

TDK-dolgozat

Szerző:

Szincsák Tamás

Debreceni Egyetem
Informatika Kar
Információ Technológia Tanszék

Offline menetrend és útvonaltervező alkalmazás készítése mobiltelefonra

Témavezető:
Vágner Anikó
egyetemi tanársegéd

Készítette:
Szincsák Tamás
Programtervező Informatikus
MSc, 1. félév

Debrecen
2012

Tartalomjegyzék

1.	Bevezetés.....	1
1.1.	A témaválasztás okai	1
1.2.	A bemutatott szoftverről röviden	2
1.3.	Köszönetnyilvánítás.....	3
2.	Terv és implementáció	4
2.1.	Célok és követelmények	4
2.2.	Felhasznált szoftverek és eszközök	5
2.3.	Architektúrális terv	6
2.4.	Adatbázisterv	8
2.4.1.	Az adatok forrása.....	8
2.4.2.	Logikai adatbázisterv.....	8
2.4.3.	Adatbázisséma	17
2.5.	Felhasználói felület	18
2.5.1.	Főmenü	19
2.5.2.	Beállítások	19
2.5.3.	Járatok listája	19
2.5.4.	Megállók listája	19
2.5.5.	Útvonaltervező.....	20
2.5.6.	Névjegy.....	20
2.5.7.	Egy járat adatai	20
2.5.8.	Egy megálló adatai	21
2.5.9.	Útvonal megtekintése szövegesen	21
2.5.10.	Útvonal megtekintése térképen.....	21
2.6.	Alkalmazáslogika.....	22
2.6.1.	Tartalomkezelő	22

2.6.2.	Adatmodell	23
2.6.3.	Útvonaltervező.....	24
2.6.4.	Felhasználói felület.....	24
2.6.5.	Szolgáltatások.....	25
2.7.	Az útvonaltervező megvalósítása	26
2.7.1.	Elméleti megvalósítás.....	27
2.7.2.	Gyakorlati megvalósítás	28
2.8.	Lényegesebb optimalizálási lehetőségek	31
2.8.1.	Általános módszerek a sebesség növelésére.....	31
2.8.2.	Távolságok számítása	31
2.8.3.	További lehetőségek	33
3.	Összefoglalás.....	34
3.1.	Továbbfejlesztési lehetőségek	35
4.	Függelék	36
4.1.	Irodalomjegyzék	36
4.2.	További ábrák és képernyőképek	38
4.3.	Saját munka leírása	42

1. Bevezetés

Mai rohanó világunkban fontos tényező a pontosság, akár különleges alkalmakról, akár a tanórákra vagy épp a munkába érkezésről van szó. Ez azonban – bármennyire is szeretnénk elérni – nem csak rajtunk múlik. Főleg a diákok, és a fiatal dolgozók veszik gyakran igénybe a tömegközlekedési eszközöket, amelyek a pontossága a csúcsforgalom idején nem garantálható, bármikor előfordulhatnak kisebb késések, elterelések, így szükséges, hogy az emberek képesek legyenek a változásokhoz időben alkalmazkodni, bárhol is tartózkodjanak. Habár a közlekedési társaságok elérhetővé teszik a menetrendeket az interneten, ezek böngészésére a szabadban vagy nincs lehetőség, vagy ha van is, használatuk egy mobiltelefonról lassú és körülményes.

Jelen dolgozat elsődleges célja egy olyan alkalmazás fejlesztésének bemutatása, amely segítségével a felhasználók a jelenlegi alternatíváknál gyorsabban és kényelmesebben képesek átlátni a Debrecen tömegközlekedési hálózatát, csupán egy okos telefon segítségével, bárhol is tartózkodjanak. Az alkalmazás tervezése során fontos szempont volt az adatbevitel számának minimalizálása, aminek elérése érdekében a szoftver felhasználja a készülék szenzorai (főként óra és GPS) által szolgáltatott adatokat, valamint lehetőséget nyújt két pont közötti automatikus útvonaltervezésre is, amely eredményét szöveges módon és aktív internetkapcsolat esetén térképen is képes megjeleníteni. Mind az adatok feldolgozása, mind az útvonaltervezés a készüléken, internet kapcsolat nélkül történik, így külön kihívást jelent a fentiek megvalósítása egy alacsony számítási kapacitással rendelkező eszközön.

A másodlagos cél olyan adatstruktúrák és algoritmusok létrehozása, amelyek felhasználásával a későbbiekben más városokhoz és közlekedési hálózatok is készíthető hasonló szoftver.

1.1. A témaválasztás okai

A választás fő oka, hogy a dolgozat írásakor nem létezett mobiltelefonra optimalizált menetrend az ország több nagyvárosában, többek között Debrecenben sem, a létező megoldások pedig inkább a papír alapú menetrendek digitális változataként funkcionáltak, mintsem teljes értékű alkalmazásként. Emellett Magyarországon egyáltalán nem, külföldön pedig csak nagyon ritka esetben volt példa olyan mobil alkalmazásra, amely képes útvonalak tervezésére csupán a készülék erőforrásait felhasználva, folyamatos internetkapcsolat és egy központi szerver nélkül.

1.2. A bemutatott szoftverről röviden

Az alkalmazás fejlesztése 2012. januárjában kezdődött, egy szakdolgozat keretében. A fejlesztés a vízesés modellt¹ követte, így tehát jól meghatározott követelményei voltak. A kezdetben meghatározott külső és belső felépítéstől csak kis mértékben lehetett eltérni, ez viszont garantálta, hogy a termék időben elkészül. Ennek eredményeként állt elő a 0.0.1-es verzió, amely demonstrálta, hogy a feladat megvalósítható, valamint a segítségével lehetőség nyílt egy zártkörű tesztre, amely a felhasználók bevonásával feltárta a szoftver hiányosságait, és a felhasználók további igényeit.

A fejlesztés ezek után az agilis módszertant¹ követi, hogy a felmerülő igényekre gyorsan lehessen reagálni. Az első publikus tesztverzió 2012. májusában vált elérhetővé, ezt számtalan alpha, majd 2012. augusztusától kezdve több béta verzió is követte. A dolgozat írásakor a legújabb verzió a 0.9.5-ös (béta), amely ismert funkcionális hibát nem tartalmaz, és a felhasználói felület kisebb módosításai után készen áll a véglegessé nyilvánításra.

A végleges (1.0) verzió kiadása után lehetőség nyílik a szoftver további városokra történő kiterjesztésére, akár nemzetközi szinten is. A fejlesztés szakaszait alábbi ábra ismerteti:



1. ábra (a fejlesztés fázisai)

A dolgozat a szoftver jelenlegi állapotát tükrözi, az agilis fejlesztés miatt a későbbiekben előfordulhatnak (és nagy valószínűséggel elő is fordulnak) változások, eltérések. A tárgyalási szakasz első része tartalmazza a menetrend funkció megvalósításának leírását, beleértve a használt adatbázis szerkezetét, az alkalmazás logikát, valamint a felhasználói felület felépítését. A második részben az útvonaltervező megvalósítását, és annak mobiltelefonokra történő optimalizálását részletezem.

1.3. Köszönetnyilvánítás

Szeretnék köszönetet mondani Vágner Anikó egyetemi tanársegédnek, aki a dolgozat készítése közben témavezetőként rendelkezésemre állt, és készséggel válaszolt a felmerülő kérdéseimre.

Továbbá szeretnék köszönetet mondani a Debreceni Regionális Közlekedési Egyesületnek (DERKE), akik rendelkezésemre bocsájtották Debrecen város 2012-es tömegközlekedési menetrendjét.

2. Terv és implementáció

2.1. Célok és követelmények

A szoftver egy Android operációs rendszerre² írt alkalmazás, melynek használatához minimálisan egy érintőképernyővel és 2.1-es, vagy újabb Androiddal* rendelkező készülék, az offline adatbázis letöltéséhez egyszeri internetkapcsolat és 2 MB tártérület szükséges. Az alkalmazás továbbá képes felhasználni a GPS és az iránytű által szolgáltatott adatokat, ha azok elérhetőek.

Az Androidon futó alkalmazások életciklusa miatt feltételezhető, hogy a program a használat után hibernált állapotban még órákig a háttérben marad, ahonnan gyorsan újra működésbe hozható. Emiatt az adatok betöltése, valamint a különféle gyorsíróterek felépítése induláskor történik, és ezek az alkalmazás teljes futása alatt a memóriában maradnak. Nem szabad azonban, hogy a fenti folyamat túl sokáig tartson, mert egy hideg indítás esetén a felhasználók számára kellemetlen lehet a várakozás, így a maximálisan elfogadható betöltési idő 5 másodperc.

Indítás után egy főmenü jelenik meg, ahonnan az összes funkció elérhető. Ezek a következők:

- Az összes járat listázása, szűrési lehetőséggel
- Egy tetszőleges járat indulási időpontjainak megjelenítése valamely megállóból (a megállóhelyi tájékoztatókhoz hasonlóan)
- Egy tetszőleges járat által érintett megállók megjelenítése a menetidőkkel együtt megállóból (szintén a megállóhelyi tájékoztatókhoz hasonlóan)
- Az összes megálló listázása abc sorrendben, szűrési lehetőséggel
- A közelben levő megállók listázása távolság szerint, szűrési lehetőséggel
- Egy tetszőleges megállóból induló járatok megjelenítése (a digitális tájékoztató táblákhoz hasonlóan)
- Útvonaltervezés két megálló között, igény szerint átszálásokkal és sétával
- A testreszabható beállítások megjelenítése és megváltoztatása
- Információk megjelenítése a szoftverről

A funkciók közötti váltásnak gyorsnak kell lennie, azaz az esetek 90%-ában 0.5-1 másodperc az irányadó, de sosem haladhatja meg a 2 másodpercet.

* A választás indoklásáért lásd a függelék 5-ös és 6-os számú ábráját.

Az alkalmazás háttérbe helyezése esetén a következő indításnál a legutóbbi képernyő jelenik meg. Kivétel ez alól az az eset, amikor a szoftver nincs használatban több, mint 12 órán át, ekkor a program a hideg indításnak megfelelően a kezdőképernyővel indul.

2.2. Felhasznált szoftverek és eszközök

Az alkalmazás az Android követelményeinek megfelelően Java³ nyelven íródott, a felhasznált adatbázis-motor pedig az SQLite. A szoftver fejlesztése során az alábbi eszközöket és programkönyvtárakat vettem igénybe:

- **Eclipse**⁵
- **Android SDK**⁶

Az Android operációs rendszerre történő fejlesztéshez szükséges eszközök gyűjteménye. Az ADT Plugin for Eclipse bővítménynek köszönhetően remekül együttműködik az Eclipse IDE-vel.

- **Paint.NET**⁷

Ingyenes képszerkesztő program az ikonok és a felhasználói felület elemeinek összeállításához.

- **ImageMagick**

Parancssori képmanipuláló program a méretezések és forgatások automatizálásához.

