# S Z A K D O L G O Z A T

## BELTÉRI NAVIGÁCIÓS MEGOLDÁSOK IMPLEMENTÁLÁSA


**Pázmány Péter Katolikus Egyetem Információs Technológiai Kar**

Készítette: **Bácskai Gergely Mór**

**mérnök informatikus BSc**

Konzulens: **Tihanyi Attila**

**(PPKE-ITK)**

**Nagy István**

**(BME-TMIT)**

**2011**

# NYILATKOZAT

Alulírott **Bácskai Gergely Mór**, a Pázmány Péter Katolikus Egyetem Információs Technológiai Karának hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem. Ezt a Szakdolgozatot más szakon még nem nyújtottam be.

_____

**Bácskai Gergely Mór**

# TARTALOMJEGYZÉK

# TARTALMI KIVONAT

Ebben a dolgozatban a mobil eszközök segítségével történő beltéri navigációval foglalkoztam. Különböző rádiófrekvenciás jelek segítségével (például WiFi, Bluetooth) képesek lehetünk épületen belüli helyzetünk meghatározására, ám ezen rendszereknek szinte nélkülözhetetlen velejárója a különböző helyszínek, beltéri objektumok közötti útvonalkeresés is.

Legelőször tanulmányoztam az eddigi, beltéri navigációra megoldási javaslatokat kínáló szakirodalmat, mely mind az épület belterületének adatmodellel való lefedését, mind pedig a meghatározott helyzetű objektumok közötti útvonalválasztást vizsgálta. Ezek után betekintést nyújtottam abba a projektbe, melynek a fent említett, általam elkészítendő rendszer a részét képezi, majd felfedtem a munkám alapjául szolgáló helymeghatározó program működését, és hiányosságait a navigáció terén.

A munkám megvalósításához először szükség volt a kiindulási alkalmazás adatbázisában egy olyan adatstruktúrát kialakítanom, melynek segítségével adatszinten könnyen reprezentálható az épület belterülete. Majd nélkülözhetetlen volt egy navigációs adatmodellé átalakítanom a szerverből kinyert adatokat, amin két, az általam a szakirodalomból megismert és útvonal-választásra leggyakrabban felhasznált útkereső algoritmust alkalmaztam a felhasználó által kijelölt két helyszín közötti legrövidebb út keresésére. Ezek után szükséges volt a kiindulási programba beépítenem a navigációs részt, mind a szerver, mind a kliens, mind pedig a közöttük zajló kommunikációs réteg módosításával.

Végül tesztelés céljából adatokkal töltöttem fel a rendszer által használt adatbázist, amin megvizsgáltam az elkészült program működését. Utolsó lépésként pedig elemeztem és javaslatot adtam a rendszer továbbfejlesztési lehetőségeire.

# ABSTRACT

In this thesis, I studied indoor navigation using mobile devices.

Various radio frequency signals (e.g. WiFi, Bluetooth) can be used for indoor localization. However, the path-finding method between indoor objects and locations is an integral and inherent part of these systems.

First, I studied existing literature on indoor navigation solutions that covers both the data modeling of indoor environments and the routing between positioned objects. After that, I gave insight into the project that involves the previously mentioned system to be created. I revealed the functions and navigational flaws of the underlying localization program.

The first step of my work was to develop a structure in the prime application's database so that the building's interior could be represented easily. After this step, I had to convert the data retrieved from the server into a navigational model. I used two popular path-finding algorithms found in literature to find the shortest path between two custom locations. After this, I had to integrate the navigational part into the prime application by modifying the server, the client and the communication layer between them.

Finally, I populated the system's database for testing purposes, and checked the execution of the program. As a last step, I analyzed the system and presented a suggestion for further improvements.

# 1. A dolgozat témája

A beltéri helyfüggő szolgáltatások egyik alapvető funkcionalitása a beltéri objektumok közti navigáció. Eddig számos ötlet született arra, hogy épületen belül, rádiófrekvenciás jelek segítségével képesek legyünk a beltéri pozíciónk meghatározására, ám ennek segítségével szolgáltatások egész sora alakítható ki, mint például az épületen belüli navigáció.

Képzeljünk el egy olyan szituációt, amikor egy bevásárlóközpontban akarunk egy boltot megtalálni. Jó esetben egyenest a célhoz akarnánk menni, elkerülve a felesleges kitérőket. A helyzetet az is megnehezítheti, ha az egészségi állapotunk nem egészen a megfelelő, tehát valamilyen oknál fogva nem vagyunk képesek lépcsőkön közlekedni, a lifteket preferálnánk a leginkább. Tegyük fel ezek után, hogy miután sikeresen eljutottunk a kívánt bolthoz, és éppen bevásárolunk, valamilyen baleset folytán lángra kap az épület egyik része, és a tűz egyre csak terjed. Mi hirtelen ijedtségünkben próbálnánk menekülni, ám amiatt, mert az épület hatalmas (esetleg először járunk ott, és nem tudjuk, mit merre találunk), fogalmunk sincs a rendelkezésünkre álló kivezető, esetleg legegyszerűbben megközelíthető menekülési útvonalakról.

Szinte mindenkivel előfordult már, egy ilyen, vagy ehhez hasonló helyzet, melyben egy nagyobb épületbe belépve, lehetőleg minél gyorsabban, céltudatosan el akartunk jutni egy általunk kiválasztott konkrét (vagy esetleg konkrét típusú – például a legközelebbi mosdó) helyszínre. Ilyen, és ehhez hasonló esetekre kínál megoldást a beltéri navigáció.

## 1.1. Bevezetés

Manapság már igen elterjedtek a különböző navigációs készülékek, melyekkel bárhol is legyünk a Földön, könnyen megtudhatjuk, merre járunk éppen, mi milyen messze van, és hogy hova, milyen utakon, mennyi idő alatt leszünk képesek eljutni. Ezek az eszközök az úgynevezett GPS**[1]** (Global Positioning System) technológiát használják, melynek a lényege, hogy a Föld körül, a felszín felett mintegy 20200 km-re keringő geostacionárius műholdak jeleit fogva, háromszögeléses módszert alkalmazva, egyszerre több műholdra történő rálátás esetén képesek vagyunk a rendszer segítségével körülbelül 10m-es pontossággal megmondani a helyzetünket.

A beltéri navigáció céljai hasonlóak, ám módszerei valamelyest különböznek. Ilyenkor ugyanis épületen belül tartózkodunk, nincs rálátásunk az égboltra, vagyis a Föld körül keringő műholdakra. Azonban itt nem is az a célunk, hogy nagy távolságokat tegyünk meg úgy, hogy közben egy teljesen újfajta navigációs technikát kell alkalmaznunk. Beltéri navigáció esetén a lényeg, hogy az épület egyik pontjából eljussunk egy másik pontjába, lehetőleg a legrövidebb, optimális úton át, mindezt tehát úgy, hogy nincsen szükségünk globális adatokra, azaz például arra, hogy melyik kontinensen vagyunk, melyik szélességi/hosszúsági körön, vagy egyáltalán

arra, hogy milyen forgalmi helyzet alakult ki a környéken. Amire viszont szükségünk van az az, hogy legyenek elérhetőek valamilyen viszonyítási pontok, amik segítségével meg tudjuk mondani, hol vagyunk, mennyit menjünk, milyen irányba.

A navigációs eszközök hardverigénye mai viszonylatban mérve igen csekély; nincs szükség semmiféle speciális architektúrára. A szokványos GPS-techikát használó berendezések is már-már tenyérnyi méretűek, sőt, a mobiltelefonokba épített GPS-vevőszerkezet ma már általánossá vált. Az emberek többsége által használatos okostelefon azonban még ennél is többre képes: számos olyan funkcióval van ellátva, amelyek segítségével új szolgáltatások egész sora alakítható ki, és tekintetbe véve mind a gyártók által egyre gyorsabb hardverek beépítését a mobil eszközökbe, mind a már az eszközbe implementált szenzorok sokaságát, viszonylag jó kiindulási alapot kapunk a beltéri navigációs szolgáltatás, mint alkalmazás szintű megoldás implementálásához.

