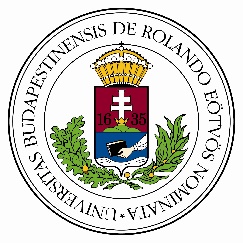
**Eötvös Loránd Tudományegyetem  
Informatikai Kar  
Adattudományi és Adattechnológiai Tanszék**

**Keretrendszer nagy erőforrás igényű számítások elosztott végrehajtására**

**Kiss Péter Polozgai Máté**

Doktorandusz, programtervező informatikus (Msc) nappali tagozat

programtervező informatikus

Budapest, 2019

Tartalom

# 1. Bevezetés

## 1.1. A szakdolgozat ötlete

Napjainkban az informatika tudományban nagy szerepet tölt be, hogy a komplex, nagy számításigényű feladatokra - amik meghaladják a lokálisan igénybe vehető erőforrásainkat - olyan elosztott keretrendszert tervezzünk, amivel külső erőforrást igénybe véve, az adott feladat végrehajtása optimálisabbá váljon, futási ideje pedig jelentősen lerövidüljön.

A témához igazodva egy olyan, az elosztott végrehajtást támogató rendszer megvalósítása volt a cél, ami egy már meglévő nagy erőforrás igényű algoritmust ültet új alapokra. Így esett a választásom egy legrövidebb út kereső algoritmusra. A szakdolgozat így egy speciális gráf kereső algoritmust valósít meg. Az algoritmussal egyszerű, összefüggő gráfokban lehet megkeresni a legrövidebb utat egy adott csúcsból kiindulva egy másikba. A kereső algoritmus működése során a gráf csúcsai fognak kommunikálni egymással üzenetsorok segítségével. A gráf csúcsait így tekinthetjük klienseknek is, akik üzeneteket fogadnak és küldenek egy megadott algoritmus alapján.

A gráfok és a velük kapcsolatos algoritmusok alapvető szerepet játszanak az informatika tudományban. Ezek közül talán a legjelentősebb, és a legtöbb felhasználási területtel rendelkező a legrövidebb utak problémája, aminek segítségével például olyan hétköznapi problémára kaphatunk egyszerű és kézzel fogható megoldást, hogy hogyan jutunk el A városból B városba.

Bár ezeknek a megoldására már számos algoritmus született, a szakdolgozat legfőbb célja, hogy rámutasson azokra az elosztott számítási módszertanokra, valamint bemutassa az ezekhez kapcsolódó technológiákat, mely során az algoritmus az adatmozgatás minimalizálására törekszik. Az eredmény pedig egy weboldalon kerül vizualizálásra.

# 2. Felhasználói dokumentáció

A következőkben ismertetésre kerül a program telepítése, összes funkciója, azoknak pontos használata. Továbbá részletezésre kerülnek a felhasználó számára szükséges lépések a program használatához. A felhasználói dokumentáció első felében az alapvető gráf elméleti fogalmakról is szó fog esni, amelyek elengedhetetlenek a szakdolgozat megértéséhez.

## 2.1. Gráfelméleti fogalmak, jelölések

**Definíció (irányítatlan gráf):** A G=(α, E, V) hármast (irányítatlan) gráfnak evezzük, ha E, V halmazok, V≠Ø, V∩E=Ø és α: E→{{v,v'}| v,v' ∈ V}. E-t az élek halmazának, V-t a csúcsok (pontok) halmazának nevezzük és α-t az illeszkedési leképezésnek nevezzük. Az α leképezés E minden egyes eleméhez egy V-beli rendezetlen párt rendel.

**Definíció (egyszerű gráf):** Ha egy él egyetlen csúcsra illeszkedik, azt hurokélnek nevezzük. Ha e≠e' esetén α(e)=α(e'), akkor e és e' párhuzamos élek. Ha egy gráfban nincs sem hurokél, sem párhuzamos élek, akkor azt egyszerű gráfnak nevezzük.

**Definíció (szomszédos csúcsok):** A v≠v' csúcsok szomszédosak, ha van olyan e∈E, amelyre v∈α(e) és v'∈α(e) egyszerre teljesül.

**Definíció (fokszám):** A v csúcs fokszámán a rá illeszkedő élek számát értjük, a hurokéleket kétszer számolva. Jelölése d(v) vagy deg(v)

**Definíció (izolált csúcs):** Ha d(v)=0, akkor v-t izolált csúcsnak nevezzük.

**Definíció (séta):** Legyen G=(α, E, V) egy gráf. A

v0, e1, v1, e2, v2,.....,vn-1, en,vn

sorozatot sétának nevezzük, v0-ból vn-be, ha:

* vj ∈ V 0 ≤ j ≤ n,
* ek ∈ E 1 ≤ k ≤ n,
* α(em) = {vm-1, vm} 1 ≤ m ≤ n.

**Definíció (út):** Ha a sétában szereplő csúcsok mind különbözőek, akkor útnak nevezzük.

**Definíció (összefüggő gráf):** Egy gráfot összefüggőnek nevezünk, ha bármely két csúcsa összeköthető sétával.

**Definíció (fa):** Egy gráfot fának nevezünk, ha összefüggő és körmentes.

**Definíció (feszítőfa):** A G gráf egy F részgráfját a feszítőfájának nevezzük, ha a csúcsainak halmaza megegyezik G csúcsainak halmazával, és fa.

**Definíció (címkézett gráf):** Legyen G=(α, E, V) egy gráf, Ce és Cv halmazok az élcímkék, illetve csúcscímkék halmaz, továbbá ce : E → Ce és cv : V → Cv leképezések az élcímkézés, illetve csúcscímkézés. Ekkor a (α, E, V, ce, Ce, cv, Cv) hetest címkézett gráfnak nevezzük.

**Definíció (élsúlyozott gráf):** Ce = R esetén élsúlyozásról és élsúlyozott gráfról beszélünk, és a jelölésből Ce-t elhagyjuk.

**Definíció (legrövidebb út):** Legrövidebb út alatt a gráfelméletben egy minimális hosszúságú utat értünk egy gráf két különböző u és v csúcsa között. Ha vannak súlyok a gráf élein, akkor pedig olyan útról beszélünk, amelynek élein szereplő súlyok összege minimális.

## 2.2. Minimális rendszerkövetelmények

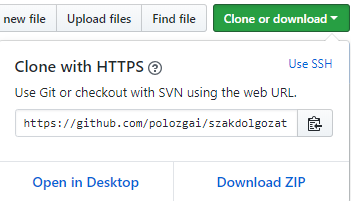
A felhasználónak az alábbiakra van szüksége ha futtatni szeretné a programot:

* Ajánlott operációs rendszer: Windows 10
* Java SE Runtime Environment 8
* Ajánlott web böngésző: Google Chrome

## 2.3. A program telepítése

A program telepítésére két lehetőség van. Ha csak futtatni szeretnénk a programot a szakdolgozathoz mellékelt CD-n megtalálható a **JMSGraphAlgorithm** nevű **".jar"** kiterjesztésű fájl, amire kétszer kattintva a program automatikusan elindul. A másik lehetőség a futtatásra, hogy a JAR-t tartalmazó mappában indítunk egy parancssort és beírjuk a következő parancsot: **java -jar JMSGraphAlgorithm.jar** . A program az indulásakor a JAR melletti **input** mappában található **graph.txt-**ből olvassa be az adatokat, de ezt a program futása során is lehetőség van módosítani. A . A weboldal eléréséhez meg kell nyitni a "**http://localhost:4567**" című URL-t egy általunk választott böngészőben. A projekt fejlesztése során a **Google Chrome** nevű böngészőt használtam. Az így elénk tároló grafikus felhasználói felület használatáról a későbbiekben esik szó.

Amennyiben a teljes projektet szeretnénk megnyitni, ezt megtaláljuk a szakdolgozathoz tartozó CD mellékleten. A <https://github.com/polozgai/szakdolgozat> weboldalról is letölthetjük a "**Clone or download**" gombra kattintva, azon belül pedig a "**Download ZIP**"-re. Miután letöltődött, csomagoljuk ki a projektet, majd nyissuk meg egy fejlesztői környezetben.



2.1. ábra. A teljes ptojekt letöltése GitHubról

## 2.4. A program megnyitása Intellij IDEA-ban

Miután a projekt fejlesztése közben az Intellij IDEA fejlesztői környezetet használtam, ezért ez az ajánlott fejlesztői környezet, de mivel a projekt egy Maven projekt, ezért más fejlesztői környezetben, például Eclipse, Netbeans is könnyen lehet importálni.

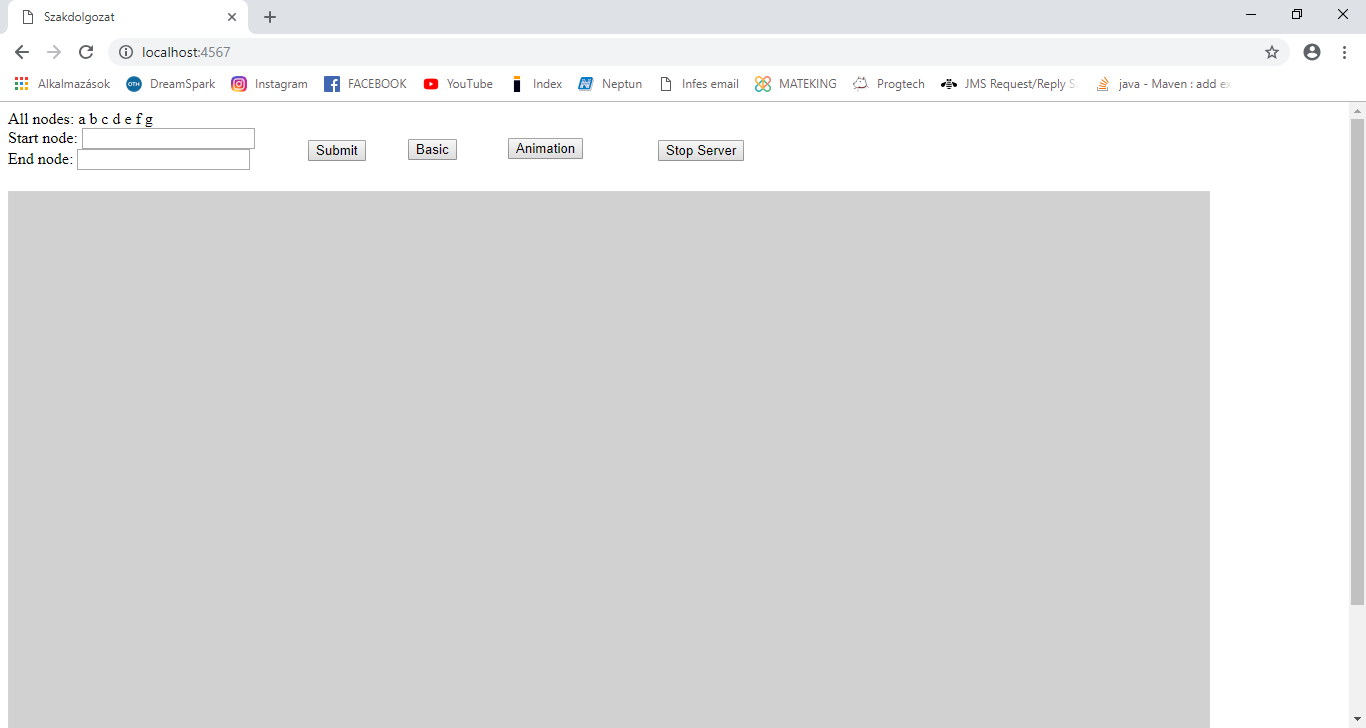
Intellij-ben a "**File**" >> "**Open..."** menüpontok segítségével lehet kiválasztani azt az elérési útvonalat, ahova a projekt kicsomagolásra kerül, majd az "**OK**" gombra kattintva egy kis idő múlva a projekt betöltésre kerül.