- **ActionBarSherlock**

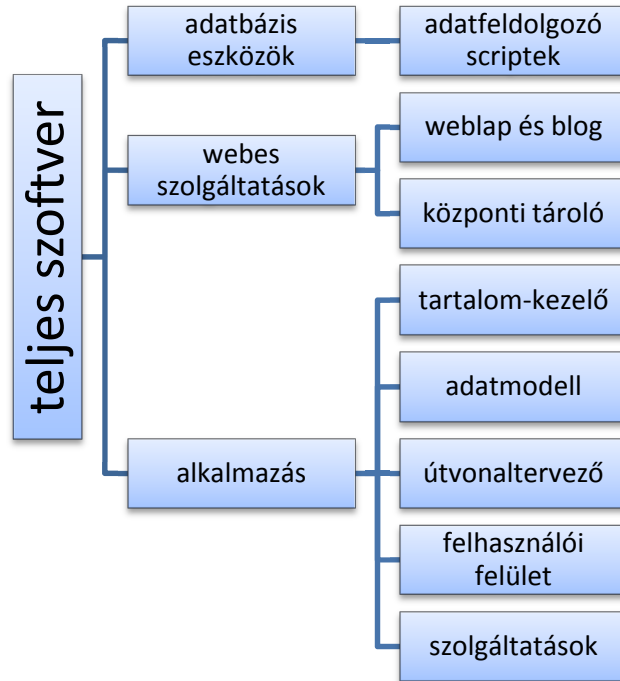
Egy népszerű programkönyvtár, amely a legújabb Android felhasználói felületét kölcsönzi a régebbi operációs rendszerekkel is kompatibilis alkalmazások számára.

- **DateTimeSlider**

Kényelmes dátumbevitelt lehetővé tevő komponens.

2.3. Architektúrális terv

Az alkalmazás, és a hozzá kapcsoló egyéb programok elemeit, és egymással való kapcsolatát az alábbi ábra szemlélteti:



2. ábra (a szoftver logikai szerkezete)

A teljes szoftvert az adatbázis eszközök, a webes szolgáltatások és a mobil alkalmazás együttesen alkotják.

Az *adatbázis eszközök* tartalmazzák az adatbázisok létrehozásához és karbantartásához szükséges alkalmazásokat és scripteket, ide értve az adatok importálását és a változások követését segítő programokat, valamint az elkészült adatbázis tömörítésére és optimalizálására szolgáló eszközöket.

A *webes szolgáltatások* fő feladata az adatbázisok, frissítések és alapértelmezett konfigurációs állományok tárolása, és azok elérhetővé tétele a mobil alkalmazás számára. Másodlagos feladata egy felület biztosítása az alkalmazás terjesztéséhez és a felhasználók tájékoztatásához.

A *mobil alkalmazás* a szoftver egyetlen eleme, amelyet a végfelhasználók futtatnak. Részegységei a következők:

- *Tartalom-kezelő*: feladatai közé tartozik az elérhető adatbázisok folyamatos nyomon követése, az új adatbázisok letöltése, a letöltött adatbázisok karban tartása, valamint a legoptimálisabbnak talált adatbázis betöltésre jelölése.
- *Adatmodell*: ellenőrzi és betölti a kijelölt adatbázist, valamint lehetővé teszi az adatok elérését az alkalmazás többi része számára az adatbázis fizikai struktúrájának ismerete nélkül.
- *Útvonaltervező*: lehetséges útvonalakat ajánl két pont között.
- *Felhasználói felület*: rendszerezi és megjeleníti az adatmodell tartalmát, valamint lehetővé teszi a felhasználó számára a szoftver különféle funkcióinak elérését.
- *Szolgáltatások*: a háttérben futva ütemezett feladatokat hajt végre, gondoskodik az adatbázis periodikus frissítéséről.

A továbbiakban csak a mobil alkalmazás működését részletezem, az egyéb szoftverelemekre csak említés szintjén térek ki.

2.4. Adatbázisterv

2.4.1. Az adatok forrása

Az alkalmazás működéséhez szükséges adatok két csoportra bonthatók:

- menetrendi adatok
- naptár-adatok

A naptár-adatok (munkanaptáblázat, ünnepek, tanítási szünetek) az egész országban azonosak, és évente változnak. A menetrendi adatok ezzel szemben városonként eltérőek, és akár havonta is változhatnak. Megfontolandó a két adathalmaz külön adatbázisban történő tárolása a naptári adatok újrafelhasználása és a frissítésekhez szükséges adatforgalom csökkentése érdekében. A naptári adatok azonban a menetrendhez képest olyan kis méretűek (kb. 1%), hogy nem éri meg szétválasztani őket a menetrendtől, így habár külön forrásból érkeznek, és külön is frissülhetnek, egy adatbázisban tárolom őket.

A menetrendi adatok minden városban különböző formában érhetőek el, és általában ezeket nem automatizált feldolgozásra szánják, így a közlekedési társaságok együttműködése nélkül nem lehet, vagy nehézkes a megfelelő formátumra alakítani. Az adatok feltöltése éppen ezért kézzel történik (a kisebb részműveletek esetenként természetesen automatizálhatók).

Debrecen esetében egy alkalommal CSV fájlok formájában megkaptam a menetrendet, amelyből könnyedén feltölthetők voltak a megállók és a járatok, az azóta történt változásokat azonban a DKV Zrt által kiadott közlemények és menetrendek alapján kell frissíteni.

Nemzetközi viszonylatban a közlekedési társaságok általában kiadják az adott város menetrendjét a Google által kidolgozott GTFS (General Transit Feed Specification)^{9,10} formátumban, Magyarországon azonban egyedül a Budapesti Közlekedési Központ^{11,12} alkalmazza ezt a megoldást. A GTFS formában leírt menetrend mérete nagy (általában több száz megabájt), elsősorban a Google Maps útvonaltervezőjéhez készült, mobiltelefonos felhasználásra közvetlenül nem alkalmas. A későbbi bővítés érdekében érdemes viszont egy feldolgozó szoftvert elkészíteni, amely a GTFS adatokat a szoftver saját formátumára konvertálja, de ezt jelenleg sem az alkalmazás, sem a dolgozat nem tartalmazza.

2.4.2. Logikai adatbázisterv

Az adatok tárolása táblákban történik. A táblák azonos típusú rekordokat tartalmaznak, opcionális mezők nincsenek. NULL érték nem létezik (hisz szoftveres feldolgozása

körülményes), ahol mégis szükség lenne rá, ott valamilyen jelzőbit, vagy egy speciális érték jelzi, hogy az adott mező tartalma nincs értelmezve. Jelenleg az adatok fizikai tárolására egy relációs adatbázist használok, de ez nem követelmény.

A mezők típusai az alábbiak lehetnek:

Mezőnév	Jelentés
boolean	Logikai, tárolása true/false vagy 1/0 alakban történhet
uint8	8 bites, előjel nélküli egész
uint32	32 bites, előjel nélküli egész
string	UTF-8 karakterlánc

A táblák és azok tartalma a következő:

calendar_override

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
from_date	string	yyyy-mm-dd formátumú karakterlánc, inkluzív
to_date	string	yyyy-mm-dd formátumú karakterlánc, inkluzív
new_type	uint8	0-6, az egyes értékek a hét napjait jelölik, 0 = hétfő, 1 = kedd, ..., 6 = vasárnap
description	string	Tetszőleges karakterlánc

Tartalmazza, hogy egy adott dátum a hét melyik napjára esik. Az alkalmazás képes kiszámolni az alapértelmezett értékeket, így elég csak a kivételeket felsorolni. Az ünnnapok vasárnapnak, a szabadnapok szombatnak számítanak. A munkanap áthelyezések esetén a munkanapnak számító szombatot olyan napként kell feltüntetni, amit azon a napon pótolunk. A description mező egy tetszőleges leírást tartalmaz, amely a sor létezését magyarázza, pl.: „Karácsony”.

calendar_school_holidays

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
from_date	string	yyyy-mm-dd formátumú karakterlánc
to_date	string	yyyy-mm-dd formátumú karakterlánc
description	string	Tetszőleges karakterlánc

A tanítási szüneteket tartalmazza. Minden olyan nap, amely ebben a táblában nem szerepel iskolai tanítási napnak számít. A hétvégéket, valamint a szabad- és munkaszüneti napokat nem kell felvenni külön, hisz ezek is tanítási időszaknak számítanak, de ha egy szünet eleje vagy vége ilyen napokra esik, ezeket is bele kell foglalni az intervallumba. Az egyetemi tanítási szünetek ettől eltérhetnek, azokat ebben a táblában nem kell megadni. A description mező egy tetszőleges leírást tartalmaz, amely a sor létezését magyarázza, pl.: „Nyári szünet”.

day_types

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
name	string	Tetszőleges karakterlánc
notes	string	Tetszőleges karakterlánc
rule	string	Speciális karakterlánc, értelmezését lásd később

Az összes olyan naptípust tartalmazza, amelyet a járatok indulása szempontjából megkülönböztetünk egymástól, például: „Szabadnapokon”, „Munkanapokon, tanítási időszakban”, vagy „Hétfőn”. Tetszőleges számú rekord sorolható föl, ezeknek nem kell egybeesniük a naptár napjaival, vagy a köznapi életben megszokott fogalmakkal. A járatok indításánál az adott napra az összes szabály ki lesz értékelve, és minden olyan indulási idő, amelynél a hozzátartozó day_type rekord rule feltétele igazra értékelődik ki, indításra kerül. A name és notes mező tartalma tetszőleges, de az előbbi a felhasználók, utóbbi a fejlesztők számára szól. Ezt érdemes figyelmebe venni a tábla feltöltésekor, és a name mezőbe egy mindenki számára érthető leírást írni. A notes mező tartalmazhatja például a rule mezőben megadott szabály könnyebben olvasható változatát, vagy olyan dolgokat, amelyekre érdemes figyelni a szabály módosításakor. A rule mező egy logikai kifejezés, amelynek értéke igaz vagy hamis lehet. A kifejezésben változóként a következő adatok jelennek meg:

Változó	Típus	Rövidítés	Leírás
is_schoolday	boolean	s	Az adott nap tanítási időszakban van-e
real_day	int	r	A hét napja munkanap áthelyezések nélkül
virtual_day	int	v	A hét napja munkanap áthelyezésekkel
date	int	d	Dátum, hónap * 100 + nap alakban.

Hivatkozni rájuk a rövidítésükkel lehet, egészek esetén csak a < > = operátorok használhatók, logikai esetben nem használhatók operátorok, a változó önmagában képviseli a részkifejezés értékét. Bármely részkifejezés tagadható, ekkor ! jellel kell kezdeni. A teljes kifejezés formátuma szokatlan: elemeit vesszők választják el egymástól, amely „és” kapcsolatnak felel meg. „Vagy” kapcsolatot az elemek előtt elhelyezett + önálló elemmel lehet elérni, amely egybevonja a következő 2 részkifejezést (több + esetén többet), ekkor elegendő egynek teljesülni közülük. Ez a szokatlan leírási forma a feldolgozás gyorsítását célozza. Példák:

- **Munkanapokon:** $v < 5$

Jelentése: a munkanap áthelyezések figyelembevételével kapott hét napja 0..4 között van (azaz hétfő-péntek, tehát munkanapokról beszélünk).

- **Munkaszüneti napokon:** $v = 6$

Jelentése: a munkanap áthelyezések figyelembevételével kapott hét napja 6, azaz vasárnap. Ez a kifejezés teljesül ünnepnapok esetén is.

- **Vasárnap:** $r = 6$

Jelentése: a hét valós napja (a munkanap áthelyezések figyelembevétele nélkül) 6, azaz vasárnap. Ez a szabály nem válik igazzá abban az esetben, ha egy adott napot vasárnapként kezelünk, de ténylegesen nem vasárnapra esik (pl. A nemzeti ünnepek).

