**Cuprins**

[Introducere 2](#_Toc11851117)

[1 Specificații Funcționale 4](#_Toc11851118)

[1.1 Editorul de cod 5](#_Toc11851119)

[1.2 Editorul de grafuri 6](#_Toc11851120)

[1.3 Management-ul datelor 8](#_Toc11851121)

[1.4 *Flow*-ul de lucru 9](#_Toc11851122)

[2 Implementare si Testare 11](#_Toc11851123)

[2.1 Tehnologii folosite 11](#_Toc11851124)

[2.1.1 JavaFX 11](#_Toc11851125)

[2.1.2 .NET Core 12](#_Toc11851126)

[2.1.3 Angular 12](#_Toc11851127)

[2.1.4 Alte librării 13](#_Toc11851128)

[2.2 Detalii tehnice 14](#_Toc11851129)

[2.2.1 Compilarea algoritmului 14](#_Toc11851130)

[2.2.2 Rularea algoritmului 14](#_Toc11851131)

[2.2.3 Colectarea comenzilor 15](#_Toc11851132)

[2.2.4 Execuția comenzilor 16](#_Toc11851133)

[2.2.5 Serializare si Deserializare 17](#_Toc11851134)

[2.2.6 Generatorul de grafuri 18](#_Toc11851135)

[2.2.7 Upload si Download 20](#_Toc11851136)

[2.2.8 Management-ul bazei de date 20](#_Toc11851137)

[2.3 Testare 22](#_Toc11851138)

[3 Concluzii 23](#_Toc11851139)

[Bibliografie 24](#_Toc11851140)

# Introducere

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

Ideea mea pentru acest proiect de licență este dezvoltarea unei aplicații desktop ce poate servii drept unealtă educativă în materia “Algoritmică 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 trebuie să ofere modalități de a controla cât mai intuitiv editarea codului și a grafului. La rularea algoritmului aplicația trebuie să ofere utilizatorului controlul asupra vitezei la care algoritmul operează asupra grafului, pentru a asigura coerentă animațiilor. Totodată, mi-am propus să dezvolt și un generator de grafuri pe baza de constrângeri, pentru a ajută la testarea rapidă a algoritmilor. Fiecare graf și algoritm poate fi stocat local, însă aplicația oferă și posibilitatea de a le încarcă într-o baza de date centralizată, de unde, după ce au fost acceptate de către un administrator, vor putea fi descărcate de către alți utilizatori.

Un graf este format dintr-o mulțime finită de noduri și una de muchii, fiecare muchie având rolul de a conecta o pereche de noduri. Există două tipuri de grafuri: grafuri orientate și neorientate. Deci, graful, că structura de date, poate fi folosit foarte ușor pentru a descrie rețele și comportamente din viață 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 cai 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. Această lucrează pe mai multe structuri de date: liste, grafuri, vectori și altele. 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, am încercat să reproduc pe cât posibil acest look&feel. 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 bună înțelegere a unor rutine.

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

Proiectul este realizat în limbajul de programare Java. În cadrul dezvotarii aplicației m-am folosit de JavaFX, o platforma software dedicată creeari de aplicații client moderne, care pot fi rulate pe o gama largă de sisteme de operare. Am mai utilizat RichTextFX, o librărie specifică JavaFX, pentru a avea acces la evidențierea sintaxei, numerotarea liniilor și alte caracterisitici necesare unui editor de cod modern. Majoritatea stilizarilor 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 JavaCompiller oferită de către system și de Reflection, un API Java folosit pentru a modifică, examina și rula metode ale unor obiecte java la runtime. Prin Reflection am avut posibilitatea de a instantia clasa fișierului rezultat în urmă 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 Dată Storage-ul pentru blob-uri de date au fost hostate în Cloud-ul Microsoft Azure. Pentru management-ul bazei de date m-am folosit de un REST API implementat în framework-ul .NET Core, expunand și o interfață web destinată administratorilor realizată în Angular.

# Specificații Funcționale

Scopul aplicației este dezvoltarea unei aplicații desktop capabilă să ofere utilizatorilor o platforma 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 viteză cu care respectivul algoritm manipulează graful. Utilizatorul poate alege să suspende flow-ul algoritmului sau să avanseze manual sau chiar automat prin pașii acestuia.

