Eötvös Loránd Tudományegyetem

Informatikai Kar

Média- és Oktatásinfromatika

Tanszék

Útvonaltervezés vizualizálása a BKK hálózatán

Témavezető:
Erdősné Németh Ágnes
Egyetemi adjunktus

Szerző:
Szabó-Galiba Máté
Programtervező informatikus BSc

Budapest, 2024

Tartalomjegyzék

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 sokaknak 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 érthetővé 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 tömegközlededés, 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.

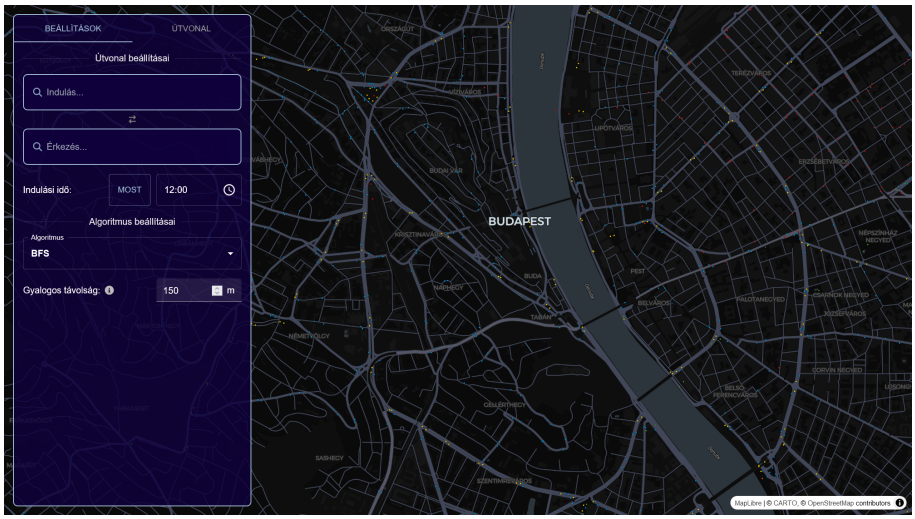
Az alkalmazás használata során a felhasználók kiválaszthatnak egy kiinduló és egy célállomást, majd az alkalmazás lépésről lépésre bemutatja az adott algoritmus működését a két állomás közötti útvonal megtalálásához. A grafikus felület segít az egyes algoritmusok előnyeinek és hátrányainak a megértésében, 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

színe[1] 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 modellezzük, így "gráf" alatt a továbbiakban ezt értjük, amennyiben mást 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:

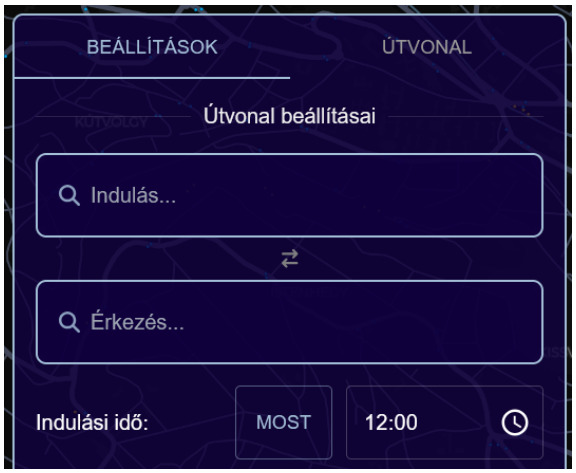
  1. **Utazási él:** Két megálló közötti közvetlen járatot jelképez, melynek a súlya az utazás időtartamának és a járatra való várakozás idejének az összege. Ez utóbbi alól az első utazási él mentesül, hiszen az csak későbbi indulást jelent, nem várakozást.
     Megjegyzés: A program csak olyan járatokat vesz figyelembe, melyekre a várakozási idő nem haladja meg a 60 percet.
  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 összekeverni választáskor, és ezzel a módszerrel akadályozza meg a

[1] A 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 időpont[2], 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).

