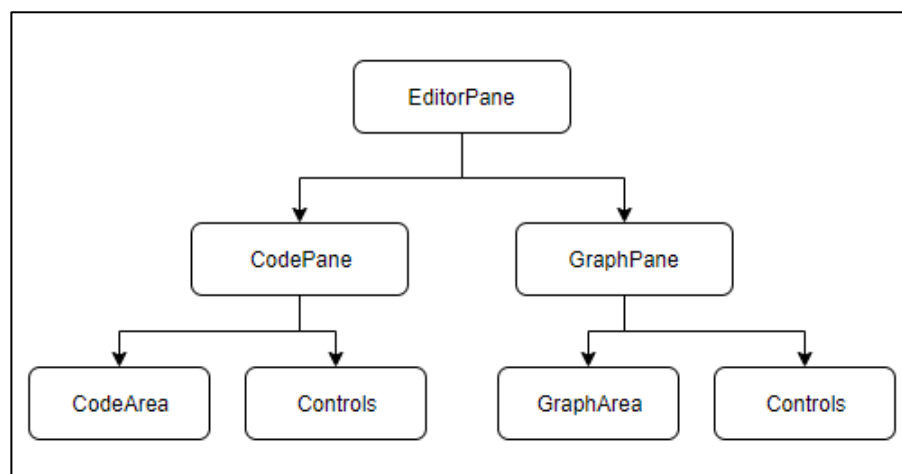


Figure 10 - Schema editoarelor

2.2 API-ul REST

Realizat în .NET Core, API-ul aplicației se folosește de Dependency Injection și de separarea logică a funcționalității pentru a asigura o structură coerentă în aplicație. Dependency Injection este o tehnică ce presupune asigurarea dependențelor, acestea fiind obiecte sau servicii folosite, unei clase de către o altă clasă care se ocupă strict de acest lucru.

API-ul este alcătuit din trei module stratificate. Primul este mulțimea de controller-e. Acestea reprezintă legătura cu exteriorul, fiind primele obiecte ce intră în contact cu cererile unui utilizator. Fiecare resursă are nevoie de un controller care oferă rute pentru crearea, modificarea, ștergerea și interogarea resursei respective. În funcție de operațiunea realizată de o ruta, aceasta poate sau nu să fie autorizată cu drepturi de administrator.

```
private readonly IGraphsService graphsService;  
  
public GraphsController(IGraphsService graphsService)  
{  
    this.graphsService = graphsService;  
}
```

Figure 11 – Constructorul unui controller care primește un serviciu prin dependency injection

```
services.AddScoped<IGraphsService, GraphsService>();  
services.AddScoped<IProgramsService, ProgramsService>();  
services.AddScoped<IDatastoreService, DatastoreService>();
```

Figure 12 - Inregistrarea serviciilor pentru dependency injection

Al doilea modul este cel format din serviciile API-ului. Un serviciu este o clasă care înglobează funcționalități complexe asupra unui set de resurse, fie prin utilizarea unui context de bază de date propriu, fie prin comunicarea cu un alt API extern. Fiecare serviciu implementează o interfață care descrie capabilitățile lui, astfel ne putem folosi de dependency injection pentru a injecta aceste servicii acolo unde este nevoie. În cadrul aplicației am creat 3 servicii, câte unul pentru manipularea tabelii de grafuri și a tabelii de programe, și încă un serviciu pentru comunicarea cu Data Storage-ul de blob-uri aflat în Cloud.

Al treilea modul este format din contextul bazei de date. O instanță a contextului reprezintă o sesiune cu baza de date, această sesiune poate fi folosită pentru a interoga și salva entități în baza de date. Crearea și modificarea bazei de date a fost realizată folosindu-mă de migrații Entity Framework Core, acestea asigură un mod programatic și complet reversibil de modificare a schemei bazei de date.

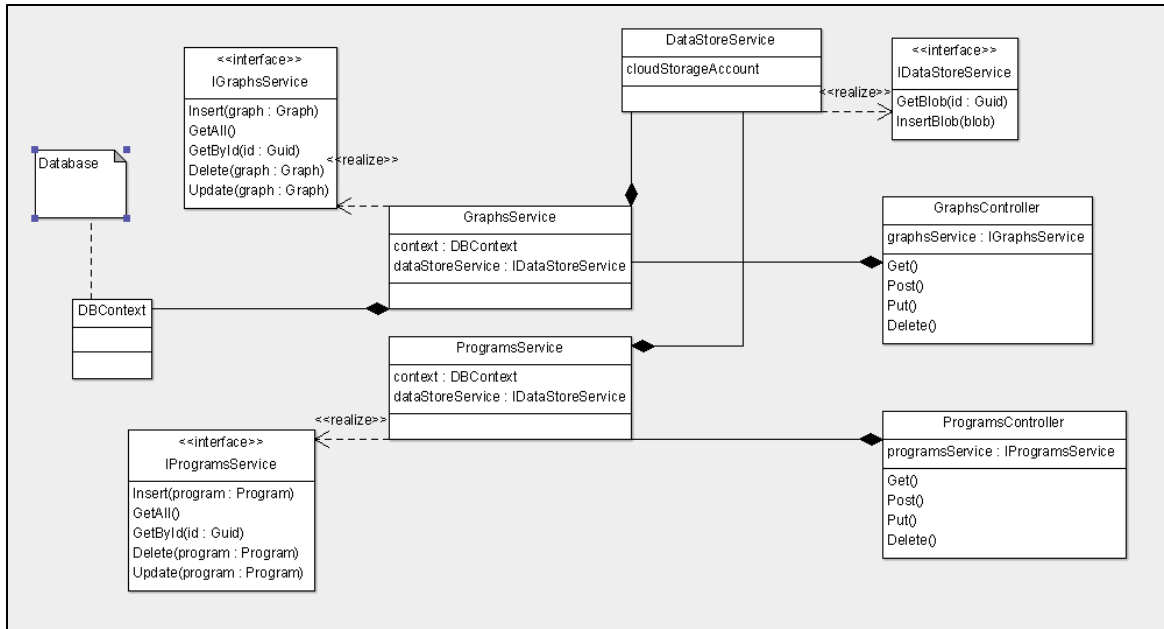


Figure 13 - Schema API-ului

2.3 Aplicatia web

Aplicația realizată în Angular se folosește de ierarhizarea componentelor pentru a asigura o structura intuitivă. Orice aplicație Angular trebuie să conțină o componentă rădăcină App care reprezintă cel mai jos nivel din ierarhie. Această componentă conține lucrurile prezente în orice pagină a aplicației, precum un *header* sau un *footer*, dar și un *tag* router-outlet. Router-outlet desemnează locul în care v-a fi așezată următoarea componentă din ierarhie, după cum spune și numele, această componentă este aleasă pe baza unui modul de rutare.