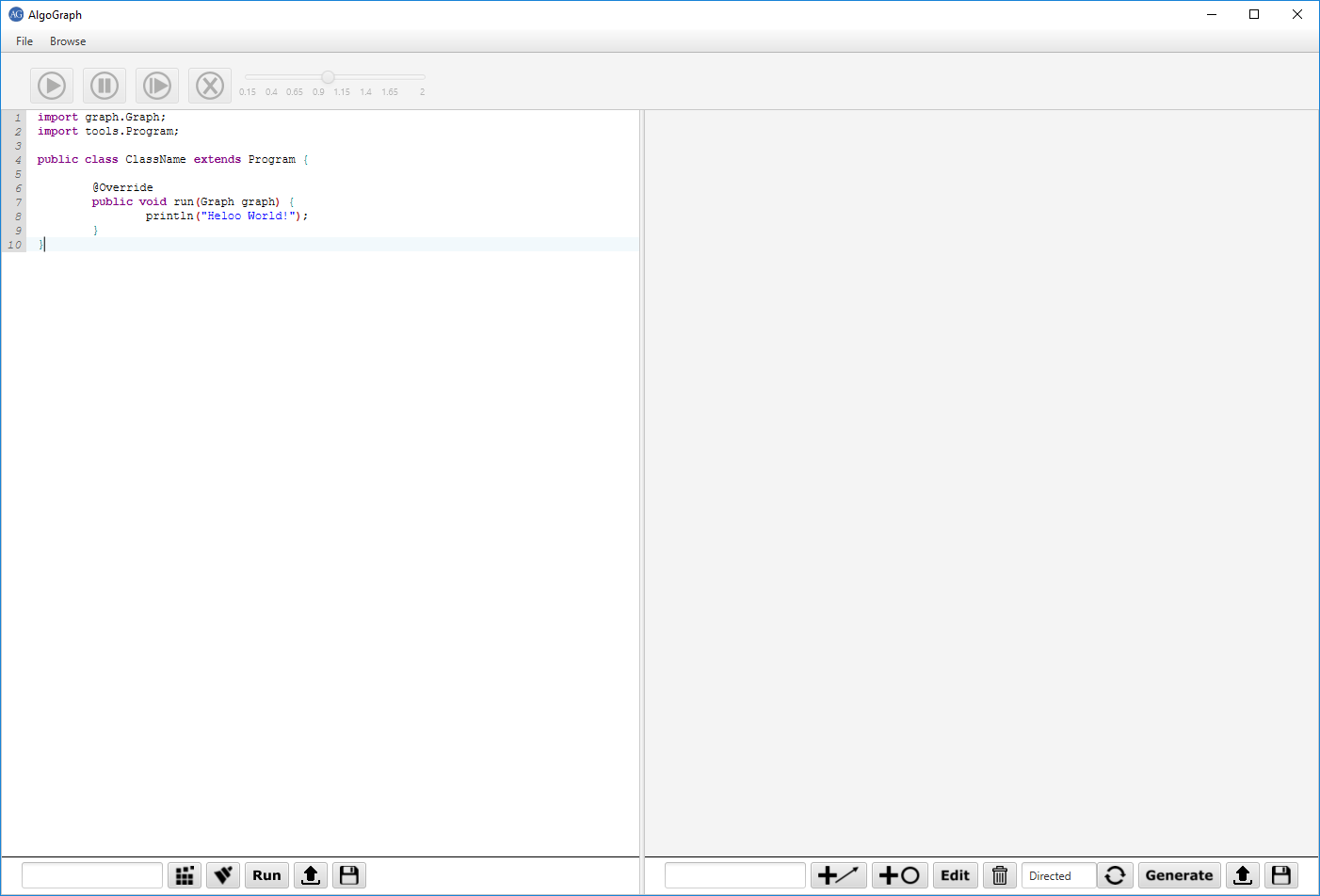
O altă funcționalitate majoră este generatorul de grafuri pe baza de constrângeri. Odată pornit, acesta interoghează utilizatorul cu privire la constrângerile dorite, că mai apoi să încerce să genereze un graf ce se încadrează în limitele impuse de către utilizator. Aceste constrângeri ar putea viza numărul de noduri sau muchii, conectivitatea grafului, valorile etichetelor componentelor și altele. Aplicația oferă și posibilitatea de a face sharing de 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- Interfata 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 a resurselor utilizatorului și o infrastructură inteligență care face legătură între cod și graf, rezultând o serie de animații coerente.

## 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, că limbaj folosit de utilizator pentru contrucția algoritmilor, fiind modern, cunoscut și cu o sintaxa prietenoasă. Permite toate operațiunile necesare unui editor de text: copiere, tăiere, lipire, selectare etcetera. Fiind dezvoltat cu ajutorul RichTextFX, acesta profită de anumite caracteristici importante că 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, este 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 comform 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.