## 2.5. A program funkciói

Ebben a fejezetben bemutatásra kerül a felhasználó számára, hogy mely funkciókat hogyan érheti el és azokat hogyan használhatja.

### 2.5.1. A program indulása

A program indulásakor a felhasználó a választott böngészőjében a "**http://localhost:4567**" URL címen érheti el a grafikus felhasználói felületet. A megjelent képernyő felső részében láthatóvá válik a gráf összes csúcsa, a "**Start node**" és "**End node**" feliratok, a mellettük található szövegdobozok, és a négy gomb: a "**Basic**", az "**Animation**", a "**Submit**", és a "**Stop** **Server**".



2.2. ábra. A weblap kinézete.

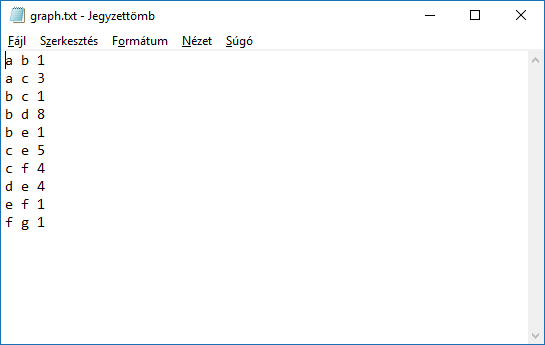
A felhasználó ezeknek a gomboknak a segítségével tudja inicializálni az algoritmust, és az algoritmus működését illetve annak a vizualizálását kiválasztani, és lehetősége van a szerver leállítására is. Ezek alatt található meg az a szürke hátterű panel, amelyben a gráf, és a két választott csúcs közötti legrövidebb út kerül szemléltetésre. A panelbe az egér görgője segítségével lehet bele nagyítani és kicsinyíteni is, valamint a bal egérgombot lenyomva tartva lehet navigálni a panelen belül.

## 2.6. Felhasználói esetek

A következőkben részletezésre kerül a mappák és a bennük található fájlok szerepe, valamint a weblapon elérhető funkciók részletezése.

### 2.6.1. A gráf beolvasása

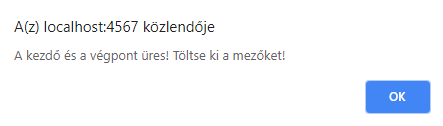
A gráf megadása az input mappa **graph.txt-**ben történik, méghozzá a gráf két csúcsának és a köztük futó él súlyának a megadásával. Először meg kell adni a gráfnak azt a két csúcsát mely közt az él futni fog, ezeket szóközzel elválasztva, majd még egy szóköz hozzáadásával az adott élhez tartozó élsúlyt is meg kell adni. Ügyeljünk arra, hogy minden ilyen élét új sorban kell megadni, valamint, hogy a gráfnak összefüggő, egyszerű gráfnak kell lennie. Ha a programot a kész JAR fájllal futtatjuk a **graph.txt-**t bármikor módosíthatjuk a JAR futása közben is, csupán frissítenünk kell az oldalt a következő futtatás előtt.



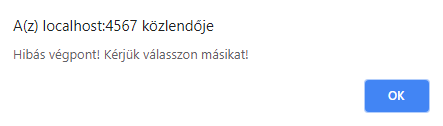
2.3. Egy példa a graph.txt helyes kitöltésére.

### 2.6.2. Az keresett út kezdő és vég pontjának megadása

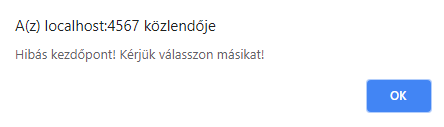
A böngésző ablak bal felső részében elhelyezkedő "**All nodes**" felirat mellett találhatja a felhasználó a gráf összes csúcsát, melyek közül kettőt kell beírnia a "**Start node**", és "**End node**" feliratok melletti szövegdobozba, majd vagy az Enterrel, vagy a szövegdobozok mellett található "**Submit**" gombra kattintva kerül továbbításra a szerver számára. Amennyiben nem a felsorolt csúcsok közül választ kezdő, illetve végpontot, vagy amennyiben valamelyik csúcs értékét üresen hagyja, illetve megegyezik a kezdő és a végpont, a program egy figyelmeztető dialógust dob fel a böngésző ablakában.



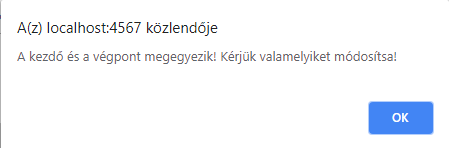
2.4. ábra. Kitöltetlenül hagyott csúcspontok esetén a **Submit** gombra kattintva kapott hibaüzenet.



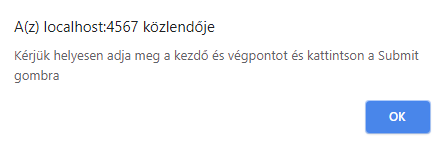
2.5. ábra. Rosszul megadott végpont esetén a **Submit** gombra kattintva kapott hibaüzenet



2.6. ábra. Hibásan megadott kezdőpont esetén a **Submit** gombra kattintva kapott hibaüzenet.



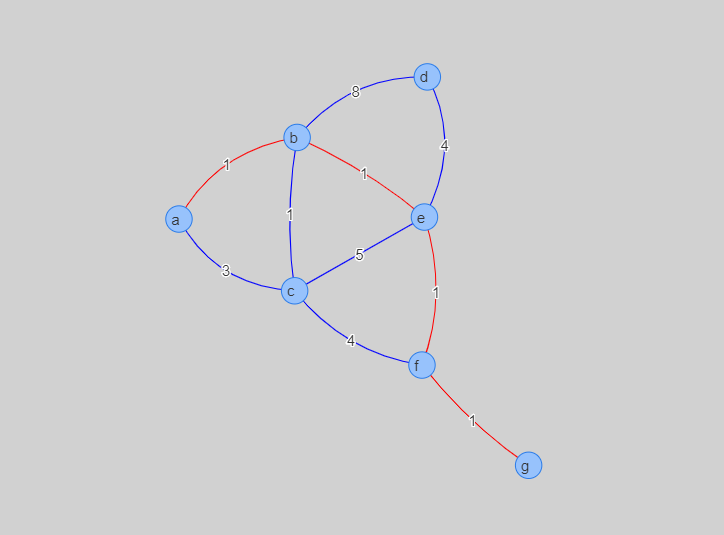
2.7. ábra. Egyező kezdőpont és végpont esetén a **Submit** gombra kattintva kapott hibaüzenet.



2.8. ábra. Hibásan kitöltött kezdő és végpont esetén kapott hibaüzenet a **Basic** vagy az **Animation** gombra kattintva

### 2.6.3. A "Basic" gomb funkciója

Amennyiben a felhasználó hibaüzenet nélkül megadta a kezdő, illetve a végpontot, és amennyiben a "**Basic**" gombra kattint, akkor a képernyő szürke hátterű panel részén a vizualizációs **Javascript** algoritmus ki fogja rajzolni az egész gráfot, és a korábban megadott két csúcs között a legrövidebb utat tartalmazó éleket **pirossal** rajzolja ki. Ha a gráfot kesze-kuszán rajzolná ki a vizualizáció, a gráf egyik csúcsára kattintva kell el kezdeni a gráfot jobbra-balra rángatni, majd ha már normál formára állt vissza a gráf kattintani kell egyet a szürke panelre, és így léthatóvá válik a keresett legrövidebb út is. Ez a kirajzolási forma ritkán fordul elő, sajnos ez a használt javascript könyvtár sajátossága.

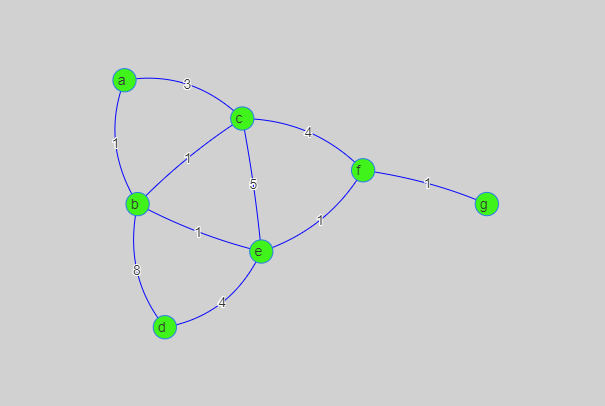


2.9. ábra. A **Basic** gomb megnyomásakor kirajzolódó gráf.

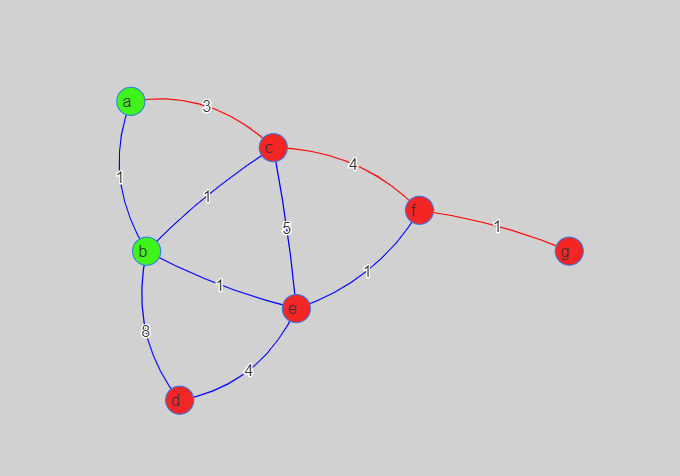
### 2.6.4. Az "Animation" gomb funkciója

Amennyiben a felhasználó hibaüzenet nélkül megadta a kezdő, illetve a végpontot, és amennyiben az "**Animation**" gombra kattint, akkor a képernyő szürke hátterű panel részén a minimális utat kereső gráf algoritmus működése lépésről-lépésre be lesz mutatva, mégpedig a következő módon:

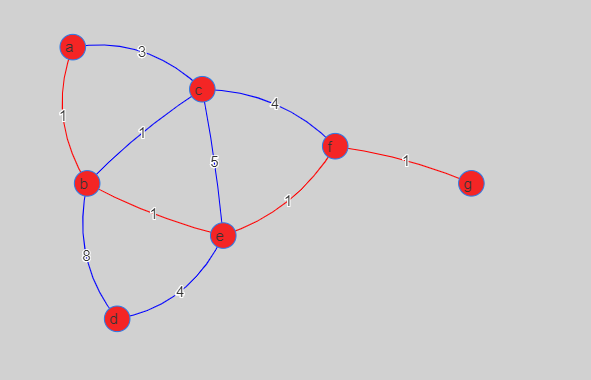
1. A grafikus felületen a kezdő ponttól kiindulva csúcsról-csúcsra haladva a gráf utolsó csúcsáig, függetlenül attól, hogy melyik csúcsot választottuk végpontnak megjelennek a gráf csúcsai **zöld** háttérrel.
2. Az előbb említett sorrend fordítottjából haladva kezdi meg az algoritmus a futásának második szakaszát. Az algoritmus csúcsról-csúcsra lépeget vissza egészen a kezdő pontig, miközben a gráf csúcsai **zöldről** **pirosra** váltanak. Ez a színváltás azt jelenti, hogy az adott csúcs ha kapott adatot egy másik csúcstól azt már feldolgozta, és ezután ő is tovább küldte ezeket a már frissített adatokat. Az információ küldése csak két szomszédos gráf közt mehet végbe, ahol a fogadónak, még aktívnak kell lennie, vagyis zöldnek kell lennie. Egy ilyen csúcsnak pedig már nincs több feladata, így magát inaktív állapotba helyezi, vagyis nem fogad és nem is küld adatokat.
3. A gráf csúcsai közötti lépegetés során amennyiben a kezdő csúcs így talál egy esetleges legrövidebb utat, ennek az útnak az **éleit kiszínezi piros színnel**. Ha az algoritmus következő lépéseiben talál egy az előbb említett útnál kevesebb összköltséggel bírót, akkor az előbb említett utat **törli**, vagyis az **éleit visszaállítja kék színre**, és ennek az **új útnak az éleit színezi ki piros színnel**.
4. Az algoritmus futása akkor ér véget, ha a gráf **összes csúcsa piros** színű lesz, és az így látható piros színű élek alkotta út lesz a **legrövidebb út** a korábban választott két csúcs között.
5. Megjegyzés: Ha az "**Animation**" funkciót választjuk, meg kell várnunk az algoritmus befejezését, ezáltal a funkció gombok letiltásra kerülnek, hogy a felhasználó még véletlenül se tudja hibás működésre késztetni a programot.



