



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSINFORMATIKA

TANSZÉK

Útvonaltervezés vizualizálása a BKK hálózatán

Témavezető:

Erdősné Németh Ágnes
Egyetemi adjunktus

Szerző:

Szabó-Galiba Máté
Programtervező informatikus BSc

Budapest, 2024

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	5
2.1. Áttekintés	5
2.2. Definíciók	6
2.3. Útvonal beállításai	7
2.4. Algoritmus kiválasztása	9
2.4.1. Választható algoritmusok	9
2.4.2. Algoritmusok beállításai	11
2.5. Útvonal tervezése	12
2.5.1. Az algoritmus futtatása	13
2.5.2. Az algoritmus állapota	13
2.5.3. Útvonalak a térképen	14
2.5.4. Megjegyzés az átszállásokról	16
3. Fejlesztői dokumentáció	18
3.1. Bevezetés	18
3.2. Adatforrás	18
3.3. Tervezés és követelmények	19
3.3.1. Nem funkcionális követelmények	19
3.3.2. Funkcionális követelmények	21
3.3.3. Felhasználási eset diagram	22
3.3.4. Felhasználói történet	22
3.3.5. Felhasználói felület tervezek	25
3.4. Magas szintű áttekintés	27
3.4.1. Alkalmazás felépítése	27
3.4.2. Verziókövetés	28
3.4.3. Backend áttekintés	28

TARTALOMJEGYZÉK

3.4.4. Frontend áttekintés	28
3.4.5. Fejlesztői környezet felállítása	29
3.4.6. Alkalmazás futtatása	29
3.5. Backend	30
3.5.1. Könyvtárszerkezet	30
3.5.2. Adatbázis	31
3.5.3. Szerkezet	35
3.5.4. REST API	37
3.5.5. Tesztelés	37
3.6. Frontend	38
3.6.1. Definíciók	39
3.6.2. Algoritmusok	40
3.7. Implementáció	44
3.7.1. Algoritmusok	44
3.7.2. Felhasználói felület	47
3.7.3. API hívások	48
3.7.4. Tesztelés	48
3.8. Telepítés	50
3.9. További fejlesztési lehetőségek	50
4. Összegzés	52
Köszönetnyilvánítás	53
Irodalomjegyzék	53
Ábrajegyzék	56
Táblázatjegyzék	57
Forráskódjegyzék	58

1. fejezet

Bevezetés

Az informatika - különösen egyetemi környezetben való - oktatásában az elméleti háttér kiemelkedő szerepet kap. Ez természetes, hiszen megfelelő elméleti alapok nélkül a gyakorlatban is csak korlátozottan lehet eredményeket elérni. Azonban sok tanuló számára a száraz elméleti anyag nehezen érthető, és gyakorlati alkalmazás hiányában gyakran érdektelennek tűnik. Az elvont elméletet nehéz lehet a gyakorlattal összekapcsolni, és soknak ez okozza a legnagyobb nehézséget az informatika tanulásában. Ez a probléma különösen szembetűnő az algoritmusok tanulásakor, hiszen ezek eredendően gyakorlatiasak; céljuk a program gyorsabb, vagy más szempontból eredményesebb működése. Ám ez a gyakorlatiasság sokszor elveszik az elméleti leírásokban, és a hallgatók számára nehezen érhetővé válik.

Ennek a programnak a célja, hogy segítséget nyújtson az útkereső algoritmusok megértésében az elmélet és a gyakorlat összekapcsolásával. Az alkalmazás egy webes felületen keresztül teszi lehetővé a felhasználók számára, hogy különböző algoritmusok működését vizsgálhassák lépésről lépésre. Erre célra mi sem jobb adatforrás, mint a budapesti tömegközlekedés, amivel a magyar diákok jelentős része nap mint nap találkozik. Az alkalmazásban a felhasználók négy alapvető útkereső- és gráfbejáró-algoritmus működését hasonlíthatják össze: a szélességi keresést, a mélységi keresést, a Dijkstra-algoritmust és az A*-algoritmust.

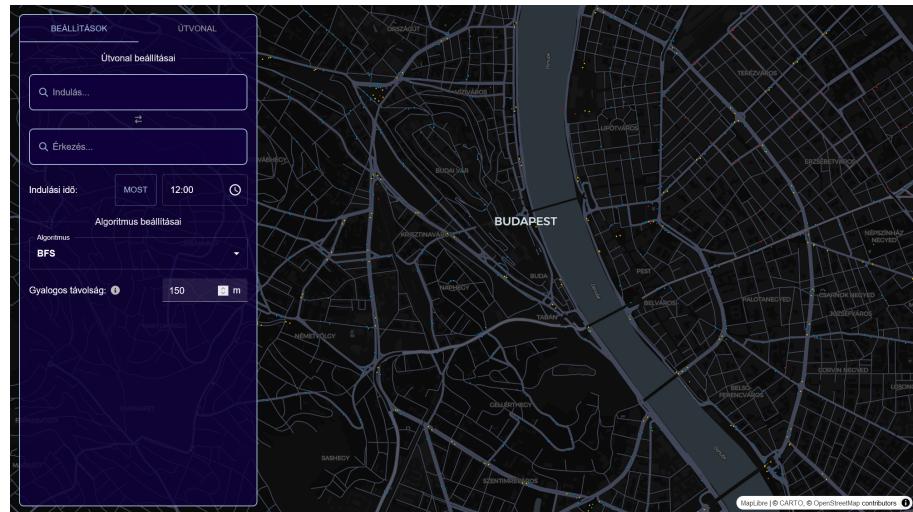
Az alkalmazás használata során a felhasználók kiválaszthatnak egy kiinduló és egy célállomást, majd az alkalmazás lépésről lépésre bemutatja az adott algoritmus működését a két állomás közötti útvonal megtalálásához. A grafikus felület segít az egyes algoritmusok előnyeinek és hátrányainak megértésében, és akár az algoritmusok ismeretében kevésbé jártas felhasználók számára is egy képet ad azok működési elvéről.

2. fejezet

Felhasználói dokumentáció

2.1. Áttekintés

Az alkalmazás böngészőben fut. Egy oldalból áll, mely nagy részét egy interaktív térkép foglalja el (2.1). A képernyő bal oldalán egy vezérlőpanel található, melyen keresztül az alkalmazás irányítható. Az alkalmazás használatához nincs szükség regisztrációra vagy bejelentkezésre.



2.1. ábra. Az alkalmazás felülete térképpel és vezérlőpanellel

A térkép navigációja egérrel történik; a térkép nagyítása és kicsinyítése a görgetőkerékkel, a térkép mozgatása pedig az egér bal gombjának lenyomásával és húzásával. A térképen a BKK és a MÁV-HÉV járatainak a megállói láthatók, amik egy-egy színes körrel van jelölve, melyeknek a színét a megállóhoz tartozó járatok

színe¹ határozza meg.

2.2. Definíciók

- **Gráf:** Formálisan csúcsok és élek halmaza, ahol minden él két csúcsot köt össze. Az alkalmazás által használt modellben a közlekedési hálózatot egy irányított, súlyozott gráffal modellezük, így "gráf" alatt a továbbiakban ezt értjük, amennyiben más nem jelzünk.
- **Csúcs:** A gráf eleme, mely egy megállót jelképez. A továbbiakban a "csúcs" és a "megálló" kifejezések egymás szinonimájaként értendőek, kontextustól függően használjuk őket. A csúcsokhoz egyedi azonosítók tartoznak, melyek a forrásadatokból származnak, és az azonos nevű megállókat megkülönböztetését segíthetik elő.
- **Él:** A gráf eleme, mely két csúcsot köt össze. Az élek irányítottak, azaz csak egy irányba utazhatóak; illetve súlyozottak, azaz minden élhez egy súlyt rendelünk, mely az élen való áthaladás költségét jelképezi. Az alkalmazásban két féle él található:
 1. **Utazási él:** Két megálló közötti közvetlen járatot jelképez, melynek a súlya az utazás időtartamának és a járatra való várakozás idejének az összege. Ez utóbbi alól az első utazási él mentesül, hiszen az csak későbbi indulást jelent, nem várakozást.

Megjegyzés: A program csak olyan járatokat vesz figyelembe, melyekre a várakozási idő nem haladja meg a 60 percent.
 2. **Átszállási él:** Két megálló közötti gyaloglást jelképez, melynek a súlya a $1\text{perc} + 1\frac{\text{másodperc}}{\text{méter}}$ képlet alapján számolódik, vagyis minden átszállás alapsúlya 1 perc, ehhez adódik annyi másodperc, amennyi méter az utak közötti távolság légvonalban.

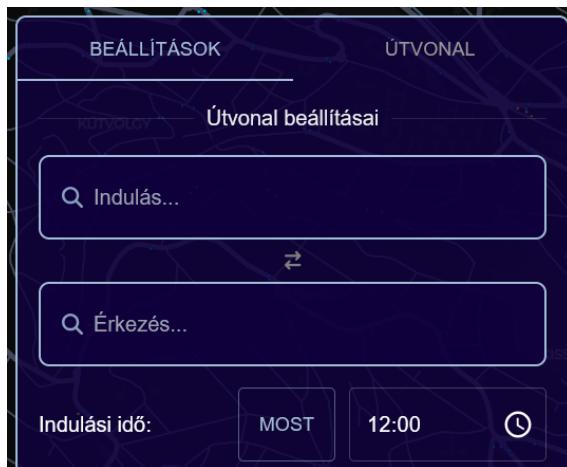
Megjegyzés: Amennyiben az útvonal első éle átszállási él, annak 0 a súlya — ez azért van, mert könnyű az azonos nevű és egy helyen lévő megállókat összekoverni választáskor, és ezzel a módszerrel akadályozza meg a

¹A színek a közlekedési társaságok által használt, közismert színek (pl. a villamosok sárgák, a trolik pirosak).

program azt, hogy egy ilyen hiba miatt hosszabbnak tűnjön az út, mint amilyen valójában.

2.3. Útvonal beállításai

Útvonal tervezéséhez szükség van egy indulási időpont², illetve egy kiinduló- és egy úticél megadására. Célpontoknak a térképen szereplő megállók közül kell választani, egyéb koordináta/cím megadása nem lehetséges. Ezeknek a megadására a vezérlőpanel "BEÁLLÍTÁSOK" fülén van lehetőség (2.2).

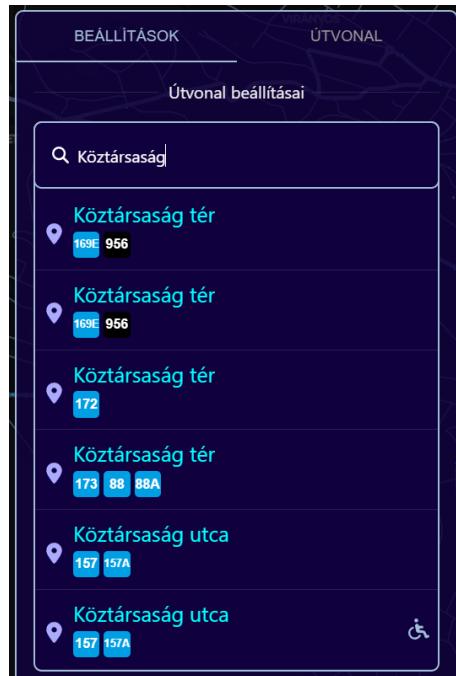


2.2. ábra. Az indulási idő, illetve a kiinduló- és célállomás beállítása

Állomások választásához a megfelelő mezőbe kell írni, majd kattintással kiválasztani a megfelelő megállót — nem elég a nevét beírni, hiszen több, egymástól távoli megálló is rendelkezhet ugyanazzal a névvel (2.3). Megfelelő megálló választásához segítségképpen a listában a megállókból induló járatok is megjelennek az adott megálló neve alatt.

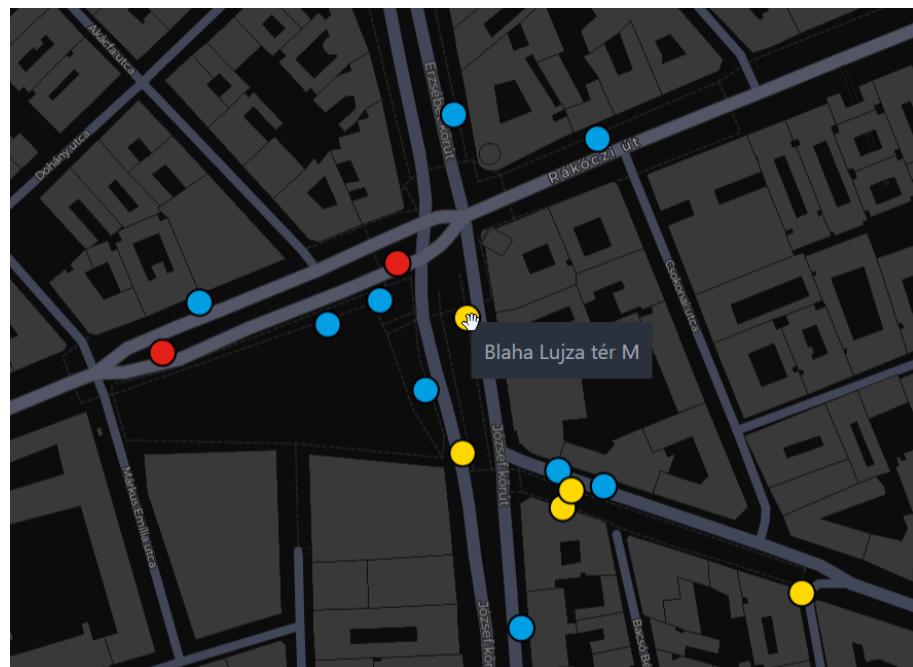
A kiinduló- és célállomás felcserélése a két beviteli mező közti dupla nyílra kattintva lehetséges (2.2).

²Az indulási idő budapesti időzóna szerint értendő, az alapértelmezett értéke az aktuális helyi idő.



2.3. ábra. Keresési találatok: az egyik *Köztársaság tér* nevű megálló Törökbálinton, a másik Pécelen található

A térképen az egeret egy megálló fölé helyezve megjelenik annak a neve (2.4); ez segítséget nyújthat, ha nem ismerjük a célpontunk közelében lévő megállók nevét.



2.4. ábra. A kurzor alatt lévő megálló neve egy információs buborékban jelenik meg

2.4. Algoritmus kiválasztása

2.4.1. Választható algoritmusok

Az útvonalkereséshez négy különböző algoritmus használható, ezekről részletesebben a 3.6.2. fejezetben olvashatunk.