- **Munkanapokon, egyetemi tanítási időszakban:**

$v < 5, d > 0200, d < 1222, +, d < 0599, d > 0900$

Az ehhez tartozó könnyen olvasható logikai kifejezés:

`virtual_day in [monday..friday] && (date in [02-01..05-31] || date in [09-01..12-21])`

comments

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
content	string	Tetszőleges karakterlánc

Tetszőleges karakterláncokat tartalmaz, amelyekre a további táblák megfelelő helyén lehet hivatkozni az id mezőn keresztül. Ebbe a táblába kerülnek többek között a

megállókra és az indulási időkre vonatkozó megjegyzések, pl.: „A Köztemető nyitvatartási idején kívül "2A" jelzéssel, csak a Kassai útig közlekedik.”.

stops

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
name	string	Tetszőleges karakterlánc
subname	string	Tetszőleges karakterlánc
street	string	Tetszőleges karakterlánc
group_id	uint32	Tetszőleges 0-tól nagyobb szám
latitude	int	GPS latitude * 10 ⁶ , egészekre kerekítve
longitude	int	GPS longitude * 10 ⁶ , egészekre kerekítve

A megállók listáját tartalmazza. Minden táblával jelölt helyet fel kell venni, így előfordulhat több azonos névvel rendelkező megálló is. A name, subname és street mezőknek azonban együttesen egyértelműen meg kell határozni egy megállót. A name mező a megálló nevét tartalmazza, amely a hirdetményeken és a táblákon szerepel, pl. „Nagyállomás”. A street az utcát adja meg, amelyiken ez a megálló található, a subname pedig további információt ad, pl. „induló oldal”, vagy „bolt mellett”. Az azonos group_id-vel rendelkező megállók az alkalmazás fel fogja ajánlani további lehetőségként egy megálló megtekintésekor. A latitude és longitude mezők a megálló GPS-koordinátáit adja meg, a nagyobb teljesítmény érdekében egész számokként. Amennyiben a pozíció ismeretlen, 0-0-t kell használni, ez esetben viszont a megálló kiesik az útvonaltervezésből.

route_categories

Mezőnév	Típus	Leírás
id	uint32	Egyedi azonosító, 1-től kezdődik, folytonos
name	string	Tetszőleges karakterlánc, egyedi.
color	string	6 hexadecimális számjegy
icon_name	string	Az angol abc kisbetűit tartalmazó karakterlánc
public	boolean	1 vagy 0

A járatok típusait tartalmazza, például „Villamos” vagy „Busz”. Kategorizálást és szín alapján történő megkülönböztetést szolgál. A `color` mező tetszőleges színt tartalmazhat, de ajánlott, hogy ne legyen se túl sötét, se túl világos, és elegendő különbség legyen különböző rekordok színei között. Az `icon_name` tetszőleges, az alkalmazásba ágyazott ikon nevét tartalmazhatja, pl. „bus”, „airport”. Ha nem létezik ilyen ikon, egy szürke kérdőjel fog megjelenni a megfelelő helyen. A `public` mező meghatározza, hogy a kategória járatai igénybe vehetők-e bármiféle megkötés nélkül. Amennyiben ez az érték hamis, az alkalmazás ezeket a járatokat nem fogja az útvonaltervezéshez felhasználni.

routes

Mezőnév	Típus	Leírás
<code>id</code>	<code>uint32</code>	Egyedi azonosító, 1-től kezdődik, folytonos
<code>name</code>	<code>string</code>	Legfeljebb 15 karakter, egyedi.
<code>long_name</code>	<code>string</code>	Tetszőleges karakterlánc
<code>description</code>	<code>string</code>	Tetszőleges karakterlánc
<code>category_id</code>	<code>uint32</code>	Hivatkozás a <code>route_categories.id</code> mezőre

A járatok listáját tartalmazza. Annyi rekordot érdemes felvenni, ahány különböző járat azonosító létezik az adott településen. Ha például 10 járat közlekedik, amelyet 1-től 10-ig számoztak, akkor a tábla 10 rekordot kell, hogy tartalmazzon, függetlenül attól, hogy ezek a járatok hány irányban közlekednek, vagy van-e olyan speciális menet, amikor az adott jármű nem közlekedik a teljes útvonalon. A `name` mező a járat számát tartalmazza, a `long_name` pedig megtett útvonalról ad tájékoztatást. A `description` mező speciális információkat tartalmaz, például útirány, vagy „Gyorsjárat”. A `category_id` egy érvényes hivatkozás a `route_categories` táblára.

route_lines

Mezőnév	Típus	Leírás
<code>route_id</code>	<code>uint32</code>	Hivatkozás a <code>routes.id</code> mezőre
<code>line_id</code>	<code>uint32</code>	1-től kezdődik, folytonos, az <code>id</code> -vel együtt egyedi
<code>direction</code>	<code>uint8</code>	Tetszőleges szám
<code>name</code>	<code>string</code>	Maximum 15 karakter
<code>time_id_logic</code>	<code>uint8</code>	0, 1 vagy 2

Az összes egyedi pályát tartalmazza, amelyet a járatok befutnak. Minden járáshoz tartalmaznia kell legalább egy rekordnak ebben a táblában, de tetszőlegesen sok is lehet. Ha egy járat például oda-vissza közlekedik, valamint a késő esti órákban rövidebb utat tesz meg, ez a tábla 4 rekordot fog tartalmazni az adott járáshoz: teljes út oda, teljes út vissza, rövidített út oda, rövidített út vissza. A `direction` mező megad egy irányt, amelyet ez a pálya befut. Tetszőleges értéke lehet, de érdemes úgy megválasztani, hogy a hasonló irányba haladó pályák azonos értéket kapjanak, mert így az alkalmazás könnyebben tudja összekapcsolni a különböző, de egy irányba haladó járatokat. A `name` mező tartalmazza a járat nevét, amely megegyezhet a `routes.name` mező értékével, de felül is írhatja azt. Abban az esetben érdemes itt más nevet megadni, ha például az adott vonal rövidített, és a menetrendi hirdetményeken is más névvel jelölik azt, például „2” helyett „2*”. A `time_id_logic` mező egy speciális azonosító, amely meghatározza, hogy milyen algoritmussal számolja az alkalmazás az egymástól eltérő menetidőket. A használható értékek az adott várostól függenek, az alkalmazásban kell definiálni a hozzárendelt működést. Debrecen esetében 0, 1 vagy 2 lehet.

route_stops

Mezőnév	Típus	Leírás
<code>route_id</code>	<code>uint32</code>	Hivatkozás a <code>route_lines.route_id</code> mezőre
<code>line_id</code>	<code>uint32</code>	Hivatkozás a <code>route_lines.line_id</code> mezőre
<code>stop_index</code>	<code>uint32</code>	1-től kezdődő, folytonosan növekvő szám
<code>stop_id</code>	<code>uint32</code>	Hivatkozás a <code>stops.id</code> mezőre
<code>stop_time_n</code>	<code>uint32[]</code>	Tömb értékű mező

Egy adott vonalhoz tartozó megállókat tartalmazza. A `route_id` és `line_id` összetett kulcsként funkcionálnak. A `stop_index` csak a rendezésre szolgál, más szerepe nincs, a betöltés után nincs rá szükség. A `stop_id` megadja, melyik megállóról van szó, a `stop_time_n` mezők pedig az indulástól a megállóba érkezésig számított időt percben. Azt, hogy hány ilyen mező van, azaz hány különböző menetidőt tudunk megkülönböztetni az alkalmazásba van beépítve, Debrecen esetében ez az érték 3.

route_departures

Mezőnév	Típus	Leírás
---------	-------	--------

<code>route_id</code>	<code>uint32</code>	Hivatkozás a <code>route_lines.route_id</code> mezőre
<code>line_id</code>	<code>uint32</code>	Hivatkozás a <code>route_lines.line_id</code> mezőre
<code>day_type</code>	<code>uint32</code>	Hivatkozás a <code>day_types.id</code> mezőre
<code>time_offset</code>	<code>uint32</code>	0 és 2779 közötti szám
<code>comments</code>	<code>uint32</code>	Hivatkozás a <code>comments.id</code> mezőre

Egy adott vonalhoz tartozó indulási időket tartalmazza a végállomástól. A táblának nincs kulcsa, tetszőleges rekord ismételhető. A `time_offset` mező megadja az éjfél óta eltelt percek számát. Amennyiben ez az érték nagyobb, mint 1440, a járat a következő nap fog elindulni, de még az adott napot veszi figyelembe az indulási feltételek kiértékelésekor (ez az éjszakai járatoknál hasznos). A `comments` mező segítségével tetszőleges megjegyzés megadható. Amikor egy járat adott megállóból, adott időintervallumban történő indulási idejét kell meghatározni, a feldolgozás a következők szerint történik:

- A járat összes indulási idején végighaladva az alkalmazás elindítja azokat a járműveket, amelyek a `day_type` mező értéke által meghatározott naptípuson közlekednek, és beleesnek a keresett időintervallumba, vagy kevéssel megelőzik azt.
- A `route_lines.time_id_logic` mező segítségével meghatározza, hogy a lehetséges menetidők közül melyikkel fog végighaladni a pályán.
- Végigjárja a járat megállóit, és a keresett megállóhoz érve hozzáadja az indulási időhöz a megfelelő menetidőt.
- Amennyiben a kapott időpont még az intervallumon belül van, hozzáadja az eredményekhez.

Amennyiben ez a tábla sok rekordot tartalmaz, és a `time_offset` mezők értéke között is sok az olyan egyezés egy-egy járaton belül, amely csak a `day_type` értékében tér el, érdemes a új naptípusokat felvenni az eredeti típusok vagy művelettel történő összekapcsolásával, és az így keletkezett `day_type` rekordot felhasználva csökkenteni a duplikációt.

route_headsigns

Mezőnév	Típus	Leírás
route_id	uint32	Hivatkozás a route_lines.route_id mezőre
line_id	uint32	Hivatkozás a route_lines.line_id mezőre
stop_index	uint32	0, és a vonal megállóinak száma közötti szám
headsign	string	Tetszőleges karakterlánc