2.10. ábra. Az algoritmus első szakaszának az animációja.



2.10. ábra. Az algoritmus által megtalált első ideiglenesen minimális út.



2.11. ábra. Az algoritmus talált egy minimálisabb utat, így a régi út törlésre kerül, az új utat megjelöljük.

### 2.6.5. "Stop Server" gomb funkciója

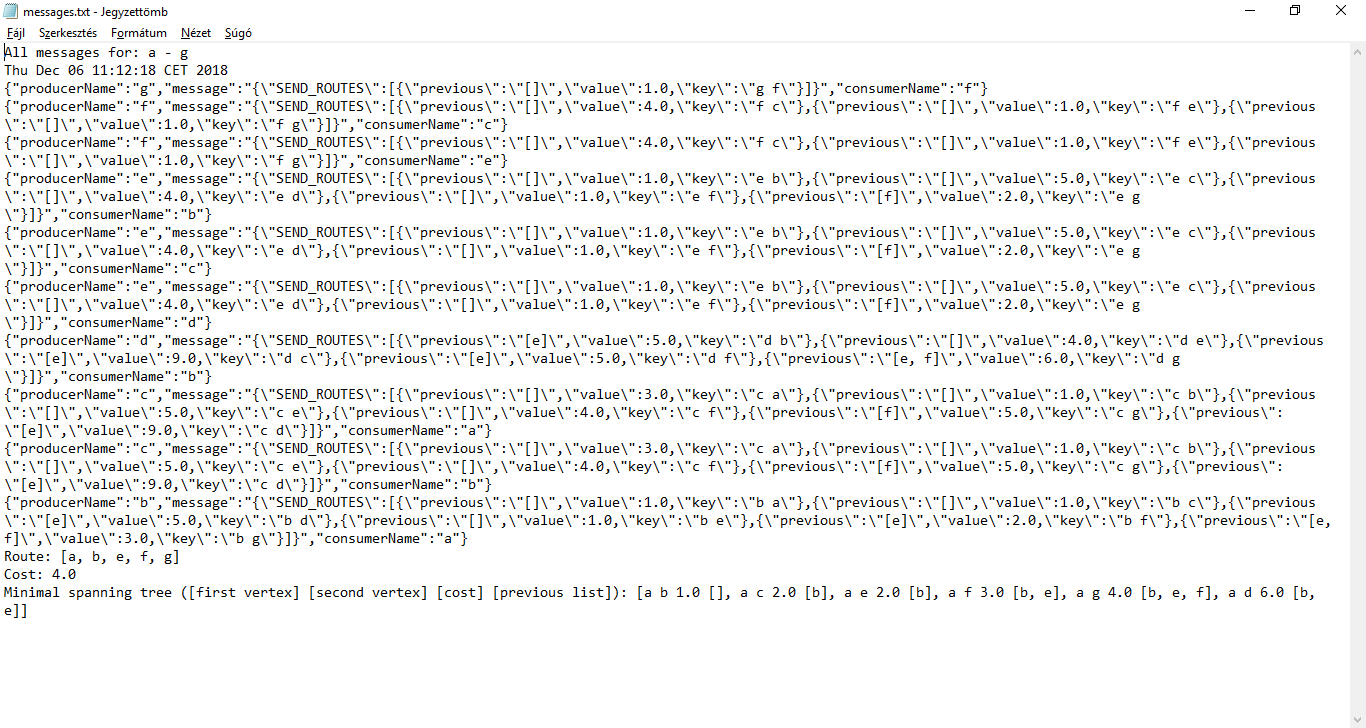
A weboldalon található egy "**Stop Server**" gomb is. Ügyelni kell arra, hogyha JAR-ból futtatjuk a programot, és be akarjuk zárni a weboldalt, ezt a gombot mindig nyomjuk meg, mert ha nem tesszük meg a futó folyamatként a háttérben ugyanúgy futni fog a szerver. A gomb megnyomása esetén az **error.html-**re kerülünk átirányításra. Ha mégis elfelejtenénk Windows operációs rendszer esetén a Feladatkezelőben a megfelelő folyamatot kikeresve le lehet állítani a futását a JAR-nak, így megelőzve az esetleges nem definiált működést.



2.12. ábra. Error.html

### 2.6.6. Output mappa

Az **output** mappában található **messages.txt** fogja tartalmazni az egyes keresésekre lefuttatott utak adatait, időpontját, a közben elküldésre került üzeneteket, a legrövidebb utat a csúcscímkéken keresztül, a minimális út összköltségét, valamint a kezdőpontból kiinduló összes minimális utat, és az ebből egyébként kirajzolható feszítőfát.



2.13. ábra. A messages.txt tartalma egy keresés lefuttatása után.

# 3. Fejlesztői dokumentáció

A következő fejezetek tartalmazni fogják a program részletes felépítését, a felhasznált technológiák leírását, valamint ezek felhasználását a projektben. Ismertetésre kerül az algoritmus részletes működése, a szerver oldal, valamint az algoritmus működésére szolgáló weblap részletes leírása. Leírásra kerül az algoritmus tesztelési folyamata is. A programnak a kereső gráf algoritmusa, valamint a szerver forráskódja **Java** nyelven íródott. A weblap **HTML** nyelvben íródott, melyhez a **CSS** stílusleírót használtam. A weblapon történő szemléltetésre a **Javascript** programozási nyelvet használtam felhasználva egy külső **Javascript** könyvtárat. A projekt fejlesztése során ügyeltem arra, hogy a forráskódban szereplő egymástól jól elkülöníthető vezérlési egységek forráskódjai különálló csomagokba kerüljenek.

## 3.1. Minimális rendszerkövetelmények a fejlesztéshez

A felhasználónak az alábbiakra van szüksége ha meg szeretné nyitni a projektet:

* Windows operációs rendszer (Ajánlott: Windows 10)
* Java SE Development Kit 8
* Apache Maven
* Ajánlott fejlesztői környezet: Intellij IDEA
* Ajánlott web böngésző: Google Chrome

## 3.2. Forráskód letöltése

Ennek a fejezetnek első része a szoftver fejlesztése során felhasznált verziókezelő megismertetésére, valamint szoftver letöltési lehetőségeire szolgál.

### 3.2.1. Verziókezelés

A program verziókezeléséhez a **Git**-et használtam, amely az egyik legismertebb ilyen szoftver. A lényege, hogy a forráskódot nem csak a saját gépünkön tárolhatjuk, hanem egy felhőben lévő tárhelyen is, egy saját **repository**-ban. Innen bárki más is letöltheti, figyelemmel kísérheti a projekt fejlesztését, illetve adott esetben másokkal is együtt lehet dolgozni az adott projekten. A tárhelyen megtalálhatók a projekt biztonsági mentései is a forráskódról, így nem kell aggódnunk, hogy elveszik a munkánk.

Tárhely szolgáltatásnak a **GitHub**-ot választottam, mely részben ingyenes, könnyen kezelhető grafikus felületet nyújt, és akár a fiókkal nem rendelkező felhasználók számára is elérhető a forráskód.

A **Git** használatához telepíteni kell magát a verziókezelő programot, amely a **Git** hivatalos oldaláról: <https://git-scm.com/downloads> letölthető a megfelelő operációs rendszer kiválasztása után.

### 3.2.2. Letöltési lehetőségek

**Github**: A szakdolgozat projektjét tartalmazó **repository** megtalálható a következő weblapon: <https://github.com/polozgai/szakdolgozat> , ahol a zöld "**Clone or download**" gombra kattintva felnyíló ablakon két lehetőség van a letöltésre:

1. Letöltés tömörített mappaként: "**Download ZIP**” gombra kattintva, majd egy tetszőlegesen választott mappába kicsomagolva a letöltött állományt.
2. Tárhely klónozás, vagyis a **Git** segítségével a projekt **repository**-ban lévő könyvtárat egy az egyben letölti egy általunk kijelölt mappába. Ehhez a művelethez le kell tölteni az előbb említett **Git** verziókezelőt, valamint rendelkezni kell **GitHub** fiókkal. Elsőként meg kell nyitni egy parancssort abban a könyvtárban, ahova szeretnénk a forráskódot letölteni. Ezután be kell gépelni a következő parancsot: **git clone https://github.com/polozgai/szakdolgozat.git** , majd **entert** kell ütni. A program a **Github** felhasználó nevünket és jelszavunkat fogja kérni, majd automatikusan végrehajtja a kiadott parancsot.

A szakdolgozathoz mellékelt CD-n is megtalálható a forráskód.

## 3.3. Projekt megnyitása

A szakdolgozat forráskódja egy **Maven** projektként lett létrehozva, így könnyen meg lehet nyitni bármely, az ilyen projektek megnyitását támogató fejlesztői környezetben. A forráskód az **Intellij IDEA** nevű fejlesztői környezetben íródott, így ez az általam is ajánlott fejlesztői környezet.

Miután betöltődött az **Intellij IDEA**, a bal felső sarokban lévő "**File**" menüpontra kell kattintani, majd az itt legördülő ablakon ki kell választani az "**Open...**" opciót. Az itt elénk táruló ablakban meg kell keresni a projektet, majd az "**OK**" gombra kattintva automatikusan betöltődik a projekt.

## 3.4. Felhasznált technológiák

A projekt részletes bemutatása előtt a következőkben a projekt létrehozásához, és működéséhez elengedhetetlen alkalmazásprogramozási interfészt, és az őt megvalósító technológiáról, valamint projekthez tartozó egyéb függőségekről lesz szó.

### 3.4.1. Maven

A Maven egy a Java programozási nyelvhez kifejlesztett csomagoló keretrendszer. Működése során feladatokat automatizál. Bevezeti a POM, azaz a Projekt Objektum Modell fogalmát. A gyakorlatban a buildelendő projekt leírását a könyvtárszerkezetben található **pom.xml**-ben lehet megadni, valamint ennek a függőségeit, amelyeket be is szerez. Lefordítja a forráskódot, amit így a Java virtuális gépe értelmezni tud, illetve összeállítja a .jar, .war fájlokat.

A Maven-t teljes körű használatához telepíteni kell, amihez a következő oldalon található az útmutató: <https://maven.apache.org/install.html>

### 3.4.2. Java Message Service

A Java Message Service (röviden JMS) egy Java API, amellyel üzeneteket lehet küldeni két vagy több kliens között. Az elosztott rendszerekben az üzenetküldés módszere aszinkron kommunikációt biztosít a kliensek számára, hogy egy üzenetsoron keresztül kommunikáljanak egymással, anélkül hogy a küldő oldali kliens vagy a fogadó oldali kliens bármilyen információval rendelkezne a másikról.

A JMS API kétféle modellt támogat:

* **point-to-point** modell
* **publish-subscribe** modell

A szakdolgozatban a **point-to-point** modell lett megvalósítva, ezért csak ez kerül bemutatásra. Erről a modellről általánosságban el lehet mondani, hogy a kommunikáció a "producer" kliens, és a "**consumer**" kliens között zajlik. Ebben az esetben a "producer" ismeri a fogadót, és közvetlenül az ő üzenetsorában publikálja az üzenetét, viszont ezt az üzenetet kizárólag egy "**consumer**" kliens kaphatja meg. Általánosságban elmondható az is, hogy mivel az üzenetküldés egy külső komponensen zajlik, ezért nem kell a "**producer**"-nek futnia, amikor a "**consumer**" megkapja az üzenetet, illetve a "**consumer**"-nek sem kell futnia, amikor a "**producer**" publikálja az üzenetet.