Áttekintés az algoritmusokról:

1. **BFS:** A "Breadth-First Search", azaz szélességi gráfbejárás egy soralapú algoritmus, ahol az újonnan felfedezett megállók egy sor (FIFO adatszerkezet) végére kerülnek be, így először az 1 megálló távolságra lévő megállók kerülnek felfedezésre, majd a 2., stb. . Az algoritmus alapvetően súlyozatlan gráfokon operál, súlyozott gráfokra való alkalmazásakor is figyelmen kívül hagyja az élek súlyát. Ennek következtében az útkeresés eredményeként garantáltan a legkevesebb megállóból álló utat kapjuk meg, attól függetlenül, hogy az adott út mennyi időbe telik. Ennek az algoritmusnak a futásidője egy hagyományos gráfon a legrosszabb esetben $O(|V| + |E|)$, ahol V a csúcsok, E az élek száma a gráfban.
2. **Dijkstra-algoritmus:** Az algoritmus a feltalálójáról, Edsger W. Dijkstra informatikusról kapta a nevét, és egy súlyozott gráfban keresi meg a legkisebb súlyú utat egy kiindulópontból az összes többi csúcsba. Az algoritmus a BFS-sel szemben egy prioritási sort használ, ahol a sor elemei a gráf csúcsai, és súly szerinti sorrendben kerülnek feldolgozásra. (Egy-egy csúcs súlya jelen esetben a kezdőállomásból való utazási távolságnak felel meg.) Az alkalmazásban elérhető algoritmusok közül ez az egyetlen, amely garantálja a legrövidebb utazási időt, viszont futásidőben a prioritási sor manipulálásának a komplexitásának³ következtében az algoritmus komplexitása is magasabb a BFS-hez képest. Az algoritmus futásidője egy hagyományos gráfon a legrosszabb esetben $O((|V| + |E|) \log |V|)$.
3. **Mohó algoritmus:** A mohó algoritmus a Dijkstra-algoritmushoz hasonlóan egy prioritásos soron alapul, azonban bevezeti a heurisztika fogalmát. A heurisztika egy olyan függvény, amely egy "megérzést" ad egy adott csúcsról, azaz

³Az alkalmazás egy kupaccal implementálja a prioritási sort, így egy elem beillesztésének és eltávolításának a komplexitása legrosszabb esetben egyaránt $O(\log n)$.

megbecsüli, hogy az adott csúcs mennyire jó választás lehet a következő lépésben. Ebben az alkalmazásban ennek az implementációja a csúcs távolsága a célállomástól, a Föld felszínén egyenes vonalban utazott méterekekben mérve⁴. Az algoritmus a prioritási sorban a heurisztika értéke szerinti sorrendben dolgozza fel a csúcsokat, így általában sokkal gyorsabban eljut a célállomásba, viszont a BFS-hez hasonlóan ez sem veszi figyelembe az utazási időt, így praktikus használatra általában nem alkalmas. Az algoritmus futásideje egy hagyományos gráfon a legrosszabb esetben $O((|V| + |E|) \log |V|)$.

4. **A* algoritmus:** Az A* algoritmus egy továbbfejlesztett mohó algoritmus, amely a Dijkstra-algoritmus és a mohó algoritmus előnyeit igyekszik ötvözni. Az algoritmus a csúcsok súlyát (azaz a kezdőállomástól való utazási időt) és a heurisztikát (a csúcs távolságát a célállomástól) együtt veszi figyelembe a prioritási sorban, így a legjobb választásnak tűnő csúcsokat feldolgozva igyekszik a lehető leggyorsabban eljutni a célállomásba. A* algoritmus választásakor módnunk van megadni egy súlyozó faktort, amellyel a heurisztika értékét szorozzuk, így az algoritmus viselkedését befolyásolhatjuk. Alacsonyabb szorzó esetén a Dijkstra-algoritmusra hasonlító viselkedést kapunk (lassabb futásidő, de rövidebb út), magasabb szorzó esetén a mohó algoritmushoz hasnlót (gyorsabb futásidő, de könnyebben eltér a legrövidebb úttól). Az alapértelmezett szorzó 1, de saját tapasztalataim szerint a 0.3 – 0.5 körüli súly jó egyensúlyt biztosít a futásidő és a "használható" eredmény között. Az algoritmus futásideje egy hagyományos gráfon a legrosszabb esetben $O((|V| + |E|) \log |V|)$.

Röviden összefoglalva:

- **BFS:** Legkevesebb megállóból álló útvonalat ad, de nem veszi figyelembe az utazási időt.
- **Dijkstra:** Garantáltan a legrövidebb utazási időt adja, de magasabb futásidővel jár.
- **Mohó:** Általában jelentősen gyorsabb a futásideje, de sem az utazási időt, sem az utazott megállók számát nem veszi figyelembe.
- **A*:** Kompromisszum a Dijkstra és a mohó algoritmus között, súlyozó faktorral befolyásolhatjuk a viselkedését.

⁴A képlet ugyan nem veszi figyelembe a tengerszint feletti magasságot, de ez Budapesten és környékén nem tesz drasztikus különbséget, így elfogadjuk közelítésnek.

2.4.2. Algoritmusok beállításai

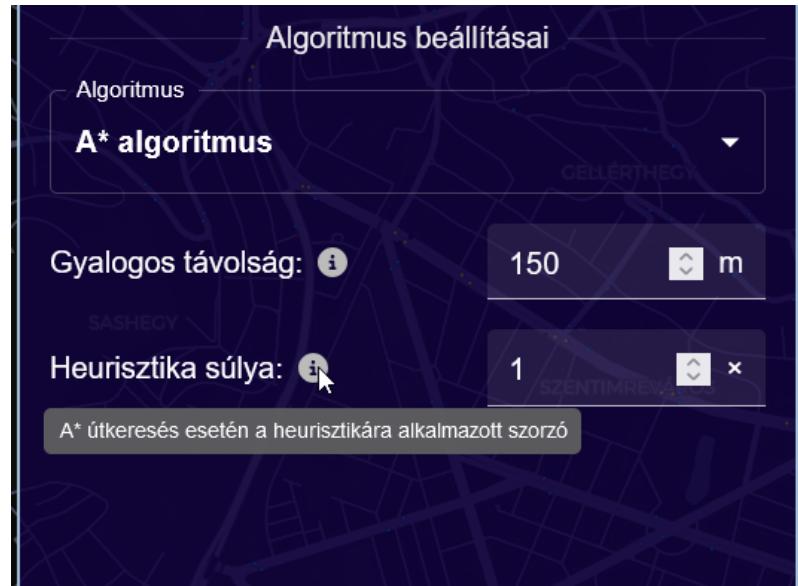
Az alapértelmezett algoritmus a BFS. Ezt az útvonal beállításai alatt, úgyszintén a vezérlőpanel "BEÁLLÍTÁSOK" fülén lehet megváltoztatni (2.5).



2.5. ábra. Az elérhető algoritmusok listája

Választott algoritmustól függetlenül beállítható az is, hogy mi a maximális sétáló távolság, amin belül az alkalmazás felajánl átszállásokat. Ennek az alapértelmezett értéke 150 méter, ami tapasztalataim szerint általában elég azonos nevű, egy csoportban lévő megállók közti átszálláshoz.

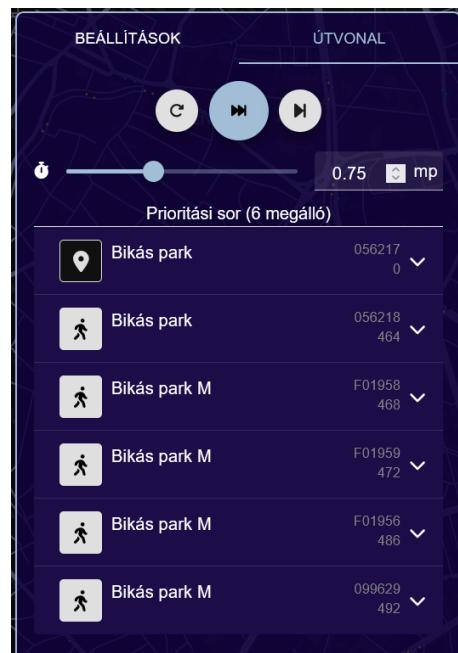
Amennyiben a választott algoritmus az A*, akkor a heurisztika súlyozó faktora is beállítható, mely alapértelmezetten 1 (2.6).



2.6. ábra. A beállításokat információs buborékok magyarázzák

2.5. Útvonal tervezése

Amennyiben megtörtént a kezdő- és célállomás megadása, illetve az algoritmust és annak paraméter(ei)t is beállítottuk, megkezdődhet az útvonal tervezése. Ez a vezérlőpanel "ÚTVONAL" fülén történik, mely érvényes beállítások megadása után válik kattinthatóvá (2.7).



2.7. ábra. Az algoritmus alapállapota az "ÚTVONAL" fülön

2.5.1. Az algoritmus futtatása

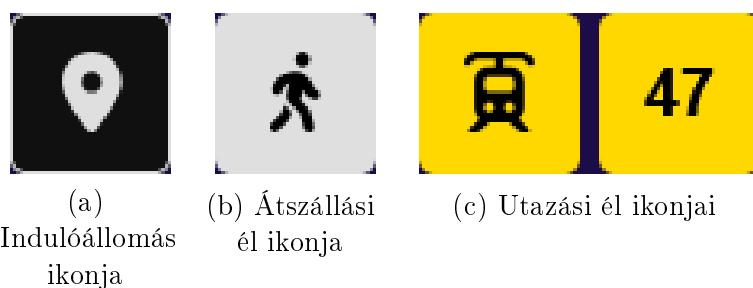
A fül tetején az algoritmus léptetésére szolgáló gombok találhatók, melyek a következők:

- **Újraindítás:** Az algoritmus visszaállítása a kezdőállapotba.
- **Futtatás:** Az algoritmus addig fut, amíg el nem éri a célállomást. Amíg az algoritmus fut, a többi gomb nem érhető el, ez pedig átváltozik **Szüneteltetés** gombbá, ami leállítja az algoritmust.
- **Léptetés:** Az algoritmus egy lépéssel halad előre, majd megáll.

Ezek alatt egy csúszka található, amelyen azt állíthatjuk be, hogy az algoritmus léptetéskor mennyit várakozik két lépés között. Az alapértelmezett értéke 0.75 másodperc. *Megjegyzés: Természetesen lehetséges, hogy egy csúcs feldolgozása tovább tart a várakozási időnél, különösen alacsony értékek esetén.*

2.5.2. Az algoritmus állapota

Amíg az algoritmus nem talált utat a célállomásba, addig a fent említett irányítógombok alatt láthatóak azok a megállók (és olyan sorrendben), amiket az algoritmus következőként fog feldolgozni. A megállók melletti ikon(ok) jelzik, hogy az adott megállóhoz milyen úton érkeztünk. Indulóállomás ez az ikon egy sötét háttéren lévő helyjelző pont, átszállási él esetén egy gyalogló ember, utazási él esetén pedig a járat ikonja és száma látható (2.8).

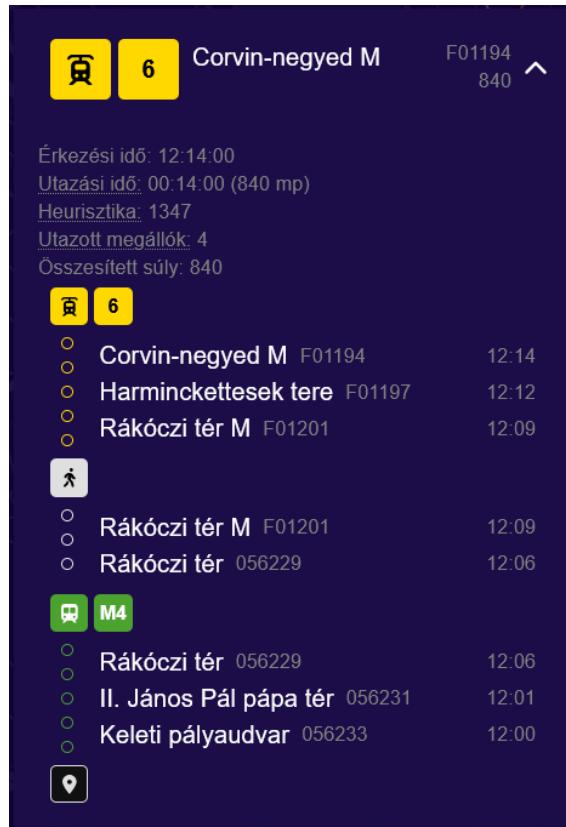


2.8. ábra. A megállók ikonjai azt jelzik, hogy milyen úton érkeztünk az adott csúcsba

Egy-egy megállóra kattintva lenyílik egy további részleteket tartalmazó információs doboz, melyben az adott megállóhoz való érkezés ideje, az odáig megtett út ideje, a csúcs heurisztikája (távolsága a célállomástól méterben), az útban lévő

utazási élek száma, illetve a csúcs súlya látható (2.9). Ez utóbbi az algoritmustól függően van a fentiek alapján kiszámítva.

Ezeken az információkon kívül a dobozban a megállóig tartó út is látható, mely a megállók neveit, azonosítóját és az utazás módját jelölő ikonokat tartalmazza.

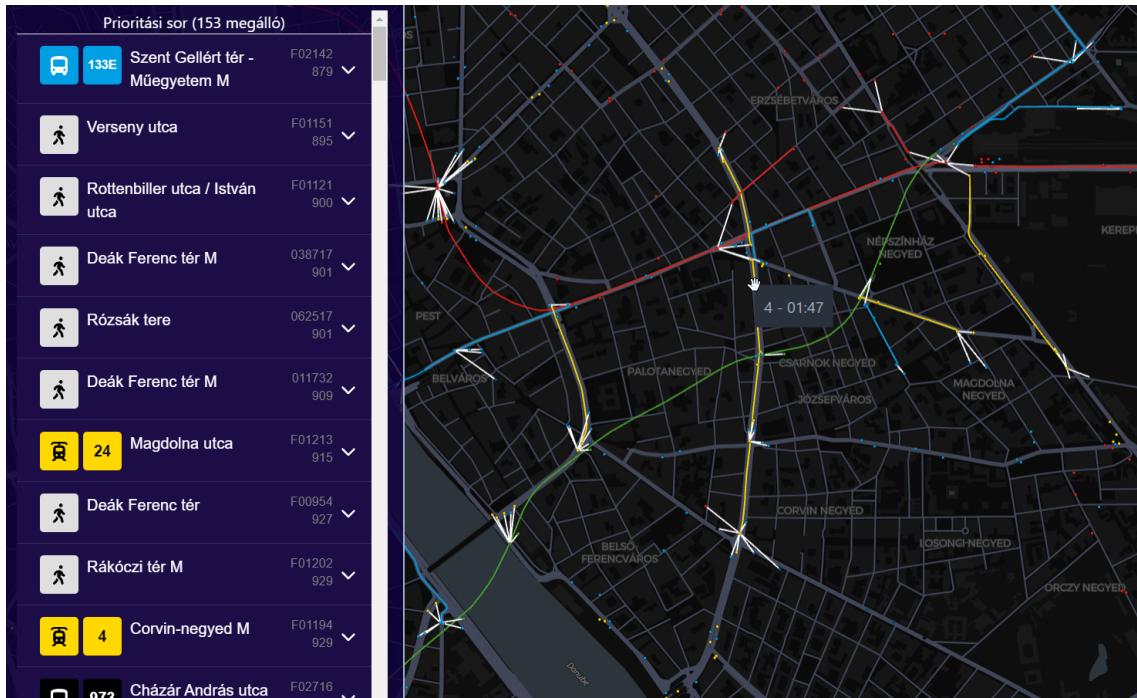


2.9. ábra. A megálló részletes információi

Ezen ismeretek birtokában nincs más hátra, mint lépésről lépésre végignézni az algoritmus futását, és megfigyelni, hogy milyen útvonalon jutunk el a célállomásig.