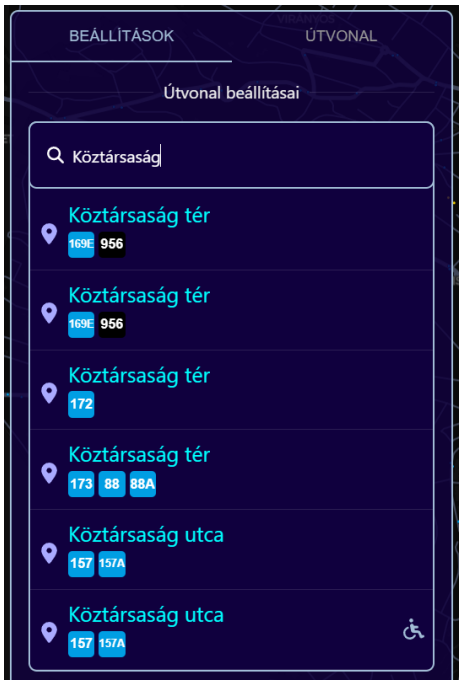


2.2. ábra. Az indulási idő, illetve a kiinduló- és célállomás beállítása

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.

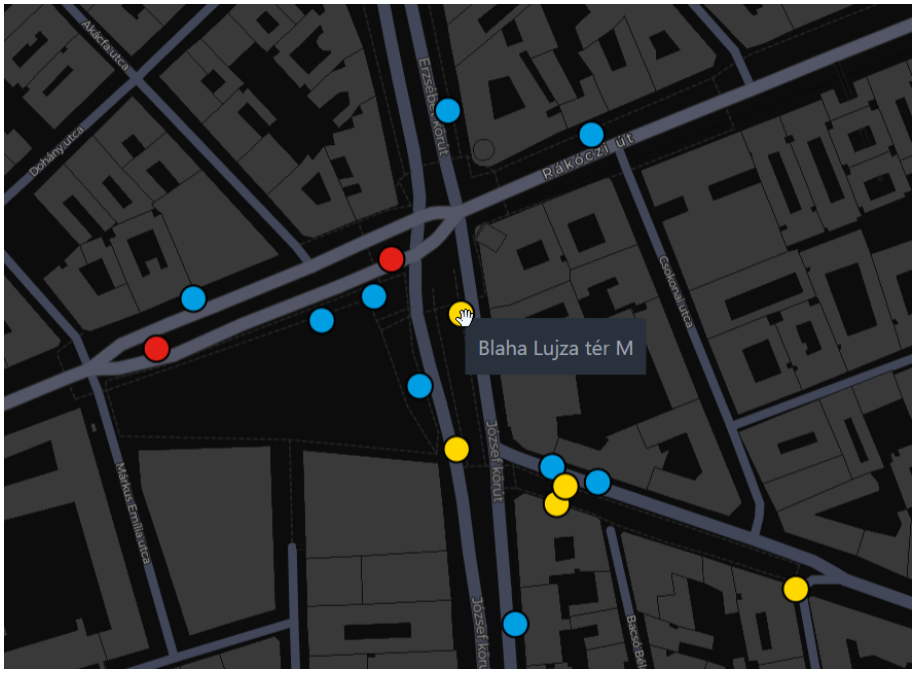
A kiinduló- és célállomás felcserélése a két beviteli mező közti dupla nyílra kattintva lehetséges (2.2).

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



2.3. ábra. 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 **??**. 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 futásideje 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 komplexitásának[3] következtében az algoritmus komplexitása is magasabb a BFS-hez képest. Az algoritmus futásideje 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

[3]Az 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 méterekben mérve[4]. 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 módunk 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 $O((|V| + |E|) \log |V|)$.