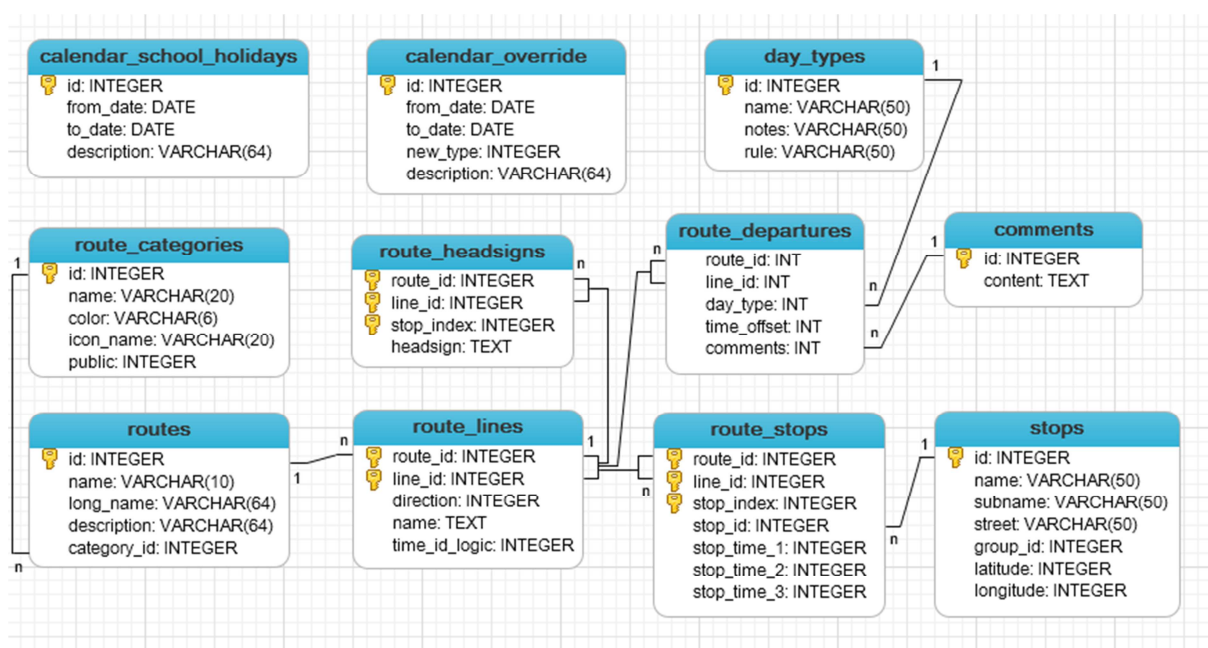
A járatok célállomásait tartalmazza, amelyet többnyire a jármű tetején elhelyezett digitális kijelző mutat. A tábla minden egyes rekordja azt írja le, hogy az adott sorszámú megálló után milyen irányba halad tovább a járat. Minden járathoz kötelező megadni a kezdő állomást, a 0-ás stop_index segítségével. Ezután meg kell adni a végállomást, vagy nem egyenes vonalban haladó járatok esetén a fontosabb érintett pontokat. A route_id, line_id és stop_index mezők összetett kulcsként funkcionálnak. A headsign mező értéke tetszőleges karakterlánc lehet, nem kell egyeznie egy megálló nevével.

A fenti struktúra segítségével csaknem tetszőleges város tömegközlekedési hálózata leírható, és elég könnyűsúlyú ahhoz, hogy az mobiltelefonokon is feldolgozható legyen.

2.4.3. Adatbázisséma

A fenti logikai terv alapján felépített adatbázis semmilyen megszorítást nem tartalmaz, a táblák pedig a beolvasásnak megfelelően rendezve vannak. Ez nem okoz problémát, hisz az alkalmazás nem módosít a kész adatbázison, azonban a megszorítások hiánya csökkenti az adatbázis méretét, és növeli a teljesítményt. Az adatbázis emellett nem tartalmaz triggereket, nézeteket és tárolt eljárásokat sem.

Az SQLite adatbázis²⁶ sémáját az alábbi diagram szemlélteti:



3. ábra (adatbázisséma)

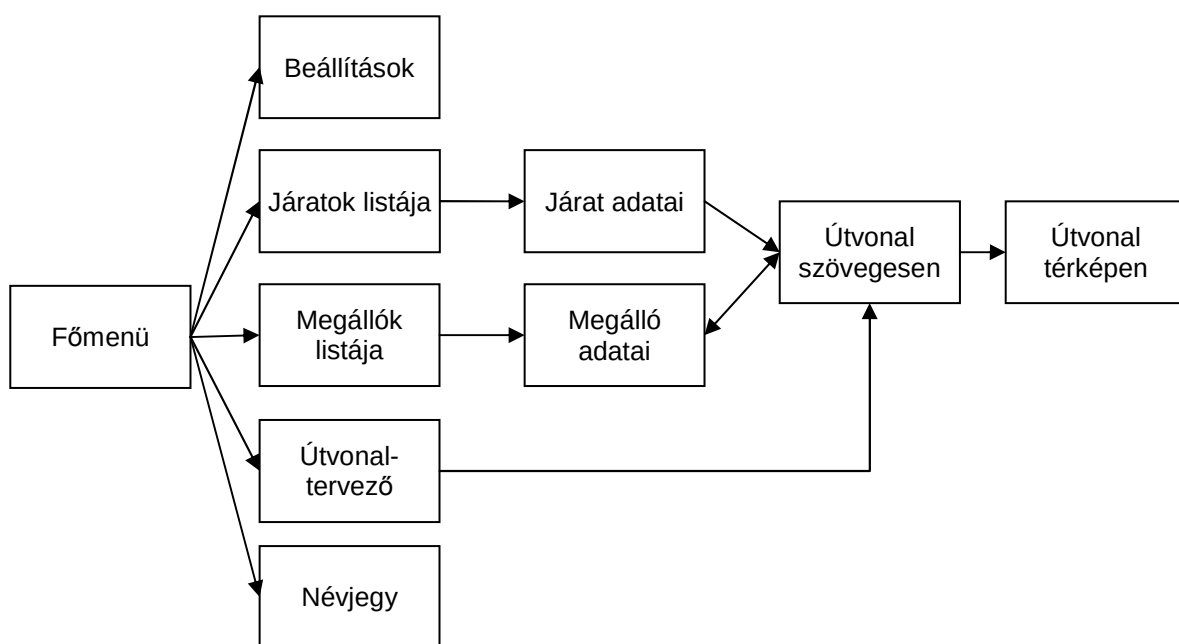
Az ábra ugyan tartalmazza a külső kulcsokat, de ezek csak az átláthatóság javítását szolgálják, a végleges adatbázisban ezek sincsenek felvéve.

2.5. Felhasználói felület

Általános elvárás, hogy a felhasználói felületeknek kényelmesen használhatónak kell lenniük¹³, és illeszkedniük kell az adott operációs rendszer témájához.

Mivel egy mobil alkalmazásról van szó, a felhasználó felület felépítését nagyban limitálják a készülékek képességei, főként a képernyő alakja és felbontása. Emiatt az alkalmazás funkcionalitását sok kis részre kell bontanom, és azokat egy-egy különálló képernyőn megjeleníteni. A felhasználó mindig csak az aktív funkcióhoz tartozó képernyőt látja, és a készülék vissza gombjával tud az előző képernyőre navigálni, vagy az alkalmazás ikonjára kattintva a főmenüre jutni. Tabletek esetén a megnövekedett terület kitöltése érdekében két képernyő is megjelenhet egymás mellett, tehát például lehetőség van az útvonal szöveges és térképes megtekintésére egyszerre.

A képernyők közötti összefüggést az alábbi ábrán szemléltetem. Az egyes képernyők a gráf csúcsainak felelnek meg, az élek pedig a lehetséges átmeneteket, és azok irányát adják meg. A visszautakat nem ábrázolom külön.



4. ábra (az alkalmazás képernyői)

2.5.1. Főmenü

A főmenü az alkalmazás indítása után jelenik meg, felépítése egyszerű és lényegre törő. Tartalmazza az alkalmazás nevét és verzióját, valamint szükség esetén egyéb figyelmeztető üzeneteket a frissítéssel, valamint az esetleges limitációkkal kapcsolatban. Az alul található gombok segítségével lehet eljutni a további képernyőkre (lásd: Függelék, 7. ábra).

2.5.2. Beállítások

A beállítások képernyőn többek között a következők megtekintésére és módosítására van lehetőség:

- Szenzorok használatának be- és kikapcsolása
- Az útvonaltervező paramétereinek megadása (séta sebessége és maximális hossza, átszállások gyakorisága, stb.)
- Az automatikus frissítés engedélyezése, az ellenőrzés gyakoriságának megadása, kézi frissítés kezdeményezése
- Az adatbázis verziójának, méretének és kiadási dátumának megtekintése

2.5.3. Járatok listája

Ezen a képernyőn jelennek meg az adatbázisban szereplő járatok, név szerint növekvő sorrendbe rendezve. A járatok mellett balra látható egy ikon, ami a járat típusát határozza meg (busz, trolis, villamos, speciális), illetve a járat neve alatt olvasható a járat leírása, amely tartalmazza a két végállomást, valamint amennyiben ez indokolt, a fontosabb közbúlsó megállókat is (lásd: Függelék, 8. ábra).

A könnyebb navigáció érdekében egy rácson is képes megjeleníteni a járatokat (lásd: Függelék, 9. ábra).

Bármely járatra történő kattintással a felhasználó az adott járat képernyőjére jut (2.6.7. fejezet).

2.5.4. Megállók listája

Megjeleníti a megállók listáját név szerint növekvő, vagy légvonalban mért távolság szerint csökkenő sorrendben, attól függően, hogy a főmenün a felhasználó melyik nézetet kérte. Lehetőség van a találatok név, utcanév vagy leírás alapján történő szűrésére is. A lista minden egyes eleme egy megállónak felel meg, és a következő információkat tartalmazza:

- a megálló neve
- az utca neve, és a megálló leírása
- azon járatok listája, amelyek áthaladnak a megállón
- a felhasználó távolsága a megállótól, és az irány, amely felől a megálló megközelíthető

A találatokra kattintva az adott megállóhoz tartozó képernyőre juthat a felhasználó (2.6.8. fejezet). Egy lehetséges képernyőképet a Függelék 13. ábrája tartalmaz.

2.5.5. Útvonaltervező

Ez a képernyő tartalmazza az alkalmazás központi részét képező útvonaltervező használatához szükséges elemeket. A funkció kiválasztása után a felhasználó megadhatja a forrás és cél megállót, valamint az indulás idejét. Az alkalmazás ezek után további beavatkozás nélkül felajánlja a lehetséges útvonalakat. Az első találat mindig gyorsan (0.5 másodperc alatt) jelenik meg, majd a következő másodpercekben további útvonalakkal bővül a lista, amelyek hossza az előzőeknél rövidebb, vagy megtétele kisebb erőfeszítést igényel a felhasználó részéről. Bármely találatra kattintva az alkalmazás kifejti az adott útvonalat a 2.6.9. fejezetben leírtak szerint. Lásd: Függelék, 16. ábra.

2.5.6. Névjegy

Ez a képernyő az alkalmazással kapcsolatos információk megjelenítésére szolgál, valamint a készítő elérhetőséget adja meg (lásd: Függelék, 21. ábra).

2.5.7. Egy járat adatai

Ezen a képernyőn láthatóak egy adott járat indulási időpontjai. A képernyő tetején a járat neve látható, melyet a kezdő- és végállomás követ. Ez alatt a felszállásra használt megálló neve, és az indulás napja adható meg (lásd: Függelék, 10. ábra). A végállomásra kattintva lehet váltani az oda-vissza irány között. Váltáskor az alkalmazás megtartja az indulási időt, és az előzőleg beállítottához legközelebbi megállót választja ki automatikusan. A megálló nevére kattintva egy felugró lista segítségével módosítható a felszállás helye (lásd: Függelék, 11. ábra), az indulás napjára kattintva pedig ez az opció is beállítható. A képernyő jobb felső sarkában található nyilakkal lehet lépkedni az előző, vagy következő járatra a járatok listájának megnyitása nélkül. Egy indulási időre kattintva az adott jármű útvonala jelenik meg (2.6.9. fejezet).