## 1.2. A beltéri navigáció eddigi eredményei

A beltéri navigáció, mint kutatási téma egyre nagyobb térhódításnak örvend manapság. A témát ugyanis több szemszögből is meg tudjuk közelíteni, hiszen felhasználása igen széleskörű lehet. Mind mobil robotok tervezésekor gondolni kell a robot emberi beavatkozás nélküli navigálására, ugyanakkor az embernek is igen sokszor szüksége lehet olyan módszerre, melynek segítségével akár egy, számára ismeretlen helyszínen is képes lehet eljutni akárhová. Ám nem csak az lehet a legnagyobb gond, ha az ember olyan helyszínen akar tájékozódni, ahol eddig nem járt. Igen sokszor probléma az optimális útvonal kiválasztása is. Hiába tehát, ha ott vagyunk a lépcső előtt, és már tudjuk, hogy egy emelettel feljebb merre találjuk a célunkat, csak fel kell menjünk a lépcsőn egy szintet, de ugyanez a megoldás egy tolószékkel közlekedő mozgássérült számára nem járható út. Az ő számára szükség van arra, hogy a neki optimális utat keressük, ami nem feltétlenül egyezik meg azzal, ami a többi, egészséges ember számára járható legrövidebb út. Az ilyen helyzetben tehát csak olyan elemeket szabad útként figyelembe venni, amik az illető képességei, igényei szempontjából megfelelőek.

Maga a beltéri navigáció szoros összefüggésben van a helymeghatározással, ugyanis ha a fent felsorolt esetek közül valamelyik bekövetkezik, legelőször szükségünk van a saját pozíciónk meghatározására, tehát arra, hogy éppen merre járunk, milyen messze vagyunk a céltól. Emellett az sem árt, ha tudjuk, hogy a számunkra kijelölt úton haladunk, vagyis a navigációs eszköz segítségével megbizonyosodhatunk arról, hogy nem tértünk le a kijelölt útvonalról.

Mindent összevetve tehát nekünk jelenleg nem a pozíció meghatározásra van szükségünk, hanem arra, hogy már ismert pozíciójú helyszínek között különböző közlekedési lehetőségek figyelembevételével hogyan lehetünk képesek legrövidebb utat meghatározni.

### 1.2.1. Nemzetközileg alkalmazott módszerek

A mobil eszközön történő beltéri navigálásra ez idáig több megoldás született. A megoldások több szempontból közelítik meg a beltéri navigáció felhasználását:

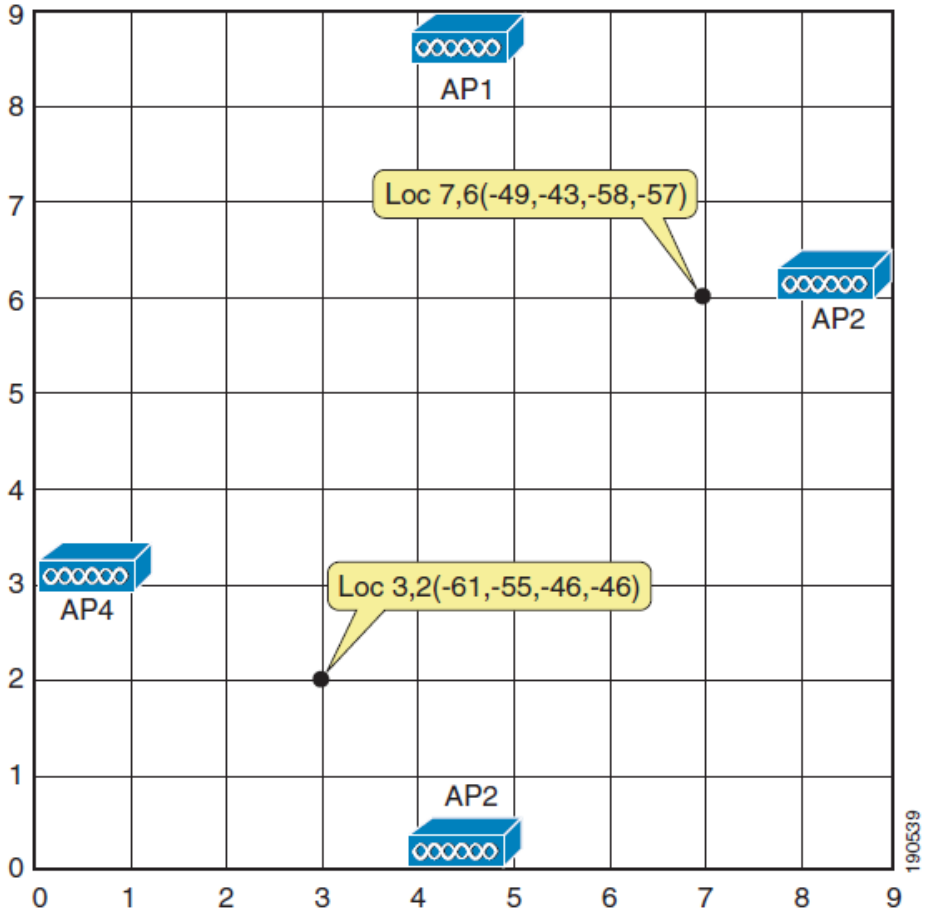
Az egyik része a kutatásoknak**[2]** abba az irányba ágazódott le, ami az autonóm mobil robotok navigációjával foglalkozik. Azon a területen az volt a cél, hogy a robot úgy tudjon emberi beavatkozás nélkül navigálni, hogy kizárólag a saját szenzoraira hagyatkozik. Ebben az esetben nincs szükség semmiféle távoli egységre (mint például egy navigációs szerver), amivel kommunikálva a robot megkapja az információt a helyzetéről, vagy a saját útvonaláról. A mobil egység tehát kizárólag saját eszközei által kibocsájtott és szenzorai által érzékelt jelek (például ultrahang) segítségével térképezi fel a környezetét, a mért adatokat feldolgozza, és így egyúttal képes eldönteni azt is, hogy merre menjen tovább. Ebben az esetben a navigáció és a pozícionálás is a kibocsájtott hullámok visszaverődésén alapszik; azaz alap esetben, ha ismerjük a hullámhosszt, és a kibocsájtás és a visszavert hullámok érzékelésének az időkülönbségét, egy egyszerű $S=V*T$ képlet alkalmazásával megkaphatjuk a hullámot visszavert tárgytól való távolságunkat. Ilyenkor azonban mindenképpen szükséges a belterület előszöri feltérképezése, azaz a szükség van arra, hogy a robot elsőként végigjárja az épületet, hogy felépítse a rendszerében a térképet, amivel a későbbiekben már nagyobb biztonsággal tud elnavigálni különböző megadott helyek között. Emellett természetesen hasznos, ha egy jól kialakított tárgyfelismerő rendszert is használunk az egységen, ami nagy mértékben hozzájárul a sikeres tájékozódáshoz.

Hasonló megoldást használunk azon esetekben**[3][4]**, amikor a mobil egységünk által kibocsájtott ultrahang-hullámokat a terem különböző pontjaiban elhelyezett adó-vevő egységek mérik, és tulajdonképpen arról kapunk információt, hogy az ismert pozíciójú egységek helyzetéhez képest az általunk kérdéses helyszín merre található. Ez a megoldás már egy fokkal közelebb van jelen témához, hiszen ezt könnyebben alkalmazhatjuk azon esetben, amikor például egy bevásárlóközpontban akarunk tájékozódni. Ilyenkor ugyanis ha már saját helyzetünk információinak birtokában vagyunk, arról is képesek vagyunk informálódni, hogy a mi helyzetünkhöz képest a cél-helyszín merre található.

Ezzel azonban még a sikeres megoldáshoz szükséges jó pár feltételt nem teljesítettünk. Legelőször ugyanis ehhez egy olyan speciális eszközökre van szükségünk, amelyet minden egyes navigálni kívánó személy rendelkezésére tudnánk bocsájtani. Ehhez természetesen új, kisméretű eszközök gyártására lenne szükség, hiszen jelenleg nincs a hétköznapi felhasználó birtokában olyan készülék, amely képes lenne az efféle felhasználásra.