### 3.4.3. Apache ActiveMQ

A Java Message Service használatához szükség van egy őt implementáló szolgálatóra. A szakdolgozat az Apache ActiveMQ nevű nyílt forráskódú üzenetbrókerét használja, melynek a függősége természetesen megtalálható a Maven által létrehozott **pom.xml** fájlban. Az ehhez tartozó bróker URL-je a következő: "**tcp://localhost:61616**" Amennyiben nem csak a forráskódban szeretnénk használni a szoftver adta szolgáltatást, de monitorozni is szeretnénk a klienseket, illetve létrehozni vagy szerkeszteni is akarjuk az üzenetsorokat, akkor lehetőségünk van letölteni az ActiveMQ legfrissebb stabil verzióját.

Telepítés lépései:

1. Keresse fel ezt a weboldalt: <http://activemq.apache.org/activemq-5158-release.html>. Itt válassza ki a számunkra megfelelő operációs rendszert, majd töltsük le a zip állományt
2. A zip állományt csomagoljuk ki egy tetszőleges mappába, majd lépjünk bele az így kapott könyvtár "**bin**" nevű mappájába, ahol nyissunk egy parancssort.
3. A parancssorba gépeljük be a következőt és üssünk entert: **activemq start**
4. Ennek a parancsnak a hatására a kiírt információk alsó felében láthatjuk, hogy a web konzol melyik porton indul el. Nyissuk meg ezt a portot a localhost-on és a /admin eléréssel. Pl (**http://localhost:8161/admin**).
5. Itt be kell jelentkezni az "**admin**"/"**admin**" párossal, és már használni is lehet az üzenetbróker által nyújtott szolgáltatásokat.

### 3.4.4. Spark Java

A Spark Java egy keretrendszer webes alkalmazások létrehozásához a Java 8-hoz. A szakdolgozat projektje ezt a webes alkalmazást használja, hogy összekösse az algoritmus modell logikáját a grafikus felülettel. Tulajdonképpen szerverként szolgál, ahol az információ továbbításra kerül, és a bejövő kérések feldolgozásra kerülnek. A használatához természetesen meg kell adni a függőségét a Maven által generált **pom.xml**-ben. A keretrendszerhez tartozó dokumentáció, valamint a használatához szükséges példa kód részletek megtalálhatók a Spark Java hivatalos honlapján: <http://sparkjava.com/>

### 3.4.5. JSON-Simple

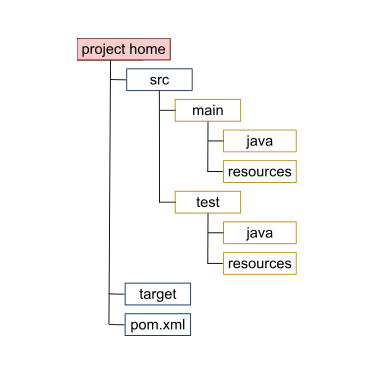
A projektben az üzenetek megkonstruálására a [javax.jms.TextMessage](https://docs.oracle.com/javaee/7/api/javax/jms/TextMessage.html) osztályt használtam, amelyben csak String típusú üzeneteket lehet küldeni az üzenetsorra. Az algoritmus működése során összetett üzenetek kerülnek összeállításra és értelmezésre. Ezeknek a könnyebb átláthatósága érdekében, az adatstruktúrák, valamint a tömbök reprezentálására is jobban alkalmasabb JSON formátumot használtam. Természetesen ezt csak úgy lehetett megoldani, hogy a kész JSON objektumunkat a küldő oldalon Stirng-é alakítjuk és így küldjük el, majd ezt a fogadó oldalán pedig átparszoljuk. A JSON-t a későbbiekben is fel tudtam használni a weboldal és az algoritmus modell rétege közti információ cserére is. A projekthez tartozó **pom.xml**-ben a JSON-Simple nevű függőség van megadva, ami egy JSON könyvtárat biztosít a Java nyelvhez.

### 3.4.6. Vis.js

A Vis.js egy **Javascript** könyvtár, melynek segítségével a weboldalon lehetőség van különböző vizualizációkat dinamikusan végrehajtani. A könyvtárral lehetősé van arra, hogy a szerver felől érkező akár nagy mennyiségű adatokat a megfelelő formátumra hozás után megjelenítésre kerüljenek a weboldalon, illetve a későbbiekben ezt a vizualizációt könnyen lehet módosítani is. Erről a könyvtárról, valamint a használatáról segítséget kaphatunk a könyvtár hivatalos oldaláról: <http://visjs.org/>

## 3.5. A program szerkezete

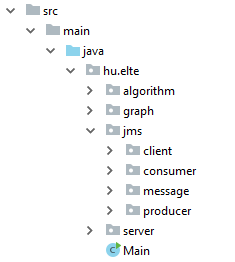
A Maven már a projekt megalkotásánál rendelkezésre bocsát egy strukturális mappa szerkezetet, ezzel teszi könnyebbé az eligazodást a projektben, valamint a benne található csomagszerkezetek között. Az ezekben a külön mappákban megtalálható csomagszerkezetek megfelelnek az általánosan elfogadott Java fejlesztési konvencióknak, azok jól elkülöníthető részekből állnak, amely megkönnyíti az eligazodást az osztályok között. A következő fejezetekben röviden bemutatásra, és jellemzésre kerülnek a projekt csomagszerkezetei. A bennük található osztályok részletes kifejtésére, működésük leírására a későbbi fejezetekben kerül sor.



3.1. ábra. Egy standard Maven által generált mappaszerkezet.

### 3.5.1. A program csomagszerkezete

A projekt jól áttekinthetősége érdekében ebben a fejezetben egy rövid leírásban kerülnek bemutatásra a projekt csomagszerkezetei, amely az **src/main/java** könyvtárban találhatók:

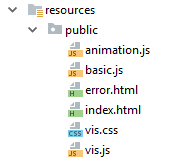


3.2. ábra. A projekt csomagszerkezete.

* **hu.elte**: A **hu.elte** a központi csomag melyből újabb csomagok vezethetők le, ebben egyetlen osztály a **Main** található, mely a későbbiekben kerül részletezésre, de ahogy a nevéből is adódik ez a program futásának a belépési pontja.
* **algorithm**: Az **hu.elte.algorithm** csomag a program szempontjából talán a legfontosabb csomag. A benne található osztály végzi a gráf algoritmus vezérlését. Az algoritmus futása során folyamatos összeköttetésben áll a Spark szerverrel, mely számára adatokat biztosít, valamint onnan az algoritmus működési beállításaihoz szükséges adatokat kapja meg.
* **graph**: A **hu.elte.graph** csomagban lévő osztályok végzik a gráf beolvasását, valamint az összes a gráfon belül jól elkülöníthető logikai egységeket, a csúcsokat, az éleket, az utakat, valamint magát a gráfot is.
* **jms**: A **hu.elte.jms** csomagban több újabb csomag is található, melyeknek a szükségességét a jól áttekinthetőség, valamint a funkcionalitásbeli különbségek indokolják. Az ezeken belül megtalálható osztályok a központi algoritmus számára biztosítják a gráf csúcsai közti kommunikáció alapját a már korábban említett **Java Message Service** segítségével az **Apache ActiveMQ** szolgáltatásán keresztül.
* **jms.client**: A **hu.elte.jms.client** csomagban található az az osztály, amely egy kliens objektumot fog tartalmazni, amelyben összpontosulni fog az üzenetsorokkal kapcsolatos összes felmerülő logika. A kereső algoritmusban minden csúcspont rendelkezni fog egy ilyen kliens objektummal.
* **jms.consumer**: A **hu.elte.jms.consumer** csomagban található az az osztály, amely az üzenetsoron történő kommunikáció fogadó objektumának a forráskódját fogja tartalmazni.
* **jms.message**: A **hu.elte.jms.message** csomagban található az az osztály, amelyben az üzeneteknek a keretét fogja adni.
* **jms.producer**: A **hu.elte.jms.producer** csomagban található az az osztály, amely az üzenetsoron történő kommunikáció küldő objektumának a forráskódját fogja tartalmazni.
* **server**: A **hu.elte.server** csomagban van a **Spark Java** által biztosított szerver működését leíró osztály, melynek segítségével először a weblapot lehet megjeleníteni, később a felhasználó választása szerint a megfelelő típusú kereső algoritmust elindítani, valamint a két komponens közti adatáramlást is biztosítani.

### 3.5.2. Resources mappa tartalma

A projekt mappaszerkezetében megtalálható **src\main\resources** elérési útvonalon lévő mappát a Maven, a projekt számára szükséges erőforrások miatt hozta létre, így az ebben lévő **public** nevű mappába kerültek a **weblap**, az **algoritmus animációját** leíró Javascript fájlok forráskódjai. A biztonságos működés érdekében bemásolásra került a **Vis.js** könyvtára is. A fájlokhoz és a forráskódokhoz tartozó magyarázatra a későbbiekben kerül sor.



3.3. ábra. A resources mappa tartalma.

### 3.5.3. Input mappa tartalma

Az **input** mappában található meg az a **graph.txt** fájl melyben az a gráf került leírásra, amelyen szeretnénk az algoritmust futtatni. Ez kerül a forráskódban beolvasásra, majd már a kiszámított eredménnyel együtt kerül megjelenítésre a weboldalon. A program helyes működése érdekében a fájl nevét ne változtassuk meg, ugyanis ez az elérési útvonallal együtt be van égetve a forráskódban! A fájlt a következőképpen lehet feltölteni a gráf adataival:

* A gráf minden egyes élét meg kell adni azzal a két csúccsal együtt, melyet az él összeköt. Először a **két csúcs** **csúcscímkéjét** kell **szóközzel** **elválasztva** beírni. Utána **szóközzel** **elválasztva** kell a **él** **súlyát** beírni, mely **double**-ként lett megvalósítva, így az erre az értéktartományra vonatkozó megszorításokat be kell tartani!
* Minden egyes ilyen él reprezentációt **új sorba kell írni**.
* Megjegyzés: Továbbá ügyelni kell arra, hogy be kell tartani a szakdolgozat korábbi részeiben már tárgyalt a gráfra vonatkozó megszorításokat is, tehát a gráfnak **összefüggőnek** és **egyszerűnek** kell lennie. A program feltételezi, hogy az így leírt gráf megfelel az előbbieknek.

Ezekre a megszorításokra nagy figyelemmel kell lenni, melyeknek a megszegése a program nem definiált működéséhez vezethet!

### 3.5.4. Output mappa tartalma

Az **output** mappában található a **messages.txt**. Ez a fájl fogja tartalmazni az összes a minimális út kereséssel összefüggő részletet. Így az összes üzenetet, ami kiküldésre került, a keresés időpontját, a keresés útját, összköltségét, és végül a kezdőpontból kiinduló a gráf összes többi csúcsába vezető utat, amelyeket ha berajzolnánk egy feszítőfát kapnánk.

## 3.6. Magyarázat a forráskódhoz

Ez a fejezet részletesen be fogja mutatni a korábbiakban már ismertetett technológiák alkalmazásait, az ismertetett csomagokban található osztályok, és ezeken belüli metódusok részletes logikáját, és felhasználását. A következetesség szerint a kisebb logikai egységektől, az ezeket magában foglaló nagyobbak felé fog haladni az ismertetés. Vagyis először a gráf tárolására szolgáló osztályok fognak következni, utánuk az üzenetsorokhoz köthető osztályok, az algoritmus osztálya, a szerver osztálya, végül egy új fejezetben fognak következni a weboldal működését kiszolgáló forráskódok.