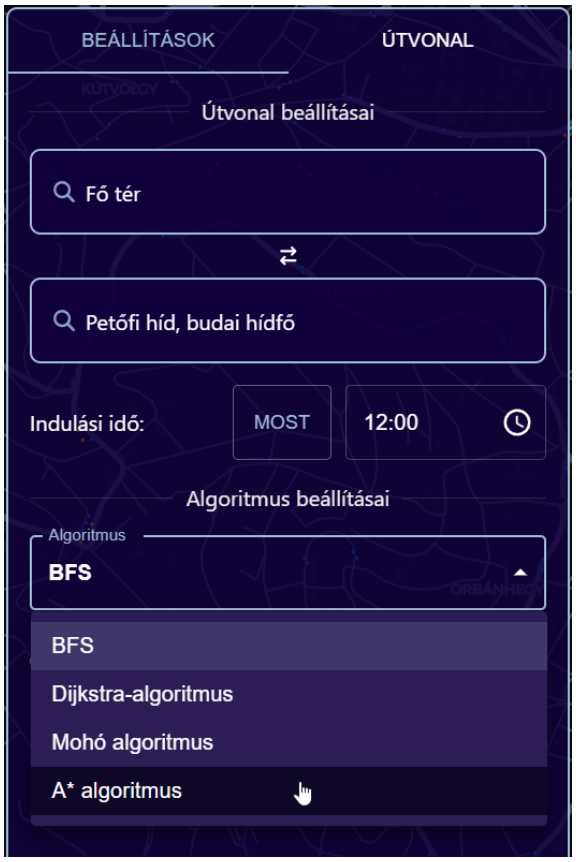
Röviden összefoglalva:

- **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.

[4]A képlet nem 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

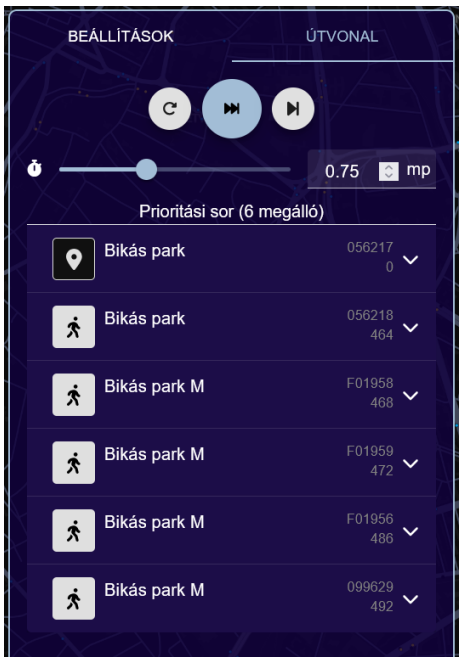
Választott algoritmustól függetlenül beállítható az is, hogy mi a maximális sétáló távolság, amin belül az alkalmazás felajánl átszállásokat. Ennek az alapértelmezett értéke 150 méter, ami tapasztalataim szerint általában elég azonos nevű, egy csoportban lévő megállók közti átszálláshoz.

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).

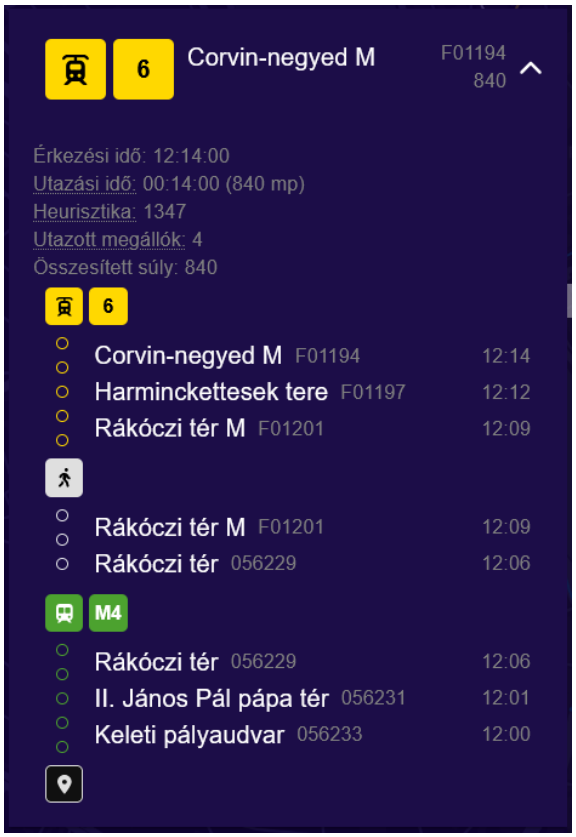
(a) Indulóállomás ikonja (b) Átszállási él ikonja (c) Utazási él ikonjai