A következő ötlet azonban lényegesen közelebb áll az általam felvetett probléma megoldásához: tegyük fel, hogy az előző esethez hasonlóan a környezetünkben elhelyezett egységek által kibocsájtott hullámok segítségét használjuk fel, ám ezek nem ultrahang

hullámok, hanem olyan vezeték nélküli jelek, amelyek a bevásárlóközpontokban, múzeumokban, egyetemeken, vagy egyéb épületekben ugyanúgy foghatóak. Ilyenek például a WiFi hozzáférési pontok és a mobilhálózatok antennái által sugárzott jelek.




1. ábra: helymeghatározás négy vezeték nélküli hozzáférési pont segítségével [5]

Ennél a megoldásnál ugyanis nincsen szükség semmilyen speciális eszközre, aminek a felhasználó rendelkezésére kéne állnia. A mai okostelefonokba beépített érzékelők képesek ezen jeleket fogni és értelmezni. Minden adóegység által kibocsájtott jel ugyanis információt hordoz a kibocsátójáról, ezáltal a helymeghatározás csupán alkalmazásszintű implementáció kérdése. Az 1. ábra épp ezt a módszert szemlélteti: van egy helységben négy elhelyezett WiFi router, amik jeleket bocsájtanak ki magukból. Ebben a helységben, ahol a helymeghatározást akarjuk végezni, akármelyik ponton figyelem meg a hozzáférési pontok által kiadott jeleket, a különböző eszközök jelerősségét mindenhol különbözőnek fogom érzékelni. Azok után pedig, hogy ezeket az egységeket a vett jel alapján meg tudjuk különböztetni egymástól (például az egyedi BSSID-juk alapján), igen jó közelítést tudunk adni a saját helyzetünkre is.

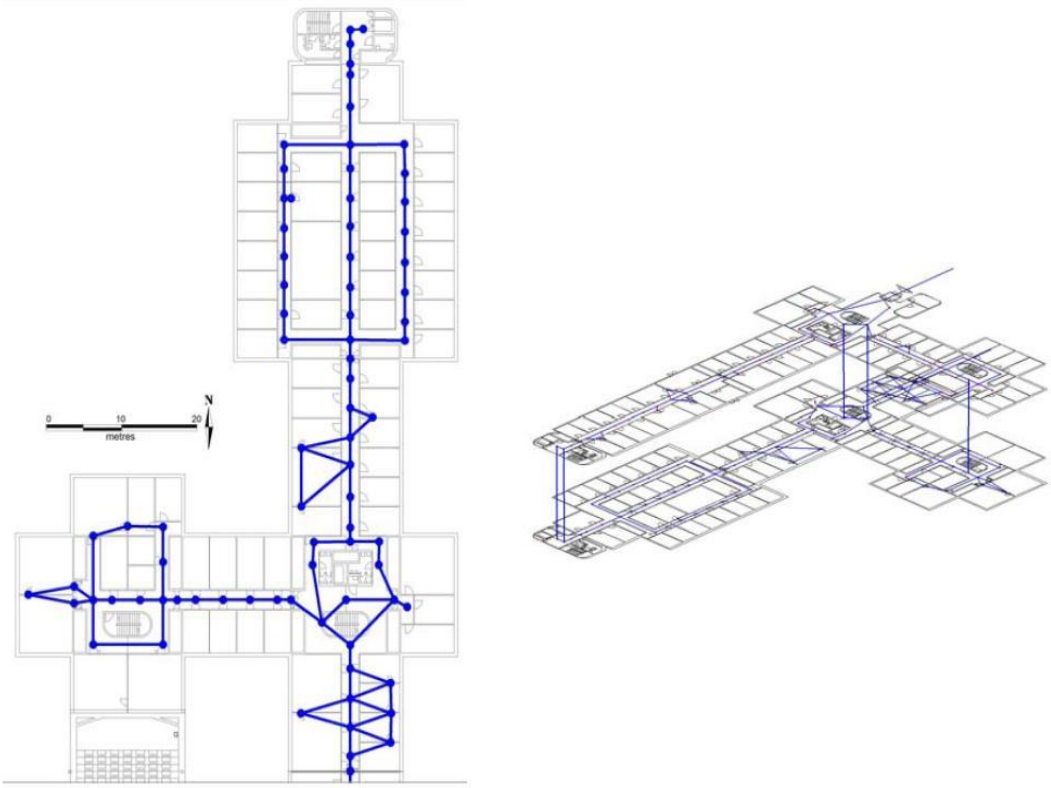
Hogy ezen módszer segítségével beltéri navigációs eljárást is implementálni tudjunk, szükségünk van először is arra, hogy a különböző helyszíneket pontként megragadjuk, és valamilyen módon egymáshoz fűzzük. Ez alatt azt értem, hogy célszerű úgy egymáshoz

rendelni a pontokat, ami alapján a valós életben is „összeköthetőek”; azaz azon pontokat kell összekötni, melyek egymásból közvetlenül elérhetőek. Nem lenne célszerű ugyanis, ha a navigációs alkalmazásunk a falon át vezetné a szomszéd szobába a felhasználót. Az eddigi, ezzel a módszerrel kapcsolatos megoldások**[6][7][8]** szinte kivétel nélkül ugyanazt a választ adták erre a felvetésre: a legjobb megoldás, ha az épület belterét valamilyen formában egy gráfként reprezentáljuk.

A gráf tulajdonképpen csomópontok és élek halmaza, ahol az egyes csomópontok közötti összeköttetéseket a közéjük behúzott élekkel jelöljük. Egy általános gráfban beszélhetünk irányított és irányítatlan élekről attól függően, hogy az adott összeköttetés egyirányú-e, azaz hogy ha egy csomópontból el tudunk jutni az egyik szomszédjába a közöttük lévő él alapján, akkor ugyanezt az utat visszafelé is meg tudjuk-e tenni. A navigáció szempontjából az esetek jelentős többségében nincs értelme az élek irányítottságának használatára, ugyanis ha az egyik helyszínről el tudunk jutni egy másik helyszínre, akkor minden valószínűséggel azt az utat ugyanazon a módon visszafelé is meg tudjuk tenni.

Ezek után szükségünk van még bevezetni az élsúly (vagy jelen esetben a csomópontok közötti távolság) fogalmát, azaz azt, hogy egy csomópontból a szomszédját milyen költséggel, milyen hosszú út megtételével érhetjük el. Szükség van tehát egy leképezési egységre, amely a valós helyzethez hűen adja meg két csomópont egymáshoz viszonyított távolságát. Erre egy megoldás**[8]** például, ha az épület különböző szintjeit azonos, de kis méretű, négyzet alakú cellákra osztjuk fel, amik egymáshoz szomszédsági alapon kapcsolódnak. Egy cellának ilyen formában 8 szomszédja lehet, melyeknél használható a cella középpontja = gráf csomópont megfeleltetés. Mivel a távolságot ily módon egységesítették, ha az egyik csomópontból el akarunk jutni az egyik szomszédjába, él súlyként a cellamérettel számolhatunk. A megoldás-felvetések közül nem biztos, hogy készen ez a fajta távolság-leképezés a legjobb módszer, de egy jó alapot nyújthat számunkra a további megoldások vizsgálatához.

Ha az épületet tehát egy gráffal fedjük le, és az egyes helyszíneket (például, ha a bevásárlóközpontos példánál maradunk: boltokat, mellékhelyiségeket, kijáratokat, stb.) a gráf egy csomópontjaként jelöljük meg, akkor egy elég jó modellt kaptunk az épületnek egyfajta navigációra alkalmas reprezentálásához.