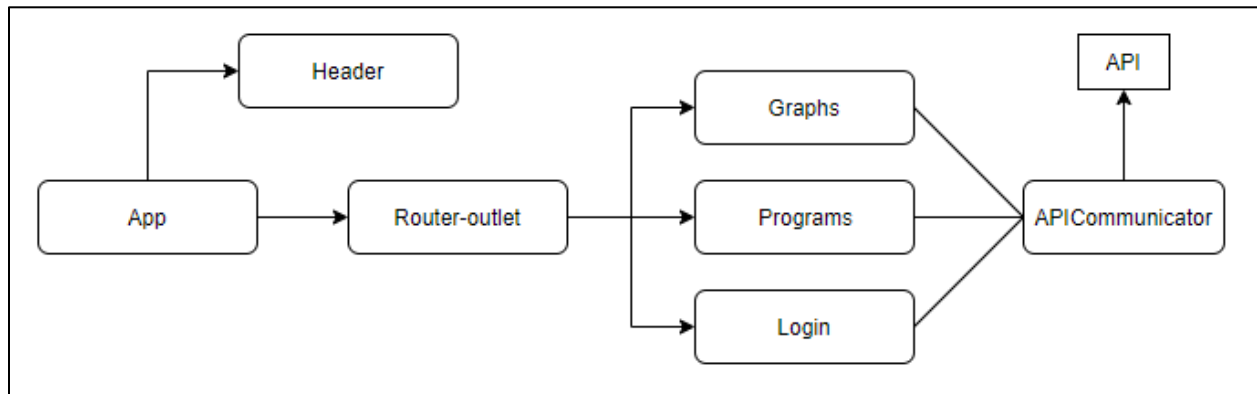


Figure 14 - Schema aplicatiei Angular

```
const routes: Routes = [
  {
    path: 'graphs',
    component: GraphsComponent,
    canActivate: [AuthenticationGuard]
  },
  {
    path: 'programs',
    component: ProgramsComponent,
    canActivate: [AuthenticationGuard]
  },
  {
    path: 'login',
    component: LoginComponent
  }
];
```

Figure 15 - Modulul de rutare

Modulul de rutare este un mic serviciu care potrivește ruta curentă cu o componentă Angular. Componenta selectată de către modul este mai apoi încărcată în locul *tag*-ului router-outlet. Tot aici adăugăm și gărzi de autentificare rutelor ce nu sunt anonime. O garda de autentificare este un serviciu care verifică dacă utilizatorul curent este sau nu autentificat, având posibilitatea de a redirecta utilizatorul către o pagină de *login*.

Aplicația conține trei componente mari: Graphs, Programs și Login. Graphs și Programs se ocupă cu reprezentarea și management-ul resurselor respective, iar Login este pagină de autentificare a administratorului. Pentru comunicarea cu API-ul am creat un serviciu APICommunicator care deține un obiect HttpClient. Cu ajutorul acestui client am făcut diferite cereri HTTP pentru interogarea și modificarea resurselor și pentru autentificarea utilizatorului.

2.4 Baza de date

Baza de date este folosită pentru a stoca grafuri și algoritmi scriși și încărcăți de către utilizatori. Creată în Microsoft SQL Server și modificată prin migrații Entity Framework Core, baza de date este formată din două tabele, Graphs și Programs. Fiecare entitate din aceste tabele conține un id, un nume, un status și un blobId. Acest blobId este utilizat în stocarea corpului resursei în Dată Storage-ul oferit de Azure. La inserarea unei noi entități în baza de date, se folosește un CloudStorageAccount pentru a crea o conexiune cu Dată Storage-ul. Prin această

conexiune se crează un nou blob identificat printr-un blobId. Mai târziu, având acest id unic putem interoga Storage-ul cu privire la conținutul acelui blob.

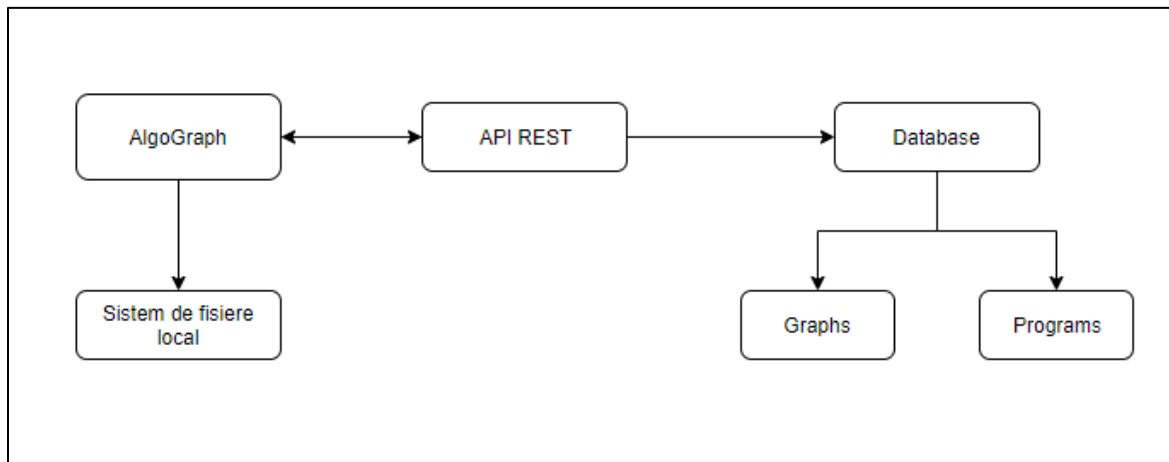


Figure 16 - Schema comunicării cu baza de date

Fiecare client al aplicației desktop deține și folosește un depozit de fișiere local drept mod de stocare principal al resurselor. Acest lucru permite citirea și modificarea mult mai rapidă a resurselor, fără să fie nevoie de o conexiune la internet. Atunci când un utilizator încarcă o resursă în baza de date, AlgoGraph interoghează calea acelei resurse și o trimite printr-o cerere HTTP către API. Similar, în momentul descărcării unei resurse, datele sunt primite de la API și un nou fișier este creat în depozitul de fișiere având numele și corpul potrivit.

3 Implementare si Testare

3.1 Tehnologii folosite

3.1.1 JavaFX

JavaFX este un set de pachete grafice care permite unui dezvoltator să creeze, să testeze și să publice aplicații desktop RCA (Rich Client Application), ce pot funcționa pe diverse sisteme de operare. Fiind un API Java, o aplicație JavaFX permite programatorului să utilizeze oricare alte biblioteci și API-uri Java native, fiind capabilă să acceseze capabilitățile sistemului sau să se conecteze la alte aplicații sau servere.

Pentru a separa design-ul de restul aplicației, stilizările pot fi făcute fișiere CSS care pot fi aplicate ulterior. Construcția scenei este făcută fie programatic, fie prin limbajul de scriptare FXML sau chiar prin *drag&drop* cu ajutorul JavaFX Scene Builder. JavaFX deține o multitudine de controale built-in (butoane, etichete, text input etc.), care pot fi folosite pentru o mai ușoară dezvoltare a scenei. Fiind o soluție menită să înlocuiască Swing (deși încă nu a făcut-o), JavaFX permite interoperabilitatea și conversia acestui tip de aplicație, dar și utilizarea componentelor Swing în cadrul proiectului.

