

BABEȘ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

GRoutes - A practical application of the travelling salesman problem

Abstract

The present paper describes a practical application of the travelling salesman problem (TSP). In the first chapters I describe the theoretical background of the work, the definition of graphs, notions in graph theory, graph problems, algorithmic approaches, TSP algorithms, reduction of the asymmetric TSP to a symmetric one. The emphasis is on selecting an algorithm that matches, and serves our use case practically.

To exemplify the travelling salesman problem, I have developed an android application, with a Firestore database, Firebase Authentication and Maps API. The functionalities of the application include searching for locations, determining the best route (visiting every node), choosing the travelling mode (by car, on foot), automatically adapting the search algorithm, saving the searches, managing routes and places as favourites.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2018

LORENZOVICI ZSOMBOR

ADVISOR:
ASSOC. PROF. DR. GASKÓ NOÉMI

BABEȘ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

GRoutes - A practical application of the travelling salesman problem



ADVISOR:

ASSOC. PROF. DR. GASKÓ NOÉMI

STUDENT:

LORENZOVICI ZSOMBOR

2018

UNIVERSITATEA BABEȘ–BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

GRoutes - Aplicarea practică a problemei comis-voiajorului



CONDUCĂTOR ȘTIINȚIFIC:
CONF. DR. GASKÓ NOÉMI

ABSOLVENT:
LORENZOVICI ZSOMBOR

2018

BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Szakdolgozat

**GRoutes - Az utazó ügynök
problémájának gyakorlati
alkalmazása**



TÉMAVEZETŐ:

DR. GASKÓ NOÉMI,
EGYETEMI DOCENS

SZERZŐ:

LORENZOVICI ZSOMBOR

2018

Tartalomjegyzék

1. Bevezetés	4
2. Gráfok	6
2.1. Alapfogalmak	6
2.2. Gráfok leszámmlálása	7
2.3. Részgráf izomorfizmus probléma	7
2.4. Gráfok színezése	7
2.5. Útproblémák	7
2.5.1. Hamilton-kör probléma	7
2.5.2. Minimális feszítőfa	7
2.5.3. Kínaipostás-probléma	8
2.5.4. Königsbergi hidak problémája	8
2.5.5. Legrövidebb út probléma	8
3. Az utazó ügynök problémája (TSP)	9
3.1. A probléma komplexitása	9
3.2. Aszimmetrikus TSP	9
3.3. Egzakt algoritmusok	9
3.3.1. Brute force	10
3.3.2. Held-Karp algoritmus	10
3.3.3. Concorde	10
3.4. Heurisztikus algoritmusok	11
3.4.1. Greedy - legközelebbi szomszéd	11
3.4.2. 2-opt algoritmus	11
3.4.3. Lin-Kernighan heurisztika	11
4. Technológiák	13
4.1. Android	13
4.2. Gradle	14
4.3. Firebase	14
4.4. Maps API	14
5. GRoutes - alkalmazás bemutatása	15
5.1. Funkcionalitások	15
5.1.1. Bejelentkezés	15
5.1.2. Főmenü	15
5.1.3. Keresési felület	16
5.1.4. Kedvencek	16
5.1.5. Múltbeli keresések	16

TARTALOMJEGYZÉK

5.1.6.	Beállítások	16
5.1.7.	Térkép felület	17
5.2.	Adatbázis	17
5.2.1.	Autentikáció	17
5.2.2.	Kollekciók, Dokumentumok	18
5.3.	Osztályok	19
5.3.1.	Activity-k	19
5.3.2.	Assist osztályok	19
5.3.3.	Általános segédosztályok	21
5.3.4.	Entitás osztályok	21
5.4.	Felmerülő problémák	21
5.4.1.	Aszinkron függvényhívások	21
5.4.2.	Adatok aktualizálása az activity-k között	22
6.	Következtetések	23

1. fejezet

Bevezetés

Dolgozatom témája az "utazó ügynök" problémájának (TSP)¹ a gyakorlatban történő alkalmazása. Az utazó ügynök problémája egy komputacionális optimalizálási probléma, amely az 1930-as évek óta nagyon intenzíven foglalkoztatja a tudományos világot. Egyszerűen megfogalmazva: adott valamennyi város, valamint a köztük levő távolságok. Melyik a legrövidebb lehetséges útvonal, ami egyszer érinti az összes várost, majd visszaérkezik a kiinduló pontba? Jelenleg nem létezik olyan polinomiális komplexitással rendelkező algoritmus, amely erre a problémára egzakt megoldást nyújtana.

Az utazó ügynök problémájának példázásának céljából egy android alkalmazást készítettem, amelyben különböző helyekre (csomópontokra) rákeresve, eredményként kirajzolja az optimális (legrövidebb) útvonalat, amely érinti az összes csomópontot. A felhasználó két utazási mód közül választhat: gyaloglás, autó. Az alkalmazás felhasználói célterülete a turistaútvonalak tervezői, a csomagkihordó szolgáltatást biztosító cégek, valamint a különböző eladási szakterülettel foglalkozó vállalkozások, ahol oda kell utazni az ügyfélhez.

Az első fejezetben a gráfelméleti alapfogalmakat fogjuk definiálni, részletezni. A következő fejezetben a különböző gráfelméleti problémákat, feladatokat taglaljuk, majd ezt követően egy külön fejezetet szentelünk a dolgozat témájául szolgáló problémára, az "utazó ügynök" problémájára. Ebben a fejezetben meg kell magyaráznunk a különböző komplexitási alapfogalmakat, hogy rávilágítsunk a téma komputacionális nehézségeire. Arról is itt fog szó esni, hogy milyen módszerekkel közelítjük meg a problémát. A későbbiekben nagyító alá helyezzük a különböző algoritmusokat az előbbiekben említett megoldási stratégiák szerint osztályozva. Bizonyos algoritmusok csak szimmetrikus TSP esetén adnak helyes eredményt, így az asszimmetrikus TSP szimmetrikusra való visszavezetését is tárgyalni fogjuk.

A dolgozat gyakorlati része a projekt során felhasznált technológiák bemutatásával fog kezdődni. Ez fontos lehet azok számára, akik alapjaiban jobban meg szeretnék érteni az applikációt, vagy egy hasonló alkalmazás elkészítéséhez szeretnének hasznos ismereteket elsajátítani. A technológiák bemutatása után maga az alkalmazás nagyító alá helyezése következik. Ebben a részben taglaljuk az applikáció különböző komponenseit, hogy ezek milyen kapcsolatban vannak egymással. Ezt osztálydiagrammal is illusztráljuk. Itt lesz szó részletesebben az adatbázisról, az autentikációról, a Maps API-ről², az alkalmazás funkcionalitásairól, hogy milyen esetben melyik algoritmus van alkalmazva és miért kell egyáltalán különböző esetekben más-más algoritmust használni, valamint a nehézségekről, melyekbe munkám során ütköztem, és ezek megoldásairól.