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, azonoítóját, és az utazás módját jelölő ikonokat tartalmazza.

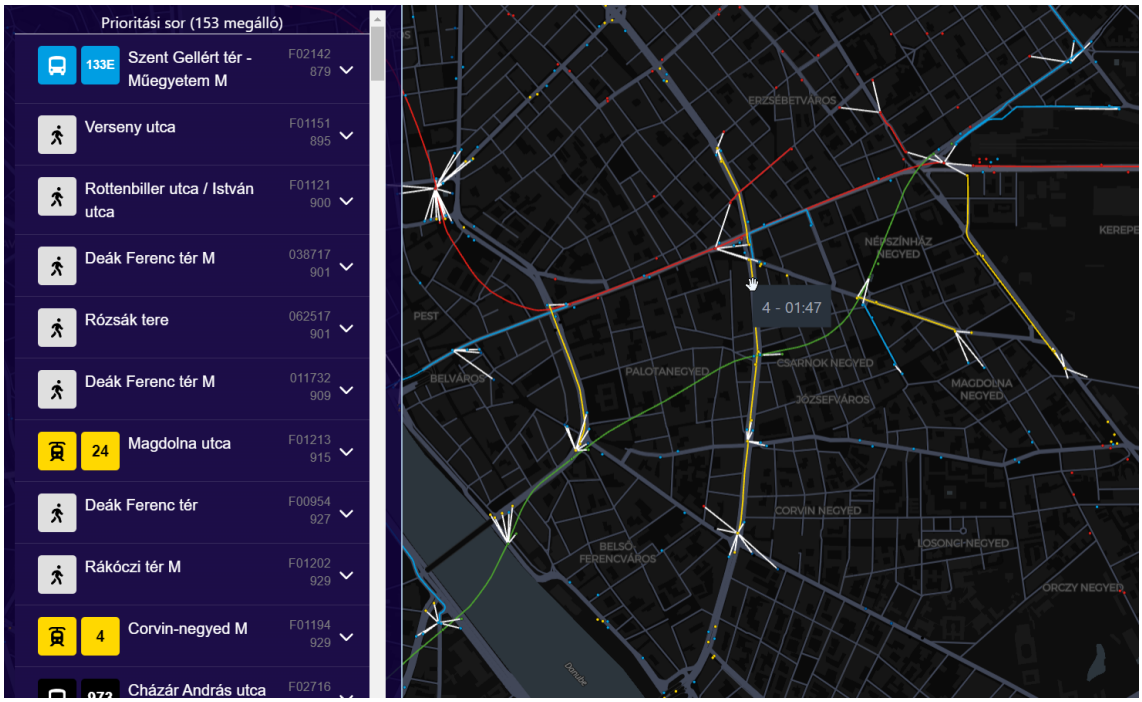


2.9. ábra. A megálló részletes információi

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

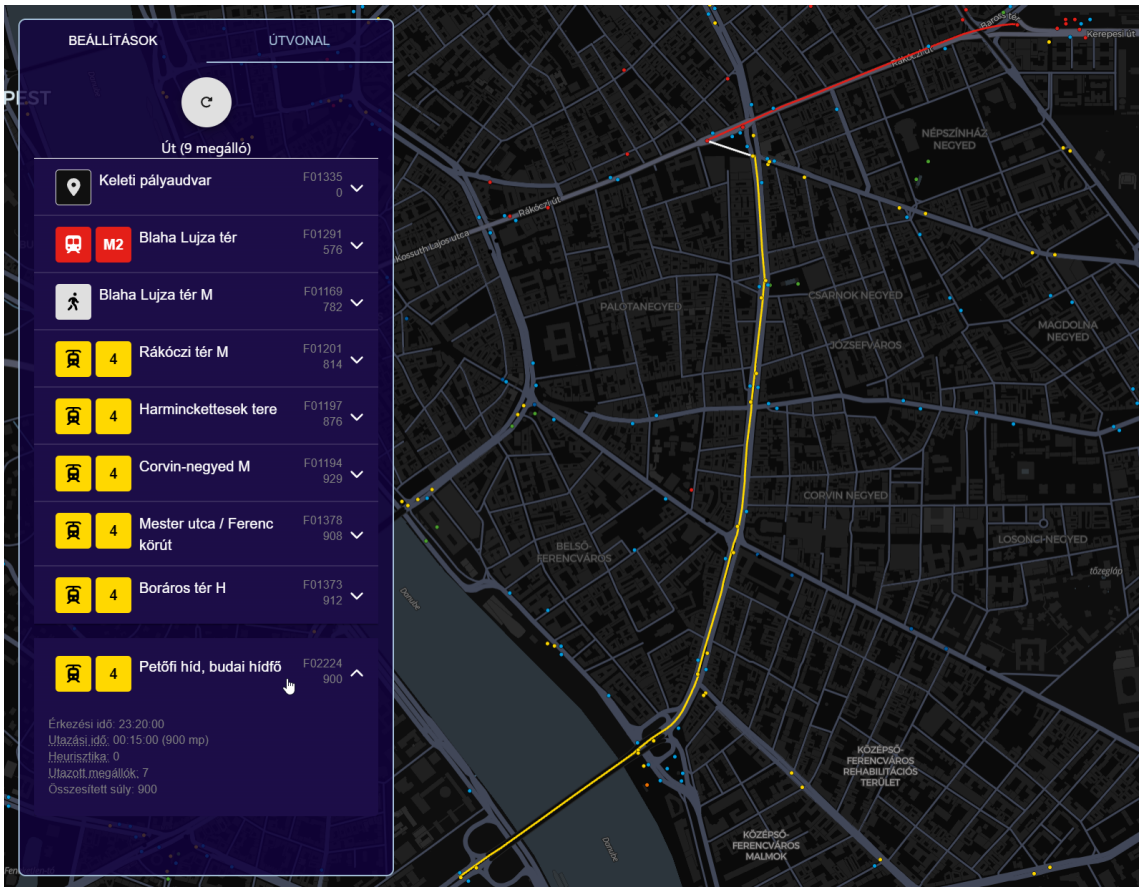
2.5.3. Útvonalak a térképen

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. Az utazás hossza 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 listája 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 élt ismer meg, a célpontjától sétatávolságra lévő megállókat is hozzáadja az ismert megállók listájához, átszállási éllel 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 várnánk, 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 figyelmbe 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 keretrendszerek 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. Magas szintű áttekintés

3.3.1. Alkalmazás felépítése

Az alkalmazás egy backendből és egy frontendből 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.

3.3.2. Verziókövetés

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

3.3.3. Backend áttekintés

A backend egy *Node.js* alapú alkalmazás, mely az *Express.js* keretrendszert használja a REST API megvalósítására. Fontos tényező volt a környezet kiválasztásában, hogy az NPM[1]-en megtalálható *node-gtfs*[4] csomag egyike volt a kevés elérhető könyvtáraknak[2], 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`

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

[2]A GTFS adatok feldolgozására való könyvtárak listája megtalálható a `https://gtfs.org/resources/gtfs/` oldalon *(*Letöltés dátuma: 2024.11.22.)