## Editorul de grafuri

Este o funcționalitate extrem de importantă în aplicație. Pe lângă necesitatea evidență de a permite utilizatorului să creeze grafuri după bunul plac, editorul de grafuri reprezintă totodată și locul în care se desfășoară toate animațiile și manipulările survenite în urmă rulării unui algoritm.

Există mai multe librării care se ocupă cu reprezentarea grafică a grafurilor într-o aplicație java. Una dintre acestea este JGraphX care, având la baza structura de date definite de librăria 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 pricina acestui inconvenient dar și a altor motive precum aspectul nesatisfăcător și lipsa controlului total asupra funcționalității grafurilor, îmi voi dezvoltă propria modalitate de a reprezenta grafuri în JavaFX.

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

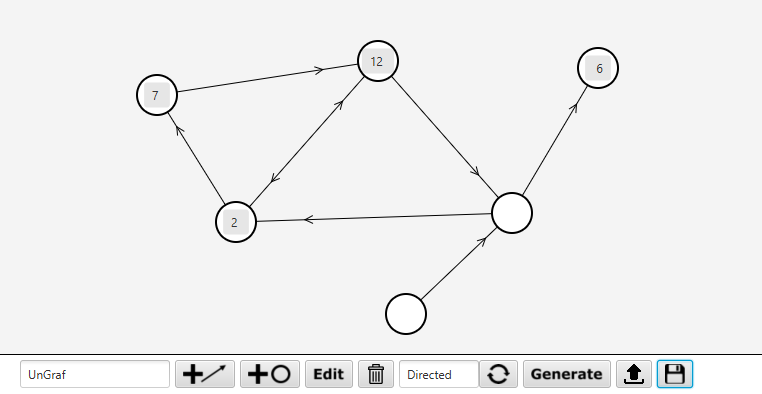


Figure 3 - Editorul de grafuri

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

Muchiile, pe de altă parte, sunt linii negre ce conectează două noduri. La fel că nodurile, muchiile pot sau nu avea 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 prezența la unul dintre capete. În cazul în care muchia respective este una dublă, reprezentarea grafică a acesteia va înfățișa săgeți la ambele capete.

Graful rezultat, reprezentat în editor, este în totalitate responsive. Utilizatorul poate modifică poziția oricăror componente prin drag and drop, iar starea componentelor afectate se va schimbă în mod corespunzător. Editorul de grafuri prezintă următoarele butoane și funcții:

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

**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 consecutive. Primul nod reprezintă sursă, 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 zona liberă din editor.

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

**Edit**: Fiecare componentă a grafului are o eticheta cu valori care este ascunsă atunci când component 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. Voi discuta modul în care funcționează mai jos.

## Management-ul datelor

Atât algoritmii 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. Algoritmii sunt salvați că fișiere .java, făcând astfel mai ușoară compilarea și utilizarea lor. Având în vedere simplitatea resursei folosite de editorul de cod (algoritumul fiind un simplu strîng), această a fost cea mai intuitivă varianta.

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 varianta 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 această descrie un model anterior al clasei, așadar la fiecare schimbare a clasei Graph, toate obiectele serializate anterior vor fi pierdute.

Căutând o soluție adecvată am descoperit formatul de fișier GraphML. Acesta este un format intuitiv și unor de folosit pentru grafuri. Are suport pentru grafuri orientate și neorientate, și este ușor de extins pentru a include diferite atribute nodurilor și muchiilor. Am căutat o biblioteca capabilă să parseze un graf și să îmi ofere serializarea acesteia GraphML, însă nu am găsit unul potrivit, fiecare fiind specializat pe grafuri descrise de librării că JGraphX sau, în cazul celor care puteau lucra cu un graf custom, adăugau prea multe atribute 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 sintaxa strictă, urmând setul de reguli propus de documentație am reușit să fac un parser satisfăcător. Cu ajutorul acestuia am rezolvat problema schimbării modelului de graf, atâta timp cât nu modific definiția grafului în sine (o mulțime de noduri și o mulțime de muchii ce conectează perechi de noduri), pot adaugă și modifică funcționalități, serializarea GraphML v-a descrie corect modelul de graf. Odată cu acest avantaj mai apare unul. 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 user poate prelua un graf interesant și îl poate încarcă în AlgoGraph, atâta timp cât serializarea acestuia respectă standardul GraphML, lucru imposibil de realizat folosind serializarea binară.