1. Travelling salesman problem

2. Application Programming Interface

1. FEJEZET: BEVEZETÉS

Végül levonjuk a következtetéseket a munkám során szerzett tapasztalatokból, valamint javaslatokat teszünk az alkalmazás jövőbeli továbbfejlesztési lehetőségeire.

2. fejezet

Gráfok

Összefoglaló: Ebben a fejezetben a későbbiekben használt fogalmakat fogom definiálni.

2.1. Alapfogalmak

Egy gráf $G = (V(G), E(G))$ vagy $G = (V, E)$ két véges halmazból áll. $V(G)$ vagy V , egy nem üres halmaz, melynek az elemeit csúcsoknak nevezzük, ezek alkotják a gráf csúcsait. $E(G)$ vagy E , egy halmaz, melynek elemeit éleknek nevezzük, ezek alkotják a gráf éleit, úgy, hogy minden $e \in E$ élet meghatároz egy rendezetlen csúcs-pár (u, v) , melyeket e csúcsainak nevezünk.

rend – definíció szerint a G gráf rendje $|V| = n$

méret – definíció szerint a G gráf mérete $|E| = m$

hurokél – olyan él, melynek mindkét végpontja megegyezik

többszörös él – ha két csúcsot több él köt össze, akkor ezeket többszörös, vagy párhuzamos éleknek nevezzük

egyszerű gráf – azon gráf, amely nem tartalmaz sem hurokért, sem többszörös éleket

teljes gráf – egy egyszerű gráf, amelynek minden különböző csúcs-párját összeköti egy él. Egy teljes gráf $n(n-1)/2$ éllel rendelkezik. Ha a teljes gráf csúcsai v_1, v_2, \dots, v_n , akkor az él halmaza megadható a következőképpen:

$$E = \{(v_i, v_j) : v_i \neq v_j; \quad i, j = 1, 2, 3, \dots, n\} \quad (2.1)$$

irányított gráf – olyan gráf, ahol az éleket rendezett (u, v) csúcspárok határozzák meg (számít, hogy melyik a kezdő- és végpont)

részgráf – Legyen H egy gráf, csúcsainak halmaza $V(H)$, éleinek halmaza $E(H)$, hasonlóan G egy gráf, csúcsainak halmaza $V(G)$ és éleinek halmaza $E(G)$. H részgráfja G -nek, ha $V(H) \subseteq V(G)$ és $E(H) \subseteq E(G)$.

séta – a csúcsok és él váltakozó véges sorozata, mely csúccsal kezdődik és csúccsal végződik, valamint minden csúcsot egy vele szomszédos él követ és fordítva

vonat – az a séta, melyben az él nem ismétlődnek.

út – az a séta, melyben a csúcspontok nem ismétlődnek.

kör – egy nem-triviális¹ zárt vonal, amelynek a kezdő- és belső pontjai nem ismétlődnek.

Hamilton-út – egy G gráf azon útja, mely minden csúcsot magába foglal

2.2. Gráfok leszámlálása

2.3. Részgráf izomorfizmus probléma

A részgráf izomorfizmus egy komputacionális probléma, adottak a H és G gráfok, és el kell dönteni, hogy létezik-e G -nek olyan részgráfja, amely izomorf H -val. Ennek bizonyítása egy NP-teljes feladat.

2.4. Gráfok színezése

Egy gráf színezése azt jelenti, hogy a csúcsaihoz (vagy az éleihez) színeket rendelünk (legtöbbször egy számmal reprezentáljuk), úgy, hogy két bármely két szomszédos csúcs (vagy él) különböző színű legyen.

2.5. Útproblémák

2.5.1. Hamilton-kör probléma

A Hamilton-kör egy gráfban, egy olyan kör, mely áthalad a gráf összes csúcsán. Mivel egy kör, minden csúcson egyszer fog áthaladni, kivéve az elsőt, ami egyben az utolsó is. Egy gráfot Hamilton-gráfnak nevezünk, ha tartalmaz Hamilton-kört.

Egy egyszerű gráf $G = (X, E)$, $n \leq 3$ csúccsal Hamilton gráf, ha $d(x) + d(y) \geq n$, minden nem szomszédos $x, y \in X$ csúcsra.

Ebből kifolyólag egy gráfban a Hamilton-kör létezése elméleti szempontból nem egy egyszerű probléma, de gyakorlati szempontból sem, mivel NP²-teljes. Az utazó ügynök problémája magába foglalja a Hamilton-kör létezését a gráfban.

2.5.2. Minimális feszítőfa

Egy fa egy olyan gráf, amely összefüggő és nem tartalmaz köröket. Bármely összefüggő G gráfban a feszítőfa G -nek egy olyan részgráfja, amely fa , és tartalmazza G összes csúcsát. A minimális feszítőfát meghatározhatjuk Prim algoritmusával segítségével.

Prim algoritmus: kezdjük bármelyik csúccsal, majd válasszuk ki a hozzá tartozó legkisebb súllyal rendelkező élet. Minden iterációban kiválasztjuk az eddig vizsgált csúcsokhoz tartozó élek közül a legkisebb súllyal rendelkező élet, majd hozzáadjuk az eddig kapott fához az eddig nem vizsgált csúcspontjával együtt. Ezt addig folytatjuk, míg az összes csúcsot megvizsgáltuk.

1. hossza nagyobb, mint 0

2. nem determinisztikusan polinomiális

2.5.3. Kínaipostás-probléma

Egy G gráfban, találjuk meg a legrövidebb zárt sétát, amely minden élen legalább egyszer halad át. Egy optimális megoldást találni úgy irányított, mint irányítatlan gráfban NP-teljes.

2.5.4. Königsbergi hidak problémája

Königsberg városában volt hét híd, amely összekötött két szigetet a város két partjával és egymással. A lakosok megpróbálták úgy sétálni, hogy minden hídon egyszer és csakis egyszer haladjanak át, és érjenek vissza a kezdőpontba. Ez soha senkinek nem sikerült Euler magyarázta meg egy 1736-ban írt cikkben, hogy ez miért nem lehetséges.