`clone [repo] && docker compose up -d` paranccsal bárhol egyszerűen futtatható a backend (ahol a Docker és a git telepítve van).

Az olvashatóság és karbantarthatóság érdekében a backend kódja TypeScript[3] nyelven íródik.

3.3.4. 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 közösség támogatása 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 megjelenítésére szükségtelen a *kepler.gl* komplexitása, illetve a *deck.gl* dokumentációját is részletesebbnek és könnyebben érthetőnek 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.

3.3.5. Fejlesztői környezet felállítása

Az alkalmazás fejlesztéséhez a következő programok telepítése szükséges:

- *Node.js (és npm csomagkezelő)*: Bár a backend és a frontend is Dockerben futtatható, 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, mert ingyenes és egyszerűen bővíthető JavaScript és Docker támogatáshoz, de használható más IDE (pl. WebStorm) is.

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

[3] A TypeScript egy nyelv, ami a JavaScriptre épül, de statikus típusokat is támogat[**typescript**]

3.3.6. Alkalmazás futtatása

Az alkalmazást fejlesztéshez is Dockerben futtatjuk, hogy a program írásakor feltételezhessük, hogy mindig 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 fájlrendszert[4], és újraindítja a kódot minden alkalommal, amikor egy fájl frissül (frontenden csak azokat a komponeneseket, amiket érinti a frissítés — ezt HMR-nek, azaz Hot Module Reload-nak hívják).
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 legkisebb lesz, és a Docker építéskor gyorsítótárban tudja tárolni a lépések közös elemeit. A különválasztott fejlesztői és a produkciós 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 produkciós 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  # Produkciós környezet indítása
5  docker compose up -d --build
```

3.1. forráskód. Alkalmazás indítása különböző konfigurációkban

Bár produkciós környezethez minden adott, hogy Dockerből futtatható legyen az alkalmazás mindkét 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 fájlokat egy webszerveren keresztül kiszolgálni, például Nginx vagy Apache HTTP szerver segítségével.

[4]A fájlrendszer figyeléséről frontend esetén a *vite*, backend esetén a *tsx* gondoskodik.

3.4. Backend

3.4.1. Könyvtárszerkezet

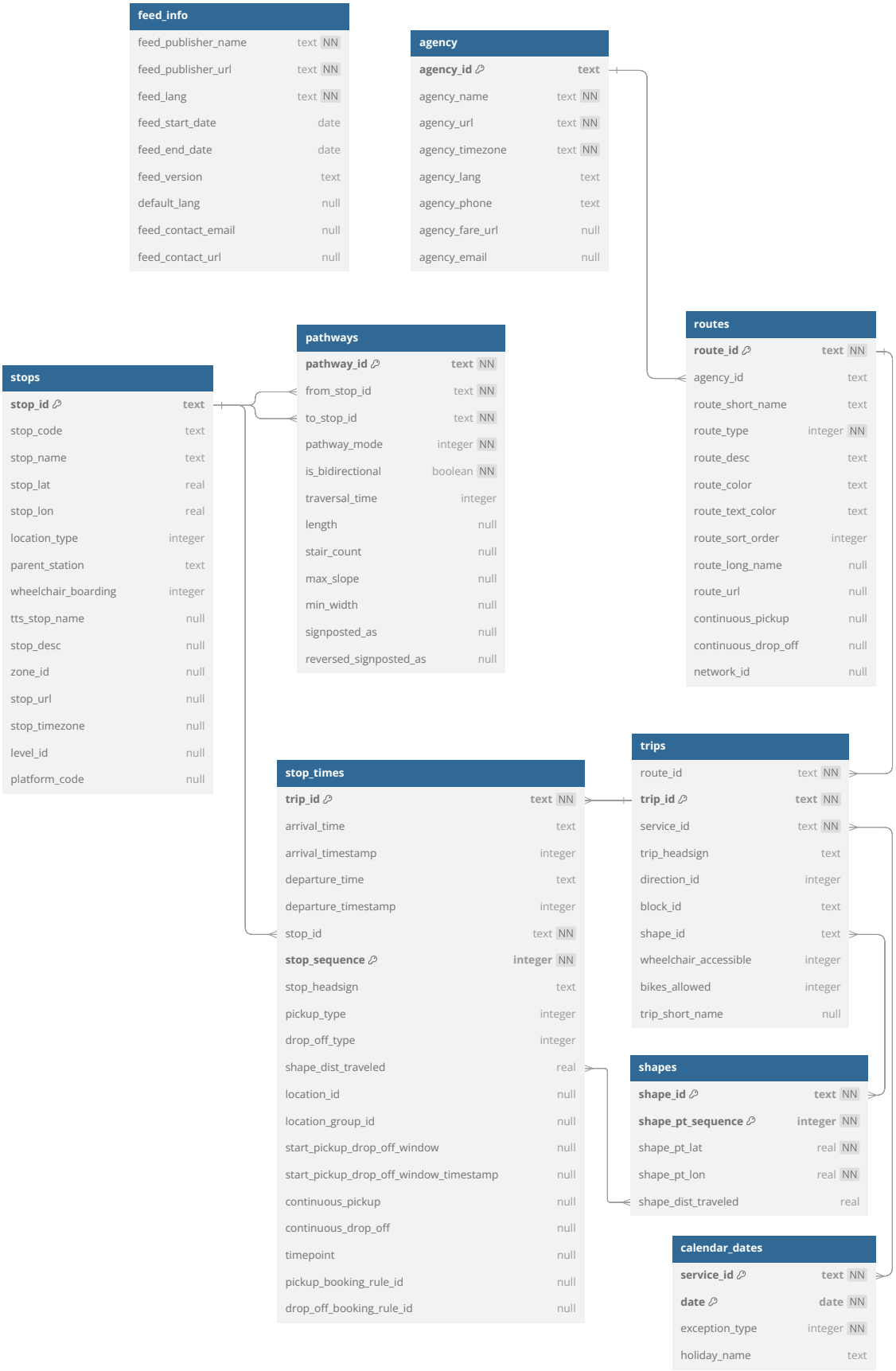
A backend könyvtárszerkezete a következő:

- `src`: A forráskódokat tartalmazó könyvtár.
- `src/configs`: 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.
- `.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 a **??** fejezetben).