2. ábra: Egy épület gráffal történő lefedésének alapötlete, két és három dimenzióban[9]

Ám ezzel korántsem oldottuk még meg a feladatot: ugyanis ha már rendelkezésre állnak a helyszínek, mint csomópontok, és a közöttük elhelyezkedő összeköttetések, mint az egymásból közvetlenül elérhetőségnek a reprezentációja, szükségünk van még olyan algoritmusokra, melyek szerint a gráfban az egyik pontból, lehetőleg a legrövidebb, optimális úton eljuthassunk abba a csomópontba, amely a célállomásunkat fedi le.

A gráf algoritmusok közül a navigációhoz használt legnépszerűbb algoritmus a Dijkstra-algoritmus segítségével történő útkeresés**[9]**, emellett előfordul az úgynevezett A* algoritmus használata is**[10]**. A cél tulajdonképpen az volt, hogy egy olyan összefüggő hálózaton, mint az épület alapján felépített gráf, minél rövidebb idő alatt, a lehető legkisebb bonyolultsággal működve az optimális utat válassza a rendelkezésre állók közül. Mivel minden algoritmus általában különböző nyelveken és gépeken valósítható meg, ezért a bonyolultság alatt itt az algoritmus megválasztott lépéseinek és műveleteinek számát értjük.

### 1.2.2. A munkám illesztése az eddig ismertetettekhez

Mint a legtöbb fejlesztés, a jelen munkám a beltéri navigációról is egy nagyobb projekt része. A műegyetem Távközlési- és Médiainformatikai Tanszékén folyamatban van egy olyan alkalmazás kifejlesztése, mely a beltéri helymeghatározáson alapul, és szolgáltatások egész sorát kívánja majd nyújtani. A projekt alapja egy RedPin**[11]** nevű nyílt forráskódú

helymeghatározó program, melyet részletesebben egy későbbi pontban ismertetek. A cél egy olyan Android alapú mobil alkalmazás létrehozása az alapprogram alapján, ami az épületen belüli helymeghatározáson kívül képes a különböző helyfüggő szolgáltatásokat is kezelni, különféle authentikációkkal felruházva működhessen mind a saját szervere, és (lévén androidos alkalmazásról szó) mind a Google felé, emellett nem mellékesen képes legyen a beltéri navigációra is. A navigációs fejlesztés is több felé ágazódik el: van, aki a vektorgrafikus megjelenítésen dolgozik, és van, aki egy olyan térképszerkesztő programot készít, amivel könnyen lehet az újonnan készülő alkalmazás számára térképeket szerkeszteni, és a központi adatbázisba a program számára eltárolni azokat. Az én feladatom több részre oszlik: először is egy olyan adatmodell kidolgozása a rendszer számára, amelybe a megszerkesztett térképeket, mint navigációhoz és helymeghatározáshoz szükséges adatelemeket tárolni lehet. Miután az adatok bekerültek az adatbázisba, szükség van a szerveroldalon egy navigációs adatmodellbe leképezni a térkép-adatokat (amit én eddigi kutatásaim eredményeként gráfként azonosítok), aminek a segítségével az épület különböző helyszínei között képesek leszünk a navigálni. Miután a gráf elkészült, szükség van az irodalomból megismert útvonalkereső algoritmusok implementációjára is, melyek alapján optimális úton eljuthatunk az épületen belül az egyik helyszínről a másikra. Végül pedig utolsó lépésként integrálni kell az alapprogramba a navigációs részt, majd tesztelni azt.

# 2. A megoldás lépései

Ahhoz, hogy a navigációs megoldásunk sikeresen működjön, szükség van tehát az adattárolási adatmodell és a gráf átgondolt implementálására.

Minden rendszer alapja egy olyan eszköz, mely képes a rendszer által kezelt adatokat tárolni. Ezen kívül igen fontos szempont az is, hogy odafigyeljünk arra, hogy az adatoknak nem csak a nagy mennyiségben való eltárolása a fő szempont, hanem könnyen el is érjük, módosíthassuk, vagy törölhessük azokat. Szinte kivétel nélkül találkozhatunk olyan rendszerekkel, melyek az adataikat strukturáltan, rendszerezve tárolják, melyeknek a különböző részeire a mindennapos használat során szükség van.

A különböző helymeghatározó és navigációra alkalmas alkalmazások sikeres működése (természetesen az algoritmusok mellett) elsősorban az adatbázisban eltárolt adatok felépítésétől függ. Jelen esetben például a navigációhoz szükséges legfontosabb elemeket kell csak eltárolni, amelyek főbb vonalaiban meghatározhatnak egy épületet, amiben a navigáció működhet. Emellett az is fontos tényező, hogy a projektmunka idekapcsolódó elemei könnyedén kezeljék a megoldást. Elsőként tehát szükség volt egy olyan adatmodellre, amelybe az éppen készülő térképszerkesztő program az adatokat tárolni tudja, és amelyből az

információt kinyerve könnyen felépíthető adatszinten az épület, és végrehajtható rajta a navigáció.

Miután ez elkészült, szükség van a szerveroldalon az adatbázisból lekérdezni a számunkra lényeges adatokat, melyekből felépíthető a gráf, mint az épület adatszintű térképe. A sikeres működéshez elengedhetetlen feltétel, hogy a felépített gráfunk metrikus legyen, vagyis ha két csomópont között adott egy úthossz, akkor ha az úthoz hozzáveszünk két csomópont közé egy harmadikat, az újonnan kapott, egyel több élt tartalmazó út költsége nem lehet kisebb az eredetinél. Emellett, mint már korábban írtam, szükség van az épületen belüli távolságok és a pontok közötti útköltségek optimális egymáshoz illesztésére is.

Végül, de nem utolsó sorban meg kell említeni, hogy az alkalmazás, amit továbbfejlesztünk, jelenleg a szoba szinten történő pontosságú helymeghatározásra képes. Ezen kívül azt is hozzá kell tenni, hogy az általunk fejlesztés alatt álló rendszernek nincs is szüksége ennél pontosabbra, hiszen nem arra vagyunk kíváncsiak, hogy egy nagyobb épületen belül elhelyezkedő kisebb szobán belül mi merre található, hanem arra, hogy az épületen belül melyik helyiséghez számunkra milyen optimális úton lehet eljutni. Ezek alapján pontosíthatjuk a célt azzal, hogy nekünk elég a keresett helyszín ajtajáig elnavigálnunk, hiszen ezek után már csak be kell lépnünk a jelzett helyiségbe, hogy elérjük a célunk. Ez gráf-adatokban annyit jelent, hogy csomópontokat kell felvennünk az ajtókhoz, hogy ez által a mögöttük lévő helyiségek útvonalak végcéljaként megjelölhetőek legyenek (kijárat esetén pedig értelemszerűen a külvilág).

Végül pedig meg kell említenünk azt az esetet, ami nem túl gyakori, ugyanakkor bizonyos épületekben előfordulhat: találhatóak olyan helyiségek, melyeknek egynél több ajtaja van, jellemzően a szoba különböző oldalain. Ezt az esetet azért kell számításba venni, mert helytelen lenne az útvonalkeresés, ha egy optimális út kalkulálásakor belevenné azt az utat, ami átvezet egy szobán. Tegyük fel ugyanis, hogy ez egy magáncélra kialakított iroda; ilyenkor nem örülne a tulajdonos, hogy idegen személyek átjáróként használják a helyiségét. Ezen felsorolt pontok tehát a legfőbb tulajdonságok, melyeknek teljesülnie kell a megvalósított elemekre ahhoz, hogy végeredményként valósághű és gyakorlatban jól használható alkalmazást kaphassunk. A sikeres megvalósítás utolsó lépéseként szükséges még a kliens és a szerver felkészítése az eddig munkának a teszteléséhez. Ehhez a projektmunka során az általunk továbbfejlesztendő alkalmazás kliens oldali programjának az Androidos változatát fogom felhasználni.