Egy Euler-út egy gráfban egy olyan út, amely minden élet magába foglal, egyszer és csakis egyszer. Egy Euler-út zárt, ha ugyanabban a csúcsban végződik, mint ahonnan elindul, egyébként nyíltnek nevezzük.

Egy gráf akkor és csakis akkor tartalmaz zárt Euler-utat, ha minden csúcshoz páros számú él tartozik, valamint minden él ugyanahoz a komponenshez tartozik. Egy gráf akkor és csakis akkor tartalmaz nyílt Euler-utat, ha pontosan két csúcshoz tartozik páratlan számú él, és minden él egyazon komponenshez tartozik.

2.5.5. Legrövidebb út probléma

Dijkstra algoritmus megoldja a legrövidebb út problémát és visszatéríti a legrövidebb utat a kiinduló csúcsból az összes többi csúcsba. Negatív súlyú élek esetén nem működik. Az algoritmus komplexitása $O(n^2)$, ahol n a csúcsok száma.

3. fejezet

Az utazó ügynök problémája (TSP)

Az utazó ügynök problémája magába foglalja a Hamilton-kör létezésnek a problémáját is. Adott valamennyi város, amelyeket az utazó ügynök úgy kell meglátogasson, hogy mindegyik városba egyszer és csakis egyszer menjen be, valamint érjen vissza a kiindulási pontba.

3.1. A probléma komplexitása

P – azokat a problémákat foglalja magába, melyeket egy determinisztikus Turing-gép polinomiális időben képes megoldani

NP – azokat a problémákat foglalja magába, melyeket egy nem-determinisztikus Turing gép polinomiális időben képes megoldani. Ezeknek a megoldását polinomiális időn belül le lehet ellenőrizni determinisztikus Turing-géppel.

NP-nehéz – egy probléma amely "legalább olyan nehéz, mint a legnehezebb probléma az NP-ben"

NP-teljes – egy probléma NP-teljes ha úgy az NP, mint az NP-nehéz halmaznak is eleme, így tehát ezek a legnehezebb komputacionális problémák

Az utazó ügynök problémája magába foglalja a Hamilton kör problémáját is, ami NP-teljes, így tehát a TSP¹ is NP-teljes.

3.2. Aszimmetrikus TSP

Amennyiben irányított gráfokkal dolgozunk, abban az esetben az aszimmetrikus TSP-ről beszélünk. Nem minden TSP algoritmus működik ATSP-re is, ezért előfordulhat, hogy át kell alakítanunk a gráfot, visszavezetve a TSP-re. A visszavezetés után, n csomópontból álló ATSP egy $2n$ csomópontból álló TSP-t fog eredményezni.

3.3. Egzakt algoritmusok

Az egzakt algoritmusok minden esetben az optimális megoldást térítik vissza, komplexitásuk azonban exponenciális. Így a csomópontok növekedésével a futási sebesség exponenciálisan fog nőni. Kutatási

1. travelling salesman problem - az utazó ügynök problémája

3. FEJEZET: AZ UTAZÓ ÜGYNÖK PROBLÉMÁJA (TSP)

szemtpontot leszámítva, nem lehetne ezeket az algoritmusokat átlagos felhasználói környezetben alkalmazni, ugyanis már 20 csomópont után nem lehet kivárni a megoldást.

3.3.1. Brute force

A legegyszerűbb megközelítés, az összes permutációt kipróbálni, és kiválasztani ezek közül a legjobbat. Ennek a komplexitása azonban $O(n!)$, így ezt már 20 városnál se lehet alkalmazni.

3.3.2. Held-Karp algoritmus

Az algoritmus komplexitása a legrosszabb esetben $O(n^2 * 2^n)$.

Legyen $S \subseteq 2, \dots, N$ részhalmaza a városoknak, és $c \in S$ úgy, hogy $D(S, c)$ a minimális távolság kezdve az első várostól, meglátogatva az összes várost S -ből, majd visszaérve c városba.

ha $S = c$, akkor $D(S, c) = d_{1,c}$, különben:

$$D(S, c) = \min_{x \in S-c} (D(S - c, x) + d_{x,c}) \quad (3.1)$$

Ezután a minimális út az összes város érintésével:

$$M = \min_{c \in \{2, \dots, N\}} (D(\{2, \dots, N\}, c) + d_{c,1}) \quad (3.2)$$

Egy $\{n_1, \dots, n_N\}$ út minimális, ha teljesíti:

$$M = (D(\{2, \dots, N\}, n_N) + d_{n_N,1}) \quad (3.3)$$

3.3.3. Concorde

A *Concorde* TSP megoldó egzakt megoldást nyújt az utazó ügynök problémájára. ANSI C-ben íródott és a "cutting plane" módszer segítségével iteratíván oldja meg a TSP lineáris programozási relaxációit. A felhasználói grafikus felület lehetőséget ad arra, hogy különböző heurisztikus algoritmusokkal számoljuk ki a megoldást.

A *Concorde* segítségével megtalálták az optimális megoldást a TSPLIB² mind a 110 példájára, ahol a legnagyobb 85900 várost tartalmaz.

Az fennebb látható eredmények egy 2.8 GHz-es Intel Xeon processzor egy magjának használatával lettek kiszámolva, az *ILOG CPLEX* lineáris programozási feladatmegoldó segítségével. Mindegyik adathalmaz 100 csomópontból áll.

2. a TSP-re példa adatokat tartalmazó könyvtár

3. FEJEZET: AZ UTAZÓ ÜGYNÖK PROBLÉMÁJA (TSP)

Adathalmaz	Futási idő (másodperc)
kroA100	0.31
kroB100	0.58
kroC100	0.30
kroD100	0.33

3.4. Heurisztikus algoritmusok

3.4.1. Greedy - legközelebbi szomszéd

Mivel az egzakt algoritmusokat csak igazán kevés használati esetben lehet alkalmazni, ezért a mohó algoritmusok mellett is dönthetünk. Ezek polinomiális idő alatt elvégzik a feladatod, azonban nem mindig az optimumot térítik vissza. Az eltérés mértéke függhet azonban az implemetációtól.