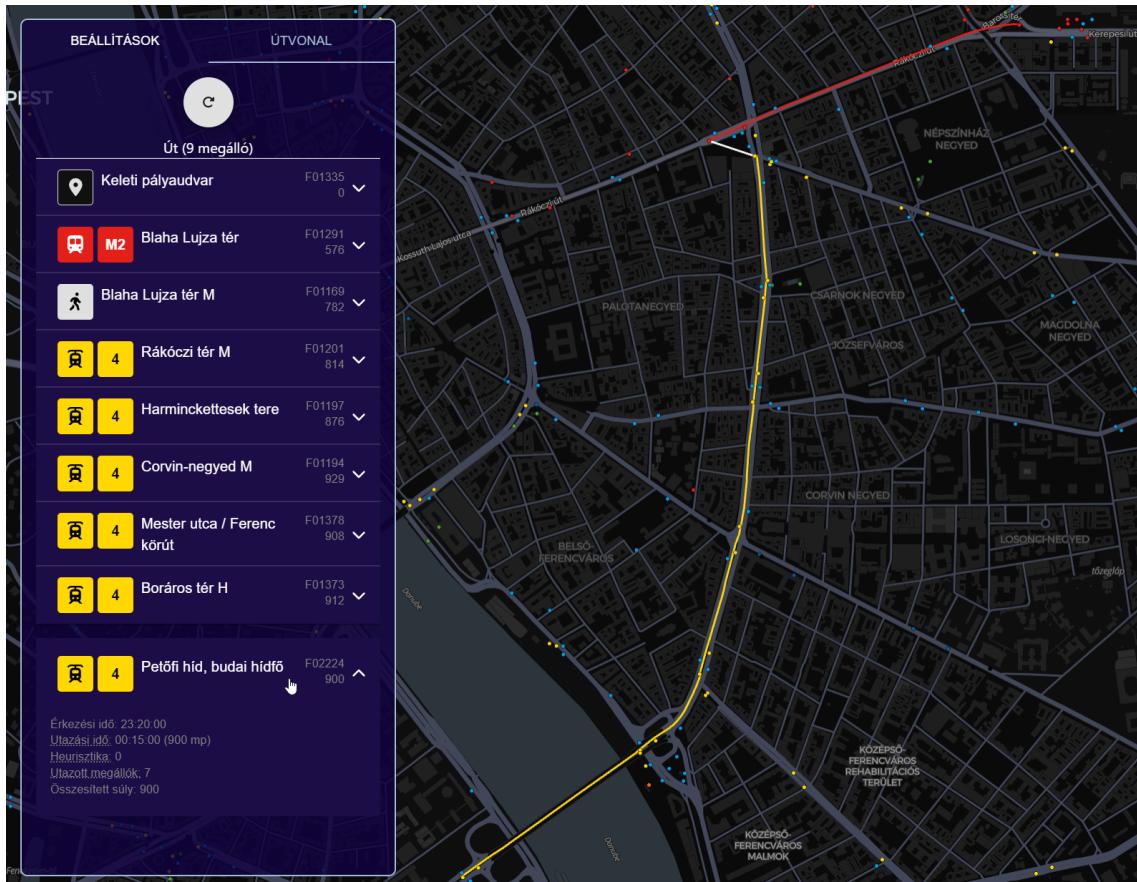
2.5.3. Útvonalak a térképen

Az algoritmus futása közben a térképen is ki vannak rajzolva a sorra következő megállók, illetve az a kiindulóállomásból vezető út. Az út a megfelelő közlekedési eszköz színével, illetve gyalogos él esetén szürkével (és egy egyenes vonallal) van jelölve. Az egeret egy utazási él fölé helyezve megjelenik annak a járat neve és az utazás hossza is (2.10).



2.10. ábra. Utazás hossza a térképen: egy megálló távolságra értendő

Amennyiben az algoritmus megtalálta a célállomást, a térképen csak az az út marad megjelenítve, és az oldalsó menüben is az út megállóinak a lista lesz látható (2.11).



2.11. ábra. Kész útvonal a térképen

2.5.4. Megjegyzés az átszállásokról

Ugyan a hétköznapokban általában "egy megállóként" szokás gondolni az egymás mellett lévő, azonos nevű megállókra, a valóságban nem ilyen egyszerű a helyzet. A forrásadatban a fizikai megállók külön-külön vannak azonosítva, így a program számára akár két olyan megálló is teljesen különállónak tűnhet, ami a valóságban ugyanahhoz a vonalhoz tartozik, csak az egyiknél a járat az egyik irányban, a másiknál az ellentétes irányban áll meg. Ennek következtében, ha pusztán tömegközlekedéssel keresnénk utat, akkor az alkalmazás nem javasolna olyan átszállásokat, ahol két járat egymás mellett lévő megállói között kellene átszállni; még a célállomást is eltévesztené, amikor az út rossz oldalára érkezik.

Erre természetes megoldás a gyalogos élek bevezetése, így az egymás közelében lévő megállókat szinte egy megállóként kezelhetjük. A program ezt úgy valósítja meg, hogy amint egy új utazási élét ismer meg, a célpontjától sétávolságra lévő megállókat is hozzáadja az ismert megállók listájához, átszállási éssel összekötve

őket az eredeti megállóval. Ugyanígy viselkedik egy útvonaltervezés megkezdésekor, amikor az indulóállomás közelében lévő megállókat "fedezi fel".

Ennek mellékhatásaként amennyiben a választott úticélunkhoz el lehet jutni közvetlen oda érkező járattal is, és egy korábban felfedezhető (pl. Dijksra-algoritmus esetén egy 1 perccel hamarabb elérhető) megállóból való sétálással is, akkor a program a sétálást tartalmazó utat fogja megtalálni, akkor is, ha a sétálás 1 percnél tovább tartana. Ez egy apró, de említésre érdemes bökkenő, hiszen a Dijkstra-algoritmustól azt várunk, hogy minden esetben a leggyorsabb utat találja meg a két végpont között.

3. fejezet

Fejlesztői dokumentáció

3.1. Bevezetés

Az alkalmazáshoz való technológiák kiválasztásakor fontos mind a felhasználó, mind a fejlesztő igényeit figyelembe venni. Szerencsére, jelen esetben van megoldás, amely minden oldal számára a legtöbb kényelmet nyújtja, mégpedig a webalkalmazás. A felhasználó számára könnyű elérést, platformfüggetlenséget, és egy megszokott felületet hordoz magával, ami különösen fontos egy oktatási céllal rendelkező alkalmazásnál, hiszen még kevesebb akadályt helyez a felhasználó és a "tananyag" közé. Fejlesztői szempontból is kényelmes egy ilyen alkalmazást a böngészőre írni, hiszen a JavaScript ökoszisztémában könyvtárak és keretrendszerök tömkelege áll rendelkezésre, melyek segítségével gyorsan és hatékonyan lehet egy webalkalmazást fejleszteni.

3.2. Adatforrás

Az adatok a BKK által szolgáltatott OpenData Portálon[1] nyilvánosan elérhető adatbázisból származnak. Az adatokat a BKK a GTFS (General Transit Feed Specification) formátumban teszik elérhetővé, ami egy Google-nél kifejlesztett[2] nyilvánosan elérhető specifikáció, mely egy szabványos formátumot definiál a tömegközlekedési adatok szolgáltatására.

3.3. Tervezés és követelmények

3.3.1. Nem funkcionális követelmények

Termék követelmények

1. Hatékonyság

- A szoftver kezdeti megnyitásakor legkésőbb 5 másodperc alatt teljesen betöltődik és használhatóvá válik
- A szoftver általános használat közben folyamatosan, megakadás nélkül fut egy középkategóriás számítógépen, egy modern böngészőben
 - Kivétel ez alól az animált útvonaltervezés, amelynek a futása közben a szoftver akadozhat, de nem annyira, hogy használhatatlanná váljon, vagy megakadályozza az animáció leállítását
- A backend válaszideje API hívásokra nem több, mint 5 másodperc (bár-milyen lehetséges, érvényes API hívásra)
- A szoftver felhasználói bevitelre adott válasz ideje nem több, mint 100 ezredmásodperc
 - Ebbe beleértendő a betöltést jelző válasz, amíg az alkalmazás API hívásokra várakozik
- A szoftver nem használ a szükségesnél több processzorkapacitást (pl. nem használja processzortól függetlenül az összes elérhető teljesítményt)

2. Megbízhatóság

- A szoftverbén ne legyen olyan egyszerűen előidézhető vagy gyakran bekövetkező hibajelenség, ami előfordulása esetén ellehetetleníti vagy jelenősen megnehezíti a szoftver használatát

3. Biztonság

- A szoftver ne tároljon felhasználói adatokat
- A szoftver ne tároljon érzékeny adatokat
- A szoftver ne tároljon jelszavakat

4. Hordozhatóság

- A szoftver kliens oldala bármilyen WebGL-t támogató modern böngészőben fut, különös tekintettel a Chromium alapú böngészőkre

- A szoftver a kliens oldalon állandó, stabil internetkapcsolatot és a szerverttel való kapcsolatot igényel
- A szoftver szerver oldala az adatok letöltéséhez stabil internetkapcsolatot igényel, ezt követően csak a klienssel szükséges kommunikálnia

5. Felhasználhatóság

- A szoftver intuitív és könnyen használható
- A szoftver kliens oldalának a használatához nem szükséges külön telepítés, komoly számítógépes tapasztalattal nem rendelkező felhasználók számára is egyértelmű
- A weboldal felülete egy átlagos számítógéphasználó számára külső segítség nélkül elsajátítható, amennyiben ismerik a szoftver által bemutatott útkereső algoritmusokat
- A szoftver szerver oldalának az üzemeltetése hálózati és Docker-compose ismereteket igényel

Menedzselési követelmények

1. Környezeti

- A szoftver kliens oldalon egy egeret és egy billentyűzetet igényel
- A szoftver szerver oldalának a futtatásához legalább egy középkategóriás számítógépnek megfelelő hardver szükséges, eltekintve a perifériáktól

2. Működési

- A szoftver legfeljebb egy órás összefüggő időtartamokban lesz használva a kliens oldalon
- A szerver oldalon a szoftver folyamatosan fut, legfeljebb 4-5 naponta lehet szükséges karbantartási műveleteket végezni rajta, pl. a szoftver újraindítása (eltekintve az adatforrások frissítésétől)

3. Fejlesztési

- Frontenden Node.js, React, TypeScript
 - Térkép és adatok megjelenítéséhez deck.gl, react-map-gl
- Backenden Node.js, Express, TypeScript
 - REST API szerver

- SQLite adatbázis
- Adatok importálásához node-gtfs
- Visual Studio Code fejlesztői környezet
- Docker
- 64-bit architektúra
- Verziókezeléshez Git kliens

4. Fenntartási

- minden API végpont tesztelve és OpenAPI 3 formátumban dokumentálva van

Külső követelmények

- A szoftverhez felhasznált külső forrásból származó médiafájlok jogtiszták
- A szoftver nem tartalmaz erkölcsileg megkérdezhető, sértő tartalmat

3.3.2. Funkcionális követelmények

Funkciók

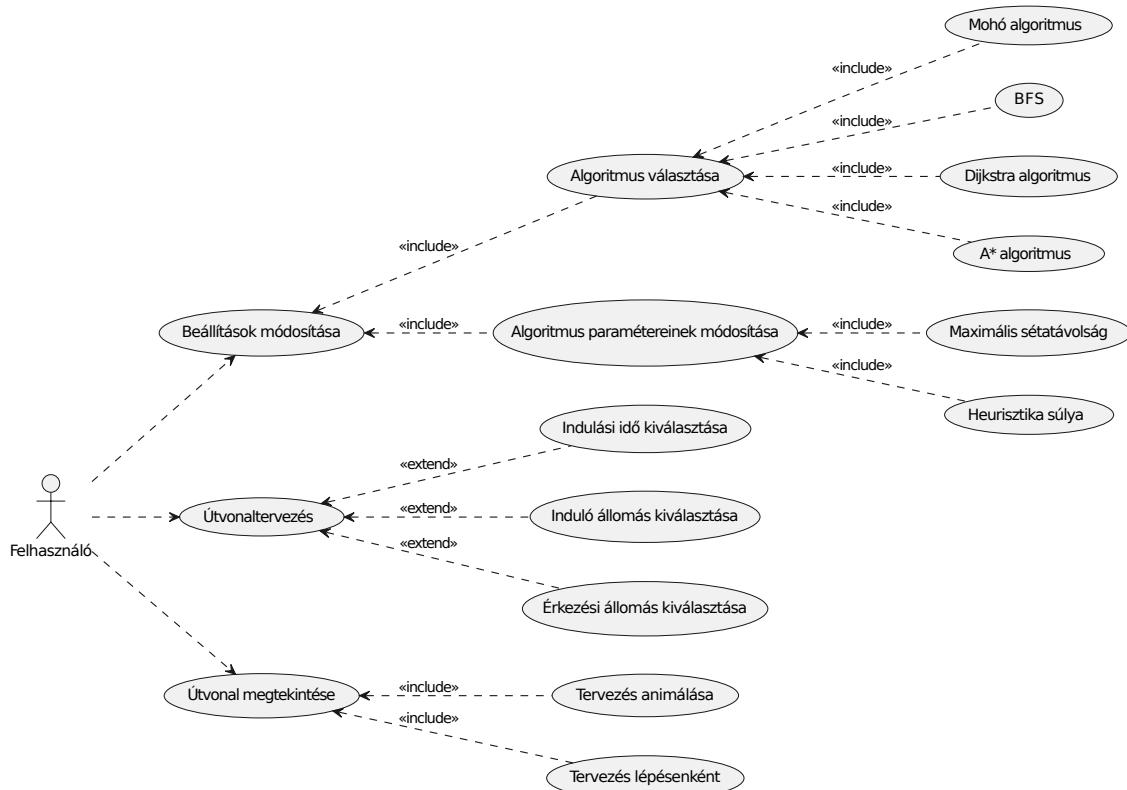
1. Útkeresés két megálló között budapesti tömegközlekedési járatokon

- Beállítások megadása
 - Indulási és érkezési megálló kiválasztása
 - Indulási idő megadása
 - Útvonaltervezési algoritmus kiválasztása
 - (a) BFS keresés
 - (b) Dijkstra algoritmus
 - (c) Mohó algoritmus
 - (d) A* algoritmus
 - Útvonaltervezési algoritmus paramétereinek megadása
 - * Maximális sétaátmérő átszállások között
 - * A* algoritmus esetén: heurisztika súlyozása
- Útvonaltervezés
 - Algoritmus léptetése
 - Algoritmus futtatásának indítása

- Algoritmus futtatásának szüneteltetése
- Algoritmus futtatásának folytatása
- Algoritmus visszaállítása alapállapotba
- Algoritmus eredményének megjelenítése
 - * Potenciális útvonalak megjelenítése a térképen
 - * Soron következő megállók megjelenítése

3.3.3. Felhasználási eset diagram

A program felhasználói eseteit a 3.1 ábra mutatja be.