### 3.6.1. Edge osztály

A **hu.elte.graph** csomagban található osztály, ahogy a nevéből is adódik a gráf egy élét fogja megvalósítani. Látható, hogy ez az osztály csupán egy **double** típusú mezőt tartalmaz, amelyet a konstruktor is megkap. Ehhez a változóhoz tartozik még egy **toString()** és egy **getWeight()** metódus, melyben az előbbi az érték szöveges reprezentációját adja vissza, utóbbival pedig az él súlyát adja vissza. A fejlesztés során fontosnak tartottam, hogy a gráf szempontjából az ilyen kis egységeket is, amilyen egy él, külön osztályt hozzak létre, így az esetleges későbbi továbbfejlesztés során ebben az egy osztályban kell megváltoztatnunk az él súly típusát.

### 3.6.2. VertexRoute osztály

A hu.elte.graph csomagban található VertexRoute osztály valósítja meg a gráfban található utak reprezentációját, benne eltárolva a kiindulási és a végcsúcsot, a köztük lévő él súlyok összegét, valamint az esetlegesen bejárt köztes csúcsokat.

Az osztálynak négy privát adattagja van:

* StartVertex: Vertex típusú változó, az út kezdőcsúcsát jelöli.
* EndVertex: Vertex típusú változó, az út végcsúcsát jelöli.
* Distance: Double típusú változó az út költségét jelöli.
* Previous: Lista, amely Vertex típusú objektumokat tartalmaz. A minimális súlyú útnak a kezdő és a végpontja közti csúcsok fognak ebbe a listába kerülni.

### 3.6.3. Vertex osztály

A hu.elte.graph csomagban található a Vertex osztály, amely a gráf egy csúcsát fogja megvalósítani. A Vertex objektum konstruktora a gráf csúcsának a nevét várja paraméterként. Ennek az osztálynak, illetve a funkcióinak központi szerepe van a gráf algoritmus működése során. Az algoritmus az innen származó adatok alapján tudja elvégezni az üzenetsorokba történő publikációt, amit egy másik szintén Vertex típusú objektum fog megkapni, a benne lévő információk alapján pedig frissíteni önmagát. Fontos kihangsúlyozni, hogy végeredményként csak egy útvonal kerül szemléltetésre a weblapon, de mivel a gráf csúcsai, ahogy korábban már említésre került kliensekként fognak viselkedni, így a kezdőcsúcs rendelkezni fog az összes többi csúcsba vezető minimális úttal. A kezdőcsúcsból így egy minimális feszítőfát is ki fog hozni az algoritmus eredményként, amit a messages.txt-ben láthatunk.

Az osztálynak négy privát adattagja van:

* Name: A csúcs privát String típusú csúcscímkéje. A változóhoz tatozik egy getter metódus is.
* Active: A csúcsnak rendelkezni kell egy privát boolean típusú változóval, amely eredetileg false értékű. Az algoritmus ha az adott csúcshoz ér, így tudja először aktiválni az csúcsot (vagyis átállítani az értékét true-ra), amely azt jelenti, hogy a későbbiekben ennek a csúcsnak még információt kell közölnie. Ha már minden adatot közölt, az algoritmus visszaállítja ezt az értéket false-ra. A változóhoz tartozik egy getter, és setter metódus is.
* Neighbours: A csúcs szomszédjait tartalmazó privát lista, amely Vertex típusú objektumokat fog tartalmazni. A változóhoz tatozik egy getter metódus is.
* Routes: A csúcs útjait tartalmazó privát lista, amely VertexRoute típusú objektumokat fog tartalmazni. Ebbe a listába kerülnek bele először a csúcs szomszédjai is, és ez fog kibővülni a beérkező adatok alapján, melynek során az adott csúcshoz tartozó út elérése, valamint az út költsége is változhat. A változóhoz tartozik egy getter metódus is.

Legfontosabb metódusok bemutatása:

routesToMessage(): Egy String visszatérésű függvény, amelynek a feladata, hogy összeállítsa az adott csúcs által nyilvántartott utakat. Az üzenet összeállítása a következőképpen történik: Létrehozunk egy JSON objektumot, amivel később String-é alakítva vissza is térünk. Ennek a kulcsa "SEND\_ROUTES" lesz , értéke pedig egy JSON tömb lesz, amiben JSON objektumokat fogunk eltárolni. A belső JSON objektumhoz egy for ciklussal az adott csúcs által nyilvántartott utakon végig iterálunk, és a következő kulcs-érték párokat rendeljük hozzá:

* "key": Az út kezdő csúcsának a címkéje, egy szóköz kihagyás, és az út végpontjának a csúcs címkéje. Megjegyzés: Ebben az esetben a kezdő csúcs címkéje mindig az éppen aktuális gráf csúcsának a címkéje lesz, így ha, kapni fogunk egy üzenetet az első csúcscímke mindig az egyik velünk szomszédos,és egyben az üzenetet küldő csúcs címkéje lesz.
* "value": Az út összköltsége.
* "previous": Az esetleges köztes csúcsok listája.

processRoutes(String message): A csúcshoz tartozó kliensben, ha egy adott csúcshoz üzenet érkezik az üzenetsoron, ez a függvény fog meghívódni. Ennek az üzenetnek a szerkezete megegyezik az előbb bemutatott routesToMessage() függvény által létrehozandó üzenetével. Így az első lépés, hogy az így visszaalakításra kerülő JSON objektumokon egy while ciklus segítségével végig kell iterálni. A benne található adatok feldolgozása a következő módon fog történni:

A while ciklus első felében megtalálható egy elágazásban egy continue utasítás. Erre azért van szükség, mert egy üzenet csak a szomszédos csúcstól érkezhet, és ebben megtalálható a fogadó csúcsra vonatkozó üzenet is, amely már saját magának is meg van, hiszen ez már a gráf beolvasása során rendelkezésre állt.

Az üzenetcsere folyamán, mivel ott csak String-ek kerülnek ki küldésre, így az egyes referenciák elvesznek a gráf objektumaihoz. Valójában ez nem is lenne probléma az algoritmus működése szempontjából, azonban az erre a célra írt függvényekkel én ezeket a referenciákat lekérem, így az aktuális objektum tudja frissíteni az adattagjait valódi objektumokkal. Itt szeretném kihangsúlyozni, hogy ezáltal a jogosultságunk meg lenne ahhoz, hogy az így kapott külső objektumokon módosításokat hajtsunk végre, ez azonban semmi esetben sem történik meg, csupán a fejlesztés szempontjából tartottam fontosnak, hogy ha valaki szeretné részletesen látni, vagy debug-olni a projektet, annak könnyebb dolga legyen ezekkel a műveletekkel.

A következő objektumok kerülnek lekérésre a referenciavesztés miatt:

* receivedVertex: Értékét a getRouteByName(String name) függvényből kapja, amely metódus ellenőrzi, hogy az objektum útjai közt, már rendelkezünk-e erről a csúcsról adattal, ha igen visszatér ezzel a Vertex-el, ha nem null-t ad vissza.
* neighbour: Értékét a Graph.getVertexByName(String name) függvényből kapja, itt szintén az elveszett referencia miatt kell meghívni ezt a metódust. Ez a függvény végigmegy a gráf csúcsain, és visszaadja a keresett csúcscímkéhez tartozó csúcsot. Ez a gyakorlatban mindig az aktuális gráf egyik szomszédja lesz, ezért értéke sosem lesz null.
* neighbourDistance: Értékét a getRouteWeightByName(String name) függvényből kapja meg. Mivel az előbb említésre került, hogy adatokat csak a szomszédoktól fogad egy csúcs, így le kell kérni az ebbe a szomszédos csúcsba vezető él költségét, amit a referenciavesztés miatt ezzel a segédfüggvénnyel lehet megtenni.
* previousList: Egy Vertex objektumokat tároló lista, értékét a getRoutePreviousByName(String name) függvény segítségével kapja meg. Erre azért van szükség, mert előfordulhat olyan eset, amikor ezt a bizonyos szomszédos csúcsot egy másik szomszédos csúcson keresztül vezető úton kisebb költséggel érjük el, így el kell tárolnunk ezeket a korábbi csúcsokat is.

A függvény futása során a következő lépésben a receivedVertex-et fogjuk megvizsgálni:

* Ha ez az érték nem null, akkor a getRouteIndexByName(String name) függvénnyel kell kikeresni az indexét a csúcs által tárolt routes listájából, hogy lekérje a nyilvántartott él költséget hozzá, amely a previous nevet kapta. Ez után egy elágazásban meg kell vizsgálni, hogy az üzenetben megkapott, valamint még a szomszédhoz vezető él költsége együttesen kisebb lesz-e a previous változó értékénél? Ha nem, akkor nem kell semmit tenni, hiszen az a minimális költségű út az adott pillanatban. Ha igen, akkor meg kell hívni a setRouteDistance(int indexOf, Vertex neighbour, Double distance, String[] previousVertexArray, LinkedList<Vertex> previousVertex) függvényt. Ebben a már lekért index alapján történik meg az útvonalra vonatkozó adatok frissítése.

1. Beállításra kerül az út költsége.
2. Kitörölésre kerül az esetlegesen megadott korábbi bejárási útvonal.
3. Az új útvonalhoz hozzáadásra kerül a küldő csúcs.
4. Az új útvonalhoz hozzáadásra kerül a korábban bemutatott previousList a nulladik indextől kezdődően.
5. Az új útvonalhoz hozzáadásra kerül az üzenetben megkapott köztes csúcs.

* Ha az érték null, akkor létre kell hozni egy új VertexRoute-ot. Az útvonal köztes csúcsaihoz először hozzá kell adni azt a szomszédot, ahonnan az üzenetet kaptuk, majd az üzenetben megkapott lehetséges előzőleges csúcsokat is hozzá kell venni. Az így megkapott utat pedig fel kell venni az üzenetet fogadó csúcs útjai közé.

### 3.6.4. Graph osztály

Ez az osztály a hu.elte.graph csomagban található. Ez fog a gráf csúcsainak a tárolására szolgálni. A gráf csúcsai egy osztályszintű privát Vertex objektumokból álló vertices nevű listában kerülnek tárolásra, amelyhez egy getter metódus fog még tartozni. Az osztály rendelkezik egy saját konstruktorral is, amely az előbb említett listát példányosítja. Az osztály a következő két további publikus metódussal rendelkezik:

* void addEdge(Edge edge, Vertex vertexOne, Vertex vertexTwo): A függvény a gráf beolvasásánál kerül meghívásra, amely a GraphReader osztályban található meg. A metódus végzi el egyrészt az adott csúcsok felvételét a központi gráfba, valamint az ellenkező csúcsot felveszi a szomszédjai közé, és megkonstruálja az abba vezető utat is, amit szintén eltárol.
* static Vertex getVertexByName(String name): Ez a metódus kerül meghívásra a Vertex osztályban, aminek segítségével az eredeti Vertex objektumot le lehet kérni a vertices listájából.

### 3.6.5. GraphReader osztály

Ez az osztály a hu.elte.graph csomagban található. Ez szolgál arra, hogy a graph.txt-ből beolvassa az adatokat, és feltöltsön egy Graph objektumot. Két publikus metódusa van:

* static Graph graphFromFile(String fileName): A metódus egyszerre több feladatot is elvégez. A segítségével a projekttel egy szinten található mappából beolvassa a graph.txt-t, és benne sorról sorra haladva meghívja a Graph osztály addEdge(Edge e, Vertex v1, Vertex v2) metódusát, aminek segítségével a beolvasás végére rendelkezésre áll az összes szükséges kiinduló alap a gráf algoritmus futásához. Összeállításra kerül egy JSON objektum is, tulajdonképpen itt magát a beérkező .txt állományt alakítjuk át JSON szerkezetűvé. Ez azért szükséges, hogy később a szerver szolgáltatni tudja ezt az adatot a weboldal vizualizációja felé.
* static JSONObject getObject(): Szimpla getter metódus, az előbb említett JSON objektumot fogja visszaadni.

### 3.6.6. AlgorithmMessage osztály

Az osztály a hu.elte.jms.message csomagban található. Az osztálynak csak az a szerepe, hogy keretet adjon a majd kiküldendő üzenet számára. Az üzenet kerete egy JSON objektum lesz, amely a következő kulcs-érték párokkal fog rendelkezni.