- 1. lépés** – kezdjük egy véletlenszerűen kiválasztott csomóponttal, melyet beállítunk aktuálisnak
- 2. lépés** – keressük meg a legrövidebb élet, amely összeköti az aktuális csúcsot, és egy meg nem látogatott V csúcsot
- 3. lépés** – beállítjuk V -t aktuális csúcsnak
- 4. lépés** – megjelöljük, hogy már meglátogattuk V -t
- 5. lépés** – ha minden csomópontot meglátogattunk, akkor algoritmus vége
- 6. lépés** – menjünk a 2. lépéshez

3.4.2. 2-opt algoritmus

A 2-opt algoritmus egy lokális kereső algoritmus, mely egy meglévő utat javít fel. Ezt az algoritmust elsőként Croes javasolta 1958-ban, az alapművetet azonban már Flood is javasolta 1956-ban. Ez abból áll, hogy egy meglévő körútból kitörünk két élet, úgy hogy ezeknek nincs közös csúcsuk, majd a csomópontokat újra összekötjük. Erre egy lehetőség van úgy, hogy ne az előző utat kapjuk. Amennyiben az újonnan kapott körút rövidebb, megtartjuk. Ezeket a cseréket minden lehetséges kombinációra elvégezzük. Az így kapott körutat 2-optimálisnak nevezzük.

3.4.3. Lin-Kernighan heurisztika

A 2-opt algoritmus általánosítása révén született meg az egyik leghatékonyabb approximációs algoritmus a szimmetrikus TSP megoldására, a Lin-Kernighan algoritmus. Egy körút k -optimális, ha nem lehetséges egy rövidebb körutat kapni k darab él, más k darab éllel való helyettesítés után. Minél nagyobb a k értéke, annál valószínűbb, hogy az algoritmus végrehajtása után az optimális megoldást kapjuk, ugyanakkor nagyobb adathalmazra gyorsan növekszik a k darab élcsere ellenőrzéséhez szükséges műveletek száma. Egy hátránya tehát a $k - opt$ algoritmusoknak, hogy futás előtt meg kell adni a k értékét. Ezt a hátrányt igyekszik kiküszöbölni a Lin-Kernighan algoritmus, azáltal, hogy futása során változtatja k -nak az értékét.

3. FEJEZET: AZ UTAZÓ ÜGYNÖK PROBLÉMÁJA (TSP)

Legyen T az aktuális körút. Minden iterációban, az algoritmus igyekszik találni két $X = \{x_1, \dots, x_k\}$ és $Y = \{y_1, \dots, y_k\}$ élekből álló halmazokat, melyekre igaz az, hogy a T körútból az X -ben található éleket az Y -ból vett élekkel helyettesítve egy jobb körutat kapunk. Ezeknek az éleknek a felcserélését $k - opt$ lépésnek nevezzük. Ahhoz, hogy kellően hatékony algoritmust kapjunk, az X és Y halmazokhoz tartozó éleknek meg kell felelniük bizonyos kritériumoknak:

- (a) **A szekvenciális csere kritériuma** – x_i -nek, valamint y_i -nek rendelkezniük kell egy közös csúccsal, ugyanígy y_i -nek és x_{i+1} -nek is. Így tehát a $x_1, y_1, x_2, y_2, \dots, x_k, y_k$ sorozat egy szomszédos élekből álló láncot alkot.
- (b) **A megvalósíthatósági kritérium** – szükséges továbbá, hogy $x_i = (t_{\{2i-1\}}, t_{\{2i\}})$ úgy legyen kiválasztva, hogy ha a $t_{\{2i\}}$ -t csatlakoztatjuk a t_1 -hez, az így kapott gráf körút maradjon. Ez a kritérium az algoritmus futási idejének csökkentéséért, valamint a kódolás leegyszerűsítéséért lett bevonva.
- (c) **A pozitív nyereség kritériuma** – az y_i -t úgy kell kiválasztani, hogy a nyereség a javasolt cserék után pozitív maradjon. Ez a feltétel kulcsfontosságú az algoritmus hatékonyságának szempontjából.
- (d) **A diszjunktivitás kritérium** – az X és Y halmazoknak diszjunktak kell lenniük

4. fejezet

Technológiák

***Összefoglaló:** Ebben a fejezetben, a projekt elkészítése során felhasznált technológiákat ismertetem.*

4.1. Android

Az Android egy nyílt forráskódú, Linux kernel alapú többfelhasználós operációs rendszer, ahol minden applikáció egy külön felhasználó. Hivatalos nyelvei a Java és a Kotlin. Alapértelmezetten, a rendszer minden applikációnak kioszt egy egyedi Linux felhasználói ID-t (az ID-t csak a rendszer használja, és ismeretlen az applikáció számára). Az operációs rendszer úgy osztja ki a hozzáférési jogokat az applikáció állományai számára, hogy csak az a felhasználói ID férjen hozzájuk, amivel az adott applikáció rendelkezik. Minden folyamat (process) rendelkezik a saját virtuális gépével, tehát minden applikáció kódja egymástól teljesen izoláltan fut. Alapértelmezetten minden applikáció a saját Linux folyamatában fut. Az Android operációs rendszer elindítja a folyamatot, amikor az applikáció valamelyik komponensének szüksége van rá, majd leállítja azt, mikor többé már nincs rá szükség, vagy ha a rendszernek memóriát kell lefoglalnia, más applikációk számára. Az Android operációs rendszer “a legkevesebb kiváltság elvét” (“the principle of least privilege”) alkalmazza, tehát alapértelmezetten, minden applikációnak csak azokhoz a komponensekhez van hozzáférése, amik szükségesek a feladatának elvégzéséhez, és semmi egyébhez.

Egy android applikációnak négy fajta komponense lehet: Activity, Service, Broadcast receiver, Content provider. Ezek közül az Activity szolgál a felhasználóval történő interakció eszközeként, ez ugyanis egy felhasználói felülettel rendelkező képernyő. Activity-k, Service-k és Broadcast receiver-ek egy aszinkron üzenet által aktiválódnak, amit Intent-nek (szándék) nevezünk.

Az Activity egy osztály (class), amely a logikát tartalmazza, a felhasználói felületért azonban egy ehhez tartozó XML állomány felel, ami a különböző UI elemeket (gombok, szövegdobozok, konténerek) tartalmazza.

Az AndroidManifest.xml egy konfigurációs állomány, ami a projekt gyökeri könyvtárában található. Itt vannak deklarálva az applikáció komponensei, valamint a szükséges hozzáférési jogokat is itt kell feltüntetni.

4.2. Gradle

A Gradle egy nyílt forráskódú projektépítő eszköz (build automation tool). A Groovy, vagy Kotlin nyelvű scriptekbe, megadhatjuk a projektünk külső függőségeit (például API-k, library-k), melyeket letölti, majd lekompillálja és lefordítja a forráskódot.

