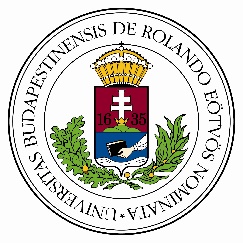
**Eötvös Loránd Tudományegyetem  
Informatikai Kar  
Adattudományi és Adattechnológiai Tanszék**

**Keretrendszer nagy erőforrás igényű számítások elosztott végrehajtására**

**Kiss Péter Polozgai Máté**

Doktorandusz, programtervező informatikus (Msc) nappali tagozat

programtervező informatikus

Budapest, 2018

**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!**

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 speciális gráf algoritmust valósít meg. Az algoritmussal irányítás nélküli gráfokban lehet megkeresni a legrövidebb utakat egy adott csúcsból kiindulva. Előfeltételként teljesülnie kell annak, hogy a gráfnak összefüggőnek kell lennie, valamint nem szabad hurokélt tartalmaznia.

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.

Elsőként az alapvető gráfelméleti fogalmakról esik szó, amelyek elengedhetetlenek a szakdolgozat megértéséhez, utána következik a felhasználói és a fejlesztői dokumentáció.

# 2. Gráfelmélet

## 2.1 Fogalmak jelölések

Ebben a fejezetben bemutatom, azokat a jelöléseket, és fogalmakat, amelyekre szükségünk van a téma kifejtéséhez, illetve amelyek a szakdolgozat megértését szolgálják.

**Definíció (irányítatlan gráf):** Egy G irányítatlan gráfot a G=(V,E) rendezett párral jellemezzük, ahol:

* V a csúcsok halmaza, mely egy véges nem üres halmaz,
* E az irányítatlan éleknek megfelelő csúcsok rendezetlen párjainak halmaza.

Az e={u,v} élről azt mondjuk, hogy az u és v között fut, összeköti u-t és v-t.

Ebben az esetben, ahogy a bevezetésben már utaltunk rá, a csúcsokra objektumokként is tekinthetünk, így a köztük lévő kapcsolat jelentheti út vagy kommunikáció lehetőségét. Ilyenkor rendszerint költségek vagy súlyok tartoznak az élekhez. Ezt a kapcsolatot egy valós értékű függvénnyel fogjuk leírni, melynek értelmezési tartománya a gráf él halmaza, az érték készlete pedig a valós számok halmaza és ce-vel jelöljük. Tehát ce : E R. A súlyokat az élekre szokás írni. ???

# 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.

## 2.1 Minimális rendszerkövetelmények

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

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

## 2.2 A program telepítése

A program telepítésére két lehetőségünk van. Ha csak futtatni szeretnénk a programot a szakdolgozathoz mellékelt CD-n megtaláljuk a ??? nevű ".jar" kiterjesztésű fájlt, amire kétszer kattintva a program automatikusan elindul. Meg kell nyitnunk "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

## 2.3 A program megnyitása Intellij IDEA

Miután a projekt fejlesztése közben az Intellij IDEA fejlesztői környezetet használtam, ezért ez az ajánlott fejlesztői környezet, de mivel a projekt a Maven-t használja a build folyamatok automatizálására, ezért más fejlesztői környezetben, például Eclipse, Netbeans is könnyen tudjuk importálni.

Intellij-ben a "File" >> "Open..." menüpontok segítségével válasszuk ki azt az elérési útvonalat, ahova a projektet kicsomagoltuk, majd az "OK" gombra kattintva várjuk meg, amíg a projektünk betöltődik.

## 2.3 A program funkciói

Ebben a fejezetben bemutatom a felhasználó számára, hogy mely funkciókat hogyan érhet el és azokat hogyan használhatja.

### 2.3.1 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 három gomb: a "Basic", az "Animation", és a "Submit". A felhasználó ezeknek a gomboknak a segítségével tudja inicializálni az algoritmust, és az algoritmus működését kiválasztani. Ezek alatt található meg az a szürke hátterű panel, melyben a gráf, és a két választott csúcs közötti legrövidebb út kerül szemléltetésre.