3.1. ábra. Felhasználási eset diagram

3.3.4. Felhasználói történet

A felhasználói történetek a 3.1., 3.2., 3.3., és a 3.4. táblákban olvashatóak.

1	AS A	Felhasználó
	I WANT TO	Megváltoztatni a használt algoritmust
	SO THAT	Más algoritmusok vizualizációját tekintsem meg
1	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	A legördülő menüben a BFS algoritmust választom
	THEN	Az útvonaltervezés a BFS algoritmussal fog történni
2	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	A legördülő menüben a Mohó algoritmust választom
	THEN	Az útvonaltervezés a Mohó algoritmussal fog történni
3	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	A legördülő menüben az A* algoritmust választom
	THEN	Az útvonaltervezés A* algoritmussal fog történni
4	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	A legördülő menüben a Dijkstra algoritmust választom
	THEN	Az útvonaltervezés a Dijkstra algoritmussal fog történni

3.1. táblázat. Felhasználói történet: algoritmus választása

2	AS A	Felhasználó
	I WANT TO	Kiválasztani az tervezendő útvonalat
	SO THAT	Megtekinthetem az útvonaltervezést a választott útvonalon
1	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	Az "indulási állomás" mezőbe beírom egy állomás nevét
	THEN	Az útvonaltervezés a kiválasztott állomástól fog indulni
2	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	Az "érkezési állomás" mezőbe beírom egy állomás nevét
	THEN	Az útvonaltervezés a kiválasztott állomást fogja megkeresni
3	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	Az "indulási idő" mezőben kiválasztok egy időt
	THEN	Az útvonal első járata a kiválasztott idő után fog indulni

3.2. táblázat. Felhasználói történet: útvonal kiválasztása

3	AS A	Felhasználó
	I WANT TO	Megváltoztatni a használt algoritmus paramétereit
	SO THAT	Más paraméterekkel tekintsem meg az útvonaltervezést
1	GIVEN	A "beállítások" fül van kiválasztva
	AND	Az A* algoritmus van kiválasztva
	WHEN	A "heurisztika súlya" mezőben módosítom a súlyt
	THEN	Az útvonaltervezés a választott súlyt fogja használni
2	GIVEN	A "beállítások" fül van kiválasztva
	WHEN	A "gyalogos távolság" mezőben módosítom a távolságot
	THEN	Az algoritmus által tervezett útvonal részeként nem fog szerepelni két járat között olyan átszállás, ami a megadottnál nagyobb távolságú

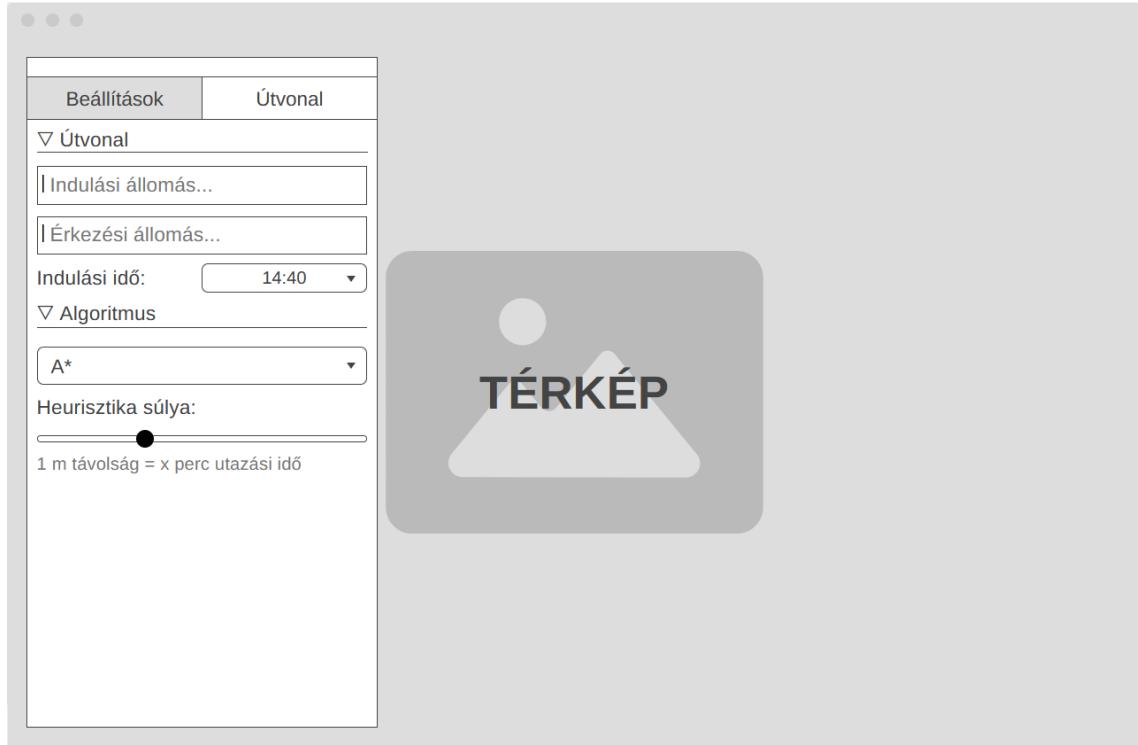
3.3. táblázat. Felhasználói történet: algoritmus paramétereinek módosítása

4	AS A	Felhasználó
	I WANT TO	Interaktívan megtekinteni a tervezett útvonalat
	SO THAT	Láthatom az útvonaltervezés lépésein
1	GIVEN	Az "útvonal" fül van kiválasztva
	AND	Válaszottam indulási állomást, érkezési állomást és indulási időt
	AND	Az algoritmus még nem talált utat a kiválasztott állomások között
	WHEN	A "következő lépés" gombra kattintok
	THEN	Az algoritmus elvégzi a következő lépését
2	GIVEN	Az "útvonal" fül van kiválasztva
	AND	Válaszottam indulási állomást, érkezési állomást és indulási időt
	AND	Az algoritmus még nem talált utat a kiválasztott állomások között
	WHEN	Az "animáció indítása" gombra kattintok
	THEN	Az algoritmus addig fut, amíg nem találja meg a cél állomást, vagy nem szüneteltetem
3	GIVEN	Az "útvonal" fül van kiválasztva
	AND	Az algoritmus az "animáció indítása" gombra való kattintás hatására fut
	WHEN	Az "animáció szüneteltetése" gombra kattintok
	THEN	A program további interakció nélkül nem végez több lépést
4	GIVEN	Az "útvonal" fül van kiválasztva
	AND	Az algoritmus talált utat a kiválasztott állomások között
	WHEN	A "visszaállítás" gombra kattintok
	THEN	Az algoritmus "elfeleji" a talált utat és megállókat, és az alapállapotába áll

3.4. táblázat. Felhasználói történet: Algoritmus irányítása

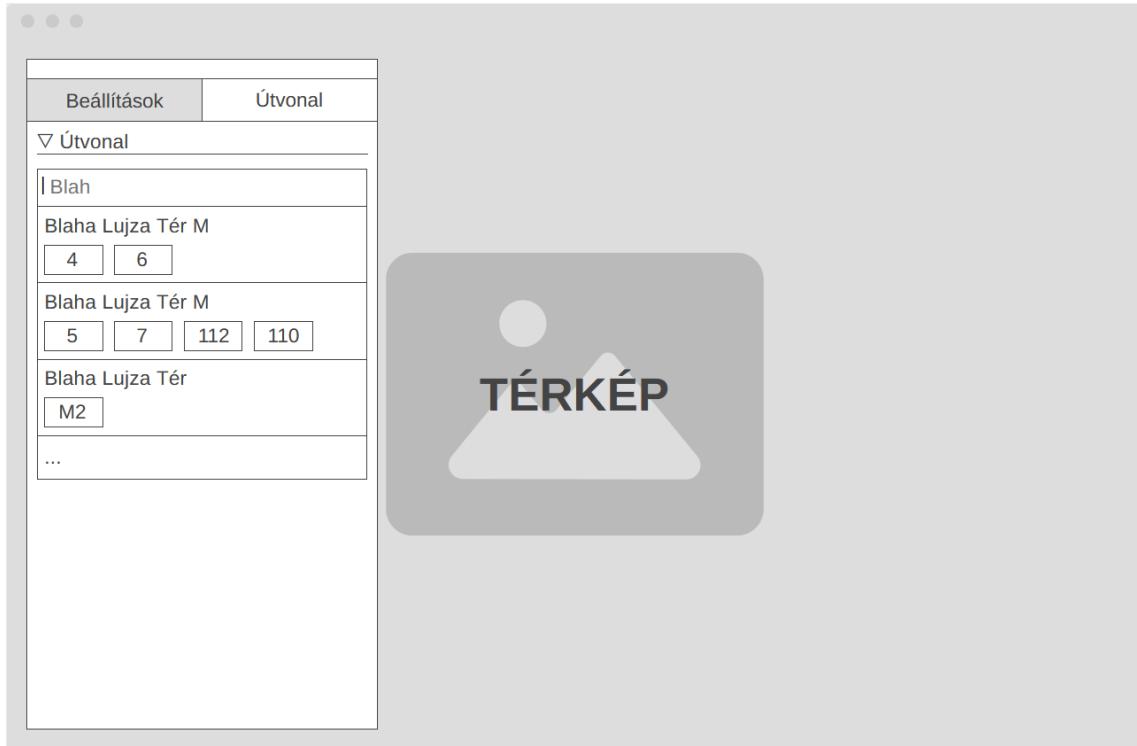
3.3.5. Felhasználói felület tervez

Az alkalmazás nyitóoldala a 3.2. ábrán látható.



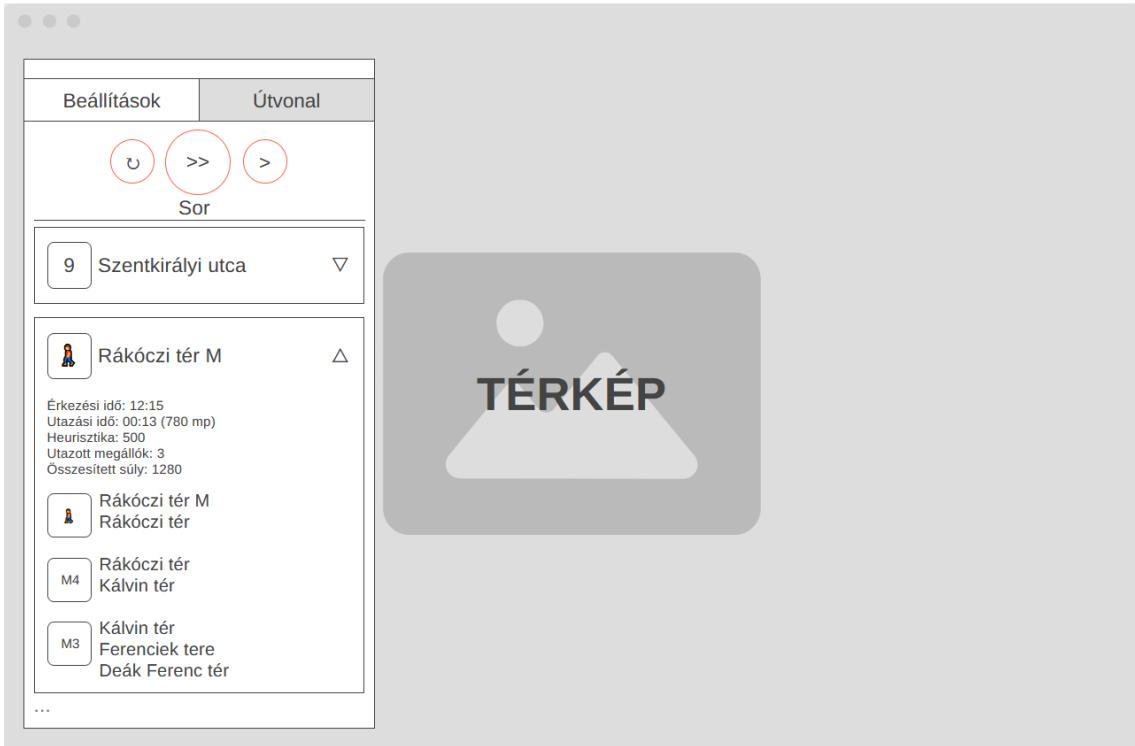
3.2. ábra. Beállítások felület terve

Innen az "indulási állomás" és az "érkezési állomás" mezőkbe beírva egy állomás nevének a részletét rákereshetünk arra, és kiválaszthatjuk azt. A keresési találatok megjelenítését a 3.3. ábra mutatja be.



3.3. ábra. Megálló keresésének felület terve

Ezen állomások kiválasztása után az "indulási idő" mezőben kiválaszthatjuk az indulási időt (vagy az alapértelmezettet használhatjuk), majd az "útvonal" gombra kattintva a 3.4. ábrán látható felületen irányíthatjuk az algoritmus futását, és tekinthetjük meg annak állapotát és eredményét.



3.4. ábra. Algoritmus irányításának felület terve

3.4. Magas szintű áttekintés

3.4.1. Alkalmazás felépítése

Az alkalmazás egy backendből és egy frontendból áll, REST API-n keresztül kommunikálnak egymással. A backend feladata a GTFS formátumban elérhető adatok adatbázisba való betöltése, valamit ezen adatok szolgáltatása a frontend számára. A frontend egy webalkalmazás, mely a felhasználói felületet biztosítja a felhasználók számára.