- `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.

3.4.2. 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 (3.1).



3.1. ábra. GTFS adatbázis sémája

Megjegyzés: A teljes GTFS szabvány ennél jóval több táblát tartalmaz, de a legtöbbjük opcionális. A 3.1 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 sémában, de null *típussal vannak jelölve.*

A táblák a következő információkat tartalmazzák[7]:

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 mindegyike 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 használjuk[5].

7. *shapes*: Az egyes járatok útvonalát tartalmazza pontokban; koordinátapárokat tárol, amelyeket összekötve az adott járat pontos útvonalát kapjuk a térképen. A frontend ezt a táblát 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 információ az egy csoportba tartozó megállók összekötésére. Egyszerűbb és a

[5]A program célja nem pontos információk szolgáltatása, hanem algoritmusok demostrálása 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.

felhasználó számára is következetesebb bármelyik 2 megálló közötti gyaloglást azonosan kezelni.

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" szöveges formátumban szereplő időpontokat egész számokká alakítják, amiket sokkal gyorsabban lehet összehasonlítani. 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 legynek.

Amint az adatbázis betöltése megtörténik, az alkalmazás GeoJson fájlokat generál a frontend számára, melyek úgyszintén a *data* mappába kerülnek (3.2). Ezek a fájlok tartalmazzák a megállók és az útvonalak geometriáját, amelyeket a frontend a térképen megjelenít.

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

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

4. fejezet

Összegzés

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus porttitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Köszönetnyilvánítás

Amennyiben a szakdolgozati / diplomamunka projekted pénzügyi támogatást kapott egy projektből vagy az egyetemtől, jellemzően kötelező feltüntetni a dolgozatban is. A dolgozat elkészítéséhez segítséget nyújtó oktatók, hallgatótársak, kollégák felé is nyilvánítható külön köszönet.

A. függelék

Szimulációs eredmények

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque facilisis in nibh auctor molestie. Donec porta tortor mauris. Cras in lacus in purus ultricies blandit. Proin dolor erat, pulvinar posuere orci ac, eleifend ultrices libero. Donec elementum et elit a ullamcorper. Nunc tincidunt, lorem et consectetur tincidunt, ante sapien scelerisque neque, eu bibendum felis augue non est. Maecenas nibh arcu, ultrices et libero id, egestas tempus