## *Flow*-ul de lucru

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

Pentru început se verifică existența fișierului produs în urmă compilării codului scris de către utilizator. Apoi, folosindu-ne de Reflection API creăm o instanța a clasei descrise de acel fișier. Un lucru foarte important de menționat este că obiectul rezultat trebuie să fie o instanța 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. Asta presupune derularea manuală sau, setand un interval, automată a comenzilor asupra grafului. Deci nu putem permite algoritmului să lucreze direct pe graful din editor, întrucât asta 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 problema poate fi rezolvată în două moduri:

**Modul 1:** Putem “coopera” cu utilizatorul. Instruind utilizatorul să foloaseasca apeluri de sleep ale sistemului pentru a creea iluzia de parcurgere secvențială a comenzilor asupra grafului. Totuși, acest approach nupermite 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 adaugarii de funcționalitate. Folosindu-ne de asta am putea modifică obiectele “dummy” în mici istoricuri, capabile să memorize 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, că mai apoi să fie rulate pe graful autentic, după bunul plac.

Evident, am ales a două varianta. Astfel, preluăm graful din editor și ne folosim de el pentru a creea un graf dummy. În continuare, algoritmul utilizatorului este rulat prin Reflection API, oferindu-I graful dummy că parametru. Rezultatul rulării este o lista de comenzi, care, puse în ordine descriu perfect evenimentele ce trebuie parcurse pentru a respectă algoritmul utilizatorului. Această lista 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.

# Implementare si Testare

## Tehnologii folosite

### JavaFX

JavaFX este un set de pachete grafice care permit unui developer să creeze, testeze și 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 librării și API-uri Java native, fiind capabilă să acceseze capabilitățile sistemului sau să se conecteze la alte aplicații sau servere.

Pentru a separă design-ul de restul aplicației, stilizarile pot fi făcute fișiere css care pot fi applicate ulterior. Construcția scenei este făcută fie programmatic fie prin limbajul de scriptare FXML sau chiar prin drag&drop cu ajutorul JavaFX Scene Builder. JavaFX deține o multitudine de controale built-în (butoane, etichete, text input …) 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 aplicației m-am folosit în principal de mediul de dezvoltare JavaFX pentru cea mai mare parte a aplicației, cu preacadere în construcția interfeței grafice, folosind o multitudine de controale specific JavaFX dar și librării dezvoltate ulterior, menite să îmbogățească capabilitățile utilizatorului, precum RichTextFX. Pentru desenari mai complexe ale grafului m-am folosit de Canvas API, un feature built-în JavaFX care permite desenarea a diferite forme geometrice pe o arie din scenă aplicației. Deși aveam și alte soluții mai ușoare am ales să construiesc scenă principala a aplicației în mod programmatic, având aceeași soluție și în cadrul stilizării, cu mici excepții.

### .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 coe 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 injectrion este un feature built-în, posibilitatea de a fi hostat 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 creări, modificării și interogării respectivelor resurse. Pentru comunicarea cu baza de date am folosit Entity Framework Core, un framework ce permite maparea tabelelor la entități .Net, ușurând foarte mult dezvoltarea aplicației.

### Angular

Fiind un framework folosit în crearea de aplicații client, Angular este un set de librării 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 coerente. 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 clasa 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.

### Alte librării

            În cadrul proiectului am folosit diferite librării menite să îmi ușureze muncă. RichTextFX este una dintre acestea, o librărie destinată aplicațiilor JavaFX, această oferă niște unelte avansate în dezvoltarea unui editor de text modern. Cu ajutorul acesteia cât și a unor tutoriale explicative găsite pe pagină 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 librăriei 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 formatării unui strîng, această operațiune a fost simplu de executat, înlocuind codul din editor cu textul rezultat din formatarea codului respectiv.

            Jersey este un framework open-source  utilizat pentru dezvoltarea serviciilor web REST. Cu toate acestea eu nu m-am folosit de această latura, 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 folosit pentru a trimite și primii date cu privire la grafuri și algoritmi de la un API.

## Detalii tehnice

### Compilarea algoritmului

Algoritmul utilizatorului este stocat local într-un fișier .java, facilitând ușurință compilării și a editării. Am implementat clasa ProgramCompiller care, folosindu-se de o instanța JavaCompiller oferită de către Environment, poate compilă fișiere java a căror cale a fost primită anterior. 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 cu pricina. Folosindu-mă de una din funcționalitățile RichTextFx creez un mapping observabil care dictează că orice linie a cărei număr este egal cu numărul liniei cu eroarea găsită anterior v-a prezența un mic icon ce o evidențiază. Un mapping asemănător folosesc și pentru numerotarea liniilor.