Fontos megemlíteni, hogy az útvonal tervezése és az algoritmusok futtatása a frontenden történik. Azért választottam ezt a megoldást, hogy az API-n átvitt adatok komplexitását minimalizáljam; mivel a frontendnek egyébként is szüksége van az összes információra az algoritmus belső állapotáról, így a számításokat a frontendre helyezve elég az adatbázis-lekérdezéseket és azok eredményét kommunikálni a kettő között.

3.4.2. Verziókövetés

A változtatásokat *git* használatával tartom számon, így a fejlesztés során bármikor visszaállítható egy korábbi verzió, vagy összehasonlítható két verzió közötti különbség.

3.4.3. Backend áttekintés

A backend egy *Node.js* alapú alkalmazás, mely az *Express.js* keretrendszeret használja a REST API megvalósítására. Fontos tényező volt a környezet kiválasztásában, hogy az NPM¹-en megtalálható *node-gtfs*[4] csomag egyike volt a kevés elérhető könyvtáraknak², amelyek képesek GTFS adatok adatbázisba való betöltésére és lekérdezésére. További előnye a *Node.js* backend választásának, hogy a frontenddel azonos a fejlesztői környezet, így a fejlesztéshez nem kell új programokat telepíteni, és a frontend és a backend fejlesztése közötti váltást is egyszerűvé teszi.

Hogy a backend akár távoli szerveren is egyszerűen beindítható legyen a teljes tesztkörnyezet reprodukálása nélkül, az alkalmazást Dockerizáljuk; így egy `git clone [repo] && docker compose up -d` parancssal bárhol egyszerűen futtatható a backend (ahol a Docker és a git telepítve van).

Az olvashatóság és karbantarthatóság érdekében a backend kódja *TypeScript* nyelven íródik.

3.4.4. Frontend áttekintés

A frontend egy *React* webalkalmazás, mely a backendhez hasonlóan *TypeScript* nyelven van írva. A *React* egy komponens alapú könyvtár, melynek segítségével a felhasználói felületet kisebb, önállóan működő komponensekre bonthatjuk, így a kód olvashatóbb és karbantarthatóbb lesz. Azért erre esett a választásom Angular és Vue helyett, mert a React népszerűsége messze túlszárnyalja ezekét[5], így a fejlesztők számára könnyen elérhetőek a segédanyagok és a közösségi támogatás is.

Az utak térképen való megjelenítéséhez a két fő lehetőség a *deck.gl*, és az erre épülő[6] *kepler.gl*, amit az Ubernél fejlesztettek nagy volumenű utazási adat megjelenítésére. A döntésem a *deck.gl*-re esett, mert egyszerű utak és megállók megjele-

¹Az NPM egy csomagnyilvántartás JavaScript csomagoknak, saját állításuk szerint a világ legnagyobb csomagnyilvántartása[3]

²A GTFS adatok feldolgozására való könyvtárak listája megtalálható a <https://gtfs.org/resources/gtfs/> oldalon (Letöltés dátuma: 2024.11.22.)

nítésére szükségtelen a *kepler.gl* komplexitása, illetve a *deck.gl* dokumentációját is részletesebbnek és könnyebben érhetőnek találtam. A React választása melletti érv volt az is, hogy a *deck.gl* a Reacthoz biztosít előre elkészített komponenseket (más könyvtárakkal ellentétben), így a két technológia jól egymásra épül.

3.4.5. Fejlesztői környezet felállítása

Az alkalmazás fejlesztéséhez a következő programok telepítése szükséges:

- *Node.js* (és *npm csomagkezelő*): Bár a backend és a frontend is futtatható Dockerben, az Intellisense számára érdemes a host gépen is telepíteni a használt csomagokat.
- *Docker*
- *git*
- *IDE/szövegszerkesztő*: Én a Visual Studio Code-ot használom és javaslom, de használható más IDE (pl. WebStorm) is.

Függőségek telepítéséhez a backend és a frontend mappákban a `npm install` parancsot kell futtatni.

3.4.6. Alkalmazás futtatása

Az alkalmazást fejlesztéshez is Dockerben futtatjuk, hogy a program írásakor feltételezhessük, hogy minden ugyanaz a környezet áll rendelkezésre. A Dockerfile a frontend és a backend esetében egyaránt négy stage-et tartalmaz:

1. **base**: Az alap image, amely node:lts-alpine-t használ. Erre épül az összes többi stage.
2. **development**: Az `npm run dev` parancsot futtatja. Ennek a futtatásakor az alkalmazás figyeli a fájlrendszerét³, és újraindítja a kódot minden alkalommal, amikor egy fájl frissül (frontenden csak azokat a komponenseket, amiket érinti a frissítés — ezt HMR-nek, azaz Hot Module Replacement-nek hívják[7]).
3. **build**: Az `npm run build` parancsot futtatja, ami JavaScript-re fordítja a TypeScript kódot.
4. **production**: A `build`-ből lemásolja a lefordított kódot és futtatja azt.

A stage-ek külön bontásának köszönhetően a `production` image a lehető legkevesebb lesz, és a Docker építéskor gyorsítótárban tudja tárolni a lépések közös elemeit.

³A fájlrendszer figyeléséről frontend esetén a *vite*, backend esetén a *tsx* gondoskodik.

A különválasztott fejlesztői és a telepítési környezetnek megfelelően két külön Docker Compose fájl is található a projektben: a `docker-compose.debug.yml` a fejlesztői környezetet állítja fel, míg a `docker-compose.yml` a telepítési környezetet. Az indításkor az ennek megfelelő parancs használandó (3.1).

```

1 # Fejlesztői környezet indítása
2 docker compose -f docker-compose.debug.yml up -d --build
3
4 # Telepítési környezet indítása
5 docker compose up -d --build

```

3.1. forráskód. Alkalmazás indítása különböző konfigurációkban

Bár telepítési környezethez minden adott, hogy Dockerből futtatható legyen az alkalmazás minden két része, a frontend statikusan kiszolgálható fájlokra fordul le, így ezt nem érdemes egy Docker konténerben futtatni. Ehelyett javasolt az `npm run build` által a `dist` könyvtárba lefordított fájlokot egy webszerveren keresztül kiszolgálni, például Nginx vagy Apache HTTP szerver segítségével.

3.5. Backend

3.5.1. Könyvtárszerkezet

A backend könyvtárszerkezete a következő:

- `src/`: A forráskódokat tartalmazó könyvtár, aminek a tartalma fordításra kerül.
- `src/configs/`: Futásidőben használt konfigurációs fájlok helye.
- `src/models/`: Adatbázis sémák helye.
- `src/routes/`: A REST API végpontokat tartalmazó fájlok.
- `src/utils/`: Segédfüggvények adatok betöltésére és lekérdezésére.
- `src/index.ts`: Az alkalmazás belépési pontja.
- `test/`: Teszteléshez szükséges fájlok.
- `.dockerignore`: `.gitignore`-hoz hasonlóan a Docker által figyelmen kívül hagyott fájlok listája.
- `Dockerfile`: A Docker image felépítéséhez szükséges fájl.
- `drizzle.config.ts`: A *drizzle-kit* konfigurációs fájlja (részletek: 3.5.2).
- `package.json`: A függőségeket és egyéb metainformációkat tartalmazó fájl.

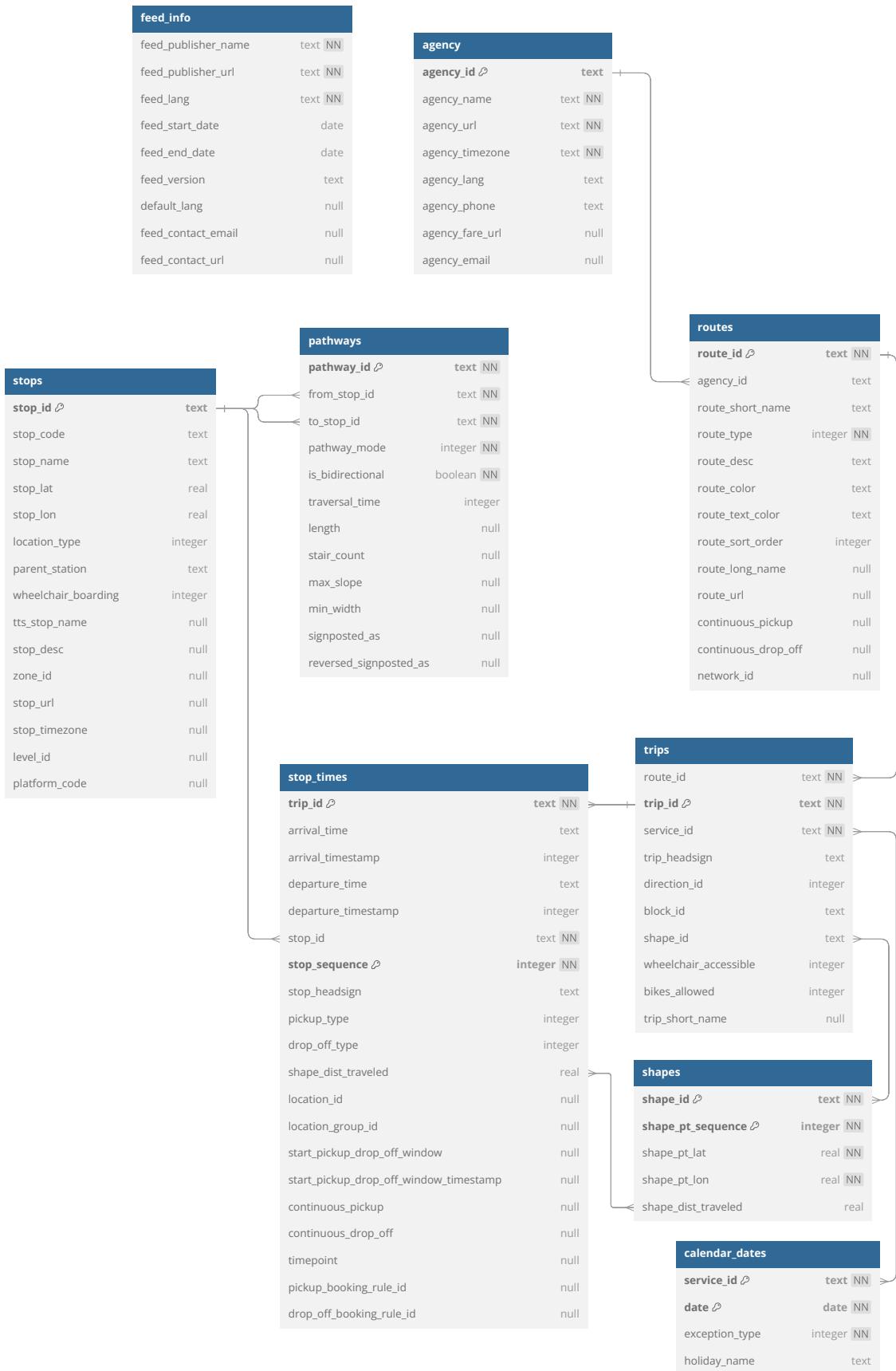
- `package-lock.json`: Az `npm` csomagkezelő által generált fájl, mely a telepített függőségek pontos verzióit tartalmazza.
- `generate-types.ts`: Segédscript, mely az adatbázis sémából TypeScript típusokat generál a frontend számára.
- `tsconfig.json`: A TypeScript konfigurációs fájl.

3.5.2. Adatbázis

A program indításakor az első lépés az adatbázis létrehozása és feltöltése a GTFS adatokkal. Az adatok betöltésére és lekérdezésére a korábban említett `node-gtfs` csomagot használom. (NPM-en és kódban egyszerűen `gtfs`-nek hívják, korábban a GitHub-on szereplő nevét használtam a szabványtól való megkülönböztetés végett. A továbbiakban kisbetűvel írva, `gtfs`-ként fogok hivatkozni rá.)

A `gtfs` a GTFS adatokat egy SQLite adatbázisba tölti be, melynek sémája a GTFS specifikációban meghatározott táblákat tartalmazza (3.5).

3. Fejlesztői dokumentáció



3.5. ábra. GTFS adatbázis sémája

Megjegyzés: A teljes GTFS szabvány ennél jóval több táblát tartalmaz, de a legtöbbjük opcionális. A 3.5. ábra csak azokat a táblákat tartalmazza, amelyeket a BKK OpenData Portálán elérhető adatok tartalmaznak. Ezen táblákban is vannak olyan opcionális mezők, amelyekről a BKK nem szolgáltat adatokat. Ezek a mezők a teljesség kedvéért szerepelnek a sémaiban, de null típussal vannak jelölve.

A táblák a következő információkat tartalmazzák[8]:

1. *agency*: A közlekedési társaságok adatait (pl. weboldal) tartalmazza. Ebben a táblában a BKK és a MÁV-HÉV adatai szerepelnek.
2. *feed_info*: Az adatbázis verziószámát és az érvényességi időszakot tartalmazza (ez itt letöltés napjától az év végéig tart).
3. *routes*: Ez a tábla tartalmazza a járatokat (pl. 4-es villamos, 9-es busz).
4. *trips*: Ez a tábla tartalmazza a járatok útjait — ha egy járat óránként közlekedik 9:00-tól 20:00-ig, akkor 24-szer fog szerepelni ebben a táblában: 12 oda- és ugyanennyi visszaút minden egyike egyedi azonosítóval.
5. *stop_times*: minden *trips*-ben szereplő út egyes megállóit tartalmazza, az érkezési és indulási időpontokkal. Ez a legnagyobb tábla, jelenleg közel 6 millió rekorddal.
6. *calendar_dates*: Arról tartalmaz információt, hogy melyik járat melyik napon lett szolgálatba állítva, illetve kiállítva. Ezt a táblát nem használjuk⁴.
7. *shapes*: Az egyes járatok útvonalát tartalmazza pontokban; koordinátárokat tárol, amelyeket összekötve az adott járat pontos útvonalát kapjuk a térképen. A frontend ezeket az adatokat használja az útvonalak megjelenítésére.
8. *stops*: A megállók adatait tartalmazza (pl. nevük, koordinátájuk).
9. *pathways*: Megállók közti átjárókat tartalmaz (pl. aluljárók). Ezt a táblát sem használjuk — csak néhány átjáró van benne (jelenlegi adatok szerint 6000-nél több megállóra 500-nál kevesebb átjáró), így önmagában nem lenne elég

⁴A program célja nem pontos információk szolgáltatása, hanem algoritmusok demonstrálása egy ismert környezetben. Ha a cél pontos menetrendi információk megjelenítése lenne, akkor valós idejű adatokat is figyelembe kell venni, ami jelentősen növelné a program bonyolultságát, és nem tartozik a dolgozat témájába.

információ az egy csoportba tartozó megállók összekötésére. Egyszerűbb és a felhasználó számára is következetesebb bármelyik 2 megálló közötti gyaloglást azonosan kezelní.