2.5.8. Egy megálló adatai

Az előzőhöz hasonlóan ennek a képernyőnek is az indulási idők listázása a célja, azonban ez nem egy adott járat aznapi indulásait jeleníti meg, hanem egy adott megállóból induló összes járatot (lásd: Függelék, 14. és 15. ábra). Az indulási időkre kattintva a felhasználó a megtekintheti az adott járat útvonalát (2.6.9. fejezet). A lista alján a megjelenített megálló közelében található további megállók találhatók, ezek használhatók például az egy utcán, de az ellenkező oldalon található megállók közötti váltásra. A képernyő tetején gyorsgombok találhatók, amelyekre kattintva útvonaltervezés indítható az adott megállóba vagy megállóból.

2.5.9. Útvonal megtekintése szövegesen

A képernyőn egyetlen lista található, amely egy útvonalat jelenít meg az érintett pontokkal, és a hozzájuk tartozó részidőkkel együtt (lásd: Függelék 17-18. ábra). Ez a képernyő szolgál egy adott járat által megtett út ábrázolására, de az útvonaltervező által visszaadott eredmény megjelenítésére is. A jobb felső sarokban található ikonnal a felhasználó térképes megjelenítésre válthat (2.6.10. fejezet).

2.5.10. Útvonal megtekintése térképen

Ez a képernyő ugyanazt a feladatot látja el, mint a szöveges változat, de az eredményeket értelem szerűen térképen mutatja meg. A térkép tetszőlegesen nagyítható / kicsinyíthető. A vonalak színe az adott járat típusát jelölik, a világos szürke pedig a gyaloglást szimbolizálja. Lásd: Függelék, 19. ábra.

2.6. Alkalmazáslogika

Az adatbázisséma („miből”) és a felhasználói felület („mit”) részletezése után elérkezett az idő az alkalmazás logika, azaz a „hogyan” bemutatására.

Mint az korábban elhangzott, az alkalmazás kódja teljes egészében Java nyelven íródott. A gyökér névtér a `hu.donmade.menetrend.debrece` nevet viseli, az egyes részekységek ez alatt helyezkednek el a saját névtérükben:

Részekység	Névtér
Tartalomkezelő	<code>content</code>
Adatmodell	<code>model</code>
Útvonaltervező	<code>pathfinder</code>
Felhasználói felület	<code>gui</code>
Szolgáltatások	<code>services</code>

A részekységek között a publikus interfészekeken tetszőleges kommunikáció engedélyezett.

2.6.1. Tartalomkezelő

A tartalomkezelő feladata a frissítések ellenőrzése, letöltése és a letöltött fájlok kezelése. Fontosabb osztályai a következők:

- `ContentManager`
- `RepositoryIndex`
- `SyncOperationList`

Az adatbázisokat egy tároló (repository) keretében kezeli, amelynek gyökerében egy `updates.xml` nevű fájl található. Ez a fájl tartalmazza azokat az URL-eket, ahol elérhetőek az adatbázisok, valamint minden egyes adatbázishoz néhány metaadatot (kiadás dátuma, tömörített méret, kibontott méret, verzió, fájlformátum, leírás, stb).

Indításkor az alkalmazás megnyitja a helyi `updates.xml` állományt, és annak tartalmát beolvassa a `RepositoryIndex` osztály egy példányának mezőibe, ahonnan a későbbi feldolgozás történik. Az itt található információk alapján képes eldönteni, hogy melyik a legfrisebb adatbázis, azaz melyiket kell betölteni.

A frissítés során az alkalmazás a helyi `RepositoryIndex` alapján csatlakozik egy véletlenszerűen kiválasztott tükörszerverhez, majd onnan letölti a friss `updates.xml`

állományt, és ennek tartalma szerint frissíti az objektum mezőit. Az új indexet a meglevő fájlokkal összehasonlítva eldönti, hogy szükség van-e a frissítésre, és a végrehajtandó műveletek (letöltés, törlés, átnevezés, stb.) leírására egy `SyncOperationList` osztályt példányosít. Ezt átadja a felhasználói felületnek, amely értesíti a felhasználót a frissítés lehetőségéről és méretéről, majd annak beleegyezése esetén végrehajta azt. Ezután az alkalmazás újraindul, és már a frissített adatbázist tölti be.

2.6.2. Adatmodell

Ezen csomag osztályai reprezentálják az adatbázis tartalmát, és itt történik a számítások jelentős része is. Fontosabb osztályai a következők:

- `DataStore`
- `WorkdayOverride`, `SchoolHoliday`, `DayType`, `Route`, `RouteCategory`, `RouteLine`, `Stop`
- `LocalCalendar`, `LocalDate`
- `Location`, `Orientation`
- `RouteInstance`

A `DataStore` osztály felelős a tartalomkezelő által kiválasztott adatbázis megnyitásáért, és annak tartalmának beolvasásáért. A beolvasás során minden adat lekérdezésre kerül, majd az adatbázis bezárul, az alkalmazás további futása során nincs rá szükség. Az egyes táblák tartalma listákba kerül, amelyekben minden egyes elem egy rekordot képvisel az adott táblából. Minden táblához tartozik egy saját osztály, amely a tábla rekordjait képes reprezentálni. Ezek a táblához hasonló nevet viselnek, például a `routes` tábla rekordjait `Route` objektumok írják le. A beolvasás során az alkalmazás a szükségtelen mezőket elhagyja, majd a külső kulcsokat referenciákra cserélve az adatokból hierarchiát épít. Ezután különféle gyorsítótárakat épít fel, például minden egyes megállóhoz hozzárendeli a rajta áthaladó járatok listáját. A beolvasás részeként létrejön a `LocalCalendar` osztály, amely a dátum-műveleteket képes kezelni. Az osztály metódusai által visszaadott értékek (munkanap eldöntése, timestamp – dátum konverziók) azok időigényessége miatt erősen gyorsítótárazva vannak.

A `Location`, `Orientation` és `LocalDate` osztályok több primitív típus együtt történő kezelését szolgálja, és az általuk reprezentált adatokon értelmezhető műveleteket (távolság számítás, összehasonlítás, stb.) is elvégzik.

A számítások középpontjában a `RouteInstance` osztály áll. Ennek egy példánya tulajdonképpen egy jármű, amely egy adott időben a pályán tartózkodik. Tartalmazza az indulási időt, a megállókat és a pontos menetidőket. Bármilyen művelet, amelyet az alkalmazásnak el kell végeznie először az érintett `RouteInstance` osztályok példányosításával kezdődik. A központosított megoldásnak köszönhetően csak egy helyen kell feldolgozni az egyébként meglehetősen összetett indulási szabályokat, így egyrészt javul a kód karbantarthatósága, másrészt a változó értékek korai kiszámolásával javítható a többi funkció teljesítménye.

2.6.3. Útvonaltervező

Ez a részegység tartalmazza az útvonal tervezéshez szükséges algoritmusokat. Részletesen a 2.7-es fejezetben tárgyalom.

2.6.4. Felhasználói felület

A felhasználó felület több, mint 30 osztály tartalmaz, ezek két csoportra bonthatók:

- Vezérlőelemek
- Adapterek

A vezérlőelemek (control) a felhasználói felület egy részét valósítják meg. Ide tartoznak a teljes képernyőt betöltő `Activity`-k, az ezeken belül szabadon elhelyezkedő `Fragment`-ek, valamint az összes egyedi komponens (többszintű listák, párbeszédablakok) implementációja. Ezek közvetlenül csak ritkán (többször az inicializációjuk során kommunikálnak az adatmodellel), az adatok megjelenítéséhez az adaptereket veszik igénybe.

Az adapterek olyan osztályok, amelyek egyrészt manipulálás céljából kapcsolódnak a felhasználói felület elemeihez, másrészt eléri az adatmodellről kapott objektumokat is, és ezek segítségével építik fel a listákat, beviteli mezőket, stb.

Egy járat megjelenítése például – némileg leegyszerűsítve – úgy történik, hogy a felhasználói felület megfelelő vezérlője (`RouteActivity`) lekérdezi a járat adatait az adatmodellről, válaszul egy `Route` típusú objektumot kap, majd a statikus mezőket (például címsor) feltölti ennek segítségével. Ezután a felhasználói felület többi részét adapterekből építi fel, amelyek lekérdezik a járhoz tartozó megállók listáját, valamint példányosítják az adott napon, adott irányba közlekedő összes járművet, és ezek indulási idejét a felszálló megálló figyelembe vételével lista formájában megjelenítik.

2.6.5. Szolgáltatások

Az alkalmazás egyetlen szolgáltatása az `UpdateService` nevet viseli, amely felelős a tartalomkezelő periodikus aktiválásáért és a frissítések felhasználói beavatkozás nélküli lebonyolításáért. Az Android rendszer ütemezőjét¹⁶ használva aktiválódik, majd feladata befejeztével leáll.

2.7. Az útvonaltervező megvalósítása

Az útvonaltervező egyetlen funkciója két megálló közötti útvonaltervek visszaadása. Mivel a tervezés többnyire nem előre történik, hanem közvetlenül az utazás közben, ezért a pontosság mellett fontos szempont a sebesség is. Emiatt megengedett, hogy az alkalmazás ne az optimális utat adja meg, csak annak egy közelítését, ha az jelentősen kevesebb idő alatt kiszámítható. Elvárás viszont, hogy az útvonal teljesíthető legyen, azaz kellően pontosan becsülje a sétához, átszálláshoz szükséges időt, és vegye figyelembe a menetrendben szereplő indulásokat, menetidőket.

Mivel Debrecen domborzata meglehetősen egyhangú, a számítások során nincs szükség a magasság kezelésére, valamint az egyenes utat elzáró objektumok (hegyek, folyók, stb.) kikerüléséről sem kell gondoskodni. Emiatt az útvonaltervezőnek nincs szüksége domborzati térképre, és a forgalmas, utcákkal sűrűn behálózott területek viszonylagos nagy száma miatt az utcaterkép is elhagyható. Ez utóbbi döntés szükségszerű is, hisz nincs pontos, részletes, offline, szabadon felhasználható utca szintű térkép Debrecenhez. Szintén nem cél a házszámok kezelése sem, hiszen ehhez is szükség lenne a megfelelő offline térképekre, azonban élő internetkapcsolat esetén ezek leolvashatók az útvonalat ábrázoló térképről. A megállók közötti távolságok becslése légvonalban történik, amely néhány összetettebb közlekedési csomópont kivételével kielégítő eredményt ad.

Ahhoz, hogy kiválasztható legyen egy közel optimális útvonal, rangsorolni kell a lehetséges utakat. Ez több szempont szerint is történhet: a megtett út hossza, az eltelt idő, az átszállások száma, a séta hossza, stb. A gyakorlatban a felhasználókat kevésbé érdekli, hogy milyen hosszú utat kell megtenniük, hiszen egy tömegközlekedési eszközön ülve nem fáradnak el. Számít viszont az eltelt idő, az átszállásokkal vállalt kockázat, valamint a séták által okozott kellemetlenség. Emiatt az alkalmazás az útvonalakat elsődlegesen az eltelt idő alapján rangsorolja, de ezt az időt az átszállások és séták miatt különféle büntetőpontokkal növeli. Előfordulhat tehát, hogy egy látszólag hosszabb útvonal előrébb kerül a rangsorban, ha a megtételéhez szükséges idő rövidebb, mint az utána következők esetén. Ez esetenként meglepő lehet, de a gyakorlatban nem hiba.