### Rularea algoritmului

După compilarea cu success 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 instantiate folosindu-ne de Reflection API. Algoritmul scris de către utilizator este forțat să implementeze clasa abstractă Program, deci în continuare este verificată această condiție. Având confirmarea necesară știm că instant clasei deține o metodă run(graph) pe care o putem instantia și rula după bunul plac.

### Colectarea comenzilor

O comandă este o acțiune de manipulare a grafului survenită în urmă unor metode apelate de către algoritmul utilizatorului. O astfel de comandă poate definii accesarea unor noduri, ștergerea unei muchii, schimbarea culorii unei componente, printarea unui mesaj la consola și multe altele. Fiecare comandă extinde clasa abstractă Command. Acest lucru obligă clasa să dețînă o variabilă commandOrder care definește ordinea comenzii în întreagă colecție de comenzi survenite în urmă rulării algoritmului. La execuția comenzii această primește graful pe care trebuie să opereze și un număr ce semnifică durata animației. Această durata este seta de utilizator printr-un slider, în funcție de viteză dorită.

            Întrucât nu vrem să rulăm algortimul direct pe graful din editor (viteză manipulării grafului ar fi mult prea mare pentru utilizator), am dezvoltat o funcționalitate care mapează graful și fiecare componentă a să 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 că un istoric al acțiunilor luate de către algoritm asupra structurii de date. Așadar, orice acțiune asupra unei componente din graf v-a 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.

  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, interogand atât graful cât și fiecare nod și muchie a să. Astfel obținem o lista 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.

### 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 metodă 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 v-a există un delay de lungime variabilă nu putem parcurge comenzile pe firul de execuție principal, apelul unor metode de tipul sleep ar suspendă 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 4 - 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.

Pentru a fi mereu la curent cu alegerile utilizatorului, clasa commandsRunner deține o instanța de panou de control pe care o consulta 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 că delay. Lucrul care ne permite o bună tranziție între modurile de execuție a comenzilor (Automatic, Manual, Pause) este un obiect de tipul lock. Acesta suspendă și pornește firul de execuție în funcție de alegerea utilizatorului.



Figure 5 – Bucla principala din CommandsRunner

### 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.

Fiind un fișier XML, construcția să este destul de ușor de realizat folosindu-mă de clasa Document, care facilitează ușoară 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 adaugă aceste entități trebuie să adăugăm o serie de noduri atribut, fiecare având un id și o entitate țintă. În cazul meu am 5 astfel de atribute: 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, o muchie sau un nod, are nevoie de un id numeric cu ajutorul căruia poate fi referita în diferite relații. Drept exemplu, o muchie trebuie să prezinte un atribut source și unul target, valorile acestor attribute fiind id-urile nodurilor ce îndeplinesc acel rol.

### Generatorul de grafuri

Acesta are două mari componente: interfață grafică și generatorul în sine. Interfață grafică este o nouă scenă care, odată activată, apare că un popup pe ecranul utilizatorului. Această prezintă diferite variații de constrângerii sub formă de checkmark-uri, la selectarea unei constrângeri vor apărea în view și input-urile specifice ei. Avem trei tipuri mari de constrângeri:

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

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

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

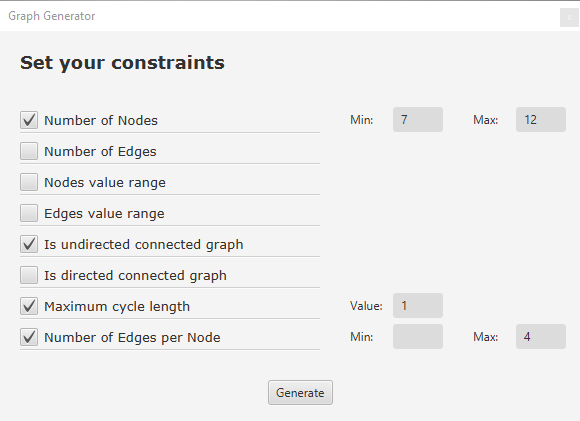


Figure 6 – Interfata grafica a generatorului de grafuri

Fiecare constrângere trebuie să implementeze clasa abstractă Constraint, astfel fiind forțate să implementeze metodă 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 posiblitatea că această 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.