## 2.4 Felhasználói esetek

### 2.4.1 A gráf beolvasása

??? A gráf megadása a graph.txt-ben történik, méghozzá a gráf éleinek a megadásával: Először meg kell adni a gráf éleinek kezdő és végpontját szóközzel elválasztva, majd szóközzel elválasztva az adott élhez tartozó élsúlyt. Ügyeljünk arra, hogy minden ilyen élt új sorban kell megadni, valamint, hogy a gráfunk összefüggő legyen, és ne tartalmazzon hurokélt.

### 2.4.2 Az algoritmus inicializálása

A böngésző ablak bal felső részében található "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 adhatja meg a kezdőcsúcs és a végcsúcs címkéjét. Amennyiben nem a felsorolt csúcsok közül választ kezdő, illetve végpontot, vagy amennyiben valamelyik csúcs értékét üresen hagyja, a program egy figyelmeztető dialógust dob fel a böngésző ablakában.

### 2.4.3 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 az algoritmus ki fogja rajzolni az egész gráfot, és a korábban megadott két csúcs között a legrövidebb utat tartalmazó éleket pirossal rajzolja ki.

### 2.4.4 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 az algoritmus működése lépésről-lépésre be lesz mutatva, mégpedig a következő módon:

1. A grafikus felületünkön a kezdő ponttól kiindulva csúcsról-csúcsra haladva a gráfban található utolsó csúcsig, 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. A zöld gráf csúcsok pirosra váltanak, vagyis arról a csúcspontról az algoritmus a működéséhez szükséges minden információval rendelkezik. A szükséges módosításokat a csúcspont attribútumain végrehajtotta, ezért ismét csúcsról-csúcsra lépeget vissza, egészen a kezdő csúcsig.
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.

# 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, szerver oldal, valamint az algoritmus működésére szolgáló weblap részletes leírása. Leírásra kerül az algoritmus tesztelési folyamata is. A programnak az algoritmus logikája, 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. 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

Ez a fejezet 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 forráskódunkat 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 tudunk dolgozni az adott projekten. A tárhelyen megtalálhatók a projekt biztonsági mentései is a projektről, így nem kell aggódnunk, hogy elveszik a munkánk.

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

A Git használatához telepítenünk 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 lehetőségünk van a letöltésre:

1. Letöltés tömörített mappaként: "Download ZIP” gombra kattintunk, majd egy tetszőlegesen választott mappába kicsomagoljuk a letöltött állományt.
2. Tárhely klónozás, vagyis a Git segítségével a projekt repository-ban lévő könyvtárat egy az egyben letölti az általunk kijelölt mappába. Ehhez a művelethez le kell tölteni az előbb említett Git verziókezelőt, valamint rendelkezni kell GitHub fiókkal. Elsőként nyissunk egy parancssort abban a könyvtárban, ahova szeretnénk a forráskódot letölteni. Ezután gépeljük be a következőt: git clone https://github.com/polozgai/szakdolgozat.git , majd üssünk entert. 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 tudjuk nyitni bármely, az ilyen projektek megnyitását tartalmazó fejlesztői környezetben. A forráskód az Intellij IDEA nevű fejlesztői környezetben íródott, így ez az általam is ajánlott.

Miután megnyitottuk az Intellij IDEA-t, kattintsunk a bal felső sarokban lévő „File” menüpontra, majd az itt legördülő ablakon válasszuk az „Open...” opciót. Az itt elénk táruló ablakban keressük meg 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 tudjuk 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: 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 információt továbbíthatjuk, és feldolgozhatjuk a bejövő kéréseket. A használatához természetesen meg kell adni a függőségét a Maven által generált pom.xml-ben. A keretrendszerhez tartozó dokumentáció, valamint oktató kód részletek megtalálhatók a Spark Java hivatalos honlapján: <http://sparkjava.com/>

### 3.4.5 JSON-Simple

A Java Message Service-ben az üzenetek megkonstruálására a TextMessage osztályt használtam, melyben 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 web oldal 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. Ezzel a könyvtárral lehetőségünk van arra, hogy a szerver felől érkező akár nagy mennyiségű adatokat megjelenítsük a weboldalunkon, illetve a későbbiekben ezt módosítsuk. 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 kerül a projekt csomagszerkezete. A bennük található osztályok részletes kifejtésére, működésük leírására a későbbi fejezetekben kerül sor.