2.7.1. Elméleti megvalósítás

Ahhoz, hogy két pont között optimális útvonalat találjunk, a megállókat tartalmazó gráfban kell megkeresni a legrövidebb utat. Ez a probléma a Dijkstra algoritmus¹⁷ segítségével könnyen megoldható, heurisztika¹⁸ alkalmazásával („A” algoritmus¹⁸) pedig gyorsítható. A valóságban azonban a helyzet ennél bonyolultabb, ugyanis az megállókat összekötő útvonalak költsége időben változik (a pályán mozgó járatok aktuális pozíciója, így a várakozási idő miatt). Ennek a problémának a megoldása érdekében az, hogy mit tekintek állapotnak némileg változik:

- A felhasználó **állapota** két tényezőből áll össze: a jelenlegi pozíciója, és az indulás óta eltelt idő. Ahhoz, hogy ne legyen végtelenül sok állapot, a maximális keresési időt meg kell adni, ez a jelenlegi implementációban 3 óra. A fenti feltételekkel bármely állapot esetén egyértelműen meghatározható, hogy a felhasználó abból az állapotban állva milyen járatokat érhet el, és ezekkel melyik megállóba juthat, így definiálható az operátor fogalma. Az állapotokra két **kényszerfeltétel** adható meg:
 - az megállóknak a megállók halmazából kell kikerülnie
 - az eltelt idő nem lehet negatív
- A **kezdőállapot** a (kezdőmegálló, 0) páros.
- **Célállapot**nak tekintek minden olyan állapotot, ahol a pozíció komponens megegyezik a célmegállóval.
- **Operátornak** tekintem a sétát, vagy egy járaton történő közlekedést egy tetszőleges megállóba. Az operátor alkalmazásaként egy olyan állapot jön létre, ahol a megálló komponens az operátor célja, az idő komponens pedig az előző állapot időkomponensének, és az operátor végrehajtásához szükséges időnek az összege. Egy állapotra alkalmazható operátorok listáját megkaphatom az összes adott távolságon belüli megálló iterálásával (séta), valamint az adott megállón egy adott időn belül áthaladó járatok feldolgozásával (utazás).
- Egy operátor **költsége** az operátor végrehajtási idejéből és a felhasználónak okozott kellemetlenség összegéből adódik. Jelenleg egy átszállás 3 percnyi, egy séta elkezdése pedig 5 percnyi utazásnak felel meg (de ez a beállítások segítségével változtatható).
- Egy megálló **heurisztikáját** a céltól való távolsága határozza meg. Ez közel sem pontos (hisz a felhasználót általában nem érdekli a távolság, csak az eltelt idő), de egyszerű, gyors, és meglepően jó eredményre vezet.

A fenti definíciók alapján már lehetőség van a probléma megoldására az „A” algoritmus segítségével. Másik algoritmus is használható lenne, de azok eredménye alulmaradna a választottal szemben: a szélességi keresés például garantáltan megtalálná a legjobb utat, de túl sokáig futna, a mélységi keresés¹⁹ pedig a szerteágazó tömegközlekedési hálózat miatt nem lenne hatékony.

Az algoritmus nem fogja garantáltan megtalálni a legrövidebb utat, mivel a heurisztika nem teljesíti azt a feltételt, hogy a cél elérésének tényleges költségét minden esetben alulról közelítse, azonban ez nem is követelmény.

2.7.2. Gyakorlati megvalósítás

A fenti elvek megvalósítását hosszas gyakorlati tesztelés és optimalizálás követte, így alakult ki az útvonaltervező jelenlegi állapota, amely az alábbiakban tér el az előző fejezetben leírtaktól:

- Mivel az eredeti elvek szerint kódolt „A” algoritmus túlságosan sok időt töltött az útvonal apró finomításával, a gyorsabb eredmény elérése érdekében a heurisztika súlyozásra került (weighted A). Az útvonalak tervezése több lépésben történik. Kezdetben a heurisztika többszörösen kerül beszámításra, ami gyorsítja az algoritmust végrehajtását, így az hamar ad egy viszonylag jó eredményt. Ezután további keresések történnek, amelyek során a heurisztika súlyozása csökken, és végül visszaáll 1-re, ezzel együtt az algoritmus lassul, de az eredmény egyre optimálisabb lesz. A felhasználó így azonnal lát egy elfogadható utat, ami a további feldolgozás során folyamatosan finomításra kerül.
- Az előfeltételek vizsgálata és az operátorok alkalmazása összevonásra került a segédváltozók többszöri kiszámításának megelőzése érdekében
- Az operátorok a szokásos egy helyett tetszőlegesen sok állapotot adhatnak vissza. A séta operátora az összes közeli megállót feldolgozza egyszerre, az utazás operátor pedig az összes lehetséges leszállási lehetőséget figyelembe veszi. Ez egyrészt tovább csökkentette a végrehajtandó műveletek számát, másrészt az algoritmust mohóvá is tette: a megoldás során nem megállónként araszol előre, hanem a lehető legtöbb időt tölti el egy járaton, majd a célhoz legközelebbi megállónál száll le, és onnan próbál tovább utazni. Ez többnyire megegyezik az emberek természetes viselkedésével is, azaz amellett, hogy ez a módosítás akár 5-10-szeres sebességbeli javulást hozott,

sikerült elérnie, hogy rögtön a legelső találatként kapott útvonal pontosan az legyen, amit a felhasználó maga is megtalált volna – percekig tartó keresés után. A következő iterációk ezután tovább pontosítják a találatot, és olyan útvonalat adnak, amelyek valószínűleg elkerülték volna egy ember figyelmét. A régi és az új megvalósítás közötti különbséget a Függelék 22. ábrája szemlélteti.

Ezen módosítások figyelembe vételével az implementáció a következő képpen alakul:

A `PathFinder` osztály megfelelő metódusait meghívva egy útvonaltervezés kezdeményezhető. A hívó feliratkozhat a keresés állapotának változásaira, ezzel értesítést kap egy új útvonal felfedezéséről, vagy a keresés befejezéséről. A keresés megkezdésekor példányosítja az `AStar` osztályt, és beállítja a kezdő és célállapotokat.

Az `AStar` osztály az adatmodell segítségével példányosítja a keresés időintervallumában pályán levő összes járatot (`RouteInstance`), rendezi őket, felépíti a gyorsírotárakat, majd beállítja a kezdő és végállapotot, és példányosítja az operátorokat. Ezek után elindít egy keresést csak a séta operátorok segítségével, hogy tájékoztassa a felhasználót a kért útvonal gyalog történő megtételének hosszáról. A kezdeti keresés befejeződése után 3 iteráció következik, amelyben a heurisztika értékeit folyamatosan csökkenti. Minden iteráció után közli a kapott végeredményt a `PathFinder` osztállyal, amely az ismétlődések kiszűrése után értesíti a felhasználói felületet a változásokról.

A `State` osztály reprezentál egy állapotot. Tárolja az eltelt időt, a jelenlegi megállót, és optimalizációs célból a jelenlegi járatot is (ha van), de ez utóbbi nem vesz részt a számításokban. Az állapotok összehasonlítása csak a megálló segítségével, ezzel csökkenthető az állapotok száma a végeredmény befolyásolása nélkül.

Az `Operator` interface definiálja azokat a metódusokat, amelyeket minden operátornak implementálnia kell. Jelenleg ez egyetlen metódust tartalmaz (`applyAll`), amely alkalmazza az operátort egy állapotra, és visszatér az új állapotokkal, vagy egy üres listával, ha nem alkalmazható.

A `WalkOperator` a sétát kezeli, alapértelmezés szerint 650 méteres távolságlimittel és 5 km/h-s sebességgel. Ez a beállításokban megváltoztatható, de a tesztek alapján ezek bizonyultak a legjobb értékeknek.

A `ChangeRouteOperator` kezeli egy járatra történő felszállást. Minden egyes járat pályához egy példány tartozik, amely tartalmazza az összes járművet indulási sorrendben.

Az útvonaltervező megvalósításánál általános cél volt, hogy csökkentsem a memóriefoglalások és a fölösleges metódushívások számát, mivel ezek emésztették fel a készülékek számítási kapacitásának legnagyobb részét. Ennek érdekében a legtöbb memóriefoglalás a keresés megkezdése előtt történik, az algoritmus futása alatt már csak az ágak leírásához szükséges memória kerül lefoglalásra. Eltávolítottam továbbá a gettereket és settereket, helyettük minden kívülről is elérhető mező láthatósága default-ra változott.

A már bejárt csúcsokat az általában megszokott `ArrayList` egy `HashMap`-ben tárolom, a legjobb elem kiválasztását pedig egy redundáns tartalommal rendelkező `PriorityQueue` biztosítja.

Az így megvalósított algoritmus nagyon jól skálázódik a járatok számának növekedésével, viszont a megállók feldolgozása sok időt vesz igénybe. Egy jó optimalizálási lehetőség a séta operátorok használatának ritkítása, és a város rács-szerű felosztása, hogy minél kevesebb összehasonlításra legyen szükség a közeli megállók meghatározásához. Ezeket a módosításokat az alkalmazás jelenlegi verziója még nem tartalmazza, alkalmazásukról később fogok dönteni.

2.8. Lényegesebb optimalizálási lehetőségek

Az útvonaltervező optimalizálása mellett fontos a teljes alkalmazás sebességének növelése is. A következő fejezetekben kiemelem a legfontosabb teljesítménnyel kapcsolatos módosításokat.

2.8.1. Általános módszerek a sebesség növelésére

A legtöbb sikeres optimalizálást mérések előznek meg, ehhez az Android DDMS²⁰ és Traceview²¹ eszközeit használtam. Ezek segítségével megmértem, hogy mi kerül a legtöbb időbe az alkalmazás futása során. Az eredmények azt mutatták, hogy a számítások több, mint 70%-át a példányosítások és a garbage collector tette ki. Ennek megoldása érdekében átszerveztem a kódot, és minden lehetséges helyen átírtam az ideiglenes objektumokat használó kódrészleteket úgy, hogy a lehetőségekhez mérten minél kevesebb objektumot hozzanak létre, illetve használják újra a már létrehozottakat.

További javulást hozott az ArrayList és HashMap osztályok tömbökre történő cserélése, ahol erre lehetőség volt, még akkor is, ha ez a kód átszervezésével járt.

Mivel az adatmodell által létrehozott objektumok egyediek, így fölösleges a megszokott hashCode és equals implementációkat használni. Az adatbázisban található elsődleges kulcsok tökéletes hashCode-ot alkotnak, míg az egyenlőség vizsgálata a == operátor segítségével is elvégezhető. A sebességkritikus kódrészletekben mellőztem ezen függvények használatát, és az implementációjukat illesztettem be a megfelelő helyre.

Általánosan is igaz, hogy érdemes a fölösleges függvényhívásokat kerülni, így minden olyan osztály mezőit, amelyek csak a konstruktorból kapnak értéket `public final` módosítóval láttam el, és töröltem a gettereket.

A fenti módszerek jelentősen növelték az alkalmazás teljesítményét, viszont néhány területen speciális megoldásokra volt szükség. Ezek egyike az útvonaltervező, amelynek optimalizálását a fentiekben már leírtam. A másik gyakran végrehajtott kódrészlet a távolságok kiszámításával kapcsolatos.