* "producerName": Az üzenet küldőjének a neve. A gyakorlatban ez a küldő gráf csúcscímkéjét fogja jelölni.
* "message": Az üzenetnek a szövege, amely a gyakorlatban a csúcs által ismert utakról fog információt tartalmazni.
* "consumerName": Az üzenet fogadójának a neve. A gyakorlatban ez az üzenetet fogadó gráf csúcscímkéje lesz.

### 3.6.7. Producer osztály

Az osztály a hu.elte.jms.producer csomagban található. Az osztály feladata, hogy kacsolódjon a korábban már említett Apache ActiveMQ kiszolgálójához, majd ezt követően elküldi a megfelelő üzenetsorba az adatokat, végül pedig lezárja a kapcsolatot. Tehát a kapcsolat csak addig van fenntartva, ameddig az üzeneteket publikálja az algoritmus. Az osztálynak van egy publikus konstruktora is, amely egy String-et vár paraméterül, ez az érték lesz, majd a Client osztályban beállítva a csúcscímkéjére, ez lesz a küldő neve. Az osztálynak öt privát adattagja van, amelyek a következőek:

* connection: Egy [javax.jms.Connection](https://docs.oracle.com/javaee/7/api/javax/jms/Connection.html) típusú változó, amely a kapcsolat létrehozását biztosítja.
* session: Egy [javax.jms.Session](https://docs.oracle.com/javaee/7/api/javax/jms/Session.html) típusú változó, amely a munkamenet környezetét biztosítja az üzenetek felhasználásához és előállításához.
* destination: Egy [javax.jms.Destination](https://docs.oracle.com/javaee/7/api/javax/jms/Destination.html) típusú változó, amely az üzenetek eljuttatását biztosítja.
* messageProducer: Egy [javax.jms.MessageProducer](https://docs.oracle.com/javaee/7/api/javax/jms/MessageProducer.html) típusú változó, amely üzeneteket küld egy megadott rendeltetési helyre.
* producerName: Egy String típusú változó, amely az üzenetküldő nevét fogja tartalmazni.

Az osztály fontosabb metódusai:

* create(String queueName): Egy privát visszatérés nélküli metódus, paraméterként az üzenetsor nevét várja. Feladata létrehozni a kapcsolatot, valamint az osztály privát adattagjait inicializálni.
* close(): Egy privát visszatérés nélküli metódus, amely az osztályhoz tartozó erőforrásokat zárja le.
* send(String message, String queueName): Egy publikus visszatérés nélküli metódus, amely üzenetet publikál az üzenetsorba. Az első paramétere lesz az üzenet, amit küldésre fog kerülni. A gyakorlatban ez a Vertex osztály routesToMessage() metódusa által biztosított üzenet lesz. A második paraméter pedig az a üzenetsor neve lesz, ami a gyakorlatban annak a csúcsnak a címkéje lesz, ahova az üzenet küldésre kerül. A metódusban az üzenetet az előbb bemutatott AlgorithmMessage osztály metódusa segítségével kerül becsomagolásra, majd létrehozásra kerül a kapcsolat (create() metódus meghívása). Az üzenetet a [javax.jms.TextMessage](https://docs.oracle.com/javaee/7/api/javax/jms/TextMessage.html)-ként kerül publikálásra az üzenetsorba, a kapcsolatot pedig ez után bezárásra is kerül (close() metódus meghívása).

### 3.6.8. Consumer osztály

Az osztály a hu.elte.jms.consumer csomagban található. Az osztály feladata, hogy az a Apache ActiveMQ kiszolgálójával kapcsolódjon, amelyet egészen a gráf algoritmus végéig fenn is tart. Biztosít egy javax.jms.MessageConsumer típusú objektumot, amelyre a Client osztályban be lehet állítani egy [javax.jms.MessageListener](https://docs.oracle.com/javaee/7/api/javax/jms/MessageListener.html) objektumot, amellyel aszinkron módon lehet üzeneteket fogadni. Az osztály egy publikus konstruktorral rendelkezik, amelyik egy String-et vár paraméterül, amely a Client osztályban kerül beállításra, Ez lesz a fogadó neve.

Az osztálynak a következő három privát és egy publikus adattagjai vannak:

* connection: Egy privát [javax.jms.Connection](https://docs.oracle.com/javaee/7/api/javax/jms/Connection.html) típusú változó, amely a kapcsolat létrehozását biztosítja.
* session: Egy privát [javax.jms.Session](https://docs.oracle.com/javaee/7/api/javax/jms/Session.html) típusú változó, amely a munkamenet környezetét biztosítja az üzenetek felhasználásához és előállításához.
* consumerName: Egy privát String típusú változó, amely a fogadó nevét fogja jelölni.
* messageConsumer: Egy publikus [javax.jms.MessageConsumer](https://docs.oracle.com/javaee/7/api/javax/jms/MessageConsumer.html) típusú változó, amely üzeneteket fogad az üzenetsorról. Ehhez az adattaghoz tartozik egy getter metódus is.

Az osztály a következő metódusokkal rendelkezik:

* create(String queueName): Egy visszatérés nélküli publikus metódus, amely paraméterként az üzenetsor nevét kapja meg. Feladata létrehozni és fenntartani a kapcsolatot a kiszolgálóval, inicializálni az osztály adattagjait.
* close(): Egy publikus visszatérés nélküli metódus, amely az osztályhoz tartozó erőforrásokat zárja le.

### 3.6.9. Client osztály

Az osztály a hu.elte.client csomagban található. Az osztály feladata, hogy minden egyes Vertex típusú objektumhoz hozzárendeljen egy Consumer, és egy Producer objektumot, amelyek segítségével az üzenetküldés meg tud valósulni. A gráf csúcsaiból így egy különálló fogadó klienst és egy küldő klienst lehet csinálni, aminek az adattagjait később az algoritmus el tudja érni, és a neki szükséges metódusokat meg tudja hívni az útkeresés különböző szakaszaiban. Az osztály továbbá megvalósítja a [javax.jms.MessageListener](https://docs.oracle.com/javaee/7/api/javax/jms/MessageListener.html) osztályt. Erre azért van szükség, hogy amint egy üzenet bekerül az üzenetsorba a fogadó fel tudja dolgozni. Az osztály egy publikus konstruktorral rendelkezik, amely egy Vertex típusú objektumot vár paraméterként.

Az osztály a következő adattagokkal rendelkezik:

* vertex: Egy privát Vertex típusú változó, amely a gráf egy csúcsát fogja tartalmazni. Ehhez az adattaghoz tartozik egy getter metódus is.
* consumer: Egy privát Consumer típusú változó, ez lesz a fogadó kliens funkcionalitására irányuló változó. Ehhez az adattaghoz tartozik egy getter metódus is.
* producer: Egy privát Producer típusú változó, ez lesz a küldő kliens funkcionalitására irányuló változó. Ehhez az adattaghoz tartozik egy getter metódus is.

Az osztály a következő saját metódusokkal rendelkezik:

* createQueue(): Egy void visszatérésű értékű függvény, amely először létrehozza az üzenetsorhoz tartozó kapcsolatot a fogadó számára, majd beállítja a MessageListener-t az osztályra.
* fullMessageProcessing(JSONObject message): Egy visszatérés nélküli privát metódus, amely egy JSON objektumot kap paraméterként. Ez a metódus szolgál arra, hogy az üzenet kerete "lefejtésre" kerüljön az értékes út információkról, így a belső üzenet kulcs-érték párja alapján tudunk tovább dolgozni a ténylegesen hasznos adatokkal. Ez a kulcs-érték pár tovább küldésre kerül a partMessageProcessing(String key, String value) számára.
* partMessageProcessing(String key, String value): Egy visszatérés nélküli privát metódus. Az első paramétere a kulcs lesz, amely itt mindig csak "SEND\_ROUTES" értékű lesz, amivel az üzenet érkezett. Értéke pedig a tényleges adatok, amiket tovább kell küldeni az adott csúcs processRoutes() metódusához. Ez a függvény, és vele együtt az előző kialakítása is alkalmas arra, hogy továbbfejlesztés esetén többfajta utasítás alapján is végezzen számításokat.

A MessageListener osztály megvalósításához szükséges metódus:

* onMessage(Message message): Egy visszatérési érték nélküli metódus, amely a synchronized kulcsszóval is rendelkezik, hogy az üzenet ne kerüljön egyszerre írásra és olvasásra. A paraméterként megkapott [javax.jms.Message](https://docs.oracle.com/javaee/7/api/javax/jms/Message.html) objektumot vissza kell alakítani JSON objektummá, hogy tovább lehessen küldeni a fullMessageProcessing() számára. Továbbá ennek a JSON-nek a String-é alakított változatát hozzá kell adni az Algorithm osztály messages listájához, hogy a későbbiekben az összes üzenetet ki lehessen írni az output mappa messages.txt-be.

### 3.6.10. Algorithm osztály

A hu.elte.algorithm csomagban található Algorithm osztály az algoritmus vezérlését irányítja, a szerver számára szolgáltatja a JSON formátumú adatokat, valamint az adatokat egy fájlba is kiírja. Az osztály a következő adattagokkal rendelkezik:

* graph: Egy privát osztályszintű Graph objektum, amelynek a példányát a konstruktorban kerül megadásra.
* clients: Egy privát Client objektumokat tároló lista.
* startVertex: Egy privát Vertex típusú objektum, amely a keresett út kezdő pontja lesz, hozzá tartozik egy setter metódus is.
* endVertex: Egy privát Vertex típusú objektum, amely a keresett út végpontját fogja tartalmazni, hozzá tartozik egy setter metódus is.
* messages: Egy privát String-ekből álló lista, amely a Client osztályban fog értékeket kapni, és ezek lesznek kiíratva egy fájlba az algoritmus végén. Ehhez tartozik egy getter metódus is.
* mainQueue: Egy privát Vertex objektumokat tartalmazó lista. A gráf felderítése során lesznek ebbe pakolva a gráf csúcsai, és amint inaktívak lesznek a csúcsok ki fognak törlődni a listából.
* minRoute: Egy privát Vertex objektumokat tartalmazó lista, ez fogja a bejárási sorrendben tartalmazni a keresett legrövidebb útnak a csúcs címkéit.
* jsonGraphByNode: Egy privát osztályszintű JSON objektum, amelybe a gráf feltérképezése során fognak a gráf csúcsai sorrendben belekerülni. Ez az esetleges weblap animáció miatt szükséges. Ehhez az adattaghoz kapcsolódik egy getter metódus is.
* jsonGraphByNodeForColorChange: Egy privát osztályszintű JSON objektum, amelybe az algoritmusnak a második szakaszában a gráf csúcsainak inaktiválásakor fognak belekerülni az egyes csúcscímkék, hogy az esetleges animáció esetén a weblap le tudja kérni a publikált adatot. Ehhez az adattaghoz kapcsolódik egy getter metódus is.
* jsonMinRouteForAnimation: Egy privát osztályszintű JSON objektum, amelybe az algoritmus futása folyamán az éppen minimális utat rakja, amit a weblap animáció meg tud jeleníteni. Ehhez az adattaghoz kapcsolódik egy getter metódus is.
* allNodes: Egy privát osztályszintű JSON objektum, amelybe a gráf csúcsai fognak kerülni, ez fog megjelenni a weblap bal felső sarkában.
* finalDistanceCost: Egy privát Double típusú objektum, amely a keresett legrövidebb út összköltségét fogja jelenteni.
* input: Egy osztályszintű végleges String, ami az input fájl helyét és nevét tartalmazza, ami a input/graph.txt elérési út lesz.