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

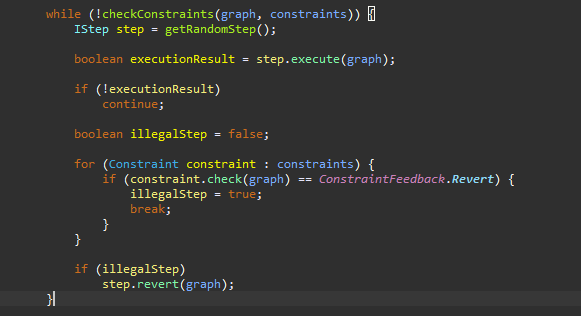


Figure 7 - Secventa de cod din GraphGenerator

Fiecare pas din procesul de generare al grafurilor trebuie să implementeze interfață IStep, asigurându-ne astfel că fiecare pas oferă posibilitatea de execuție și rollback. 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.

### Upload si Download

Aplicația oferă utilizatorului posibilitatea de a încarcă algoritmi și grafuri într-o baza de date centralizată astfel încât, prin intermediul unui panou de browsing, alți utilizatori le pot descarcă și folosii. Baza de date este administrate de către un API Rest care expune câteva rute folosite de către aplicație pentru a poștă 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.

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

|  |  |
| --- | --- |
| Figure 8 – Fereastra de browsing a algoritmilor | Pe de altă parte, pentru a descarcă 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. |

### 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. Această conține două tabele, unul pentru grafuri și unul pentru programe. Pentru a stoca corpul unei astfel de resurse m-am folosit de un Dată Storage pentru blob-uri hostat în Microsoft Azure, astfel fiecare entitate din tabele conține câte un id în plus care face legătură între ea și blob-ul asociat ei din Dată 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 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.

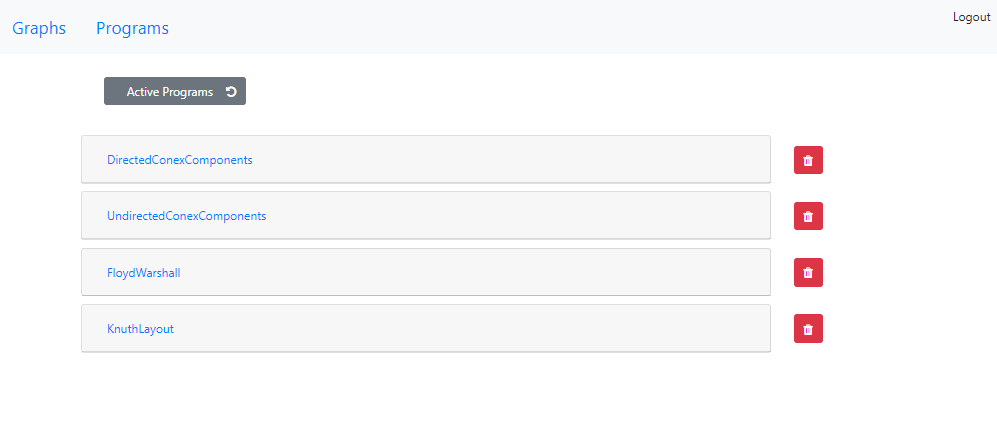
Pentru a oferii o interfață grafică atractivă, AlgoGraph pune la dispoziție o aplicație web dezvoltată în Angular ce permite management-ul resurselor. Această necesită autentificare cu un cont de administrator și permite mai multe operații pe grafurile și algoritmii încărcați de către utilizatori. În momentul în care o nouă resursă este încărcată în baza de date, această 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 schimbă statutul după care se 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 9 - Interfata administratorului, realizata in Angular

## Testare

Testarea unei aplicații software este o investigație continuă menită să ofere o vizualizare obiectivă și independența asupra capabilității 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 principala, 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 capabilitățile 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.

# Concluzii

În concluzie sunt de părere că AlgoGraph este o unealtă educativă excelență 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.

Algoritmică 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 modifică un algoritm după bunul sau plac mi se pare o funcționalitate extrem de importantă, adăugând o nouă dimensiune aplicațiilor existente. Algoraph dispune din start de o gama destul de variată de algoritmi ajutatori, meniți să demonstreze capabilitățile aplicației și să-i introducă utilizatorului modul de lucru cu aplicația.

Că 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 fereastră de browsing ar putea fi ierarhizate după numărul de voturi, fiind un insemt 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. Deasemenea, 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/>