În cadrul licenței m-am folosit în principal de mediul de dezvoltare JavaFX pentru cea mai mare parte a aplicației, cu preacădere în construcția interfeței grafice, folosind o multitudine de controale specifice JavaFX, dar și biblioteci dezvoltate ulterior, menite să îmbogățească capabilitățile utilizatorului, precum RichTextFX. Pentru desenări mai complexe ale grafului m-am folosit de Canvas API, un feature *built-in* JavaFX, care permite desenarea de diferite forme geometrice pe o arie din scena aplicației. Deși aveam și alte soluții mai ușoare, am ales să construiesc scena principală a aplicației în mod programatic, având aceeași soluție și în cadrul stilizării, cu mici excepții.

3.1.2 .NET Core

.NET Core este un *framework open-source*, capabil să creeze aplicații *cross-platform*, conectate la internet și bazate pe cloud. Printre beneficiile .NET Core se numără: o arhitectură ușor testabilă, abilitatea de a rula atât pe Windows, macOS și Linux, posibilitatea de integrare cu *framework-uri* pe partea de client, *dependency injection* ca *feature built-in*, posibilitatea de a fi găzduit pe diferite *servere*, un manager de pachete modern și unelte ce simplifică procesul de *web development*.

Folosind .NET Core am realizat un API REST, o aplicație web menită să expună informații despre un set de resurse, oferind posibilitatea creării, modificării și interogării respectivelor resurse. Pentru comunicarea cu baza de date am folosit Entity Framework Core, un *framework* ce permite crearea de corespondențe între tabele și entități .NET ușurând foarte mult dezvoltarea aplicației.

3.1.3 Angular

Fiind un *framework* folosit în crearea de aplicații client, Angular este un set de biblioteci care, utilizând HTML, CSS și TypeScript, o variație tipizată a limbajului JavaScript, facilitează și structurează procesul de dezvoltare web. Pentru a ușura în așezarea fundațiilor unei aplicații web, Angular oferă, printre altele, trei unelte:

Modulul descrie un context de compilare și un set de componente ce descriu un *workflow*, o funcționalitate concretă. Foarte folositor în structurarea proiectului, modulele permit exportul și importul de funcționalitate cu alte module, o rutare modernă și suport pentru *lazy-loading*, o practică ce permite încărcarea codului strict necesar, minimizând costurile.

Componentele sunt o asociere între logică de business, scrisă într-o clasă TypeScript, și *view* reprezentat printr-un fișier HTML. Capabile să folosească alte componente, acestea permit construcția unui *view* complex și extensibil, facilitând reutilizarea codului.

Serviciile sunt simple clase TypeScript care nu sunt asociate unui *view*. Acestea sunt importate în diferite componente cu ajutorul *Dependency Injection* și pot descrie o gama variată de funcționalitate, de la constituirea unui element de legătură între componente la clase care se ocupă cu comunicarea cu un API.

3.1.4 Alte biblioteci

În cadrul proiectului am folosit diferite biblioteci menite să îmi ușureze munca. RichTextFX este una dintre acestea, o bibliotecă destinată aplicațiilor JavaFX, aceasta oferă niște unelte avansate în dezvoltarea unui editor de text modern. Cu ajutorul acesteia cât și a unor tutoriale explicative găsite pe pagina de github, am reușit să creez editorul de cod, o componentă foarte importantă în aplicație. RichTextFX oferă o interfață ușor de folosit în ceea ce privește stilizarea textului, lucru greu de făcut cu primitive JavaFX. Cu ajutorul bibliotecii am reușit să implementez evidențierea sintaxei, numerotarea liniilor și evidențierea liniilor cu probleme. Pentru formatarea codului am utilizat Google Java Format, un API ce formatează cod Java după standardele Google.

Deoarece API-ul oferă posibilitatea formătării unui *string*, această operațiune a fost simplu de executat, înlocuind codul din editor cu textul rezultat din formatarea codului respectiv.

```
Formatter formatter = new Formatter();
String formattedText = formatter.formatSource(getText());
this.setText(formattedText);
```

Figure 17 - Formatarea textului

Jersey este un *framework open-source* utilizat pentru dezvoltarea serviciilor web REST. Cu toate acestea eu nu m-am folosit de această latură, alegând să implementez API-ul REST în .NET Core, însă am folosit client-ul HTTP oferit de Jersey 2, JAX-RS. Acest client poate fi folosit pentru a face diferite *request*-uri HTTP către un WebTarget. În aplicație, acesta este utilizat pentru a trimite și primii date cu privire la grafuri și algoritmi de la un API.

3.2 Detalii tehnice

3.2.1 Compilarea algoritmului

Algoritmul utilizatorului este stocat local într-un fișier .java, facilitând ușurința compilării și a editării. Am implementat clasa ProgramCompiller care, folosindu-se de o instanță JavaCompiller oferită de către Environment, poate compila fișiere java a căror cale a fost primită anterior.

```
public List<Diagnostic<? extends JavaFileObject>> compile(File file) throws Exception {
    DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<JavaFileObject>();
    StandardJavaFileManager fileManager = javaCompiler.getStandardFileManager(diagnostics, Locale.getDefault(),
        null);

    JavaFileObject javaObject = getJavaObject(file, fileManager);

    CompilationTask compilerTask = javaCompiler.getTask(null, fileManager, diagnostics, null, null,
        Arrays.asList(javaObject));

    compilerTask.call();

    return diagnostics.getDiagnostics();
}
```

Figure 18 - Metoda de compilare a unui fisier

Această creează un CompilationTask care, odată rulat, generează fișierul .class corespunzător și stochează anumite rezultate într-un DiagnosticsContainer. Dacă este cazul, erorile

la compilare vor fi stocate în acel container care este returnat de către metodă implementată de mine.

Odată ce task-ul de compilare este complet, verific diagnosticul și, în cazul în care există erori de compilare, aflu numărul liniei respective. Folosindu-mă de una dintre funcționalitățile RichTextFX, creez un *mapping* observabil, care dictează că orice linie cu numărul egal cu numărul liniei cu eroarea găsită anterior va prezenta un mic *icon* ce o evidențiază. Un *mapping* asemănător folosesc și pentru numerotarea liniilor.