Az osztály a következő saját metódusokkal rendelkezik:

* readGraph(): Egy privát metódus, amely a gráfnak a fájlból történő beolvasását végzi.
* createClients(): Egy privát visszatérési érték nélküli metódus, amelyben létrehozásra kerülnek a gráf csúcsai alapján a Client objektumok.
* createQueues(): Egy privát visszatérési érték nélküli metódus, amely a Client objektumokhoz létrehozza a saját üzenetsorukat.
* allNodes(): Egy publikus osztályszintű metódus, amely egy JSON objektummal fog visszatérni. Az objektum kulcsa az "allNodes" lesz, az értékei pedig sorban a gráf csúcsai lesznek. Ez a kulcs-érték pár az allNodes változóban kerül tárolásra.
* jsonRoute(): Egy publikus osztályszintű metódus, amely egy JSON objektummal fog visszatérni. Ennek az objektumnak a kulcsa a "route" lesz, értékei pedig a már elkészült legrövidebb út bejárását fogják tartalmazni egymás mellé téve két-két csúcscímkét.
* jsonGraphSize(): Egy publikus osztályszintű metódus, amely egy JSON objektummal fog visszatérni. Ennek az objektumnak a kulcsa a "size" lesz, értéke pedig a gráf csúcsainak száma lesz.
* getClient(Vertex v): Egy privát Client visszatérési értékű metódus, amely a paraméterként megkapott objektumhoz, visszaadja a hozzá tartozó Client objektumot.
* closeClients(): Egy privát visszatérési érték nélküli metódus, amely az algoritmus legvégén a Client objektumokhoz tartozó Consumer objektumokat zárja be.
* getMinRouteWithWeight(): Egy publikus String visszatérési értékű metódus, amely a keresett legrövidebb utat, és a hozzá tartozó költséget adja vissza.
* writeAllMessagesToFile(): Egy privát visszatérési érték nélküli metódus, amely az output/messages.txt fájlba írja ki az algoritmus keresése folyamán keletkezett összes üzenetet, a keresett úthoz tartozó összes keletkezett adattal együtt.
* sleep(JSONObject object, boolean isSleeping, int time): Egy privát visszatérés nélküli metódus. Arra szolgál, hogy a felhasználó későbbi döntése alapján a weblapon, ha az "Animation" gombot választja - vagyis szeretne animációt - az algoritmust meg kell lassítani, hogy az általa közölt adat valós időben meg tudjon jelenni a weblapon. Ebben az esetben a második paraméter mindig true értékű lesz, az első paraméter pedig az esetlegesen az előzőekben a JSON változókba került elévült adatot törli ki. A harmadik paraméter pedig a tényleges várakoztatási értéke lesz az algoritmusnak.
* graphDiscovery(boolean isSleeping): Egy privát visszatérés nélküli metódus, amely egy logikai értéket vár paraméterül. Ez az előbbiekben tárgyalt weblap animáció miatt tudja meglassítani az algoritmust. A metódusban először létre kell hozni egy listát. Ez a "queue" nevű lista csak átmeneti szerepet fog betölteni. Ehhez a listához, valamint a "mainQueue" listához először bele kell rakni a kezdőcsúcsot. Ez a csúcs aktiválásra is kerül, vagyis az "active" mezőjét true értéket kapja meg. A következőkben a külső while ciklus addig fog futni, ameddig tartalmaz elemet. Egy-egy lefutás végén az első elemet törölni kell a "queue" listából. A ciklus elején le kell kérni az első elemet, majd ennek a szomszédjain egy for ciklussal végig kell menni. Ha a szomszédok még nem aktívak, és van szomszédjuk, akkor aktiválni kell őket, és bele kell rakni őket mindkét előbb írt listába.
* mapNeighbours(): Egy privát visszatérés nélküli metódus. Egy boolean paramétere van, amely az animációkor lassítja meg az algoritmust. Az előbb feltöltött "mainQueue" listán megy végig egy while ciklus, ameddig abban vannak elemek. Itt most az utolsó elemet kérjük le, és a hozzá tartozó Client objektumot. Ennek a csúcsnak a szomszédjain végig kell menni. Itt le kell kérni az eredeti szomszéd objektumot, hogy megtudjuk valóban aktív-e? Ha igen, akkor publikálni kell az adott szomszédnak az összes út információt, amivel a csúcs éppen rendelkezik. A következő elágazásnak a gyakorlati szerepe csak az animációnál van, hiszen ebben kell az éppen minimális utat a szerver számára elkészíteni. Kilépve a for ciklusból a lekért csúcs objektumot false-ra kell állítani, hiszen az már nem tud kinek adatot biztosítani, valamint a mainQueue listából is ki kell törölni.
* computeAlgorithm(String start, String end, boolean isSleeping): Egy publikus visszatérés nélküli metódus, amely az előbb felsorolt metódusokat fogja vezérelni. Első lépésként a klienseket, és a hozzájuk tartozó üzenetsorokat fogja létrehozni, illetve az első két paraméterként megkapott csúcsot fogja beállítani. Ezt követően kerül meghívásra az előbbi két metódus, melyeknek tovább adja a harmadik paramétert. Ezek lefutását követően összeállítja a minimális utat a költséggel együtt. Végül lezárja az összes klienst, és a keletkezett adatokat kiíró metódust hívja meg.

### 3.6.11. Server osztály

A hu.elte.server csomagban található Server osztály biztosítja a szervert az algoritmus működéséhez, valamint annak a szemléltetésére szolgáló weblaphoz. Az osztály egy üres konstruktorral rendelkezik Az osztály a következő adattagokkal rendelkezik:

* start: Egy privát String típusú változó, amelybe fog kerülni a keresett legrövidebb út kezdő pontja.
* end: Egy privát String típusú változó, amelybe fog kerülni a keresett legrövidebb út végpontja.

Az osztály a következő metódusokkal rendelkezik:

* startAlgorithm(boolean sleep): Egy privát visszatérési érték nélküli metódus, amely egy logikai értéket vár paraméterül. Ez a metódus fogja minden egyes alkalommal megkonstruálni egy Algorithm példányt és a megfelelő adatokkal meghívja ennek a példánynak a computeAlgorithm(start,end,sleep) metódusát.
* show(): Egy publikus visszatérés nélküli metódus, amely előbb beállítja a weblap megjelenítéséhez szükséges statikus fájlok helyét, ami a resources/public mappa lesz. Ez után megadásra kerülnek az elérési útvonalak. A szerver a localhost://4567-es porton fog futni. A GET és POST kérésekkel pedig kiszolgálja a weblapot.

A következőkben a fontosabb GET, és POST kérések kerülnek részletezésre:

* GET "/": A kérés a kezdő oldalra vezet minket, vagyis a public mappában lévő index.html fog betöltődni.
* GET "/route": A kérés a legrövidebb úthoz tartozó élek kezdő illetve végpontját fogja tartalmazni, amit az Algorithm osztályból kerül lekérésre, és egy JSON objektum lesz.
* GET "/grap": A kérés az graph.txt-ben található sorokat fogja tartalmazni, amit a GraphReader osztályból kerül lekérésre egy JSON objektumban.
* GET "/size": A kérés a gráf csúcsainak számát fogja megadni egy JSON objektumban, ezt is az Algorithm osztályból kerül lekérésre.
* GET "/graphByNode": A kérés a gráf csúcsait fogja tartalmazni egyesével, ahogy az Algorithm osztály graphDiscovery() metódusa bejárja a gráfot. Ezeket egy JSON objektumban fogja megkapni az Algorithm osztályból.
* GET "/graphByNodeForColorChange": A kérés a gráf csúcsait fogja tartalmazni egyesével, ahogy az Algorithm osztály mapNeighbours() metódusa lezárja a gráf csúcsait. Ezeket egy JSON objektumban fogja megkapni az Algorithm osztályból.
* GET "/routeStepByStep": A kérés az adott pillanatban legrövidebb utat fogja tartalmazni, ahogy az Algorithm osztály mapNeighbours() metódusa összesíti a beérkezett adatokat. Ezeket egy JSON objektumban fogja megkapni az Algorithm osztályból.
* GET "/allNodes": A kérés a gráf összes csúcsát fogja tartalmazni. Ezeket egy JSON objektumban fogja megkapni az Algorithm osztályból.
* POST "/animation": A kérés során egy üzenetet fog megkapni a weblaptól, ami egy ebben az esetben egy "true" érték lesz, aminek segítségével meghívásra kerül a startAlgorithm() metódus.
* POST "/basic": A kérés során egy üzenetet fog kapni a weblaptól, ami egy ebben az esetben egy "false" érték lesz, aminek segítségével meghívásra kerül a startAlgorithm() metódust.
* POST "/start": A kérés során egy üzenetet fog kapni a weblaptól, ami tartalmazni fogja a keresett minimális útnak a kezdő, és a végpontját. Ezek pedig beállításra kerülnek a start, end változók értékeire.
* POST "/stop": A kérés során a Spark szerver kerül leállításra.

### 3.6.12. Main osztály

A hu.elte csomagban található Main osztályban található main() metódus fogja elindítani az alkalmazást. Ebben a metódusban először is a kiírásra szolgáló messages.txt-ból kell az estelegesen a korábbi futtatásnál benne maradt adatokat kitörölni. Ez követően a Server osztály show() metódusával indul az alkalmazás.

## 3.7. A weboldal forráskódjai

Ebben a fejezetben kerülnek bemutatásra azok a Html és Javascript fájlok, amelyek az weboldalt, és annak a dinamikáját írják le. Ezek a resources/public mappában vannak. Az itt megtalálható vis.js és vis.css fájlok egy Javascript könyvtár, ami a vizualizációra szolgál, ezek nem saját fájlok.

### 3.7.1. Kezdőlap

A kezdőlap forráskódja az index.html. A kezdőoldal két részre tagolható, a felső részen találhatók, azok a gombok és mezők, amikkel az algoritmust, és a vizualizációt lehet működtetni. Ezek alatt található egy szürke rész, ahol a vizualizáció fog történni.

A kezdőoldal bal felső sarkában látható kiírva a gráf csúcs címkéi szóközzel elválasztva, ezek közül kell egyet-egyet beírni a "Start node:" és "End node:" nevű mezőkbe és a "Submit" gombbal lehet elmenteni a két értéket. Ha ugyanaz a csúcs címke kerül mindkét helyre, vagy az egyik hely üres marad hiba üzenetet fog a program megjeleníteni, megfelelő adatok esetén nem kapunk semmilyen válasz reakciót. A "Submit" gomb működése a következő: Egy "form" tagben van megvalósítva, megnyomásakor először ellenőrződik az input a validateSubmit() Javascript függvény segítségével, ha ez rendben, akkor egy POST metódussal a localhost:4567/start elérésre kerül át a két gráf csúcscímke, ahol a már ismert módon feldolgozásra kerül ez az adat.

A felső részen található még három gomb. A gombok a következők: "Basic", "Animation", "Stop Server". A gombok egy "form" tagben vannak megadva és azon belül egy "button" tagben. Ezek részletes működése a következő:

* "Basic" gomb: A gomb megnyomásakor a basicFunc() nevű Javascript függvény fog meghívódni. Ez ellenőrzi, hogy megtörtént-e már a két csúcs felvétele, ha nem a program hibaüzenetet fog kiírni és vissza is fog térni. Ha megtörtént az input felvétele a program mindhárom gombhoz kikapcsolja a kattintás funkciót, így nem lehet a háttérben futó algoritmustól érkező adatokat rosszul megjeleníteni. Ez a funkció zárolás ebben az esetben szabad szemmel nem látható, mivel az a lefutás nagyon gyors lesz, ha kis méretű a gráf. Ezeket a funkciókat a basic.js-ben kerülnek visszaállításra. A függvény egy POST metódust küld a localhost:4567/basic elérésre, amiben egy "false" érték lesz ezzel jelezve az algoritmus számára, hogy nem kell meglassítani az algoritmus futását. Ezt követően betöltődik a basic.js nevű Javascript fájlt, ami a következőkben lesz elmagyarázva.
* "Animation" gomb: A gomb megnyomásakor az animationFunc() nevű Javascript függvény fog meghívódni. Ez ellenőrzi, hogy megtörtént-e már a két gráf csúcs felvétele, ha nem a program hibaüzenetet fog kiírni és vissza is fog térni. Ha megtörtént az input felvétele a program mindhárom gombhoz kikapcsolja a kattintás funkciót, így nem lehet a háttérben futó algoritmustól érkező adatokat rosszul megjeleníteni. Ezeket a funkciókat az animation.js végén kerülnek visszaállításra. A függvény egy POST metódust küld a localhost:4567/animation elérésre, amiben egy "true" érték lesz ezzel jelezve az algoritmus számára, hogy meg kell lassítani a futást, hogy az algoritmus valós időben szemléltetve legyen. Ezt követően betöltődik az animation.js nevű Javascript fájlt, ami a következőkben lesz elmagyarázva.
* "Stop Server" gomb: A gomb megnyomásakor a stopFunc() nevű javascript függvény kerül meghívásra. Ebben egy POST metódus kerül küldésre a localhost:4567/stop elérésre, és az oldal átirányításra is kerül az error.html-re. Az error.html-ben csak egy hibaüzenet található. Ezt a funkciót mindig meg kell hívni, ha be akarjuk zárni az alkalmazást, így nem marad a háttérben futó folyamat.