# 3. A felhasznált adatmodellek

A megoldáshoz első lépésként az alapprogram adatbázisához kellett hozzáférjek, hogy kialakítsam a közösen használható adatmodellt a készülő térképszerkesztő és a saját munkám

részére. Olyan struktúra létrehozására volt szükség, mely az épület részleteit lényegeiben megragadja és számomra a navigációhoz felhasználhatóvá teszi. Először tehát felépítését tekintve felbontottam az épületet: főbb alkotóelemei a következők, melyeknek a szervezett kezelésével a programban adatszinten rekonstruálható maga az épület:

- *Épületek:* azon elemek, amiken belül a navigáció végrehajtható. Azért láttam szükségét egy ilyen tábla létrehozásának, mert a továbbfejlesztések során könnyen képessé tehetjük a rendszert összefüggő épületkomplexumok, épületcsoportok kezelésére is.
- *Szintek*, melyeken a különböző termek, irodák, folyosók helyezkednek el. Ezeknek az elemeknek a rétegei alkotják meg magát az építményt.
- *Szobák*; ezen építőelemek reprezentálják a szinteken elhelyezkedő különböző méretű zárt tereket. Ám nem csak a termek, irodák minősülnek szobának: konkrétan minden terület, amit falak vesznek körül, és ajtókon át lehet közlekedni rajta. Ilyen szempontból tehát ebbe a kategóriába tartoznak a szinteken található különböző folyosók is. Itt fontos megemlíteni, hogy mivel a folyosók átjárhatók, és az ilyen típusú szobákon folyik a navigáció, ráadásul rendkívül hosszúak és éles kanyarokat írhatnak le, ezért minden, a navigáció szempontjából átjárható szobának célszerű adnunk egy összesített globális maximum értéket, amekkora egy falának a hossza lehet. Ennél nagyobb méret esetén több, kisebb egységre felosztva ajánlott az adatbázisban tárolni ezeket. Ennek a strukturálásnak pontosabb indoklását egy későbbi bekezdésben tárgyalom.
- *Ajtók*, melyek tulajdonképpen az átjárható elválasztó elemek a szobák között. Ez annyit jelent, hogy ha két elem között megengedett az átjárás, akkor bekerül egy ajtó a kettő közé. Logikailag két fő csoportba oszthatjuk ezeket: léteznek szokásos ajtók, ezeken át lehet közlekedni a különböző típusú szobák között, emellett találhatóak folyosószakasz-folyosószakasz közötti átjárást reprezentáló ajtó-elemek is. Erre a célra szolgál a másik kategória, amik elemeit úgynevezett „virtuális ajtókként” képzelhetünk el. Ezek ugyanis csak is kizárólag az azonos szinten lévő elemek elkülöníthetőségét biztosítják, de emellett szükség van arra is, hogy jelezzük valahogyan, hogy ezeken is megengedett az áthaladás.
- *Falak* azok az elemek, amelyek a szobákat határolják. A különbség az ajtókkal szemben az, hogy ezek átjárhatatlanok, így lényegében megadják a formai határait a szobáknak.
- *Pontok:* a hierarchia legalja, az alapvető építőeleme mindennek. Természetesen nem kerül minden egyes pont külön-külön tárolásra, csupán az elemek főbb határpontjaira és referenciapontjaira van szükség.

Ezeket a főbb alkotóelemeket kell tehát táblákban, segédtábláikkal együtt eltárolni egy adatbázisban, hogy az adatreprezentáció a navigációhoz szükséges módon, hűen tükrözze az épület valódi felépítését. A különböző segédtáblák csupán arra kellenek, hogy az elemek elhelyezkedését, valódi típusát és kapcsolódási pontjait reprezentálja. Ilyen táblák például a szobákhoz kapcsolt segédtábla, melyből a szoba típusáról tudhatunk meg információkat, mint például hogy iroda-e, vagy folyosó a vizsgált elem.

Mint már korábban említettem, az átjárható szobák méretének szükséges felső korlátot adni (ennek legfőképpen a folyosóknál és olyan nagyobb termeknél, mint például az aula van szerepe), azaz ha egy ilyen típusú szoba falainak hossza egy bizonyos értéknél nagyobbak, akkor az lehetőség szerint több olyan szobára bontva kerül be az adatbázisba, aminek azon falai, amelyek egymás felé néznek, értelemszerűen átjárhatóak (ezeket hívhatjuk „virtuális ajtóknak"). Azért tároljuk ezeket ajtóként és nem falként, mert elsődleges szempontként az átjárhatóságot tekintjük az egyes elemek között.

A nagyobb méretű, átjárható szobák „tördelésének" oka a következő: az épületen belül elhelyezkedhet akármilyen hosszú és kanyargós folyosó, ezen kívül tetszőleges méretű, egybefüggő belső terek is megtalálhatóak. A navigációnál az a lényeg, hogy ezekben az elemekben egy egységes út legyen felépíthető, azaz egy tetszőleges alakú és méretű folyosón is ugyanolyan viszonyítással legyenek elbírálva a távolságok, mint egy rövid és egyenes helységben. Ehhez azonban szükséges, hogy minden belső tér méretének egy felső határt szabjunk, hogy azokon belül ugyanolyan struktúra szerint lehessen elhelyezni a csomópontokat. Közös megegyezés alapján ennek egy 10m x 10m –es felső korlátot adtunk, ám hangsúlyozva az adatnak csupán az iránymutatást szolgáló értékét, az esetek többségében nem történik probléma akkor sem, ha ettől a szoba típusától és valós méretétől függően kis mértékben eltérünk. Egy olyan ritka esetben például, amikor egy folyosót azért kellene felbontanunk, mert több helyen ír le kisebb kanyarokat, és az adatbázisba az előbb említett iránymutató értéktől felfelé eltérve vesszük fel a folyosót alkotó szobák méretét, akkor mivel gráf csomópontokat az átjárhatóságot tekintve a később részletezett módon a szoba középvonalára vettük fel, és így alapjában véve a szobán keresztülmenő gráf élek is a középvonalon mennek át, az említett élek egy bizonyos mérethatár után a szoba egyik oldala felé kezdenek konvergálni, majd túl nagy méret után kilóghatnak magából a szobából.

Miután az adatbázisban a táblák sikeresen létrejöttek, szükség van a tartalmukat a szerver oldalra lekérdezni, hogy ott objektum szinten felépülhessenek belőlük az épület térképén szereplő, navigációhoz szükséges elemek. Ehhez természetesen szükség van a szerver oldalon a megfelelő osztályok létrehozására is. Amikor ez a lekérdezés megtörtént, a létrejött objektumok és az eddig támasztott kritériumok alapján fel kell építeni egy gráfot, amin a később megadott algoritmusokkal végrehajtható a navigáció. Miután tehát adatszinten felépítettük a térkép elemeit a szerveroldalon, ahhoz még, hogy ezt átültessük egy gráfba, létre

kell hozni egy olyan interfészt, ami segítségével a létrejött objektumokból egy gráf létrehozható. Ezzel tehát meghatároztuk, hogy a szerveren három főbb dolog létrehozása szükséges a navigációs algoritmusok mellett a megbízhatóan működtethető alkalmazáshoz.

# 4. A RedPin alkalmazás

A RedPin egy nyílt forráskódú, mobil eszközön futtatható, beltéri helymeghatározásra alkalmas alkalmazás, mely kifejlesztésének a fő célja a szoba-szinten működő pontos helymeghatározás volt. Azaz ha egy épületen belül ezen program segítségével kívánunk tájékozódni, ha ugyan nem is ad teljesen pontos pozíció-meghatározást, de azt meg tudja határozni, hogy az épületen belül melyik helyiségben tartózkodunk.

A rendszer kialakításának főbb tényezői a következők voltak**[12]**:

- Olyan eszköz használata a helymeghatározásra, amelyet a legtöbb ember manapság használ.
- A működéséhez ne legyen szükség semmilyen speciális egységet az épületen belülre felszerelni.
- Könnyű legyen karban tartani, és egyszerű, felhasználóbarát legyen a használata.
- Legalább szoba-szinten határozza meg pontosan a helyzetünket.
- A környezet változásához könnyű legyen adaptálni.