3.2.2 Rularea Algoritmului

După compilarea cu succes a programului scris de către utilizator, un fișier .class este generat la o cale cunoscută de către aplicație. Această clasa este preluată de către aplicație și instantiată folosindu-ne de Reflection API.

```
Class<?> programClass = null;
List<Command> commands = new ArrayList<Command>();
Graph graph = new Graph(graphicGraph);
RunResult result = new RunResult();
double runTime;

try {
    URLClassLoader classLoader = new URLClassLoader(
        new URL[] {directory.toURL() });
    programClass = classLoader.loadClass(program);
    classLoader.close();

    Object instance = programClass.newInstance();

    if (!(instance instanceof Program))
        throw new Exception();

    Method method = programClass.getDeclaredMethod("run", graph.getClass());
    method.setAccessible(true);

    Stopwatch stopwatch = Stopwatch.createStarted();

    method.invoke(instance, graph);

    runTime = stopwatch.elapsed().toMillis() / 1000.0;

    Method getCommandsMethod = programClass.getMethod("getCommands");
    getCommandsMethod.setAccessible(true);

    commands.addAll(((List<Command>) getCommandsMethod.invoke(instance)));
} catch (Exception e1) {
    result.setSuccessful(false);
    return result;
}
```

Figure 19 - Rularea unui algoritm pe un graf

Algoritmul scris de către utilizator este forțat să implementeze clasa abstractă Program, deci în continuare este verificată această condiție. Având confirmarea necesară știm că instanța clasei deține o metodă run(graph) pe care o putem instanția și rula după bunul plac.

3.2.3 Colectarea comenzilor

O comandă este o acțiune de manipulare a grafului, survenită în urma unor metode apelate de către algoritmul utilizatorului. O astfel de comandă poate defini accesarea unor noduri, ștergerea unei muchii, schimbarea culorii unei componente sau *printarea* unui mesaj la consolă. Fiecare comandă extinde clasa abstractă Command. Acest lucru obligă clasa să dețină o variabilă *commandOrder*, care definește ordinea comenzii în întreaga colecție de comenzi survenite în urma rulării algoritmului. La execuția comenzii, aceasta primește graful pe care trebuie să opereze și un număr ce semnifică durata animației. Această durată este setată de către utilizator printr-un *slider*, în funcție de viteza dorită.

```
public class ChangeNodeColorCommand extends Command {
    int id;
    Color color;

    public ChangeNodeColorCommand(int commandOrder, int id, Color color) {
        this.id = id;
        this.color = color;
        super.commandOrder = commandOrder;
    }

    @Override
    public void run(GraphicGraph graph, int duration) {
        GraphicNode node = graph.getNodeById(id);

        FillTransition fill = new FillTransition(Duration.millis(duration), node.getShape(), node.getColor(), color);

        node.setColor(color);

        fill.play();
    }
}
```

Figure 20 - Comanda de schimbare a culorii unui graf

Întrucât nu vrem să rulăm algoritmul direct pe graful din editor (viteza manipulării grafului ar fi mult prea mare pentru utilizator), am dezvoltat o funcționalitate care mapează graful și fiecare componentă a sa la un obiect separat. Inițial, acest *dummy* reflectă exact structura de date a grafului

din editor, dar, odată trimis metodei scrise de către utilizator, devine complet independent. Fiecare componentă a grafului trimis către utilizator, cât și graful în sine, conține o colecție de comenzi care acționează ca un istoric al acțiunilor luate de către algoritm asupra structurii de date. Așadar, orice acțiune asupra unei componente din graf va fi tradusă că fiind o comandă. Drept exemplu, apelul `setColor` asupra unui nod va conduce la adăugarea unei comenzi de tipul `setNodeColorCommand` în setul de comenzi al nodului respectiv.

```
public void setColor(Color color) {  
    commands.add(new ChangeNodeColorCommand(Graph.getCommandOrder(), this.id, color));  
  
    this.color = color;  
}
```

Figure 21 - Metoda `setColor` dintr-un nod dummy

Crearea unei comenzi presupune și generarea unui `commandOrder`, adică a unui număr ce indică ordinea comenzii în întregul ansamblu de acțiuni. Acest `commandOrder` este obținut dintr-un câmp static al grafului, incrementat la crearea unei noi comenzi. La finalul rulării algoritmului aplicația trebuie să centralizeze toate comenzile survenite, interogând atât graful, cât și fiecare nod și muchie a sa. Astfel, obținem o listă de comenzi care, odată sortată crescător după câmpul `commandOrder`, descrie perfect toate tranzițiile prin care a trecut graful de la începutul până la sfârșitul algoritmului.

3.2.4 Execuția comenzilor

După rularea algoritmului scris de către utilizator obținem un set ordonat de comenzi ce trebuie executate asupra grafului din editor. Fiecare comandă implementează metoda `run` care descrie modul în care această manipulează graful, în același timp această metodă este responsabilă pentru definirea animației corespunzătoare comenzii. Deoarece între execuția a două comenzi va există un *delay* de lungime variabilă, nu putem parcurge comenzile pe firul de execuție principal, apelul unor metode de tipul *sleep* ar suspenda procesele JavaFX. Așadar, `CommandsRunner`, clasa

care se ocupă cu preluarea, management-ul și execuția comenzilor asupra grafului, extinde clasa Thread, lucrând în paralel cu thread-ul principal JavaFX.

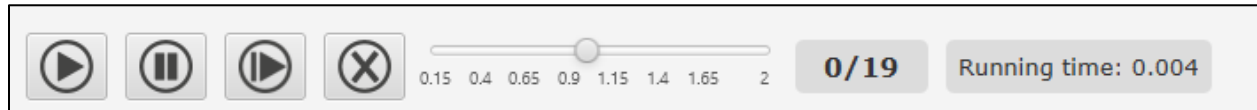


Figure 22 - Panoul de control

Aplicația pune la dispoziție și un panou de control destinat utilizatorului. Inițial acesta este dezactivat, însă, după execuția unui algoritm, panoul devine utilizabil. Pe lângă niște view-uri informative legate de numărul de comenzi și timpul de execuție al algoritmului, panoul prezintă următoarele controale: un *slider*, reprezentând durata *delay*-ului între execuția a două comenzi, și 4 butoane:

Pause: suspendă execuția comenzilor,

Play: pornește execuția automată a comenzilor, *delay*-ul fiind cel setat în slider,

Manual Step: înaintează cu o comandă, apoi se schimbă în Pause,

Exit: oprește execuția comenzilor.

```
for (Command command : commands) {
    if (runningMode == RunningMode Automatic) {
        Thread.sleep((long) (1000 * speed.get()));
    } else if (runningMode == RunningMode Pause) {
        synchronized (pauseLock) {
            pauseLock.wait();
        }
    }

    if (runningMode == RunningMode Exit) {
        break;
    } else if (runningMode == RunningMode Manual) {
        setRunningMode(RunningMode Pause);
    }

    runningControlPane.incrementCommandsNumber();
    command.run(graph, (int) (1000 * speed.get()));
}

synchronized (pauseLock) {
    while (runningMode != RunningMode Exit)
        pauseLock.wait();
}
```

Figure 23 - Bucla principală din CommandsRunner

Pentru a fi mereu la curent cu alegerile utilizatorului, clasa `commandsRunner` deține o instanță de panou de control, pe care o consultă ori de câte ori trebuie să facă o decizie. Slider-ul poate fi modificat oricând, întrucât există un *binding*, o proprietate ce permite sincronizarea a două variabile, între acesta și valoarea folosită ca *delay*. Lucrul care ne permite o bună tranziție între modurile de execuție ale comenzilor (Automatic, Manual, Pause) este un obiect de tipul *lock*. Acesta suspendă și pornește firul de execuție în funcție de alegerea utilizatorului.