4.3. Firebase

A felhasználók adatainak, beállításainak tárolására, kezelésére a Firestore nevű no-sql (.json) alapú adatbázist használtam, mely a CRUD¹ műveletekhez is biztosít metódusokat. Logikai építőelemei a kollekciók (collection) és a dokumentumok (document). Az előbbi tartalmazhat dokumentumokat, míg az utóbbi alkollekciókat, vagy magukat az adatokat. Az adatok kulcs, érték párok, az értékek lehetnek számok, karakterláncok, tömbök, geopontok vagy akár sajátos osztályok. Amennyiben sajátos osztály akarunk használni, annak rendelkeznie kell egy publikus konstruktorral, melynek nincsenek paraméterei, valamint az attribútumokhoz kell tartozzon egy-egy publikus „getter”² metódus. Az adatbázisban található adatokhoz való hozzáférést egy szabállyal kell megadni, amelyet a szerver leellenőriz a CRUD műveletek végrehajtása esetén.

A felhasználók menedzselésére, mint a regisztráció, bejelentkezés, e-mail cím aktiválása, elfelejtett jelszó visszaállítása, a Firebase Authentication szolgáltatást használtam.

4.4. Maps API

A csomópontok közti távolságok mátrixának lekérdezésére a Distance Matrix API-t, valamint két csomópont közötti útvonal meghatározására a Directions API-t használtam.

1. create, read, update, delete

2. egy paraméter nélküli metódus, mely egy attribútumot térít vissza - példa: String getName()

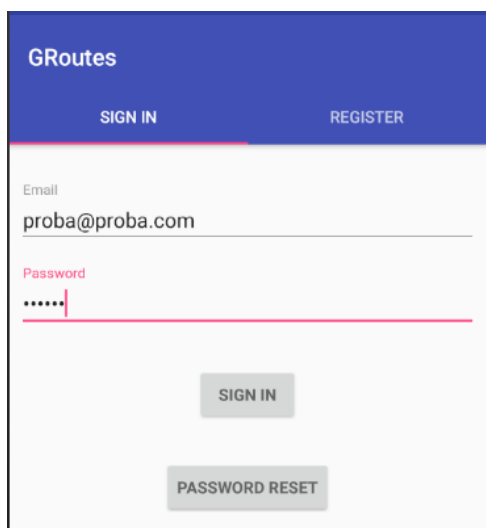
5. fejezet

GRoutes - alkalmazás bemutatása

Összefoglaló: Ebben a fejezetben a GRoutes android applikációt fogom bemutatni úgy technikai, mint funkcionális szempontból.

5.1. Funkcionalitások

5.1.1. Bejelentkezés



5.1. ábra. Bejelentkezési felület

nálóját.

Az alkalmazás elindítása után egy bejelentkezési felület fogadja a felhasználót. Itt kell megadni az e-mail címet, valamint a jelszót amivel a felhasználó regisztrált. Amennyiben elfelejtette a jelszavát, az "elfelejtett jelszó" gomb megnyomásával lehetőség van egy jelszó-visszaállító e-mail kiküldésére.

Ha egy új felhasználó szeretné igénybe venni az applikáció szolgáltatásait, akkor regisztrálnia kell. Ezt megteheti a regisztráció lapra való navigálás után, melyet a cím megérintésével, vagy a képernyőn az ujjának balra történő húzásával érheti el. Az e-mail cím és az új jelszó megadása után, a felhasználó egyből a főmenübe érkezik, mindeközben azonban egy aktivációs e-mailt is kap arra a címére, amivel regisztrált. A későbbiekben csak akkor fog tudni bejelentkezni, ha a kapott emailben az URL¹ -re klikkelve aktiválja a felhasználóját.

5.1.2. Főmenü

A főmenüből négy különböző oldalra navigálhatunk tovább: a keresési felület, a régebbi keresések megtekintése, a kedvencek menedzselése, valamint a beállítások. Az ötödik (csoportok) oldal egy jövőbeli funkciók kifejlesztésére van fenntartva.

1. Uniform Resource Locator, más néven webcím

5.1.3. Keresési felület

Ezen a felületen van lehetőségünk útvonalak, helyszínek keresésére. Az "+" gomb megjelenít egy új mezőt, ahol további címeket adhatunk meg. A "nagyító" gomb a térkép felületre navigál, és kirajzolja a tervezett útvonalat. A keresés elindítása előtt, minden cím mellett van egy "térkép" gomb, ennek segítségével tudjuk megnézni, hogy valóban az az a hely, amire gondoltunk.

5.1.4. Kedvencek

Itt jelennek meg az általunk manuálisan elmentett utak, helyszínek. Minden bejegyzésnél négy gomb jelenik meg, ezeknek az funkcióik a következők:

térkép – megjeleníti a térkép felületet, ahol ráközelít a helyszín koordinátáira, vagy kirajzolja az útvonalat.

nagyító – a keresési felülethez navigál és kitölti a keresési mezőket, az útvonal csomópontjaival, így a felhasználó könnyedén módosíthatja útvonaltervét.

ceruza – módosítja a bejegyzés nevét.

szemetes veder – törli a bejegyzést, ezáltal az adatbázisból is törlődni fog.

5.1.5. Múltbeli keresések

A keresési felület eredményeül kapott útvonalak megjelennek a múltbeli keresések menüpont alatt, nevüket a keresési dátum és időpont alkotja. A bejegyzéseknél a "Kedvenceknél" tárgyalt gombok találhatóak, azonos funkcióval, azzal a kivétellel, hogy a "ceruza" gombot a "csillag" helyettesíti, melynek segítségével elmenthetünk egy útvonalat a kedvencek közé.

5.1.6. Beállítások

Itt ki tudjuk választani a limitet, hogy mennyi csomópont esetén, melyik algoritmust használja az alkalmazás. A megadott számnál kisebb vagy egyenlő számú csomópontok esetén a *Concorde* nevű egzakt megoldásokat nyújtó algoritmus fogja kiszámolni az ideális útvonalat. Ez az algoritmus a legpontosabb megoldásokat nyújtja, azonban az "utazó ügynök" problémájának komplexitása miatt, a csomópontok növekedésével, a végrehajtási idő is exponenciálisan növekszik. Amennyiben a felhasználó nagyon sok csomóponttal szeretne dolgozni, és fontosabb neki az, hogy belátható időn belül egy elfogadható megoldást kapjon, de nem probléma, ha nem a leghatékonyabb útvonal rajzolódik ki, akkor a megadott szám feletti mennyiségű csomópontok esetén, az applikáció egy úgynevezett *greedy*² algoritmust fog használni. Ez nagyságrendekkel gyorsabb az exponenciálishoz viszonyítva, azonban nem minden esetben nyújtja a legjobb megoldást.