mauris. Etiam iaculis dui nec augue venenatis, fermentum posuere justo congue. Nullam sit amet porttitor sem, at porttitor augue. Proin bibendum justo at ornare efficitur. Donec tempor turpis ligula, vitae viverra felis finibus eu. Curabitur sed libero ac urna condimentum gravida. Donec tincidunt neque sit amet neque luctus auctor vel eget tortor. Integer dignissim, urna ut lobortis volutpat, justo nunc convallis diam, sit amet vulputate erat eros eu velit. Mauris porttitor dictum ante, commodo facilisis ex suscipit sed.

Sed egestas dapibus nisl, vitae fringilla justo. Donec eget condimentum lectus, molestie mattis nunc. Nulla ac faucibus dui. Nullam a congue erat. Ut accumsan sed sapien quis porttitor. Ut pellentesque, est ac posuere pulvinar, tortor mauris fermentum nulla, sit amet fringilla sapien sapien quis velit. Integer accumsan placerat lorem, eu aliquam urna consectetur eget. In ligula orci, dignissim sed consequat ac, porta at metus. Phasellus ipsum tellus, molestie ut lacus tempus, rutrum convallis elit. Suspendisse arcu orci, luctus vitae ultricies quis, bibendum sed elit. Vivamus at sem maximus leo placerat gravida semper vel mi. Etiam hendrerit sed massa ut lacinia. Morbi varius libero odio, sit amet auctor nunc interdum sit amet.

Aenean non mauris accumsan, rutrum nisi non, porttitor enim. Maecenas vel tortor ex. Proin vulputate tellus luctus egestas fermentum. In nec lobortis risus,

sit amet tincidunt purus. Nam id turpis venenatis, vehicula nisl sed, ultricies nibh. Suspendisse in libero nec nisi tempor vestibulum. Integer eu dui congue enim venenatis lobortis. Donec sed elementum nunc. Nulla facilisi. Maecenas cursus id lorem et finibus. Sed fermentum molestie erat, nec tempor lorem facilisis cursus. In vel nulla id orci fringilla facilisis. Cras non bibendum odio, ac vestibulum ex. Donec turpis urna, tincidunt ut mi eu, finibus facilisis lorem. Praesent posuere nisl nec dui accumsan, sed interdum odio malesuada.

Irodalomjegyzék

[1] Budapesti Közlekedési Központ. *Tervezett menetrendi adatbázis*. https://opendata.bkk.hu/data-sources. Letöltés dátuma: 2024.11.15. 2024.

[2] MobilityData. *About - General Transit Feed Specification*. https://gtfs.org/about/. 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: React összehasonlítása kompetitorokkal*. 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] MobilityData. *General Transit Feed Specification Reference*. https://gtfs.org/reference/. September 2024 revision verzió. Letöltés dátuma: 2024.11.22. 2006.

Ábrák jegyzéke

Táblázatok jegyzéke

Algoritmusjegyzék

Forráskódjegyzék

Tárgymutató