2.8.2. Távolságok számítása

A „megállók a közelben” funkció és az útvonaltervező működéséhez is kulcsfontosságú a megállók közötti távolságok és irányzék kiszámítása. Az alábbi műveleteket a Location

osztály implementálja, a `getDistance`, `getDistanceFast` és a `getAngle` metódusok segítségével.

2.8.2.1. Két pont közötti távolság kiszámítása

A Föld felszínén két, latitude és longitude értékekkel meghatározott pont közötti távolságot ortodromikus távolságnak nevezzük (az angol szakirodalomban Great Circle Distance²²). Ennek kiszámítására több lehetőség is van, ezek közül a legkézenfekvőbb megoldás a gömbi koszinusztétel használata:

$$d = R * (\arccos(\sin(lat_1) * \sin(lat_2) + \cos(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)))$$

ahol $R \approx 6371$ km. A klasszikus képlet sajnos kis távolságok esetén pontatlan, ezért a tényleges számításokhoz a haversine formulát²³ használom, amely a következőképpen alakul:

$$d = R * 2 * \arcsin\left(\sqrt{\sin^2\left(\frac{lat_2 - lat_1}{2}\right) * \cos(lat_1) * \cos(lat_2) * \sin^2\left(\frac{lon_2 - lon_1}{2}\right)}\right)$$

Valójában ez is csak egy közelítés, hisz nem veszi figyelembe a Föld ellipszoid mivoltát, de mivel a bolygónk csak 0,5%-al tér el a tökéletes gömbtől, a számítás eredményét pontosnak tekintem.

2.8.2.2. Az ortodromikus távolság közelítése²⁴

A sebesség növelése érdekében a fenti megoldás egy kevésbé pontos, ám annál gyorsabb megoldással helyettesíthető. Ehhez kihasználom azt a tényt, hogy egy tipikus közép-európai város átmérője nem haladja meg a néhány tíz km-t, így a pontok távolságát közelíthetem az euklideszi távolság segítségével is. Ehhez a föld felszínét síknak tekintem, a latitude és longitude értékeket pedig a síkon vett x, illetve y tengely irányú mérőszámoknak. Két tetszőleges pont közötti vízszintes, illetve függőleges távolság az eredeti latitude és longitude értékekből a következő képletek segítségével számítható ki:

$$x = (lon_2 - lon_1) * \cos\left[\frac{(lat_1 + lat_2)}{2}\right]$$

$$y = lat_2 - lat_1$$

Minden szög radiánban értendő.

Látható, hogy amíg a kapott távolságot Y irányban csak a latitude értékek határozzák meg, addig X irányban az függ az egyenlítőtől mért távolságtól is.

A végeredményhez a Pitagorasz-tétel alapján ki kell számolni a két komponensből a tényleges távolságot, majd megszorozni a már megismert konstanssal. A végső képlet tehát így alakul:

$$d = R * \sqrt{x^2 + y^2}$$

2.8.2.3. Initial bearing kiszámítása²³

A távolságon kívül a felhasználók kényelme érdekében szükség van arra a szögre is, amelyet egy A pontból indulva fel kell venni, hogy a föld felszínén előre haladva a B pontba jussunk. Ezt a szöget az északi sarktól mérjük az óramutató járásával megegyező irányban, és initial bearingnek nevezzük. Kiszámítása a következőképpen lehetséges:

$$\alpha = \text{atan2}(\sin(\Delta\text{long}) * \cos(\text{lat2}), \\ \cos(\text{lat1}) * \sin(\text{lat2}) - \sin(\text{lat1}) * \cos(\text{lat2}) * \cos(\Delta\text{long}))$$

Fontos megjegyezni, hogy a követendő szög nagy távolságok megtétele esetén nem marad állandó a teljes útvonalon, egy városon belül viszont - hacsak az nem a sarkok közvetlen közelében található - nem szükséges ezzel számolni, és a felhasználó számára sem kell megjeleníteni a változást. Számolni kell viszont azzal, hogy a navigáció során a felhasználó nem feltétlenül északi irányba néz, így a megjelenítés előtt a kapott szöget egy egyszerű összeadással át kell számolni az irányához képest vett relatív szögre.

2.8.3. További lehetőségek

Az eddig leírt módszerekkel sikerült az elérhető határok közelébe vinni a program teljesítményét, a további fejlődéshez már drasztikusabb megoldásokra van szükség. Ezek egyike a teljes adatmodell, és az útvonaltervező C++ nyelven²⁵ történő újraírása. Ez nagy feladat, viszont a sebesség növelése mellett egyben javítaná az alkalmazás hordozhatóságát és modularitását is, így a későbbiekben valószínűleg be fog következni.

3. Összefoglalás

Jelen dolgozatban elkészítettem egy olyan alkalmazást, amelynek segítségével a jelenleg elérhető eszközöknél egyszerűbben átlátható Debrecen város tömegközlekedési hálózata. A program teljesíti a vele szemben támasztott követelményeket, azaz egy gazdag felhasználói felületen, gyorsan és egyszerűen bocsájtja a felhasználók rendelkezésére az információkat.

Az előző fejezetekben ismertettem a szoftver felépítését a követelményektől kezdve az adatbázis és a felhasználói felület leírásán át az alkalmazás logika működéséig. A program implementálása során, és így a dolgozatban is különösen figyelmet fordítottam a kód optimalizálására. Ez a folyamat időben a teljes alkalmazás elkészítésének legalább egyharmadát tette ki, azonban a végeredmény szempontjából egyértelműen megérte.

3.1. Továbbfejlesztési lehetőségek

Habár a dolgozatban több helyen is megemlítettem néhány fejlesztési lehetőséget, azok mind az alkalmazás működésére és sebességére vonatkoztak. Fontosnak érzem kiemelni azonban, hogy a szoftver legalább ugyan olyan fontos része az adat is, amelyet folyamatosan napra készen kell tartani, hogy a felhasználók továbbra is hasznát vegyék az alkalmazásnak.

Az adatok frissítése jelenleg kézzel történik, így ez roppant időigényes folyamat. A jövőben megsokszorozná az alkalmazás használhatóságát, ha sikerülne az adatok beszerzését automatizálni, esetleg valamilyen egyezsége jutni az érintett szervezetekkel az adatok feldolgozható formában történő közzétételét illetően.

Alternatív megoldásként az alkalmazás felhasználói is bevonhatóak az adatok gyűjtésébe: a készülékek GPS szenzora által szolgáltatott adatok automatikusan feltölthetők egy központi szerverre, amely így számon tudja tartani a járművel pozícióját, azok útvonalát, és az esetleges késéseket, dugókat. Ehhez azonban megfelelő nagyságú felhasználói közösségre, és nem csak informatikai témájú kutatásra van szükség.

Másrészt, az elkészült alkalmazás jelenleg csupán egy várost fed le, holott ennél jóval több helyen lenne igény rá, így a jövőbeni fejlesztés fontos része terjeszkedési lehetőségek keresése és kiaknázása. Ehhez nem csupán a menetrendi adatok beszerzésére van szükség, hanem a helyi közlekedési viszonyok elemzésére is, beleértve a nagyobb forgalmi csomópontok megismerését, és fontosabb domborzati tényezők összegyűjtését. Szükség esetén a megfelelő térképeket is fel kell kutatni, majd az új követelményeknek megfelelő funkciókat implementálni.