Ugyanitt megadhatjuk az alapértelmezett utazási módot, ami lehet gyaloglás, valamint vezetés.

2. mohó

Bár az egyszerűség kedvéért igyekeztem sok írás helyett szimbólumokat tenni a gombokra, de az applikációban található kevés szövegnek a nyelvét szintén itt lehet beállítani.

A térképpel, navigációval kapcsolatos paraméterek módosítására is van lehetőség, mint például az alapértelmezett nagyítás, a pozíció lekérésének gyakorisága, a térképre való automatikus ránagyítás és ennek centralizálásának a gyakorisága navigáció közben, valamint a megtett út kirajzolásának a mennyisége.

5.1.7. Térkép felület

Ez a felület akkor jelenik meg, amikor megérintjük a térkép gombot egy helyszínen vagy egy útvonal mellett, valamint a keresési felület eredménye is itt rajzolódik ki. Amennyiben egy helyszíntől érkezünk, megjelenik egy *marker*³ az adott koordinátán.

A "csillag" gomb segítségével hozzáadhatunk egy helyszínt vagy egy útvonalat a kedvencekhez, ahol később visszanézhetjük, módosíthatjuk a nevét. Alapértelmezetten, a bejegyzések neve a keresési dátum és időpont lesz. A "nagyító" gomb segítségével visszatérhetünk a keresési felületre, így könnyedén módosíthatjuk tervezett útvonalunkat. Az "navigáció" gomb segítségével elkezdhetünk navigálni a tervezett útvonalon.

5.2. Adatbázis

```
implementation 'com.google.firebase:firebase-core:16.0.1'
implementation 'com.google.firebase:firebase-auth:16.0.2'
implementation 'com.google.firebase:firebase-firestore:17.0.4'
```

5.2. ábra. A Firebase függőségei

Adatbázisnak és az autentikáció menedzselésére Firebase-t használtam. Mielőtt bármilyen funkcióját is használhatnám ennek az applikációfejlesztési platformnak, be kellett jelentkezsek a Google felhasználómmal, csinálnom kellett egy projektet,

majd ezt a projektet össze kellett kapcsolnom magával az applikációval amit fejleszteni szerettem volna. Van egy külön *plugin*⁴ az *AndroidStudio*⁵ -ban, melynek segítségével könnyedén létre lehet hozni a ezt kapcsolatot. Előtte azonban az applikáció Gradle állományába be kellett jelentsem a *core*, *auth* és *firestore* modulokat, mint függőségeket.

5.2.1. Autentikáció

A web-es felületen engedélyeznem kellett az e-mail/jelszó bejelentkezést. Ugyanitt állítottam be, hogy biztonsági okokból maximum hány regisztrációt engedjen a rendszer óránként ugyanarról az IP címről, jelenleg ez 100. Regisztrációkor generálódik egy egyedi felhasználói ID, ennek segítségével oldom meg, hogy minden felhasználó csak a saját adataihoz férjen hozzá. A bejelentkezés, regisztráció, jelszó visszaállító e-mail küldése műveleteket a *FirebaseAuth singleton*⁶ osztály metódusaival oldottam meg.

3. jelző

4. kiegészítés

5. fejlesztési környezet android applikációk számára

6. egy tervezési minta, az adott osztályból egy és csak is egy objektumot lehet létrehozni

5. FEJEZET: GRUTES - ALKALMAZÁS BEMUTATÁSA

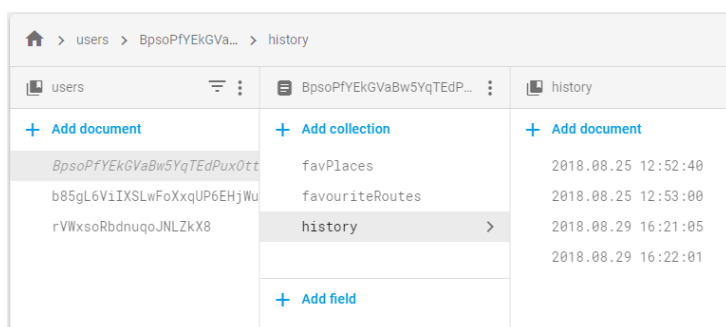
createUserWithEmailAndPassword(email, jelszó) – kreál egy felhasználót az e-mail és jelszó karakterlánc paraméterek segítségével, majd sikeres válasz után be is jelentkezik

signInWithEmailAndPassword(email, jelszó) – megpróbál bejelentkezni a megadott e-mail/jelszó kombinációval

sendPasswordResetEmail(email) – küld egy jelszó visszaállító e-mailt a paraméterként megadott címre.

Az aktiváviós e-mail küldéséhez a `FirebaseUser` osztály instanciájának a `sendEmailVerification()` metódusát használtam.

5.2.2. Kollekciónok, Dokumentumok



5.3. ábra. Firestore kollekciónok

A Firestore egy még Beta⁷ verzióban levő no-sql, kulcs-érték párokra alapuló adatbázis, mely a Firebase platformon érhető el. Építőelemei a kollekciónok és a dokumentumok, ezek váltakozva követik egymást, mivel egy kollekción csak dokumentumokat tartalmazhat, egy dokumentum viszont a kollekciónok mellett adatokat (kulcs-érték párok) is tartalmazhat.

Minden felhasználónak csak a saját dokumentumához van hozzáférése. Ezt egy úgynevezett szabály meghatározásával lehet elérni. Az adatbázis minden beérkező függvényhívást kiértékel a szabály alapján, és eldönti, hogy az adott kérésnek van-e elegendő jogosultsága. Amennyiben nincs, csak egy hibaüzenetet térít vissza.

```
1 service cloud.firestore {
2   match /databases/{database}/documents {
3     match /users/{userId}/{document=**} {
4       allow read, write, create, update: if request.auth.uid == userId;
5     }
6   }
7 }
```

5.4. ábra. Firestore szabály

favoriteRoutes és *history* kollekciónokat tartalmaznak, melyek rendre megfelelnek a "kedvenc helyek", "kedvenc útvonalak" és "múltbeli útvonalak" fogalmaknak. Ezek a kollekciónok tartalmazzák a különböző bejegyzéseknek (helyek, útvonalak) megfelelő dokumentumokat, melyek magukat az adatokat tartalmazzák. Az adatok sajátos entitás objektumok, melyeket a kódban, bizonyos szabályok alapján, de az én elképzelésem szerint hoztam létre.