Működése a lenyomatoló technikán alapul, azaz nem globális koordinátákként kapjuk meg az aktuális pozíciónkat, hanem szimbolikus azonosítókkal (mint például a szoba neve) ellátott helyszínekként. Emellett a program nagy előnye tehát az is, hogy könnyű alkalmazkodtatni a változó környezethez is olyan szempontból, ha például egy általa felhasznált hozzáférési pontot áthelyeztünk más helyre az épületen belül.

A program a helymeghatározást a mobil-eszköz különböző szenzorainak segítségével végzi: alapvetően háromféle értéket mér: WiFi-jeleket, Bluetooth-jeleket és moblilhálózat-cella adatokat. Értelem szerűen, ha egy eszközön a három közül valamelyik vevőegység nem elérhető, vagy a méréskor nincs bekapcsolt állapotban, akkor azon adatokat a program nem rögzíti, viszont ez adott esetben nagyban megnehezítheti, vagy akadályozhatja a pozícionálást. Ahhoz, hogy a helymeghatározás jól működjön, szükség van tehát ezen mért jelek részletes információira, tehát például WiFi esetében szükség van a hatótávolságon belül lévő hozzáférési pontok jelerősségei mellett az SSID-kra, BSSID-kra, és a titkosításuknak módjára, hiszen gyakran előfordul, hogy egy épületen belül több, azonos nevű (SSID-jú) hozzáférési pont van elhelyezve, ezeket pedig jól el kell különítenünk egymástól.

A rendszer három fő részből áll: kliens-, szerver- és adatbázis oldali részből. A kliens program két fő alkotó elemből épül fel: az egyik egy „hallgatózó” komponens, ami információkat gyűjt a fent felsorolt, hatótávolságon belül lévő vezeték nélküli eszközök jeleiről, ezáltal egy

lenyomatot képez az aktuális pozícióról. A másik fő komponense egy ún. „lokátor”, mely eltárolja az eddig mért lenyomatokat egy adatbázisban, emellett magába foglalja az algoritmust, amely az eddigi adatok alapján képes a mobileszközt lokalizálni. Hogy ez mind megbízhatóan, gyorsan és könnyedén menjen, a lokátor egy központi szerveren fut, melyhez a kliens, amin a másik fő komponens fut, csatlakozik. A program fő működése tehát a kliens-szerver elvének felhasználásával működik. Ezeken kívül az adatbázisban külön táblákban kerülnek tárolásra a rendszer működéséhez nélkülözhetetlen adatok, mint például az eddig mért értékek, helyszínek, lenyomatok és térképek.

## 4.1. A szerver

A szerver több funkcióval szolgál a mobil kliens részére. Elsőként egy tároló szolgáltatást, amely lehetővé teszi a kliens számára, hogy az általa a szervernek beküldött lenyomat-adatokat egy adatbázisban eltárolja. Ez a szolgáltatás hívódik meg mindig, amikor a felhasználó eltárol, vagy felüldefiniál egy pozíciót.

Egy másik funkció, a térképek fel- és letöltése, ezek úgynevezett PNG-formátumú képek a szintek alaprajzairól. Itt fontos tehát megemlíteni, hogy a program a pozíciónk vizualizálásakor az adott emelet alaprajzát, mint térképet tölti be a szerverről, és azon jelöli a helyzetünket.

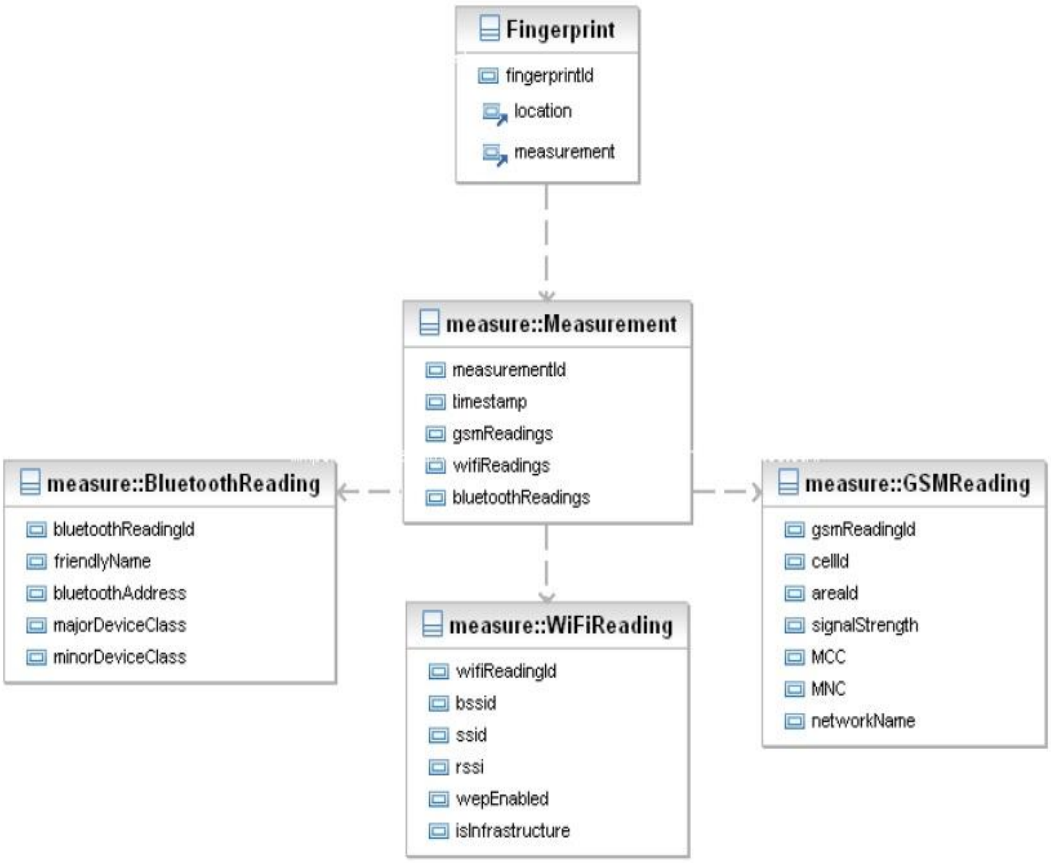
Végül pedig a legfontosabb funkciója, hogy fogadja a kliens által küldött lenyomatokat, és ezeket az adatbázisból lekérdezett, már eltárolt lenyomatokhoz hasonlítja. A mobil eszköz meghatározott pozícióját tehát az a lenyomat adja, amely adataiban és mért értékeiben a legközelebbi értékeket mutatja a szerver által az eszközről fogadott lenyomatnak, természetesen, ha az értékek különbsége egy előre definiált küszöbindexen belül esik. Nem volna értelme ugyanis annak, hogy azért, mert egy területet például nemrég építettek hozzá az épülethez (természetesen azon belterület még nincs feltöltve mérési adataival együtt az adatbázisba), és azon a helyen végeznénk a programmal pozíció-meghatározást, eredménynek értelem szerűen a legközelebbi, még az adatbázisban lévő helyet kapnánk.

## 4.2. A kliens

A kliens gyakorlatilag maga a mobil eszközön futtatandó program, mely információkat gyűjt a fent említett vezetéknélküli jelekből a készülék fent szenzorainak felhasználásával, és ebből egy információ-csomagot, úgynevezett lenyomatot képez, melyet egy bufferben tárol, és sikeres kapcsolódás esetén elküld a szervernek. A lenyomat azonban a szenzorok által mért adatok mellett egy időbélyeget is tartalmaz, hogy a szerver meg tudja különböztetni a hozzá elküldött adatokat; ugyanis fogadhat többször olyan lenyomatot, amelynek a szenzorok által mért adatai ugyanakkorák (tehát nagy valószínűséggel ugyanazon a helyen többször történt

pozícionálás), viszont nincs két olyan, amit ugyanazon időpillanatban, ugyanazon a helyen mért az eszköz, ugyanolyan mért adatokkal.