Az adatbázis a Docker Compose fájlban mountolt *data* mappába kerül, amely a *frontend* és a *backend* mappák mellett foglal helyet. Az alkalmazás induláskor ellenőrzi, hogy az adatbázis létezik-e (illetve sikeres-e egy lekérdezés), és ha nem, akkor betölti az adatokat a GTFS adatokból. Az adatforrás a *src/configs/gtfs.config.ts* fájlban állítható be.

A gtfs könyvtár a hivatalos specifikáción felül is tartalmaz néhány segédoszlopot, amelyek a gyors lekérdezést segítik. Ezek az eredeti adatokban "óra:perc:másodperc" formátumban szereplő időpontokat egész számokká alakítják. Azonban a GTFS specifikáció szerint az éjfél után közlekedő járatok időpontjai átléphetik az "24:00:00" időpontot, amit a gtfs könyvtár nem kezel helyesen. Így betöltés után az időpontokat a programnak újra kell számolnia, hogy a betöltött adatbázisban ne null értékek legyenek.

Amint az adatbázis betöltése megtörténik, az alkalmazás GeoJson fájlokat generál a frontend számára, melyek úgyszintén a *data* mappába kerülnek (3.6). Ezek a fájlok tartalmazzák a megállók és az útvonalak geometriáját[9], amelyeket a frontend a térképen megjelenít.

```
data
|- db.sqlite
-- public
|- shapes.geo.json
-- stops.geo.json
```

3.6. ábra. A *data* mappa szerkezete az adatok betöltését követően

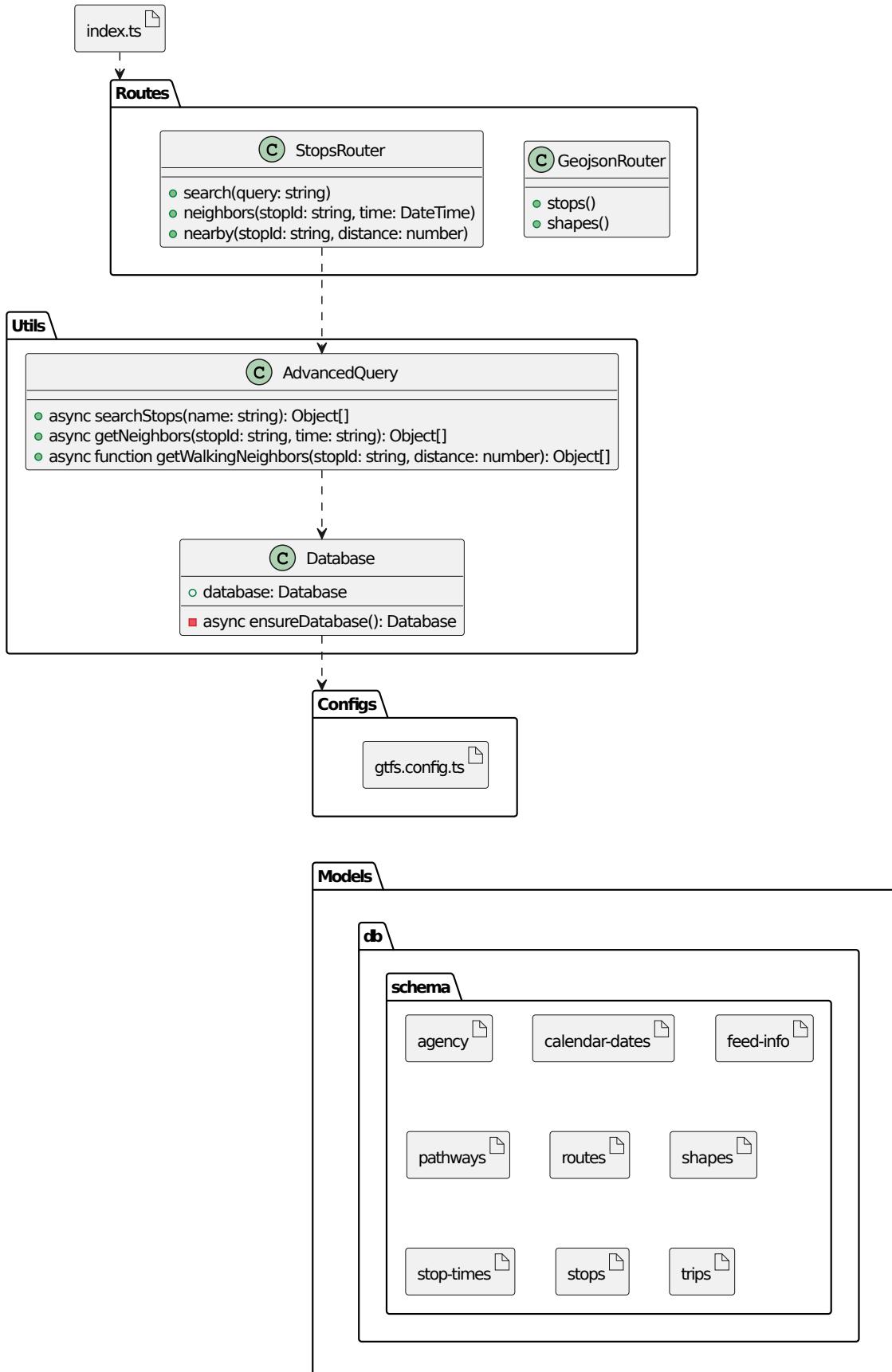
A GeoJson fájlok generálásán kívül a többi lekérdezéshez (például megállók kereséséhez) a gtfs könyvtár nem biztosít megfelelő eszközöket, így saját lekérdezéseket kell írni. Erre a *Drizzle ORM* nevű könyvtárat használom, amely egy egyszerű ORM⁵ az SQLite adatbázishoz. Azért erre esett a választásom például *Prisma* helyett, mert a Drizzle csak egy vékony réteget biztosít az SQL felett[10], így több irányításunk van a pontos lekérdezések felett, és nem kell a könyvtár által generált SQL kódot megérteni, ha egy lekérdezést optimalizálni szeretnénk. Emellett teljesítménytesztek en-

⁵Object-Relational Mapping, egy programozási technika, amely az objektumorientált programozást és a relációs adatbázisokat kapcsolja össze

is jobban teljesített a fent említett Prisma-nál[11]. Az előnye a nyers SQL írásával szemben, hogy a Drizzle TypeScript segítségével biztosítja a típusbiztonságot, így egyszerűbb és biztonságosabb a lekérdezések írása.

3.5.3. Szerkezet

A backend osztályait és azok közötti kapcsolatokat a 3.7. ábra mutatja be.



3.7. ábra. Backend szerkezete

3.5.4. REST API

Az alkalmazás REST API-jának végpontjai a következők:

- GET `/data/stops.geo.json`: GeoJson fájl, amely a megállók koordinátáit tartalmazza.
- GET `/data/shapes.geo.json`: GeoJson fájl, amely az útvonalak koordinátáit tartalmazza.
- GET `/stops/search`: Megállók keresése név alapján.
- GET `/stops/:stopId/nearby`: Egy megállótól adott távolságon belüli megállók lekérdezése.
- GET `/stops/:stopId/neighbors`: Egy megállótól az adott időpont utáni 1 órás intervallumban induló járatokkal elérhető megállók lekérdezése.

Teljes dokumentáció sémával és példákkal a *BKWay.yml* fájlban, OpenAPI 3 formátumban található.

3.5.5. Tesztelés

A backend tesztelésénél a cél az API végpontok kimerítő tesztelése volt. A tesztek futtatásához *Postman*-t alkalmaztam.

A teszteléshez egy külön gtfs adatbázist használtam, mely a BKK GTFS adatbázisának egy részét tartalmazza. Erre azért volt szükség, mert az éles adatbázis bármikor változhat, és a tesztek konkrét megállók azonosítóira hagyatkoznak. Az adatbázist a backend *test* mappájában található *generate_sample.sh* segítségével generáltam egy kicsomagolt GTFS-ből.

A tesztelés folyamata a következő:

1. Backend konfigurálása teszt adatbázis használatára:

- (a) a `src/configs/gtfs.config.ts` fájlban az adatforrás átállítása a teszt zip fájlra az ott leírtak szerint
- (b) a `data` mappában a `db.sqlite` fájl törlése (ha létezik)
- (c) a `docker-compose.debug.yml` indítása (3.4.6)

2. A Postman Collection importálása a *backend/test* mappából Postman-be az itt leírtak szerint
 - Az importáláshoz Postman fiók létrehozása szükséges.
3. A tesztek futtatása a Postman-ben.
 - Az importált collection-t megnyitva, jobb felül található a "Run" gomb.

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	1s 617ms	75	7 ms

Test Results:

- GET neighbors test 404**
http://127.0.0.1:3333/stops/hihi/neighbors?time=12:00:00
 - PASS Status code is 404
 - PASS Response body contains message in JSON format
- GET neighbors test bad format**
http://127.0.0.1:3333/stops/007884/neighbors?time=ab:cd:ef
 - PASS Status code is 404
 - PASS Response body contains message in JSON format
- GET nearby test 1**
http://127.0.0.1:3333/stops/F00954/nearby?distance=0
 - PASS Response status code is 200
 - PASS Response time is less than 5s
 - PASS Content type is application/json
 - PASS Response has the required fields
 - PASS Response is empty
 - PASS Response body matches OpenAPI schema
 - PASS Response contains the correct stops
- GET nearby test 2**
http://127.0.0.1:3333/stops/F00954/nearby?distance=50

3.8. ábra. Sikeres teszteredmények

A tesztek többek között a következőket ellenőrzik:

1. A válasz gyorsasága
2. A válasz státusz kódja (érvénytelen üzenetek esetén is, pl. 400 vagy 404)
3. A válaszban szereplő pl. megállók száma
4. A válasz megfelelése az OpenAPI sémának

3.6. Frontend

A frontend az algoritmusok futtatásáért felelős, és a felhasználói felületet biztosítja a felhasználók számára.

3.6.1. Definíciók

Csúcs: Egy gráf csúcsa egy pont a gráfban. Egy csúcsnak lehetnek szomszédos csúcsai, amelyekkel élek kötik össze. Ebben az alkalmazásban minden csúcs egy megállónak felel meg. A továbbiakban a két fogalmat szinonimaként használom.

Él: Egy gráf élé egy csúcsból egy másikba mutató irányított vagy irányítatlan kapcsolat. Irányított élek esetén a két csúcs közül az egyik a kezdőpont, a másik a végpont. Egy él egy utazást reprezentál, amely egy megállóból egy másikba vezet. Az élek lehetnek súlyozatlanok és súlyozottak. Amennyiben súlyozottak, minden élhez egy súly van rendelve, ami a két csúcs közötti utazási időt jelenti. A továbbiakban "él" alatt irányított, súlyozott élt értek. Az alkalmazás szempontjából további két csoportba sorolhatók az élek: utazási élek és gyaloglási élek.

Utazási él: Két megálló között közlekedő járatokat reprezentáló él. Az utazási élek súlya magában foglalja a járatra való várakozási időt is.

Gyaloglási él: Két megálló közötti gyaloglást reprezentáló él. A gyaloglási élek is irányítottak.

Gráf: Egy gráfot $G = (V, E)$ párként definiálunk, ahol V a csúcsok halmaza, E pedig az élek halmaza. Egy gráf a BKK utazási hálózatát reprezentálja.

Út: Egy út egy gráfban két csúcs közötti élek sorozata. Az út bármelyik két egymást követő élre igaz, hogy ami az első él végpontja, az a másik él kezdőpontja. Az út hossza a súlyozott élek összege.

Útvonaltervezés: Az útvonaltervezés (vagy útkeresés) egy gráfban két csúcs (az induló- és a célpont) között keres utat.

Algoritmus: "Algoritmus" alatt útvonaltervező algoritmusokat értek, azaz olyan utasítássorozatokat, amelyeknek a célja egy kezdő- és egy végpont közötti út megtalálása.

Heurisztika: A heurisztika egy olyan $h(n)$ függvény, amely egy algoritmus számára egy becslést ad egy adott csúcsból (n) a célig vezető legrövidebb út hosszáról[12].

3.6.2. Algoritmusok

Az algoritmusok két fő csoportba sorolhatók: informált és informálatlan algoritmusok[12]. Az informálatlan algoritmusok keresés közben nem veszik figyelembe a célt, csak a kezdőpontot és a gráfot járják be. Az informált algoritmusok a célt is figyelembe veszik, és ennek megfelelően működnek.

BFS (Breadth-First Search) útkeresés

A BFS algoritmus egy ún. *sor* adatszerkezetet használ[12], ami a következő műveleteket támogatja:

- **push(elem)**: Egy elem hozzáadása a sor végéhez.
- **pop()**: Az első elem kivétele a sor elejéről.

Az útkeresés a kiindulócsúcsban kezd, és a szomszédos csúcsokat egy sorba helyezi. Ezt követően minden lépéssében kivesz egy csúcsot a sor elejéről, majd annak szomszédjait a sor végére helyezi. Ennek eredményeként először azokat a csúcsokat fedezi fel, amik 1 élre vannak a kiindulóponttól, majd azokat, amikhez 2 megállót kell utazni, és így tovább[12]. Az algoritmus addig fut, amíg el nem éri a célt, vagy nincs több csúcs a sorban.

A BFS nem tesz különbséget a különböző súlyú élek között, így általában nem szokás súlyozott gráfokon alkalmazni[12].

Az algoritmus pszeudokódja a következő:

```
1 function BFS(graph, start, goal):
2     queue = [start]
3     visited = set()
4
5     while queue:
6         current = queue.pop()
7         if current == goal:
8             return current
9         visited.add(current)
10        for neighbor in graph[current]:
11            if neighbor not in visited:
12                queue.append(neighbor)
```

3.2. forráskód. BFS algoritmus pszeudokódja

Dijkstra algoritmus

A Dijkstra algoritmus egy súlyozott gráfokon való legrövidebb út[12] keresésére alkalmas algoritmus. Az algoritmus az indulási csúcsnál kezd, és a szomszédos csúcsokat egy prioritási sorban tárolja. Az algoritmus minden szomszédos csúcsot meglátogat, majd a prioritási sorból a legkisebb súlyú csúcsot veszi ki és annak szomszédjait tárolja. Mivel mindenkor az aktuális legkisebb súlyú csúcsot veszi ki, és egy csúcs súlya annak az időben mért távolsága az indulóállomástól, az algoritmus garantálja, hogy a célhoz vezető út a legrövidebb lesz (hiszen ha létezne rövidebb út a találtnál, azt már megtalálta volna korábban). Az algoritmus addig fut, amíg el nem éri a célt, vagy nincs több szomszédos csúcs a prioritási sorban.