7. még tesztelés alatt van

8. egyedi felhasználói ID, ami regisztrációkor generálódik

Egy út tartalmaz egy hely objektumokból álló tömböt a csomópontok számontartására, egy indexekből álló tömböt a csomópontok sorrendjének tárolására, valamint egy név mezőt.

Egy hely objektum tartalmaz egy-egy karakterlánc típusú név és cím mezőt, valamint egy GeoPoint objektumot, ami a hely földrajzi szélességi és hosszúsági fokait tárolja.

5.3. Osztályok

5.3.1. Activity-k

Egy android applikációban a UI⁹ -t XML¹⁰ állományok segítségével szerkeszthetjük meg, ezek azonban csak a külalakot adja meg. A háttérben végrehajtódó logikáért az Activity osztályok felelnek. A GRoutes alkalmazás különböző használati eseteihez más-más *activity* és XML állománpár tartozik.

LoginTabActivity – ez a bejelentkezési felület, tehát ez fogad minket az alkalmazás indításakor. Itt történik az e-mail és a jelszó mezők validálása, formátum ellenőrzése a bejelentkezési próbálkozás előtt. Használja a DatabaseAssist autentikációval foglalkozó metódusait, majd a válasz visszaérkezése után a LoggedInActivity-hez navigál.

LoggedInActivity – ez a főmenü. Amint beléptünk ide, a háttérben betöltjük a felhasználó adatait (múltbeli keresések, kedvencek). Innen tudunk tovább navigálni a főbb menüpontokhoz.

FavActivity – ez a "Kedvencek" felület. Mivel itt már komplexebb műveleteket kell végrehajtani, ezért azokat egy segédosztályba csoportosítottam, egy köztes réteget létrehozva a UI és az adatbázissal foglalkozó osztály között.

HistoryActivity – ez a "múltbeli keresések" menüpont, rendelkezik segédosztállyal

MultiRouteActivity – ez felel az útvonalak kirajzolásáért, rendelkezik segédosztállyal

SearchActivity – ez a keresés menüpont, rendelkezik segédosztállyal

ShowMapActivity – a helyek megjelenítéséért felel

5.3.2. Assist osztályok

Általánosan, az alábbi segédosztályok végzik el a különböző Activity-k feladatait. Továbbá ők vannak kapcsolatban a CacheManager osztállyal, amely az adatok frissen tartásáért felel.

FavAssist – feltölti adatokkal a "Kedvencek" felületet, dinamikusan beágyazza az XML-be a bejegyzéseket, továbbítja az activity módosítási, törlési szándékait a CacheManagernek

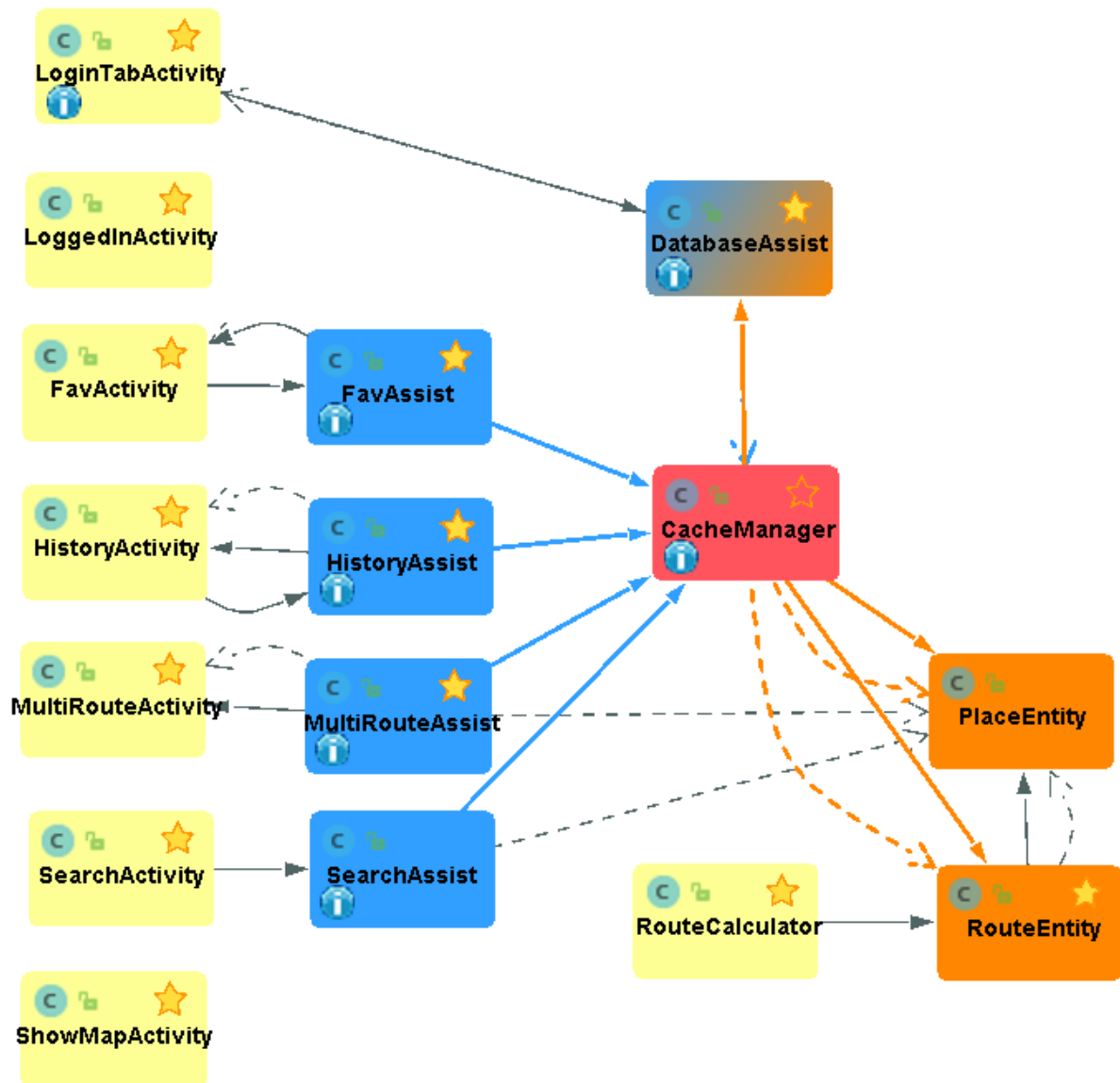
HistoryAssist – a FavAssist osztályhoz hasonló feladatokat végez el a HistoryActivity-nek. Egy múltbeli keresés kedvencekhez való hozzáadását is lemenedzseli

MultiRouteAssist – lekéri a csomópontok közti útvonalat a DirectionApi segítségével, majd kirajzolja a tervezett útvonalat a Polyline¹¹ osztály használatával. A navigálásért is felel.