??? ábra

### 3.5.1 A program csomagszerkezete

A projekt jó áttekinthetősége érdekében rövid leírás a csomagszerkezetről:

* hu.elte: A hu.elte a központi csomag melyből újabb csomagok vezethetők le, ebben egyetlen osztály a Main található, mely a későbbiekben kerül részletezésre. Ahogy a nevéből is adódik ez a program futásának a belépési pontja.
* Algorithm: Az hu.elte.algorithm csomag a program szempontjából talán a legfontosabb részegység. A benne található osztály végzi a gráf algoritmus vezérlését. 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, annak a speciális működés szempontjából a gráf minden komponensének megfelelő objektum szerinti eltárolását.
* Jms: A hu.elte.jms csomagban több újabb csomag is található, melyeknek a szükségességét a jó áttekinthetőség, valamint a később részletezett 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 Apacha ActiveMQ szolgáltatásán keresztül.
* Server: A hu.elte.server csomagban van a Spark Java által biztosított szerver működését leíró osztály, melynek segítségével először a weblapot tudjuk 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.

### 3.5.3 Input mappa tartalma

Az input mappában található meg az a graph.txt fájl melyben azt a gráfot írjuk le, 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 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 tartanunk.
* Minden egyes ilyen él reprezentációt új sorba kell írnunk.
* Megjegyzés: Továbbá ügyelnünk kell arra, hogy betartsuk a szakdolgozat korábbi részeiben már tárgyalt a gráfra vonatkozó megszorításokat is, tehát a gráfnak összefüggőnek kell lennie, és nem szabad hurok éleket tartalmaznia.

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

## 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 tudjuk megkapni. 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. Ebben lesznek tárolva a kezdő és a végpont közti csúcsok, ha léteznek ilyenek. A kezdő és végpont nem lesz tárolva.

### 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 csúcs 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 van szemléltetve, mivel azt nem lehet előre tudni, hogy melyik lesz a legrövidebb út az adott csúcsba. Az Algorithm osztályban leírt működés miatt a kezdőcsúcsból a gráf összes többi csúcsába megkapjuk a minimális utakat, vagyis egy feszítőfát fogunk kapni a kiinduló csúcsból.

Az osztálynak négy privát adattagja van:

* Name: A csúcs String típusú csúcscímkéje.
* Active: A csúcsnak rendelkezni kell egy 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.
* Neighbours: A csúcs szomszédjait tartalmazó lista, amely Vertex típusú objektumokat fog tartalmazni.
* Routes: A csúcs útjait tartalmazó 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.

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 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 a velünk szomszédos, egyben üzenetet küldő csúcs címkéje lesz.
* "value": Az út költsége.
* "previous": Az esetleges köztes utak listája.

processRoutes(String message): Az üzenetsorból, ha egy adott csúcshoz üzenet érkezik, 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 iterálunk végig. A benne található adatok feldolgozása a következő módon fog történni:

A while ciklus első felében megtalálható egy if()-ben egy continue utasítás. Erre azért van szükség, mert egy üzenet csak a szomszéd csúcstól érkezhet, és ebben megtalálható a fogadó csúcsra vonatkozó üzenet is, amely már neki is meg van, hiszen ezzel már a gráf beolvasása után rendelkezünk.

Az üzenetcsere folyamán, mivel ott csak String-eket küldünk, így az egyes referenciák elvesznek. 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. 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 ilyen objektumokat kérjük le a referenciavesztés miatt:

* receivedVertex: Értékét a getRouteByName(String name) függvényből kapja, amely függvény 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 meghívnunk ezt a hívást. Ez a függvény végigmegy a gráf csúcsain, és visszaadja a megfelelő 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 kérnünk 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. Beállítjuk az út költségét.
2. Kitöröljük az esetlegesen megadott korábbi bejárási útvonalat.
3. Az új útvonalhoz hozzáadjuk a küldő csúcsot.
4. Az új útvonalhoz hozzáadjuk a korábban bemutatott previousList-et a nulladik indextől kezdődően.
5. Az új útvonalhoz hozzáadjuk az üzenetben megkapott köztes csúcsokat.