Az algoritmus egy prioritási sor adatszerkezetet használ, ami a következő műveleteket támogatja:

- **insert(elem, weight):** Egy elem hozzáadása a sorba adott súllyal.
- **extract_min():** A legkisebb elem kivétele a sorból.

A csúcs távolságát az indulóállomásból $g(n)$ jelöli (ez az indulóállomás esetében értelemszerűen 0). Új csúcs súlyát egy $f(n)$ -nel jelölt kiértékelőfüggvény számítja ki. mivel az algoritmus csak a távolságot veszi figyelembe, a Dijkstra algoritmusnál a kiértékelőfüggvény $f(n) = g(n)$.

Az algoritmus pszeudokódja a következő:

```

1 function Dijkstra(graph, start, goal):
2     pq = PriorityQueue()
3     pq.insert((start, 0))
4     visited = set()
5
6     while pq:
7         current, current_distance = pq.extract_min()
8         if current == goal:
9             return current
10        visited.add(current)
11        for neighbor in graph[current]:
12            if neighbor not in visited:
13                new_distance = current_distance + distance(current,
14                                                neighbor)
15                pq.insert((neighbor, new_distance))

```

3.3. forráskód. Dijkstra algoritmus pszeudokódja

Mohó algoritmus

A mohó algoritmus a Dijkstra algoritmusnak egy olyan változata, amely pusztán egy heurisztika alapján végzi az útkeresést[12]. Ennek a kiértékelőfüggvénye $f(n) = h(n)$, ahol $h(n)$ a heurisztika értéke az adott csúcsra — ez azt jelenti, hogy az algoritmus pusztán heurisztika alapján dönti el, hogy melyik csúcsot válassza következőnek.

A programban használt heurisztika a két pont közötti távolság a Föld⁶ felszínén, egy egyenes vonalban. Ez jó választás[12], hiszen

1. Ennél rövidebb út nem létezhet, így a heurisztika minden alulbecsüli a távolságot, és
2. Egyetlen képlet segítségével, gyorsan kiszámítható.

Ez az algoritmus a legtöbb esetben sokkal gyorsabban találja meg a célt, mint a Dijkstra algoritmus, viszont nem garantálja, hogy a talált út a legrövidebb lesz — sőt, egy olyan hálózaton, mint a tömegközlekedés, az utazási idő figyelmen kívül hagyása a tapasztalataim szerint még többet ront az eredmény minőségén, mint például egy gyalogos útvonaltervezésnél.

⁶A program implementációjában a számításokhoz a *distance-from* csomag szolgáltatja a távolság számításához szükséges függvényeket. Ez a könyvtár a Földet egy 6371 méter sugarú gömbként modellez[13].

```

1 function Greedy(graph, start, goal):
2     pq = PriorityQueue()
3     pq.insert((start, 0))
4     visited = set()
5
6     while pq:
7         current, current_distance = pq.extract_min()
8         if current == goal:
9             return current
10        visited.add(current)
11        for neighbor in graph[current]:
12            if neighbor not in visited:
13                -           new_distance = current_distance + distance(current,
14                neighbor)
14                -           pq.insert((neighbor, new_distance))
15                +           pq.insert((neighbor, heuristic(neighbor)))

```

3.4. forráskód. Különbség a Dijkstra és a mohó algoritmus között

A* algoritmus

Az A* algoritmus⁷ a Dijkstra és a mohó algoritmusok kombinációja, amely a Dijkstra algoritmus pontosságát ötvözi a mohó algoritmus gyorsaságával. Ezt úgy éri el, hogy a prioritási sor elemeit a súlyok és a súlyozott heurisztika összege szerint rendezzi.

"Súlyozott heurisztika" alatt egyszerűen a heurisztika és egy W súly szorzatát értem. Az A* kiértékelőfüggvénye tehát a következő[12]:

$$f(n) = g(n) + W \times h(n)$$

A súly értékének változtatásával az algoritmus többet vagy kevesebbet fog támaszkodni a heurisztikára, tehát a súly csökkentéséhez közelebb kerülhetünk a Dijkstra algoritmushoz pontos eredményeihez, míg a súly növelésével a mohó algoritmus gyorsaságához.

Az algoritmus pszeudokódja a következő:

⁷A szakirodalomban az A* algoritmus nem mindig tartalmazza a heurisztika súlyozását, ilyenkor ezt a változatot *súlyozott A**-nak nevezik[12]. Én az egyszerűség kedvéért erre a variánsra egyszerűen A*-ként hivatkozom, a "súlyozatlan" változatot pedig a továbbiakban nem említém meg.

```

1 function Greedy(graph, start, goal):
2     pq = PriorityQueue()
3     pq.insert((start, 0))
4     visited = set()
5
6     while pq:
7         current, current_distance = pq.extract_min()
8         if current == goal:
9             return current
10        visited.add(current)
11        for neighbor in graph[current]:
12            if neighbor not in visited:
13                new_distance = current_distance + distance(current,
14                neighbor)
14 -                pq.insert((neighbor, new_distance))
15 +                pq.insert((neighbor, new_distance + heuristic(
16                neighbor)))

```

3.5. forráskód. Különbség a Dijkstra és az A* algoritmus között

3.7. Implementáció

3.7.1. Algoritmusok

Megfigyelhető, hogy az algoritmusok szerkezetükben nagyon hasonlóak egymás-hoz. A prioritási sor-alapú algoritmusok esetében egyértelmű, hiszen csak egy-két sor változtatásra vannak egymástól, ám a BFS sem különbözik sokban ezektől. Néhány triviális változó- illetve metódus-átnevezés után a 3.6. kódrészletben látható, hogy a BFS és a mohó algoritmus mennyire hasonlít.

```

1
2 -   function BFS(graph, start, goal):
3 +   function Dijkstra(graph, start, goal):
4 -       queue = Queue()
5 +       queue = PriorityQueue()
6
7 -       queue.insert(start)
8 +       queue.insert((start, 0))
9       visited = set()
10
11      while queue:
12 -          current = queue.pop()
13 +          current, current_distance = queue.pop()
14          if current == goal:
15              return current
16          visited.add(current)
17          for neighbor in graph[current]:
18              if neighbor not in visited:
19 -                  queue.insert(neighbor)
20 +                  queue.insert((neighbor, heuristic(neighbor)))

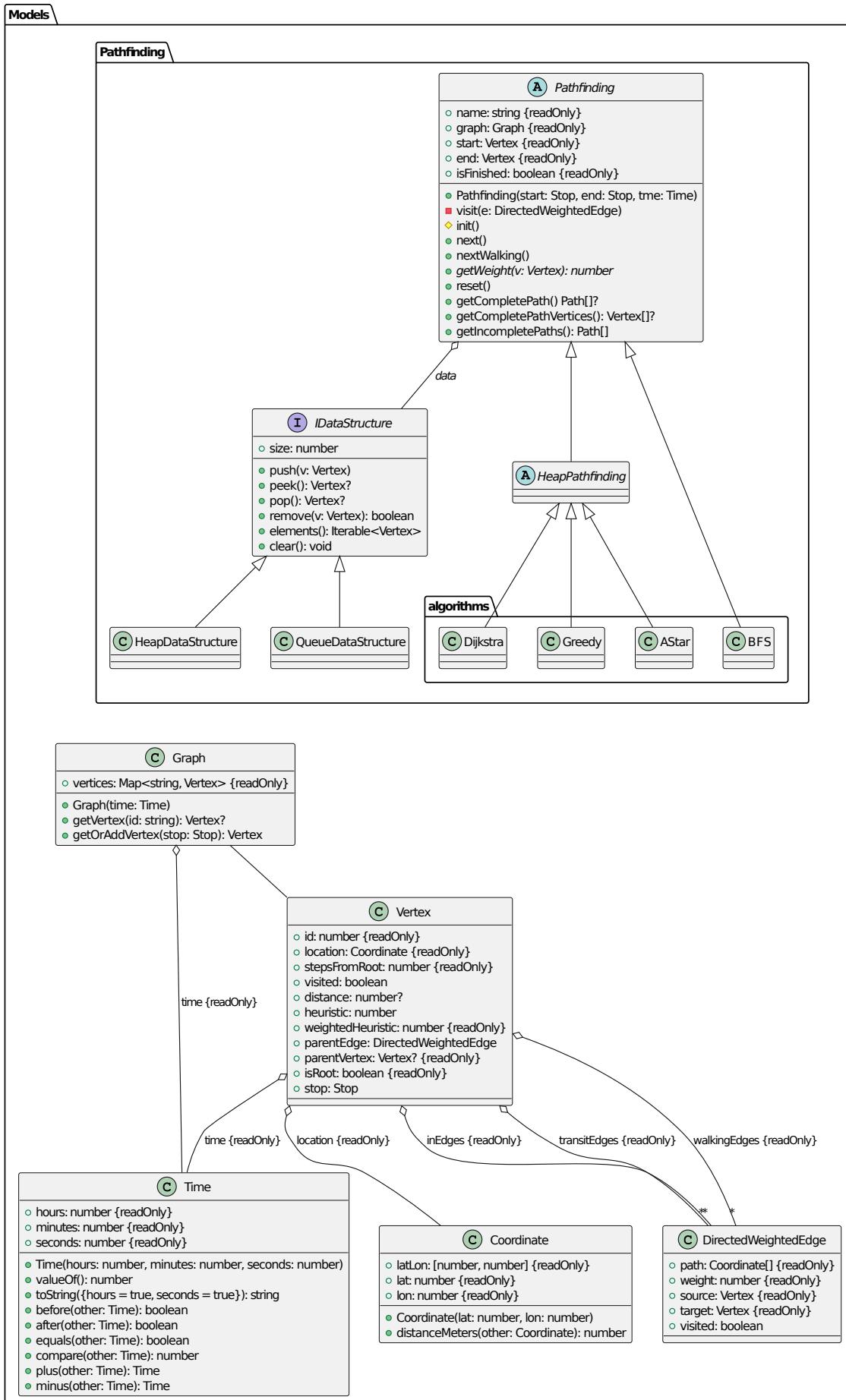
```

3.6. forráskód. Különbségek a Dijkstra és a mohó algoritmus pszeudokódja között

Mint látható, a program folyása teljes azonos köztük, az egyetlen különbség, hogy a prioritási sor alapú algoritmusok súlyt társítanak a sor elemeihez. Ez azt jelenti, hogy akár a BFS algoritmust is át lehetne írni úgy, hogy prioritási sort használjon, ha a minden csúcs súlyát eggyel nagyobbra állítanánk az előzőénél, hiszen így garantáltan a prioritási sor végére kerülne. Ezt egyszerűen el lehetne érni, hiszen az összes ismert csúcs száma soha nem csökken, és a p. sorba⁸ való beillesztéskor minden a p. sor hosszának és a látogatott csúcsok számának az összege.

Bár ez működne, hatékonyiségi megfontolásból nem emellett a megoldás mellett döntöttem. Helyette az algoritmusokat megvalósító absztrakt osztályhoz egy absztrakt *data* adattagot vettem fel. Ennek típusa, az *IDataStructure* interfész néhány alapvető függvényt tartalmaz; többek közt az új elem hozzáadását és az elem kivételét. A heurisztika, forrástól való távolság, és hasonló információk egy csúcsról az útkereső osztály helyett a csúcs (*Vertex*) osztályban kerülnek kiszámításra, így az útkeresés (*Pathfinding*) szabadon használhatja ezeket.

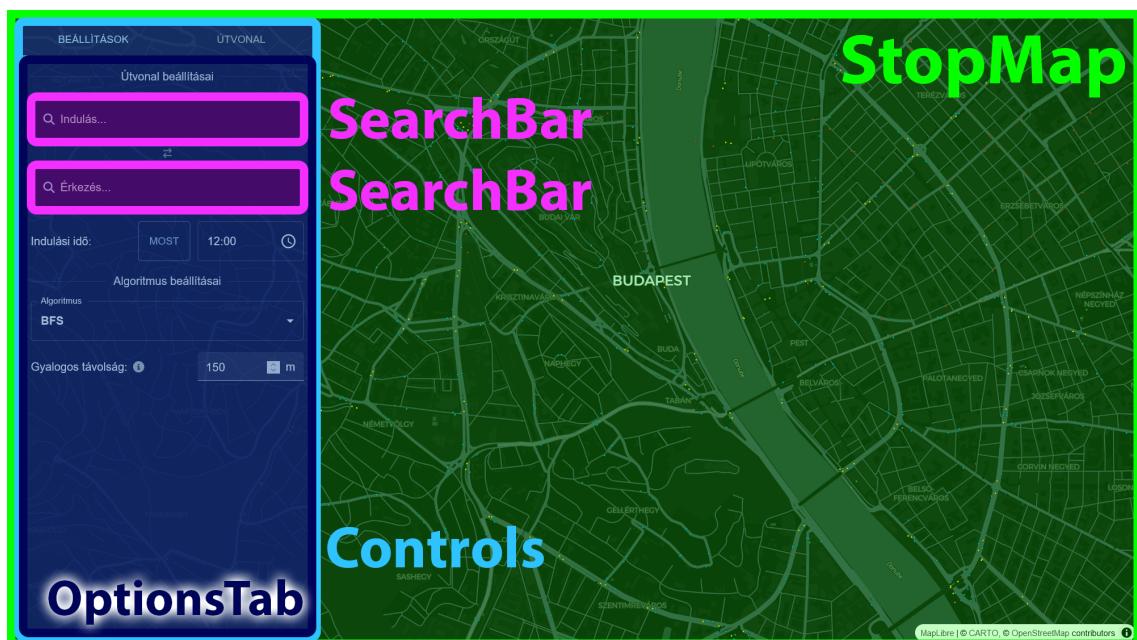
⁸Prioritási sor



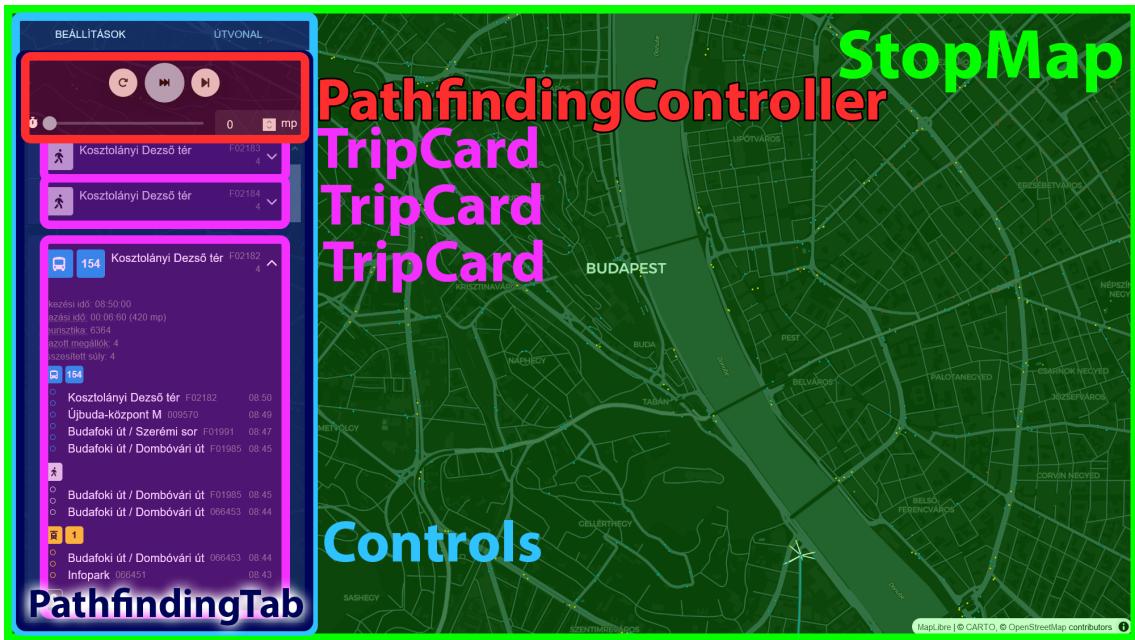
3.9. ábra. Frontend modellek

3.7.2. Felhasználói felület

A felhasználói felület React komponenseket használ, amelyek a *Material UI* React komponens könyvtár komponenseivel épülnek fel. Ez a könyvtár előre elkészített, egységes dizájnelemeket tartalmaz, amelyek a Google Material Design irányelvei alapján készültek[14]. A Material UI-hoz kapcsolódó *MUI X* könyvtár az indulási idő beállításához használt *TimePicker* komponenst biztosítja. A felhasználói felület főbb komponensei a 3.10. és 3.11. ábrákon láthatóak.