9. user interface - felhasználói felület

10. Extensible Markup Language - Kiterjeszthető Jelölő Nyelv

11. vonallánc



5.5. ábra. Osztálydiagramm

5. FEJEZET: GROUTES - ALKALMAZÁS BEMUTATÁSA

SearchAssist – feldolgozza a keresési címeket, majd a megfelelő adatstruktúrát lekommunikálja a CacheManager-nek

5.3.3. Általános segédosztályok

A fontosabb általános műveletek végrehajtására létrehoztam néhány Singleton mintára készült segédosztályt.

DatabaseAssist – ez felel mindenféle Firebase-el kapcsolatos kommunikációért. Feladatai közé tartozik a regisztráció, bejelentkezés, az ezekkel kapcsolatos e-mail küldések, valamint a CRUD operációk elvégzése.

CacheManager – a központi adatmenedzselő osztály. Ő tölti be, aktualizálja, szolgálja ki a segédosztályokat az adatokkal, továbbítja kéréseiket a DatabaseAssist-nak. A LoginTabActivity kivételével csak ő áll kapcsolatban a DatabaseAssist segédosztállyal.

RouteCalculator – az optimális útvonal meghatározásáért felel (az utazó ügynök problémájának megoldása)

5.3.4. Entitás osztályok

Az alkalmazás két nagyon fontos logikai struktúrát használ: helyek és útvonalak. Ezekkel több műveletet is kell végezni, mint például átalakítások, keresések, bizonyos paraméter alapján történő módosítások. Ezért számukra létrehoztam egy-egy entitás osztályt.

A Firestore-ba való egyszerű mentés érdekében bizonyos kritériumoknak meg kell felelniük: rendelkezniük kell egy paraméter nélküli publikus konstruktorral, valamint az elmenteni kívánt attribútumoknak publikus *getter* metódusokkal. Ezért azoknak az attribútumoknak a *getter* metódusait, amelyeket nem kívántam elmenteni, átneveztem úgy, hogy a *ret*¹²– prefixet használtam.

PlaceEntity – három attribútuma van: egy String típusú név, egy GeoPoint típusú (magába foglalja a földrajzi szélességi és hosszúsági fokokat) helyszín és egy String típusú cím

RouteEntity – négy attribútuma van: egy String típusú ID (az adatbázisban történő azonosításra), egy String típusú név, egy egész számokból álló ArrayList típusú sorrend (a csomópontok bejárési sorrendjének tárolására) valamint egy helyszínekből álló ArrayList típusú csomópont (csomópontok tárolása)

5.4. Felmerülő problémák

5.4.1. Aszinkron függvényhívások

Több olyan használati eset van, amikor egy bizonyos metódus az interneten keresztül kommunikál. Ez mindig okozhat problémákat, de a szinkronhívásokat kivételkezeléssel egyszerűen meg lehet oldani.

¹².return - visszatérít

5. FEJEZET: GROUTES - ALKALMAZÁS BEMUTATÁSA

Több esetben, mint például a bejelentkezéskor vagy az adatok lekérdezésekor, a program nem vár a válaszra, hanem tovább végzi a következő műveleteket. Ez olyan helyzetekhez vezethet, hogy megnyitunk egy felületet és még nincsenek beöltve az adataink.

Ezt a problémát úgy oldottam meg, hogy a DatabaseAssist osztály fog utólag jelezni annak az osztálynak ahonnan a kérés érkezett, amint választ kapott.

5.4.2. Adatok aktualizálása az activity-k között

Mivel az applikáció több különböző pontján van lehetőség az adatok hozzáadására, módosítására, törlésére, ezért fontos, hogy mindig, mindenhol a legfrissebb adatok birtokában legyünk.

Ezért vezettem be a CacheManager osztályt. Ez nem csak tárolja az aktuális adatokat, hanem az első lekérdezés kivételével, minden adatbázisműveletet lokálisan is elvégez, így a felhasználónak nem kell várnia, hogy a "múltbeli keresések" menüpont alatt megjelenjen a friss keresés, mivel azonnal hozzá lesz adva. Ellenkező esetben meg kéne várni, míg elmentjük az adatokat, majd lekérdezzük őket.

6. fejezet

Következtetések

A tudomány jelenlegi állása szerint nem létezik olyan algoritmus, amely polinomiális komplexitással rendelkezik, és meg tudja oldani egzakt módon az utazó ügynök problémáját. A gyakorlati felhasználói esetekben, azonban nem is biztos, hogy mindig erre van szükség. Az alkalmazásokat adaptívnak kell készítenünk, és a felhasználónak meg kell adjuk a lehetőséget, hogy a saját felhasználási igényeihez mérten testre tudja szabni azokat. Így tehát ez a projekt egy ötletet adhat azoknak, akik bizonyos helyzetekben egy, míg bizonyos helyzetekben más megoldási módszereket alkalmaznának. Midennek megvan ugyanis az előnye, valamint a hátránya. Az egzakt algoritmusoknál a bemenő adatok növekedésével párhuzamosan, a végrehajtási idő exponenciálisan növekszik. A mohó algoritmusok ezzel ellentétben nagy mennyiségű bemenő adatra is nagyon gyorsan képesek meghatározni egy megoldást, azonban az nem biztos, hogy ez az optimális megoldás lesz. Azt viszont csak a felhasználó maga tudja, hogy milyen célra, vagy mikor milyen célra szeretné használni az alkalmazást. Így tehát a döntés lehetőségét a kezébe helyeztük.

A későbbiekben szeretném az alkalmazást egy újabb funkcionalitással, valamint platformmal kibővíteni. A csoportok menüpont alatt lehetőség nyílik arra, hogy a felhasználók egymásnak tervezzenek útvonalakat, megosszák ezeket, valamint arra is, hogy cégen belül a főhadiszállásról lehessen frissíteni a terepen dolgozó kolléga további állomsait. Ehhez szeretnék majd egy web-es felületet is készíteni, hogy az irodából egyszerűbben, kényelmesebben lehessen nagyobb mennyiségű adattal dolgozni.

Irodalomjegyzék