* 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. Ősszeállításra 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 az adatokat 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 csúcsok által küldött 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 csúcs címkéjére, 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 küldeni akarunk. 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 a küldést szánjuk. A metódusban az üzenetet az előbb bemutatott AlgorithmMessage osztály segítségével csomagoljuk be, majd megkreáljuk a kapcsolatot (create() metódus meghívása). Az üzenetet a [javax.jms.TextMessage](https://docs.oracle.com/javaee/7/api/javax/jms/TextMessage.html)-ként fogjuk publikálni az üzenetsorba, a kapcsolatot pedig ez után be is zárjuk (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, melyet egészen a gráf algoritmus végéig fenn is tart. Biztosítja a MessageConsumer nevű objektumot, amelyre a Client osztályban be tudunk állítani egy [javax.jms.MessageListener](https://docs.oracle.com/javaee/7/api/javax/jms/MessageListener.html) objektumot, amellyel aszinkron módon tudunk 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 üzenetsorban 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 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 keretét "lefejtsük" az értékes út információkról, és egy kulcsot rendeljünk hozzá a valós adatokhoz. Ezt a kulcsot, és értéket tovább küldjük a partMessageProcessing(String key, String value) számára.
* partMessageProcessing(String key, String value): Egy visszatérés nélküli privát metódus. Az első paramétere az a kulcs lesz, amely itt mindig csak "SEND\_ROUTES" lesz, amivel az üzenet érkezett. Értéke pedig a tényleges adatok, amiket tovább küldünk az adott csúcs processRoutes() metódusához. Ez a függvény, és vele együtt az előző kialakítása is alkalmas arra, hogy továbbfejlesztés esetén több fajta 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 ne olvashassuk, és írhassuk is egyszerre az üzenetet. 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 tudjuk küldeni a fullMessageProcessing() számára. Továbbá ennek a JSON-nek a String-é alakított változatát hozzáadjuk az Algorithm osztály messages listájához, hogy a későbbiekben az összes üzenetet ki tudjuk 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 kapjuk meg.
* 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 ezeket fogjuk kiíratni egy fájlba. Ehhez tartozik egy getter metódus is.
* mainQueue: Egy privát Vertex objektumokat tartalmazó lista. A gráf felderítése során fogjuk ebbe pakolni a gráf csúcsait, amit az inaktiválásuk folyamán ki fogunk venni.
* minRoute: Egy privát Vertex objektumokat tartalmazó lista, ez fogja sorrendben tartalmazni a legrövidebb úthoz tartozó út csúcs címkéit.
* jsonGraphByNode: Egy privát osztályszintű JSON objektum, amely a gráf feltérképezése során fogja a gráf csúcsait sorrendben belerakni. Ez az esetleges weblap animáció miatt szükséges. Ehhez az adattaghoz kapcsolódik egy getter metódus is.
* jsonGraphByNodeForColorChange: Egy privát osztályszintű JSON objektum, amely az algoritmusnak a második szakaszában a gráf csúcsainak inaktiválása miatt szükséges, 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 az 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 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. Ezt a kulcs-érték pár az allNodes adattagban 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 fogja 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 létrehozunk egy listát. Ez a "queue" nevű lista csak átmeneti szerepet fog betölteni. Ehhez a listához, valamint a "mainQueue" listához először belerakjuk a kezdőcsúcsot. Ezt aktiváljuk is, vagyis az "active" mezőjét true értékre állítjuk. A következőkben a külső while ciklus addig fog futni, ameddig tartalmaz elemet. Egy-egy lefutás végén az első elemet töröljük a "queue" listából. A ciklus elején lekérjük az első elemet, majd ennek a szomszédjain egy for ciklussal végigmegyünk. Ha a szomszédok még nem aktívak, és van szomszédjuk, akkor aktiváljuk őket, és belerakjuk 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 megyünk végig. Itt le kell kérnünk az eredeti szomszéd objektumot, hogy megtudjuk valóban aktív-e? Ha igen, akkor publikáljuk 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