3.2.5 Serializare si Deserializare

Pentru modul de serializare al grafurilor am ales standardul GraphML, bazat pe XML acest format este prietenos și folosit de multe alte editoare de grafuri. Întrucât nu am găsit un convertor de grafuri *custom* în GraphML, am ales să îmi creez propria soluție capabilă să importe și să exporte grafuri în fișiere graphml. Acest lucru este implementat prin clasa `GraphIO`, punând la dispoziție metodele `importGraph` și `exportGraph`.

```
DocumentBuilderFactory documentFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder = documentFactory.newDocumentBuilder();
Document document = documentBuilder.newDocument();

Element root = document.createElement("graphml");
setupGraphMLDocument(document, root);
document.appendChild(root);

Element xmlGraph = document.createElement("graph");

for (GraphicNode node : graph.getNodes()) {
    Element xmlNode = getXmlNode(document, node);
    xmlGraph.appendChild(xmlNode);
}

for (GraphicEdge edge : graph.getEdges()) {
    Element xmlEdge = getXmlEdge(document, edge);
    xmlGraph.appendChild(xmlEdge);
}

root.appendChild(xmlGraph);
```

Figure 24 - Corpul metodei de export al unui graf

Fiind un fișier XML, construcția sa este destul de ușor de realizat, folosindu-mă de clasa `Document`, care facilitează adăugare de noduri și valori specifice acestui tip de fișier. Standardul GraphML suportă trei tipuri de entități: grafuri, noduri și muchii. Însă, înainte de a adauga aceste entități trebuie să adăugăm o serie de noduri atribut, fiecare având un ID unic și o entitate țintă. În cazul meu, am 5 astfel de attribute: valoarea unui nod, valoarea unei muchii, poziția unui nod (x și

y) și culoarea unui nod. Aceste atribute vor fi găsite drept noduri XML în cadrul entităților pe care le descriu. Fiecare entitate, fie un graf, fie o muchie sau un nod, are nevoie de un ID numeric, cu ajutorul căruia se poate face referire în diferite relații. Drept exemplu, o muchie trebuie să prezinte un atribut *source* și unul *target*, valorile acestor atribute fiind ID-urile nodurilor ce îndeplinesc acel rol.

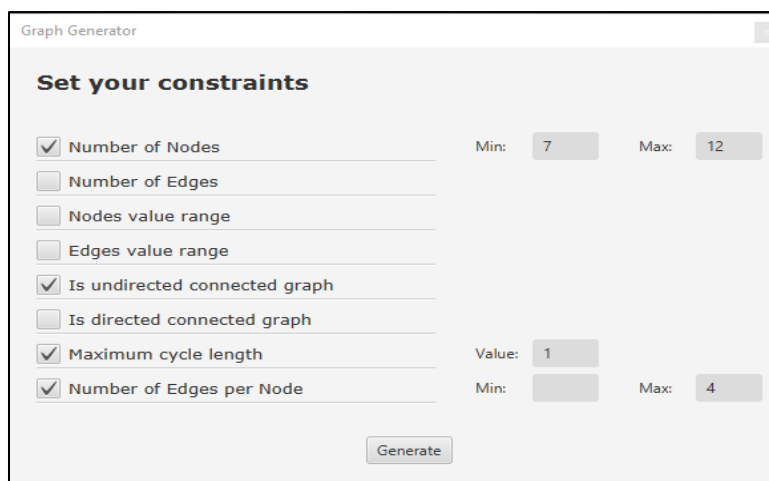
3.2.6 Generatorul de grafuri

Acesta are două mari componente: interfața grafică și generatorul în sine. Interfața grafică este o nouă scenă care, odată activată, apare ca un *popup* pe ecranul utilizatorului. Aceasta prezintă diferite variații de constrângeri sub formă de *checkbox*-uri, iar la selectarea unei constrângeri vor apărea în *view* și input-urile specifice ei. Avem trei tipuri de constrângeri:

Boolean Constraint: denotă o constrângere de tipul adevărat sau fals, care nu necesită introducerea unor limite. Exemplu: caracteristica conexității grafului rezultat.

Single Value Constraint: este o constrângere ce se folosește de o singură limită setată de utilizator. Exemplu: lungimea celui mai mare circuit din graf.

Double Value Constraint: reprezintă tipul de constrângere cu o limită inferioară și una superioară. Exemplu: numărul de noduri din graful generat.



The screenshot shows a window titled "Graph Generator" with a section "Set your constraints". It contains several checkboxes and input fields for configuring graph parameters:

- ☒ Number of Nodes: Min: 7, Max: 12
- ☐ Number of Edges
- ☐ Nodes value range
- ☐ Edges value range
- ☒ Is undirected connected graph
- ☐ Is directed connected graph
- ☒ Maximum cycle length: Value: 1
- ☒ Number of Edges per Node: Min: , Max: 4

A "Generate" button is located at the bottom right of the constraints section.

Figure 25 – Interfața grafică a generatorului de grafuri

Fiecare constrângere trebuie să implementeze clasa abstractă *Constraint*, astfel fiind forțate să implementeze metoda *check* (*graph*), care testează nivelul de completitudine al constrângerii pentru graful dat. Feedback-ul oferit de această metodă este de trei feluri: *Complete*, *Incomplete* și *Revert*. Evident, *Complete* denotă îndeplinirea constrângerii, iar *Incomplete* ne spune că, deși constrângerea nu este îndeplinită în mod current, există încă posibilitatea ca aceasta să fie respectată în viitor. Pe de altă parte, *Revert* înseamnă că limitele constrângerii au fost depășite și că nu mai există posibilitatea îndeplinirii acesteia în viitor, astfel este necesar *rollback*-ul ultimului pas din generare.

```
public class NodesNumberConstraint extends DoubleValueConstraint {  
    @Override  
    public ConstraintFeedback check(GraphicGraph graph) {  
  
        if (graph.getNodes().size() < firstValue)  
            return ConstraintFeedback.Incomplete;  
  
        if (graph.getNodes().size() > secondValue)  
            return ConstraintFeedback.Revert;  
  
        return ConstraintFeedback.Complete;  
    }  
}
```

Figure 26 - Constrângerea numărului de noduri

După selectarea constrângerilor și a limitelor dorite, utilizatorul poate începe generarea prin apăsarea butonului “Generate”. Odată cu această acțiune, aplicația interoghează interfața grafică și își crează o listă de constrângeri pe care o pasează mai departe generatorului de grafuri. Procesul de generare de grafuri este descris în ceea ce urmează. Aplicația generează aleator un pas de manipulare al grafului, executându-l asupra grafului. Întreaga listă de constrângeri este interogată pentru a decide următoarea acțiune. Dacă toate constrângerile sunt *Complete*, atunci generarea a luat sfârșit, însă dacă una dintre constrângeri returnează feedback-ul *Revert*, ultimul pas este anulat și generarea continuă. Dacă vreo constrângere returnează *Incomplete*, generarea va continua, feedback-ul primit însemnând că încă există posibilitatea îndeplinirii tuturor constrângerilor.

```

public class AddNodeStep implements IStep {
    private int id;

    @Override
    public boolean execute(GraphicGraph graph) {
        GraphicNode node = new GraphicNode();

        id = node.getUniqueId();
        graph.addNode(node);

        return true;
    }

    @Override
    public void revert(GraphicGraph graph) {
        GraphicNode node = graph.getNodeById(id);

        graph.removeNode(node);
    }
}