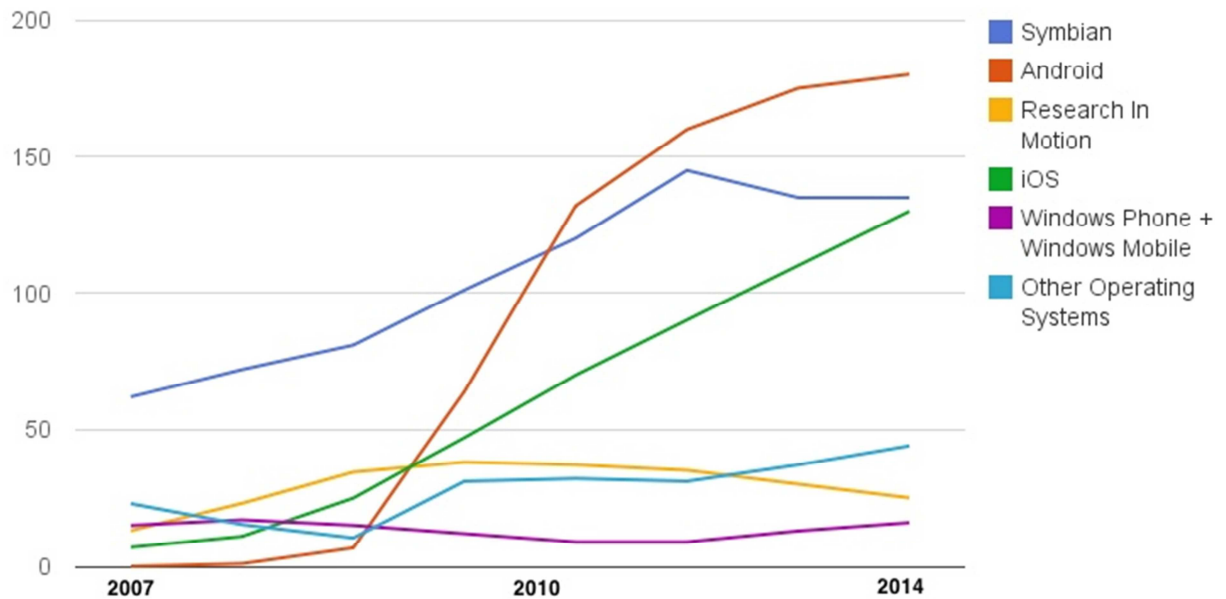
4. Függelék

4.1. Irodalomjegyzék

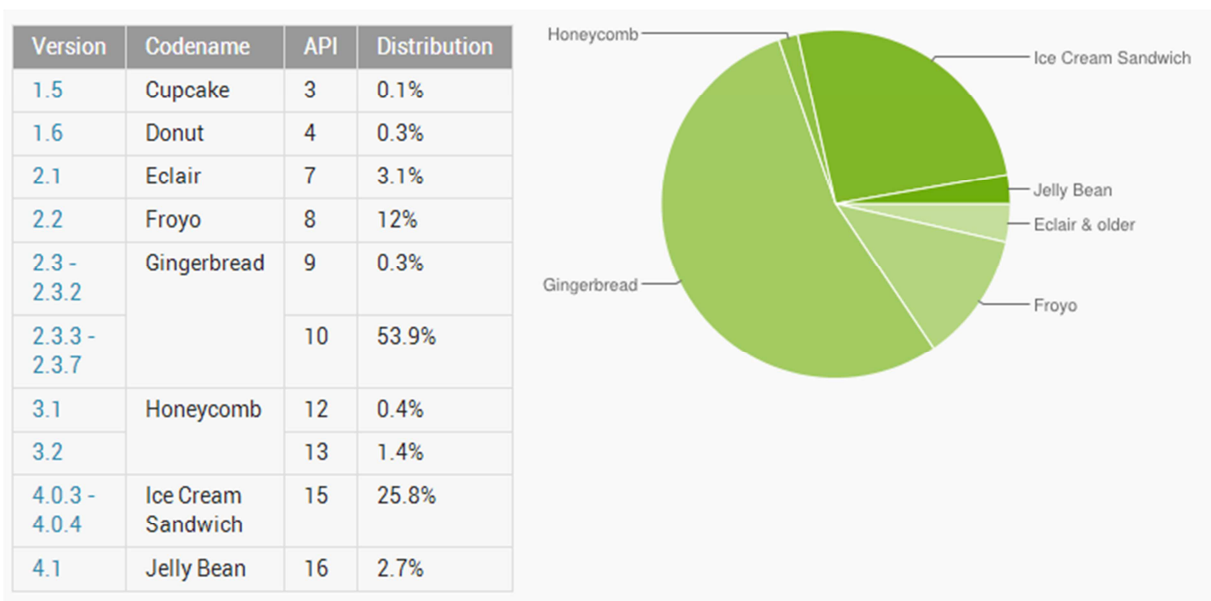
1. Ian Sommerville: Szoftverrendszerek fejlesztése, Panem, 2007
2. Rick Rogers, John Lombardo, Zigurd Mednieks, G. Blake Mekie:
Android Application Development: Programming with the Google SDK
O'Reilly, 2009
3. Bruce Eckel: Thinking in Java, Prentice-Hall, 2002
4. Joshua Bloch: Effective Java, Addison-Wesley, 2008
5. Eclipse dokumentáció
<http://help.eclipse.org/indigo/index.jsp>
6. Android SDK dokumentáció
<http://developer.android.com/sdk/index.html>
7. Paint.NET dokumentáció
<http://www.getpaint.net/doc/latest/index.html>
8. <http://www.stackoverflow.com/>
9. GTFS felhasználói dokumentáció
<https://developers.google.com/transit/gtfs/>
10. GTFS fejlesztői dokumentáció
<https://developers.google.com/transit/gtfs/reference>
11. <http://www.google.com/intl/en/landing/transit/#mdy>
12. <http://www.gtfs-data-exchange.com/agency/bkk/>
13. Webes alkalmazások használhatóságára vonatkozó alapelvek
<http://www.jaysonjc.com/programming/top-10-web-application-usability-guidelines-checklist.html>
14. Felülettervezési irányelvek Android operációs rendszerre vonatkozóan
<http://developer.android.com/design/get-started/principles.html>
15. Ikonokra vonatkozó irányelvek Android operációs rendszerre vonatkozóan
<http://developer.android.com/design/style/iconography.html>
16. Az AlarmManager osztály dokumentációja
<http://developer.android.com/reference/android/app/AlarmManager.html>
17. Fekete István, Gregorics Tibor, Nagy Sára: Bevezetés a mesterséges intelligenciába,
LSI Oktatóközpont, 1990

18. Stuart Russell, Peter Norvig: Mesterséges Intelligencia – Modern megközelítésben, Panem, 2005
19. A Debreceni Egyetem Programtervező Informatikus BSc szakának Mesterséges Intelligencia előadásának anyaga, összeállította: Dr. Várterész Magdolna
<http://www.inf.unideb.hu/~varteres/mi/tartalom.htm>
20. Android DDMS dokumentáció
<http://developer.android.com/tools/debugging/ddms.html>
21. Android Traceview dokumentáció
<http://developer.android.com/tools/debugging/debugging-tracing.html>
22. Weisstein, Eric W. "Great Circle." From MathWorld - A Wolfram Web Resource.
<http://mathworld.wolfram.com/GreatCircle.html>
23. Movable Type Scripts – Distance Calculations
<http://www.movable-type.co.uk/scripts/latlong.html>
24. Flat-surface formulae
http://en.wikipedia.org/wiki/Geographical_distance#Flat-surface_formulae
25. Bjarne Stroustrup: The C++ Programming Language, Addison-Wesley, 2000
26. SQLite Documentation
<http://www.sqlite.org/docs.html>
27. http://www.allaboutsymbian.com/features/item/12554_Oddly_Symbian_at_number_2_make.php
28. <http://developer.android.com/about/dashboards/index.html>

4.2. További ábrák és képernyőképek



5. ábra (mobiltelefon eladások becsült száma operációs rendszer szerint)²⁷



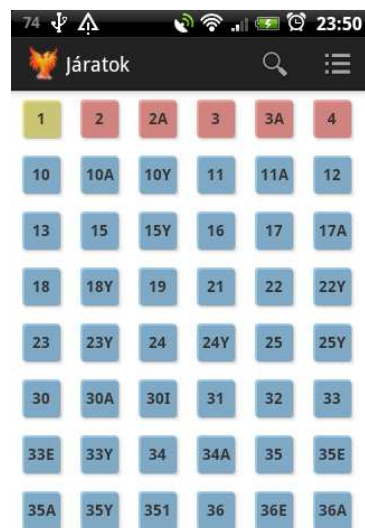
6. ábra (az egyes Android verziók elterjedtsége 2012. November 1-én)²⁸



7. ábra



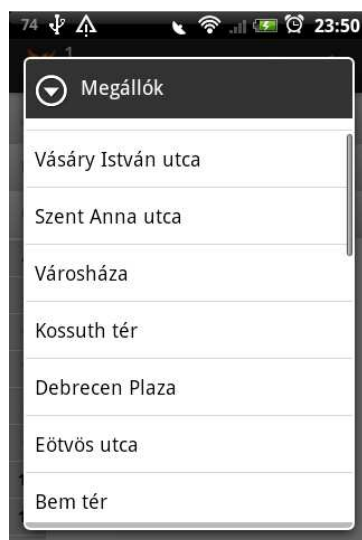
8. ábra



9. ábra



10. ábra



11. ábra



12. ábra



13. ábra



14. ábra



15. ábra



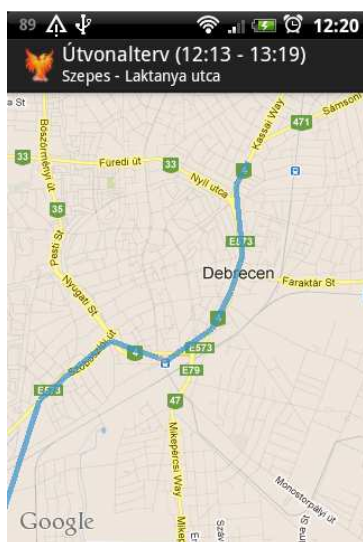
16. ábra



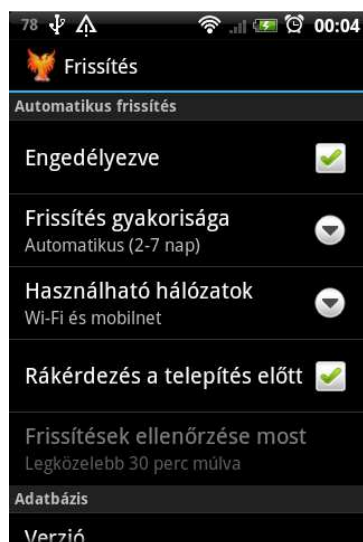
17. ábra



18. ábra



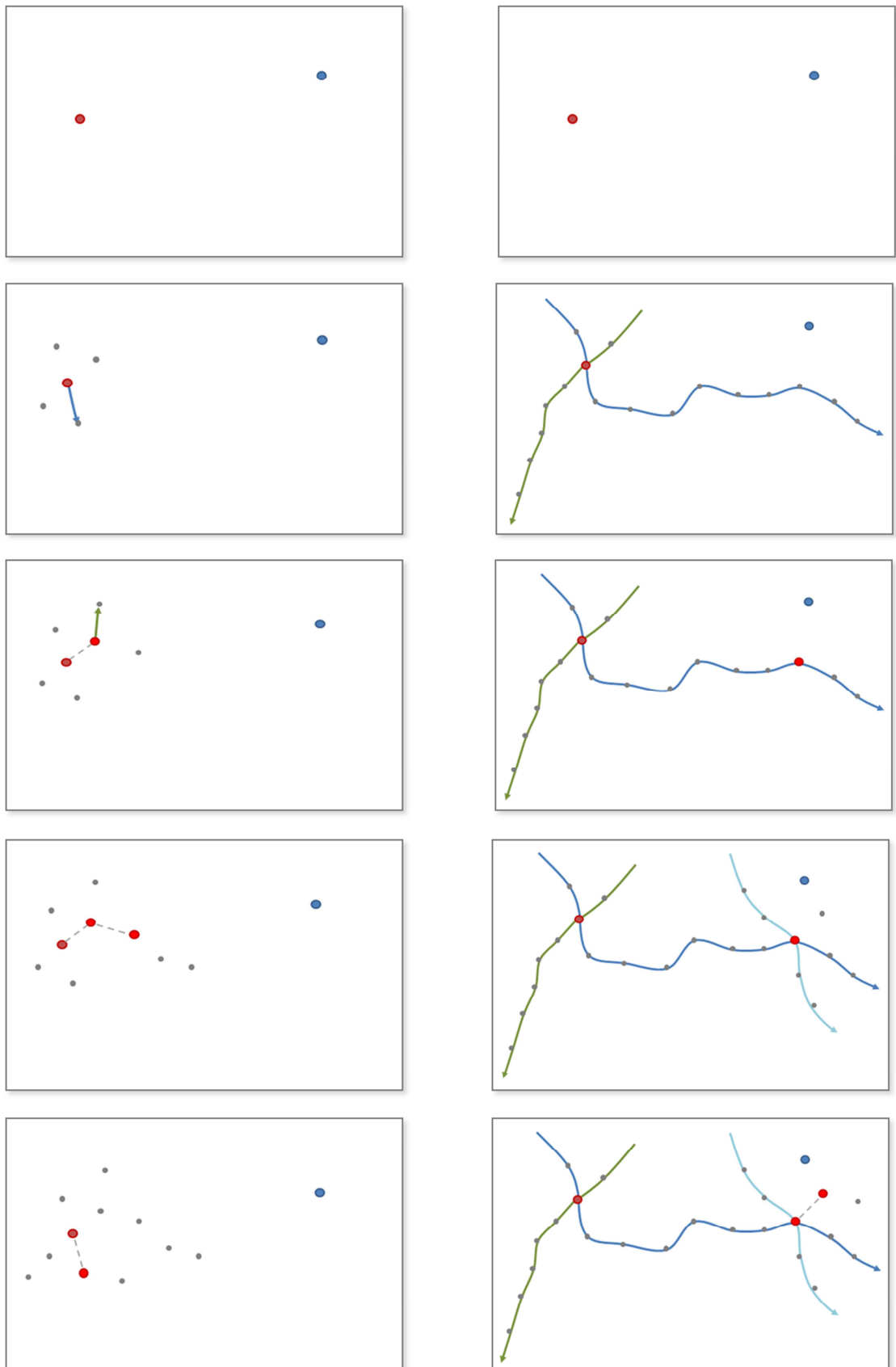
19. ábra



20. ábra



21. ábra



22. ábra - Az útvonaltervező eredeti megvalósítása (balra), és a gyorsított verzió (jobbra)

4.3. Saját munka leírása

A dolgozatban leírt szoftver terve és megvalósítása teljes egészében a saját munkám tükrözi, beleértve a következő feladatokat:

- Kis teljesítményű mobiltelefonokon is kezelhető adatbázis-formátum kidolgozása városi tömegközlekedési hálózatok leírásához
- Az alkalmazás logika megvalósítása
- A felhasználói felület megtervezése
- Az útvonaltervező megtervezése, és megvalósítása