Miután sikeresen kapcsolódtunk a szerverhez, és elküldtük a lenyomatot, ő visszaküldi nekünk az általa beazonosított pozíciónkat a térképen. A térkép fel- és letöltés az egyik alapfunkciója a szervernek, ezáltal biztosak lehetünk benne, hogy ha eddig nem tartalmazta a térképfájlt a kliensünk, a szervertől azt most megkapjuk.



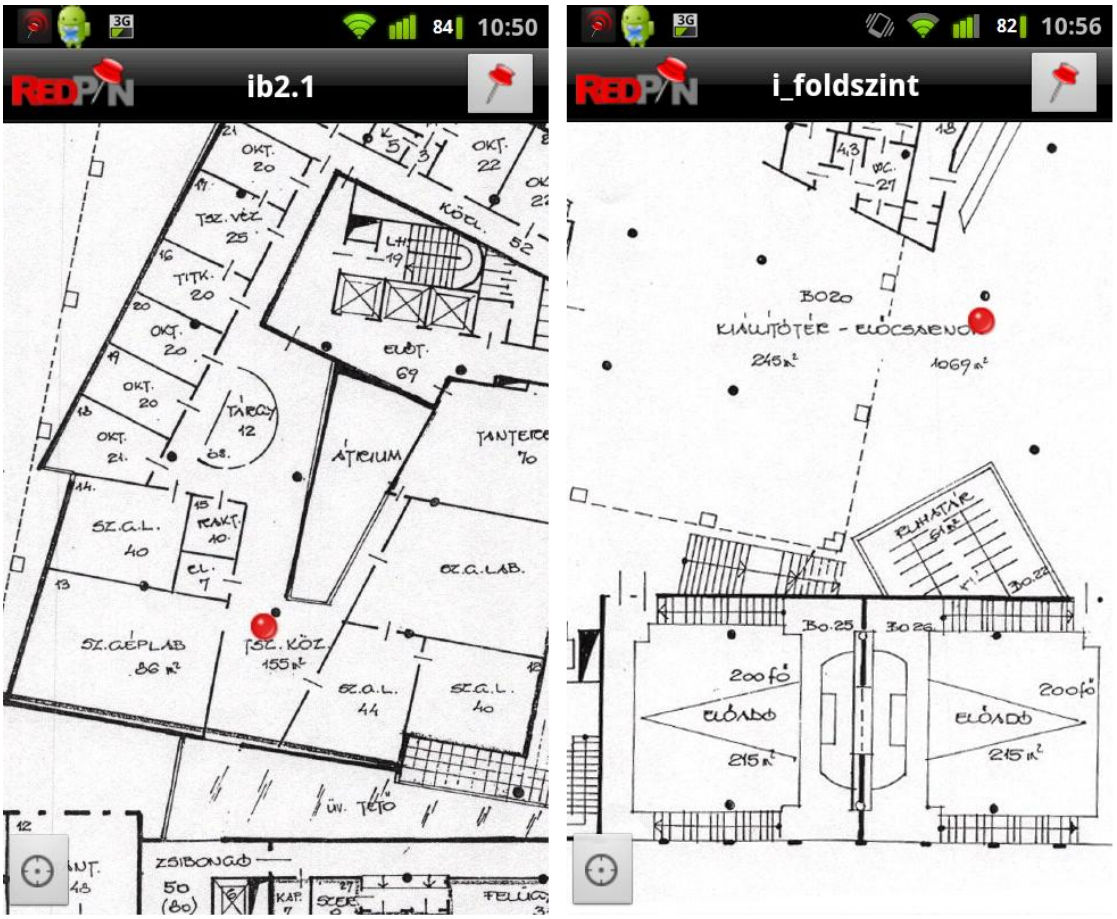

3. ábra: Lenyomatképzés a kliens-eszköz szenzor-adataiból

A térképen, mint alaprajz-képen a beazonosítás az X és Y koordináták alapján történik; a szerver által visszaküldött információ ugyanis a térkép-azonosító mellett tartalmazza az azon elhelyezkedő pont koordinátáit is. Legvégül tehát, ha minden folyamat sikeres volt, a kliens megjeleníti az aktuális térképet, amelyiken a mérések szerint tartózkodunk, és azon a kapott koordináták alapján egy piros körrel jelöli az általa meghatározott helyzetünket.

## 4.3. Működési teszt

Mivel a projekt fő célja egy, már meglévő beltéri helymeghatározó alkalmazás továbbfejlesztése, szükséges volt annak a vizsgálata, hogy ezen rendszer alkalmas-e a kitűzött

feladatok megvalósítására. Hogy ezt meghatározzam, a tanszéki épületben elvégeztem néhány helymeghatározási mérést a programmal.




4. ábra: a program kimenetei a mérések alapján

Összesen 10 mérést végeztem el az épület különböző pontjain annak vizsgálataként, hogy mennyire képes az alapprogram a szoba szinten történő helymeghatározásra. Ezek közül 9-szer helyesen megtalálta a legközelebbi helyszínre mutató lenyomatot az adatbázisból (ami annyit jelent, hogy maximum pár méteres eltérés adódott a tényleges pozícióm és a program által meghatározott pont között), egyszer pedig az épületben a megfelelő helyszínt azonosította, csak az egy szinttel lejjebb található emeleten. Természetesen ez az információ is pontosítható, ha kisebbre állítjuk a szerveren a küszöb értéket, ami azt határozza meg, hogy a mért értékekhez képest mekkora eltéréssel vegye figyelembe az adatbázisban eltárolt lenyomatokat a program.

## 4.4. Összegzés

Mint kiderült, ez a rendszer tökéletesen alkalmas a továbbfejlesztésre mind beltéri navigáció, mind pedig egyéb helyfüggő szolgáltatások terén. Mivel már alapvetően tartalmaz pontokat

(helyszíneket rájuk vonatkozó koordinátákkal), ezeket meg lehetne feleltetni a navigációs gráf csomópontjainak, ám az ezek közötti éleket lehetetlen lenne meghatározni pusztán a másik réteg, a megjelenített térkép alapján. Hogy a program mégis képes legyen a helymeghatározás mellett a navigációra is, alapjaiban véve kell átdolgozni, majd a rendszerhez igazítani a definiált navigációs módszereket, természetesen úgy, hogy az elkészült rendszerre is igazak legyenek a kiindulási feltételként megadott elemek.

# 5. A megvalósítás

Azok után, hogy minden feltétel ismert, és minden szükséges eszköz a rendelkezésünkre áll, nekiállhatunk a konkrét megvalósításnak. Mint már tudjuk, a megoldáshoz három elemen kell az implementációt végrehajtani: a kliens oldalon, a szerveroldalon, és az adatbázis oldalon.