```

Figure 27 – Pasul de adăugare al unui nod

Fiecare pas din procesul de generare al grafurilor trebuie să implementeze interfața IStep, asigurându-ne astfel că fiecare pas oferă posibilitatea de execuție și *rollback*. Un pas poate fi adăugarea unui nod, adăugarea unei muchii sau schimbarea valorii unui astfel de element. Generarea de grafuri se încheie atunci când toate constrângerile sunt îndeplinite sau când un anumit număr de pași au fost executați fără a duce la o soluție, indicând faptul că limitele utilizatorului nu pot fi respectate.

Din acest proces rezultă un nou graf reprezentat în editor care poate fi ulterior salvat. Adăugarea unor constrângeri inteligente poate duce la generarea unor clase specifice de grafuri. Drept exemplu, limitându-ne la grafuri conexe în care lungimea maximă a unui ciclu este 1, așadar este un graf aciclic, rezultă un arbore.

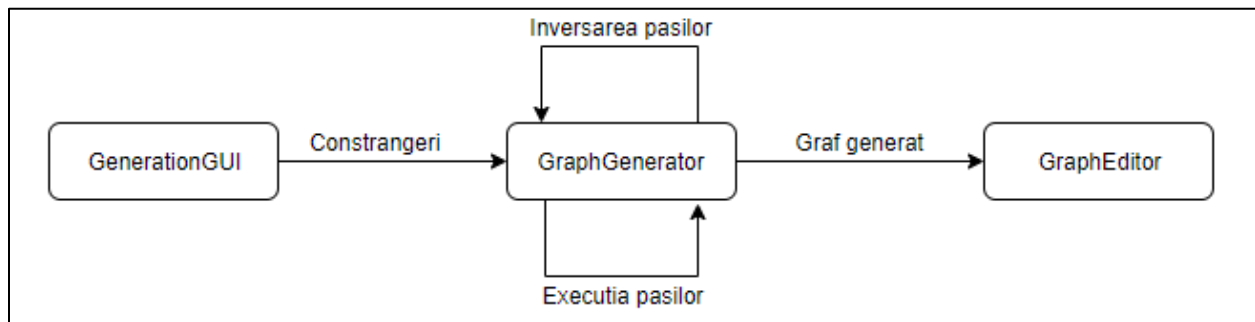


Figure 28 - Schema generarii grafurilor

3.2.7 Upload si Download

Aplicația oferă utilizatorului posibilitatea de a încărca algoritmi și grafuri într-o bază de date centralizată, astfel încât, prin intermediul unui panou de *browsing*, alți utilizatori le pot descărca și folosi. Baza de date este administrată de către un API REST, care expune câteva rute folosite de către aplicație, pentru a posta și lua algoritmi și grafuri. Pentru a face cereri la API din aplicație, m-am folosit de *framework*-ul Jersey, utilizând un client JAX-RS.

```
public void postGraph(ApiEntity entity) {  
    WebTarget employeeWebTarget = webTarget.path("api/graphs");  
  
    Invocation.Builder invocationBuilder = employeeWebTarget.request(MediaType.APPLICATION_JSON);  
    invocationBuilder.post(Entity.entity(entity, MediaType.APPLICATION_JSON));  
}
```

Figure 29 - Metoda ce incarcă un graf in API

Ambele editoare prezintă câte un buton de upload, care poate fi acționat doar când utilizatorul lucrează cu un algoritm sau cu un graf salvat. La apăsarea lui aplicația mapează resursa respectivă la un obiect `ApiEntity` și se folosește de un client JAX-RS pentru a face un *request* POST către API.

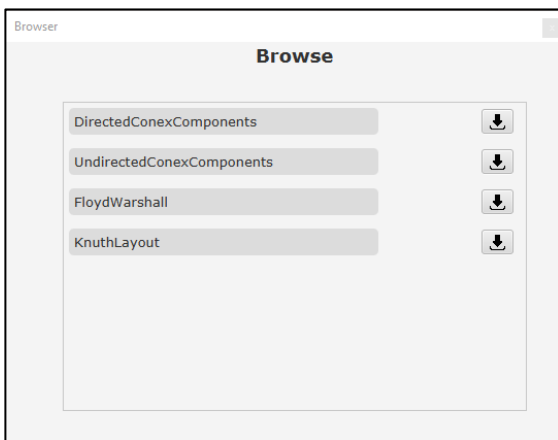


Figure 30 – Fereastra de browsing a algoritmilor

Pe de altă parte, pentru a descărca astfel de resurse, utilizatorul trebuie să selecteze *item*-ul “Browse” al meniului, care odată acționat preia toate resursele necesare de la API și le afișează într-un nou panou, de unde pot fi descărcate cu ușurință.

3.2.8 Management-ul bazei de date

Baza de date, realizată în Microsoft SQL Server, are o interfață spre exterior realizată printr-un API REST în .NET Core. Aceasta conține două tabele, unul pentru grafuri și unul pentru programe. Pentru a stoca corpul unei astfel de resurse m-am folosit de un *Data Storage* pentru blob-uri, hostat în Microsoft Azure, astfel fiecare entitate din tabele conține câte un ID în plus, care face legătura între ea și blob-ul asociat ei din *Data Storage*.

API-ul deține două *controllere*, câte unul pentru fiecare resursă. Acestea expun diferite rute ce conduc către operații specifice unui API REST, precum adăugarea, ștergerea, interogarea și modificarea unei entități. Cu excepția rutelor de adăugare și de interogare de resurse active, toate rutele necesită autentificare pentru a putea fi accesate. Această autentificare este de tipul Basic Authentication, o soluție simplă ce permite logarea unui utilizator administrator.