### 3.7.2. Basic.js

A basic.js egy Javascript fájl, ami az index.html-ben található basicFunc() függvény hatására hívódik meg. Először lekérésre kerül az algoritmus által a szerver localhost:4567/graph elérésén tárolt input gráf, ami feldolgozásra, és tárolásra is kerül, külön listákban a gráf csúcsai, és az élei. Le kell kérni továbbá a szerver localhost:4567/route elérésén található legrövidebb utat is. Ezt követően már csak a legrövidebb út mentén található éleket kell pirosra állítani. Az így kapott adatokat pedig csak át kell adni a vis.js számára, ami ki fogja rajzolni a gráfot.

### 3.7.3. Animation.js

Az animation.js egy Javascript fájl, ami az index.html-ben található animationFunc() függvény hatására hívódik meg. A működése során üzenetet küld az Algorithm osztály számára, hogy lassítsa meg a saját működését, valamint az adatokat is itt egyesével fogja megkapni a Javascript és nem ömlesztve. Itt szeretném megjegyezni, hogy az ídőbeni várakoztatás 2 másodperc minden Algorithm osztály beli sleep() hívásnál, és az animation.js-ben is, kivéve az Algorithm osztályban található legelső sleep() hívást itt ez az idő 3 másodperc. Az Algorithm osztály így technikalilag 1 másodperccel előrébb jár, így előzzük meg, hogy az adatáramlás pontos legyen. Az animation.js-ben a várakozástatás a setInterval() metódusokkal került implementálásra.

A következőkben a Javascript forráskódja kerül szemléltetésre. Az első lépésként le kell kérnünk a gráf csúcsainak a számát a localhost:4567/size elérésről. Ez azért fontos, hogy tudjuk, mikor kell megszakítani a setInterval metódust, hiszen ekkor a szemléltetés egy későbbi szakaszába kell lépnünk. A belépési pont a graph() függvény lesz. Ebben a függvényben két másodperces időközönként lekérjük a localhost:4567/graphByNode-ról az Algorithm osztály által küldött gráf csúcsokat, és beállítjuk a megfelelő értéküket a szemléltetéshez. A színük kezdetben zöld lesz. Ez a folyamat addig fog ismétlődni, amíg a gráf mérete el nem éri az előbb lekérésre került méretet. Ebben a setInterval-os belső függvényben kerülnek berajzolásra az élek is az addEdges() függvény segítségével. Ez a függvény kérést küld a localhost:4567/graph elérésre, ahonnan az összes gráfra vonatkozó adaton végig megy, és mindig ellenőrzi, hogy van-e már olyan csúcs, amit össze lehet kötni, ha van akkor ezt a műveletet végre is hajtja. Ahogy az előbbiekben leírásra került, ha a gráfunkban már szerepel az összes csúcs, akkor a lezárjuk a ezt a setInterval() metódust és tovább lépünk a colorChange() nevű függvényre.

A colorChange() függvény az algoritmus második szakaszát szemlélteti. Ez a függvény kéréseket küld szintén két másodpercenként a localhost:4567/graphByNodeForColorChange elérésre, hogy az ott érkező csúcsokat pirosra tudja színezni. Ebben a belső setInterval() függvényben kerül lekérdezésre a localhost:4567/routeStepByStep, vagyis, amennyiben a kezdő csúcshoz beérkezik egy éppen minimálisnak tekinthető útról a végpontba, annak az éleit az edgeChange(param) függvény fogja pirosra festeni. A colorChange() függvény szintén addig fut, amíg a gráf összes csúcsa piros nem lesz. Ekkor a setInterval() függvényt leállítjuk és, az ainmáció végeztével a weblapon megtalálható három gomb is újból kattinthatóvá válik.

## 3.8. A Maven konfigurálása

Ebben a fejezetben kerül rövid leírásra a Maven pom.xml-je. Az ebben a pom.xml-nek az alsó részén található <dependencies> tagek közt található függőségek a "Felhasznált technológiák" fejezetnél tárgyalt függőségeket fogják tartalmazni. Ezeket a függőségeket így könnyen lehet használni az alkalmazásban. A fájl felső részében található <build> tagek közti utasítások a következő utasításokkal bírnak. Ezek a program összeállítása miatt fontosak, itt megadásra került a Main osztály, valamint a JAR neve, ami a JMSGraphAlgorithm lett. A kész JAR a target/App mappába fog kerülni. Idekerülnek átmásolásra a input mappa és a benne helyet kapó graph.txt is, valamint az output mappa a benne található messages.txt-vel.

A futtatható JAR állományt a következő lépésekkel lehet elkészíteni:

* Meg kell nyitni egy parancssort a projekt könyvtárában, majd üssük be a következő parancsokat.
* mvn clean
* mvn package -DskipTests

A clean parancs először kitörli az összes az előző buildelés által generált fájlt. A package pedig újra generálja a szükséges fájlokat, és elkészíti nekünk a JAR-t. A -DskipTests kapcsolót nem írnánk oda, úgy is lefutna a package parancs, de úgy a teszteket is lefuttatná hozzá.

A generálás során a JMSGraphAlgorithm.jar állományt, valamint az e mellett található input mappa a benne található graph.txt-vel és az output mappa a benne található messages.txt-vel. A .jar állományt kattintással, vagy parancssorból való futtatással lehet elindítani. Ha parancssorból történik az indítás a Spark szerver logolását is nyomon lehet követni. Valamint ügyeljünk arra, hogy a weblapon található "Stop Server" gombot mindig nyomjuk meg, ha beakarjuk zárni a programot, ha ezt nem tesszük meg, akkor a Feladatkezelőben kell a megfelelő folyamatot be kell zárni. Ha ez nem történne meg az esetleges hibához vezethetne.

## 3.9. Tesztelés

Ebben a fejezetben kerülnek részletezésre a forráskódhoz tartozó JUnit tesztek, valamint azok a manuális tesztek, amik során a weboldal és az algoritmus helyes működése tesztelésre került.

### 3.9.1. JUnit tesztek

A Java forráskódokat a JUnit technikával teszteltem. A projekt írása során törekedtem arra, minél több privát metódust használjak, így a JUnit-tal történő tesztelés nem sok függvényt érintett. A privát metódusok ugyanakkor ezeken a publikus metódusokon keresztül meghívásra kerülnek, amelynek során az ő helyes működésük is igazolásra kerül. A tesztelés során hibás és helyes adatokra is tesztelésre kerültek az osztályok. A következő osztályok kerültek tesztelésre:

* hu.elte.algorithm.Algorithm
* hu.elte.graph.GraphReader
* hu.elte.graph.Graph
* hu.elte.graph.Vertex
* hu.elte.jms.client.Client
* hu.elte.jms.consumer.Consumer
* hu.elte.jms.producer.Producer
* hu.elte.jms.message.AlgorithmMessage

### 3.9.2. A weblap manuális tesztelése.

A program indítása után a szerver helyesen betöltötte az index.html állományt az ott ábrázolásra kerülő elemek és animációk jól elkülöníthető részekre bonthatók. A felhasználó könnyű eligazodását teszik lehetővé. Az esetlegesen megjelenítésre kerülő hibaüzenetek egyértelmű utasításokkal látják el a felhasználót. A megfelelő eléréseken helyesen kerül megjelenítésre az algoritmustól érkező adat.

A weblapon található gombok működése manuálisan került tesztelésre. Az oldalon található gráf csúcsok kiírása megfelelő volt, az minden esetben tükrözte a graph.txt-ben felsorolásra került csúcsokat. A két kitölthető mező a helyes adatok esetén a működés szerint elmentésre került az értékük, hibás működés esetén a megfelelő hibaüzenettel kezelődött le a hiba.

A tesztelés következtében kerültek letiltásra a gombok is, amelyek a vizualizáció befejezése után újra engedélyezésre kerülnek. Így akadályozva meg azt, hogy ugyanazon gomb egymás után többszöri lenyomásával a vizualizáció rosszul jelenjen meg. A gombok működése és az általuk meghívott Javascript függvények működése a tesztelés során megfelelt az elvártnak, minden esetben hiba nélkül kerültek lefutásra.

A "Basic" és a "Animation" gombok megnyomásakor lefutásra kerülő Javascript függvények az animáció során helyesen szemléltették a gráf algoritmus működését. A konzol ablakban egyszer sem került sor hibaüzenetre, minden egyes alkalommal helyes kérést tudtak küldeni a szerver számára.

A "Stop Server" gomb megnyomásakor a lefutó Javascript függvény helyesen küldött jelzést a Spark szerver számára, és az helyesen le is állt. Az erről szóló üzenetet parancssori futtatás esetén a Spark ki is írja a konzol ablakba. A gomb megnyomását követően sikeresen betöltődött az error.html oldal.

# 4. Összegzés

A program nagy segítséget tud nyújtani azok számára, akik a hagyományosan ismert gráf kereső algoritmusokat szeretnék a modern technológiákkal ötvözni. A kereső algoritmusnak az üzenetsorokkal való támogatása pedig lehetőséget nyújt, hogy a gráf csúcsaira kliensekként gondoljunk, akik aszinkron módon tudnak kommunikálni egymással. Így lehetőségünk van arra, hogy akár nagy mennyiségű adatot is feldolgozzanak. Az esetleges felhasználók számára a weboldal pedig nagy segítséget tud nyújtani abban, hogy megértsék az algoritmus tényleges működését.

## 4.1. Továbbfejlesztési lehetőségek

Az algoritmus jelen állapotában csak irányítatlan gráfokat tud feldolgozni így az irányított gráfokra történő tovább fejlesztés lenne célszerű. Ebben az esetben már a való életben is több helyen fellelhető szituációk modellezésére is lehetne alkalmazni a szoftvert.

## 4.2. Köszönetnyilvánítás

Ezúton szeretném megköszönni a témavezetőmnek, Kiss Péternek, hogy a szakdolgozat ötletemet felkarolta, valamint szakértelmével a rendszeres konzultációk alkalmával tanácsaival segítséget nyújtott a dolgozat elkészítéséhez.

Köszönettel tartozom szüleimnek, valamint az egész családomnak, hogy türelemmel, és megértéssel támogatták tanulmányaimat, valamint, hogy minden helyzetben mellettem álltak.

Szeretném hálásan megköszönni az Eötvös Lóránd Tudományegyetem Informatikai Kar valamennyi oktatójának, dolgozójának azt a kitartó munkát, amivel hozzájárulnak ahhoz, hogy a hallgatók magas színvonalú, és versenyképes tudáshoz jussanak.

**Köszönöm!**

## 4.3. Melléklet

A szakdolgozathoz tartozik egy CD melléklet, amely tartalmazza a program forráskódját, a lefordított állományokat, és a dokumentációt PDF formátumban.