3.10. ábra. Beállítások komponense



3.11. ábra. Keresés komponense

Megjegyzés: A StopMap komponenes nem tartalmazza magában a Controls komponenst, de a többi komponens egymásba ágyazása úgy történik, ahogy vizuálisan látható — A Controls foglalja magában a bal oldali irányítópanel összes elemét, ezen belül az OptionsTab és a PathfindingTab komponenseket, melyek a két fül elemeit tartalmazzák.

3.7.3. API hívások

Az API hívásokért az *api.service.ts* felel, amely *fetch* hívásokkal kommunikál a backenddel. A visszatérési típusok a backendben definiált adatbázis sémájából lettek generálva, így megegyeznek a backend által küldött adatok típusával.

3.7.4. Tesztelés

Az útvonaltervezés tesztelése manuálisan történik, néhány példaútvonalon keresztül. minden útvonaltervezést 12:00-kor kezdünk, ha nem szerepel más időpont. A sétálási távolságot is hagyjuk az alapértelmezett 150 méteren, ha nem szerepel más érték.

Útkeresés önmagába

Adjuk meg induló- és érkező állomásnak is ugyanazt az állomást. Erre jó alany a *Csobánc utca*, mert ilyen nevű állomásból csak egy van a térképen. Ekkor egyik algoritmusnak sem szabad elindulnia, és azonnal jelezniük kell, hogy megtalálta a célt.

BFS útkeresés

A BFS nem veszi figyelembe az élek súlyát, csak a megállók számát. Tervezzünk utat a *Népliget* (*F01282*⁹) és a *Kőris utca* (*Korányi Sándor utca*) megállók között 175m gyaloglási távolsággal. Az algoritmusnak hamarabb kell felfedeznie a Kálvin tér-ből induló járatokat, mint megtalálnia a célállomást, hiszen az M3 vonalán ez ugyanúgy 4 megállóra van, mint a 83-as vonalán a Kőris utca (*Korányi Sándor utca*) megálló, és az M3 indulóállomásából tervezünk utat. Az algoritmusnak a 83-as trolival kell eljutnia a célállomásra.

Dijkstra útkeresés

Használjuk a BFS beállításait. A Dijkstra algoritmusnak hamarabb kell hozzáadnia a Deák Ferenc tér (*F00954*) M3 metrómegállót a prioritási sorba, mint megtalálnia az utat, de nem szabad a Kálvin tér többi megállóját meglátogatnia (ami nem az M3 megállója), mert azokhoz több idő sétálni, mint a célállomásra.

Mohó útkeresés

A mohó útvonaltervezés gyakorlatilag bármilyen útvonalon ellenőrizhető, mert látványosan figyelmen kívül hagyja az utazási időt (ellenben a távolsággal). Tervezzünk utat Ráckeve és Szentendre között. Az útkeresés belátható időn belül végezni fog, és nem fedez fel Budapest nagy részét, mielőtt célba ér.

A* útkeresés

Az A* algoritmus tesztelésekor fontos, hogy a súlyozott heurisztika helyesen működjön. Tervezzünk utat a *Margit híd, budai hídfő H* (*F00189*) és a *Petőfi híd, budai hídfő* (*F02224*) megállók között, ahol megáll a 4-es és a 6-os villamos. Az algoritmus

⁹Az indulási- és érkezési megállók azonosítója a kiválasztást követően a böngésző konzolában is megjelenik

$0.3 \times$ heurisztika súly esetén a 4-es vagy 6-os villamossal fogja megtalálni a célt. $0.7 \times$ súly esetén az út Budapest *Kiskörútján* fog áthaladni, részben a 9-es busz vonalán. $2.5 \times$ súly esetén a talált út a 15-ös busz vonalán lévő a *Kossuth Lajos tér M* és a *Petőfi tér* megállókat is tartalmazni fogja.

3.8. Telepítés

A programot az itthoni szerveremen futtatom, jelenleg a <https://bee-612.space/bkway/> címen elérhető. Más szerveren való futtatáshoz a 3.4.6 alatt található használati utasítást kell követni, illetve a következő helyeken a megfelelő változókat beállítani:

- A *docker-compose.yml* fájlban a bkway-back szolgáltatásnál a *BKWAY_FRONTEND* környezeti változót a frontend domainjére állítani,
- a frontend *.env* fájljában a *BKWAY_BACKEND* változót a backend címére állítani,
- végül opcionálisan a frontend *vite.config.ts* fájljában a konfiguráció *base* értékét átállítani. Jelenlegi értéke */bkway/*, ami azt jelenti, hogy a <https://your.domain.com/bkway/> elérési úton lesz kiszolgálva.

3.9. További fejlesztési lehetőségek

Adatbázis lekérdezések gyorsítása

Jelenleg a megállókra való keresésnél az adatbázis lekérdezés költséges táblaösszekapcsolásokat végez el. Bár ezeknek az eredménye gyorsítótárazva van a szerveren, még sokszor lehetnek lassúak a lekérdezések. Mivel az adatbázis nem változik a létrehozása után, nem lenne nehéz egy kapcsolótábla létrehozása, ami a *stops* és a *routes* táblát köti össze. Ezzel azt gondolom, hogy a lekérdezések 10-100-szoros gyorsulást érhetnének el.

Animáció gyorsítása

Jelenleg a frontend az útvonaltervezés minden egyes lépése után újra kiszámítja a forrásból az összes sorban szereplő elembe vezető utat. Ez hosszú sor és folyamatos animáció esetén lassú tud lenni. Ha az algoritmus számon tartaná az egy lépésen

belül történt változtásokat, és csak azokat frissítené, akkor a program jelentősen gyorsabb lenne.

További részletek megjelenítése

Jelenleg a felhasználói felület "ÚTVONAL" fülén csak a felfedezésre váró megállók és az azokhoz tartozó információk jelennek meg. Hasznos információ lenne itt többek között az algoritmus által megtett lépések száma és a már felfedezett megállók (akár kereshető) listája is.

Új algoritmusok hozzáadása

Az A* algoritmus egy gyakori példa hatékony útvonaltervezésre, de elérhetők újabb, modernebb és hatékonyabb algoritmusok is. Ilyen például a RAPTOR[15], amely különösen tömegközlekedési hálózatokra van kifejlesztve, vagy egyszerűen az A* algoritmus súlyozott változata.

Új beállítások hozzáadása

További hasznos beállítások lehetnek az átszállások száma, a tömegközlekedési eszközök szűrése, vagy az akadálymentes útvonaltervezés.

Valós idejű adatok megjelenítése

A BKK Futár API GTFS-Realtime formátumban szolgáltat valós idejű adatokat a járatokról[1], amelyeket fel lehetne használni az útvonaltervezés során.

4. fejezet

Összegzés

Az alkalmazás célja az algoritmusok közérthetőbbé tétele volt, és véleményem szerint ez sikerült is. Az alkalmazás elkészülte után megmutattam néhány olyan ismerősömnek, rokonomnak, akiknek nincs informatikai tapasztalatuk, és kevés magyarázatot követően megértették a bemutatott algoritmusok működési elvét és egyes algoritmusok előnyeit, illetve hátrányait.

Bár a programnak van tere fejlődésre, a használata egyszerű és intuitív lett, és remélem, hogy a jövő informatikusai is hasznosnak fogják találni tanulmányaik során.

Köszönetnyilvánítás

Köszönet a konzulensemnek, Erdősné Dr. Németh Ágnesnek, hogy másodszor is elvállalta a konzulensi feladatot (miután elsőre visszamondtam) és segített a szakdolgozat elkészítésében; és köszönet a páromnak, hogy főzött rám amikor nem volt időm magamra főzni.

Irodalomjegyzék

- [1] Budapesti Közlekedési Központ. *Tervezett menetrendi adatbázis*. <https://opendata.bkk.hu/data-sources>. Letöltés dátuma: 2024.11.15. 2024.
- [2] MobilityData. *GTFS evolution*. <https://gtfs.org/about/#gtfs-evolution>. Letöltés dátuma: 2024.11.22. 2024.
- [3] OpenJS Foundation. *An introduction to the npm package manager*. <https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager>. Letöltés dátuma: 2024.11.22. 2022.
- [4] BlinkTag Inc. *gtfs*. <https://www.npmjs.com/package/gtfs>. Letöltés dátuma: 2024.11.22. 2012.
- [5] Stack Overflow. *Tag Trends - ReactJS, Vue.js, Angular, Svelte, AngularJS, VueJS 3*. <https://trends.stackoverflow.co/?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3>. Letöltés dátuma: 2024.11.22. 2024.
- [6] Uber Shan He. *From Beautiful Maps to Actionable Insights: Introducing kepler.gl, Uber's Open Source Geospatial Toolbox*. <https://www.uber.com/en-HU/blog/keplergl/>. Letöltés dátuma: 2024.11.22. 2018.
- [7] webpack. *Hot Module Replacement*. <https://webpack.js.org/concepts/hot-module-replacement/>. Letöltés dátuma: 2024.11.22. 2016.
- [8] MobilityData. *General Transit Feed Specification Reference*. <https://gtfs.org/documentation/schedule/reference/>. 2024 Szeptember verzió. Letöltés dátuma: 2024.11.22. 2006.
- [9] H. Butler és tsai. *The GeoJSON Format*. RFC 7946. 2016. aug. DOI: 10.17487/RFC7946. URL: <https://www.rfc-editor.org/info/rfc7946>.
- [10] Drizzle. *Drizzle ORM - Documentation*. <https://orm.drizzle.team/docs/rqb/>. Letöltés dátuma: 2024.11.22. 2024.

- [11] Drizzle. *Drizzle ORM - Benchmarks*. <https://orm.drizzle.team/benchmarks/>. Letöltés dátuma: 2024.11.22. 2024.
- [12] S.J. Russell, S. Russell és P. Norvig. *Artificial Intelligence: A Modern Approach*. 4. kiad. Pearson series in artificial intelligence. Pearson Higher Education, 2020. ISBN: 9780134610993. URL: <https://books.google.hu/books?id=koFptAEACAAJ>.
- [13] Ricardo Pierre-Louis. *distance-from*. <https://github.com/rickyplouis/distance-from/blob/5cc1b84f414e8355b20f0af681e1bb0bda7efe2b/lib/index.ts#L112>. 2.0.52. verzió. Letöltés dátuma: 2024.11.22. 2019.
- [14] MUI. *Material UI — Overview*. <https://mui.com/material-ui/getting-started/>. Letöltés dátuma: 2024.11.22. 2022.
- [15] Daniel Delling, Thomas Pajor és Renato F. Werneck. „Round-Based Public Transit Routing”. *Transportation Science* 49.3 (2015. aug.), 591–604. old. ISSN: 1526-5447. DOI: 10.1287/trsc.2014.0534. URL: <http://dx.doi.org/10.1287/trsc.2014.0534>.

Ábrák jegyzéke

2.1.	Az alkalmazás felülete térképpel és vezérlőpanellel	5
2.2.	Az indulási idő, illetve a kiinduló- és célállomás beállítása	7
2.3.	Keresési találatok: az egyik <i>Köztársaság</i> tér nevű megálló Törökbálinton, a másik Pécelen található	8
2.4.	A kurzor alatt lévő megálló neve egy információs buborékban jelenik meg	8
2.5.	Az elérhető algoritmusok lista	11
2.6.	A beállításokat információs buborékok magyarázzák	12
2.7.	Az algoritmus alapállapota az "ÚTVONAL" fülön	12
2.8.	A megállók ikonjai azt jelzik, hogy milyen úton érkeztünk az adott csúcsba	13
2.9.	A megálló részletes információi	14
2.10.	Utazás hossza a térképen: egy megálló távolságra értendő	15
2.11.	Kész útvonal a térképen	16
3.1.	Felhasználási eset diagram	22
3.2.	Beállítások felület terve	25
3.3.	Megálló keresésének felület terve	26
3.4.	Algoritmus irányításának felület terve	27
3.5.	GTFS adatbázis sémája	32
3.6.	A <i>data</i> mappa szerkezete az adatok betöltését követően	34
3.7.	Backend szerkezete	36
3.8.	Sikeres teszteredmények	38
3.9.	Frontend modellek	46
3.10.	Beállítások komponense	47
3.11.	Keresés komponense	48

Táblázatok jegyzéke

3.1. Felhasználói történet: algoritmus választása	23
3.2. Felhasználói történet: útvonal kiválasztása	23
3.3. Felhasználói történet: algoritmus paramétereinek módosítása	24
3.4. Felhasználói történet: Algoritmus irányítása	24

Forráskódjegyzék

3.1.	Alkalmazás indítása különböző konfigurációkban	30
3.2.	BFS algoritmus pszeudokódja	40
3.3.	Dijkstra algoritmus pszeudokódja	42
3.4.	Különbség a Dijkstra és a mohó algoritmus között	43
3.5.	Különbség a Dijkstra és az A* algoritmus között	44
3.6.	Különbségek a Dijkstra és a mohó algoritmus pszeudokódja között . .	45