Comunicarea cu baza de date este realizată prin Entity Framework Core. Acesta permite *maparea* entităților stocate în tabele la clase specifice .NET, astfel facilitează paradigmele programării orientate obiect, făcând posibilă interogarea și modificarea bazei de date într-un mod programatic.

```
public sealed class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
        Database.Migrate();
    }

    public DbSet<Graph> Graphs { get; set; }
    public DbSet<Program> Programs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Graph>()
            .Property(g => g.Pending)
            .HasDefaultValue(true);

        modelBuilder.Entity<Program>()
            .Property(g => g.Pending)
            .HasDefaultValue(true);
    }
}
```

Figure 31 - Context-ul bazei de date

Pentru a oferi o interfață grafică atractivă, AlgoGraph pune la dispoziție o aplicație web dezvoltată în Angular ce permite management-ul resurselor. Aceasta necesită autentificare cu un cont de administrator și permite mai multe operații pe grafurile și algoritmi încărcăți de către utilizatori. În momentul în care o nouă resursă este încărcată în baza de date, aceasta primește statutul de “Pending”, însemnând că are nevoie de confirmarea administratorului pentru a deveni “Active”. Aplicația expune 2 pagini, câte una pentru fiecare entitate. Aici toate resursele vor fi afișate în funcție de statutul lor, utilizatorul având posibilitatea de a schimba statutul, după care se va face filtrarea. Resursele ce se află în “Pending” pot fi acceptate sau șterse de către un administrator, pe când cele aflate în “Active” oferă doar posibilitatea ștergerii.



Figure 32 - Interfața administratorului, realizată în Angular

3.2.9 Editorul de cod

Folosindu-mă de RichTextFX, editorul de cod are la baza o instanță a clasei CodeArea. Adăugată într-un panou de tipul VBox, alături de un panou de control, zona destinată scrierii de text este situată în partea superioară a editorului de cod. CodeArea facilitează multe operații specifice unui editor de cod, greu de implementat folosind TextArea oferit de JavaFX, precum stilizarea individuală a unor cuvinte, adăugarea de obiecte grafice înaintea liniilor și altele.

```

if(matcher.group("Keyword") != null)
    style = "keyword";
else if(matcher.group("RoundParanthesis") != null)
    style = "round-paranthesis";
else if(matcher.group("CurlyParanthesis") != null)
    style = "curly-paranthesis";
else if(matcher.group("SquareParanthesis") != null)
    style = "square-paranthesis";
else if(matcher.group("PointComma") != null)
    style = "bold";
else if(matcher.group("String") != null)
    style = "string";
else if(matcher.group("SingleLineComment") != null
    || matcher.group("MultipleLineComment") != null)
    style = "comment";

```

Figure 33 - Atribuirea claselor css grupurilor de text

```

Pattern pattern = Pattern.compile("(?<Keyword>\\b("
+ String.join("|", keywords) + ")\\b)"
+ "|(?<RoundParanthesis>\\(|\\))"
+ "|(?<CurlyParanthesis>\\{|\\})"
+ "|(?<SquareParanthesis>\\[|\\])"
+ "|(?<PointComma>\\;)"
+ "|(?<String>\"([^\"]\\\\\\\\|\\\\\\\\.)*\\")"
+ "|(?<SingleLineComment>//[^\n]*)"
+ "|(?<MultipleLineComment>/\\*(.\\\\\\\\R)*?\\*/)");

```

Figure 34 - Sablonul folosit de catre Matcher

Prin ascultarea unor evenimente declanșate la introducerea de text, am apelat o metodă care se ocupă cu stilizarea codului. Aceasta își crează un Matcher care, pe baza unor șabloane, potrivește grupuri de text cu diferite etichete. După etapa de etichetare, fiecărui grup de text îi este administrată o clasa CSS în funcție de valoarea etichetei sale. Astfel avem grupuri de text ce reprezintă cuvinte cheie, comentarii, paranteze și altele, fiecare stilizat într-un mod corespunzător.

```

codeArea = new CodeArea();
codeArea.setParagraphGraphicFactory(LineNumberFactory.get(codeArea));
codeArea.multiPlainChanges().successionEnds(Duration.ofMillis(500))
    .subscribe(ignore -> codeArea.setStyleSpans(0, computeHighlighting(codeArea.getText())));

codeArea.onKeyReleasedProperty().set((event) -> this.setModified(true));

IntFunction<Node> lineNumbers = LineNumberFactory.get(codeArea);
IntFunction<Node> errorLines = new LineArrowFactory(errorLine);

IntFunction<Node> graphics = line -> {
    HBox hbox = new HBox(lineNumbers.apply(line), errorLines.apply(line));
    hbox.setAlignment(Pos.CENTER_LEFT);
    return hbox;
};
codeArea.setParagraphGraphicFactory(graphics);

```

Figure 35 - Inițializarea obiectului CodeArea

3.2.10 Editorul de grafuri

Editorul de grafuri este panoul care se ocupă cu administrarea canvasului ce cuprinde întregul graf și componentele sale. Similar editorului de cod, acest editor este format din două panouri, cel superior rezervat reprezentării grafului, iar cel inferior acționează drept un panou de control. Panoul folosit pentru desenarea grafului este o instanță de Pane, care extinde clasa abstractă

definită de mine, FlexibleCanvas. Această conferă posibilitatea unor acțiuni de tipul *drag&drop* asupra grafului și a componentelor acestuia.

```
private void handleMouseDragged(MouseEvent event) {
    double xDistance = event.getX() - xStart;
    double yDistance = event.getY() - yStart;
    xStart = event.getX();
    yStart = event.getY();

    for (Node node : getChildren()) {
        if (node instanceof GraphicNode) {
            GraphicNode gNode = ((GraphicNode) node);

            gNode.setX(xDistance);
            gNode.setY(yDistance);
        }
    }
    event.consume();
}
```

Figure 36 - Metoda ce permite drag&drop în FlexibleCanvas

Graful reprezentat în editor este o mulțime de noduri și muchii grafice. Un nod grafic este un obiect ce extinde clasa JavaFX Group, un tip de panou care permite reprezentarea mai multor noduri-copil, capabil să transfere orice transformare efectuată asupra lui, copiilor săi. Nodul grafic este reprezentat de un cerc și o eticheta ce conține valoarea acestuia. Poziția etichetei este setată printr-un *binding* cu poziția cercului desenat anterior, astfel încât eticheta este mereu în mijlocul nodului.

Muchiile grafice extind, precum nodurile grafice, clasa Group. Acestea sunt reprezentate prin linii a căror orientare este evidențiată prin prezența unei săgeți. Capetele liniei sunt conectate la sursa și destinația muchiei, cu ajutorul unor *binding*-uri JavaFX. În mod similar, poziția etichetei este fixată în mijlocul liniei, iar poziția săgeții ce evidențiază orientarea este setată că fiind la o distanță egală cu un sfert din lungimea liniei de nodul destinație.

```

public class GraphicGraph {
    private boolean directed = true;
    private FlexibleCanvas canvas;
    private List<GraphicNode> nodes;
    private List<GraphicEdge> edges;

    public GraphicGraph() {
        this.nodes = new ArrayList<GraphicNode>();
        this.edges = new ArrayList<GraphicEdge>();
        this.canvas = new FlexibleCanvas();
    }
}

```

Figure 37 - Clasa GraphicGraph

Editorul permite execuția mai multor operații asupra grafului, precum: ștergerea unui nod sau a unei muchii, adăugarea unui nod, adăugarea unei muchii și editarea valorii acestora. Ștergerea unui element este implementată prin setarea unui *listener* pentru fiecare componentă a grafului. În momentul în care un utilizator selectează o componentă, *listener*-ul apelează o metodă de ștergere a elementului respectiv. În mod similar se procedează pentru editarea valorii unui nod sau a unei muchii. La selectarea unui element, eticheta acestuia devine vizibilă și ușor de editat.

Adăugarea unui nod presupune atașarea unui *listener* direct pe canvasul grafului, astfel încât, la selectarea unui spațiu gol din canvas, un nou nod este creat și introdus în graf. Adăugarea unei muchii este puțin mai complicată. Aceasta presupune selectarea consecutivă a două noduri grafice, o muchie urmând să fie adăugată între acestea, sursa fiind primul nod selectat, iar destinația este cel de-al doilea. Pentru a nu există cazuri în care aceste operații se intercalează, am ales să creez un sistem ce asigură execuția unei singure operații la un moment dat. Acest lucru este implementat printr-o variabilă ce semnalează modul de lucru curent, valoarea variabilei poate fi schimbată prin apăsarea butoanelor din panoul de control.

```

private void addNodeHandlers(GraphicNode node) {
    node.addEventHandler(MouseEvent.MOUSE_RELEASED, (event) -> {
        if (editMode == GraphEditMode.EditingValues) {
            node.valueField.showInput();
            modified = true;
        } else if (editMode == GraphEditMode.Deleting) {
            graph.removeNode(node);
            modified = true;
        }
    });
}
}

```

Figure 38 - Adăugarea listener-ului unui nod grafic

```

public enum GraphEditMode {
    AddingNodes,
    AddingEdges,
    EditingValues,
    Deleting
}

```

Figure 39- Modurile de lucru în editarea grafului

4 Testare

Testarea unei aplicații software este o investigație continuă, menită să ofere o vizualizare obiectivă și independentă asupra capacității acesteia. Tehnicile de testare urmăresc verificarea următoarelor criterii: aplicația răspunde corect la orice input, îndeplinește cererile de design și funcționalitate, îndeplinește diferite operații într-un timp acceptabil, poate fi instalată și rulată în mediul intenționat și altele.

În cazul AlgoGraph am testat mai multe module ale aplicației în moduri diferite. Aplicația principală, realizată în JavaFX, a fost testată prin derularea a diferite scenarii asupra soluției aflată încă în dezvoltare în diferite stagii. Astfel, am putut fi sigur cu privire la cursul și capacitățile aplicației de-a lungul dezvoltării. Algoritmii prestabiliți în aplicație au fost testați folosind exemple găsite pe internet, astfel, pentru un graf-input desenat de mine, am verificat *output*-ul cu cel găsit în alte documentații. Folosindu-mă de Postman, un tool folosit pentru testarea a diferite rute și metode API, am testat și verificat comportamentul API-ului REST. Am verificat atât autentificarea, cât și diferite operații realizate de către administrator și de către un utilizator neautentificat.

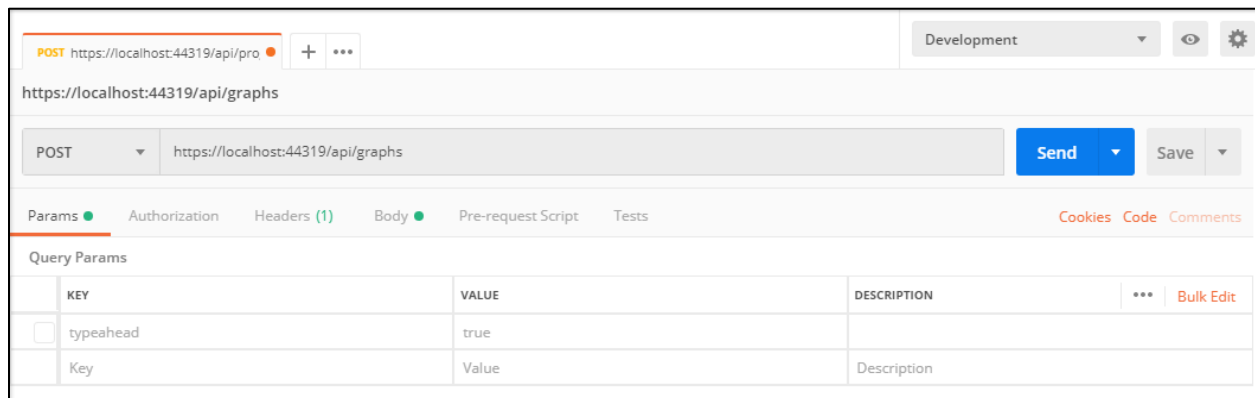


Figure 40 - Interfața aplicației Postman

5 Concluzii

În concluzie, sunt de părere că AlgoGraph este o unealtă educativă excelentă în ceea ce privește scrierea și simularea unor programe specifice „Algoritmicii Grafurilor”. Posibilitatea de a desena grafuri ajută enorm în exemplificarea comportamentului unor algoritmi, pe când generarea acestora permite atât testarea rapidă, cât și exemplificarea felului în care niște limite pot conduce la crearea unor tipuri de grafuri specifice.

Algoritmica grafurilor este foarte importantă atât pe plan teoretic, cât și din punct de vedere practic. Modelarea unor comportamente reale în structuri de date asemănătoare grafurilor au dus la apariția multor algoritmi specifici care, cu ajutorul AlgoGraph, pot fi explicați într-un mod grafic și intuitiv.

Spre deosebire de alte aplicații asemănătoare, AlgoGraph oferă un nivel mult mai înalt de libertate. În special, modul în care un utilizator poate modifica un algoritm după bunul său plac mi se pare o funcționalitate extrem de importantă, adăugând o nouă dimensiune aplicațiilor existente. AlgoGraph dispune din start de o gamă variată de algoritmi ajutători, mențiți să demonstreze capabilitățile aplicației și să-i introducă utilizatorului modul de lucru cu aplicația.

Ca îmbunătățiri și direcții viitoare, m-am gândit la o autentificare opțională în aplicația desktop. Un utilizator autentificat ar avea mai multe drepturi, precum încărcarea directă de grafuri și algoritmi, și votarea celor deja încărcate. Astfel, resursele reprezentate în fereastra de *browsing* ar putea fi ierarhizate după numărul de voturi, fiind un însemn al relevanței.

Editorul de grafuri permite mutarea nodurilor și a grafului cu ajutorul mouse-ului, însă nu suportă *zooming*-ul. Cred că ar fi o funcționalitate foarte folositoare atunci când lucrăm cu grafuri de dimensiuni mari. De asemenea, o îmbunătățire a generatorului de grafuri ar putea fi adăugarea de noi constrângeri care să permită generarea unor structuri de date specializate.

Bibliografie

<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

<https://www.baeldung.com/jersey-jax-rs-client>

<https://angular.io/guide/architecture>

<https://docs.microsoft.com/en-us/ef/>

<https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f>

<https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2>

<https://en.wikipedia.org/wiki/JavaFX>

<https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm>

<https://github.com/FXMisc/RichTextFX>

https://en.wikipedia.org/wiki/Software_testing

<http://graphml.graphdrawing.org/>