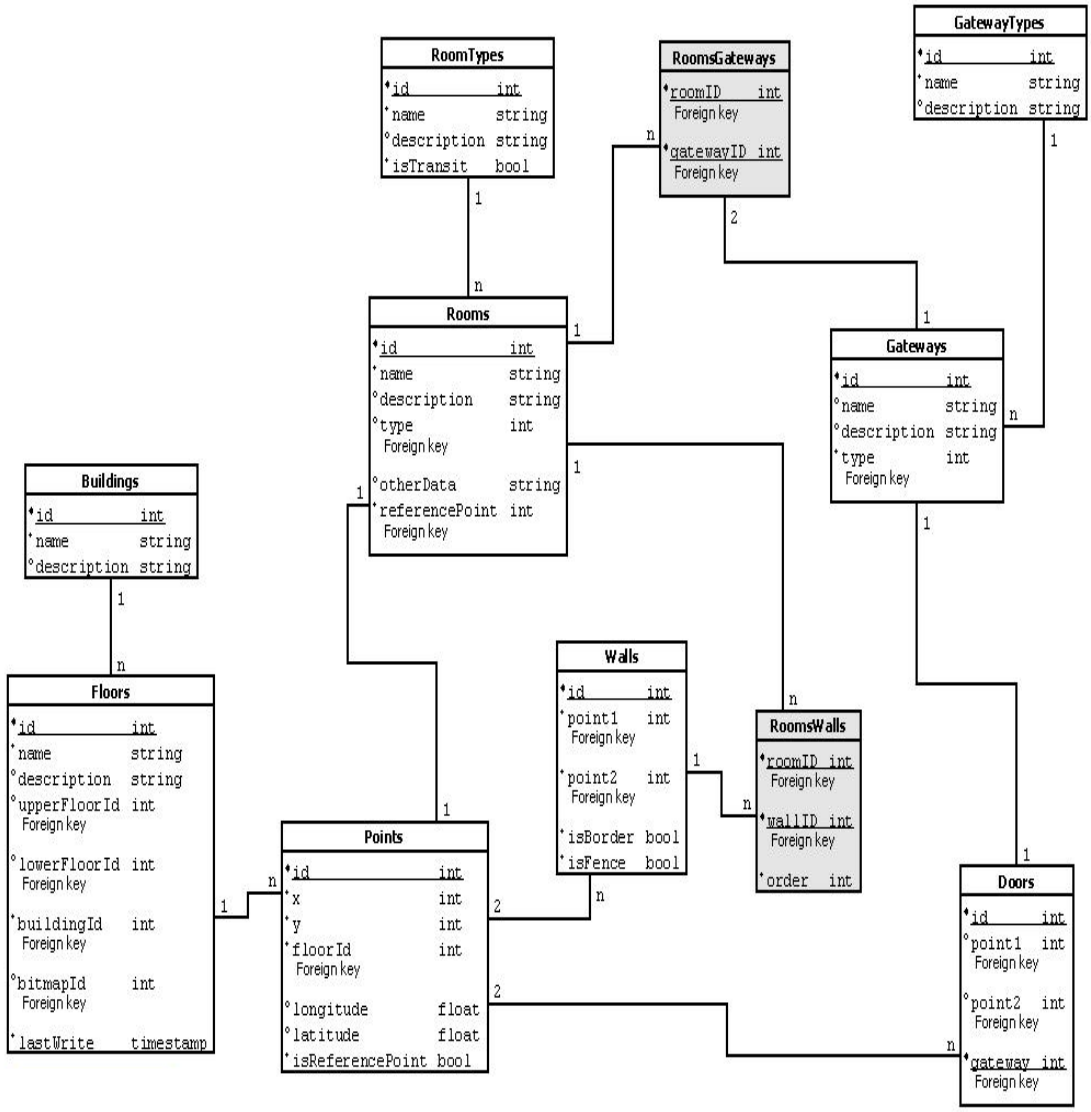
## 5.1. Az adatbázissal kapcsolatos elemek

Legelőször szükség volt a RedPin adatbázisába a navigációhoz használt térkép-adatok tárolásához szükséges táblákat megvalósítani. Ez az adatbázis egy tanszéki MySQL adatbázis szerveren helyezkedett el. Elsőként szükséges volt az említett főbb elemeknek táblát készíteni; ezek a pontok, ajtók, falak, szobák, szintek és épületek. Mindegyikhez tartozik egy, a lekérdezéshez szükséges azonosító. Ezek természetesen a szerveroldalon az objektumokká történő felépítésnél is hasznosak, hiszen egy elemre legcélszerűbb az egyéni azonosítója alapján hivatkozni. Ezen kívül az ajtóknak, szobáknak, a folyosóknak és az épületeknek szükséges, hogy adjunk emberi szemmel is olvasható, számunkra is könnyedén azonosítható nevet. Az egyszerű felhasználó a mobil eszközön a legtöbbször ezeket az adatokat tekintheti meg, amikor éppen tájékozódni akar. Azért csak legtöbbször, mert a gráfépítés során nem egy az egyben adatbázisban tárolt pontokból képződnek csomópontok, ezek ugyanis egy későbbi pontban leírt módszer szerint épülnek fel. Egy folyosószakaszon több pontot veszünk fel a gráfba, ezeknek értelem szerűen egyéb nevet kell adnunk a folyosószakaszon belül. A szinteket eltároló táblába bekerülnek ezen kívül az alatta és a felette lévő szintek azonosítói (értelem szerűen, ha van olyan), és mivel egy szint felel meg egy térképnek is, megtudhatjuk a hozzá tartozó, térkép azonosítóját is. A pontoknál szükség volt az X és Y koordináták eltárolására, ezen kívül a globális hosszúsági és szélességi adatok tárolására is van lehetőség. Ez azt a célt szolgálja, ha például egy épületcsoporton belül akarjuk majd használni a navigációt (mint például egyetemi campuson belül), akkor még több információnk lehet az elhelyezkedésről. Hozzá kell tenni, hogy ez egyelőre egy még ki nem használt funkció, csupán a továbbfejleszthetőség érdekében került bele.

A pontokat tároló táblához kapcsoltam a szobákat, melyeknek még természetesen szükségük van az őket határoló falakra és ajtókra, amikből maga az objektum létrejöhet. Nélkülözhetetlen

volt még ezen kívül egy olyan segédtábla létrehozása, melyben az tároljuk el, hogy az adott szoba milyen típusú (például folyosó vagy iroda). Ám itt azt is meg kellett jelölni, hogy az ezzel a típussal megjelölt elem átjárható-e, és ha igen, akkor csupán a többi, vele megegyező szinten fellelhető szobák közt lehet rajta keresztül közlekedni, vagy a különböző emeletek között.

A falak és az ajtók tulajdonképpen hasonlók, főleg a funkciójukat tekintve térnek el egymástól; ám a falakhoz szükség volt egy olyan azonosítóra az éppen készülő megjelenítés miatt, ami tudatja velünk, hogy az egy „igazi" fal-e, vagy csupán egy emeleten elhelyezkedő korlát, amitől az átjárhatóságot tekintve fal lesz ugyan, de egyéb szempontokat tekintve különbözik tőle.




5. ábra: az adattároláshoz szükséges adatmodell táblái és kapcsolatuk

Az ajtókat tároló tábla a legszükségesebb információkat tartalmazza: az azonosítókon kívül tehát a két pont koordinátáját, amik között az ajtó található, és egy átjáró-azonosítót. Erre azért

volt szükség, mert minden ajtó egyben átjáró is. Lehet szinten belüli átjáró, két hétköznapi szoba között, ám előfordulhat, hogy épp egy szintek közötti átjárót reprezentáló elemnek az éppen aktuális emeleten elhelyezkedő becsatlakozási pontjához jutunk általa (vagyis épp egy liftajtóról vagy lépcsőház ajtajáról van szó).

A fennmaradó három, tábla, ami a szobákkal és az ajtókkal áll kapcsolatban, arra szolgál, hogy azonosítsuk az átjárók típusát az éppen aktuálisan vizsgált szobák között. Először tehát kellett egy olyan tábla, ahol a térképen található átjáró típusokat tároljuk. Olyan elemeket tartalmazhat, mint például „lift", „lépcső", vagy „szobák közti átjáró" (két egyszinten lévő, egymás melletti szoba esetén), de épülettípustól függően ide vehetjük fel akár a mozgólépcsőket is, ha tartalmaz ilyet az épület. Ezen kívül szükség volt egy táblára, amelyben azt tároljuk, hogy melyik szobák között milyen típusú összeköttetés teremt kapcsolatot, végül pedig kellett egy tábla, amely az eddigiekkel és az egyes ajtókhoz rendeli hozzá.

Az adatbázisba a navigációhoz létrehozott táblákat az 5. ábrán tekinthetjük meg, melyben láthatóak az egyes elemek kapcsolatai és az általuk tartalmazott mezők is.

## 5.2. Algoritmusok leírása, felhasználása

Az adatbázisban létrehozott elemek felhasználásához és kezeléséhez szükség volt olyan algoritmusokra, amelyek segítségével létrehozható magukból az elemekből a komplett gráf, melyen végrehajtható a navigáció. Mivel az adatbázissal szoros kapcsolatban a szerver áll, emellett az eltérő képességű és architektúrális felszereltségű, Android operációs rendszert futtató készülékeknek ugyanazt a szolgáltatást kívánjuk nyújtani (azaz egy gyengébb hardverrel felszerelt eszköz is képes legyen a gyors útvonalválasztásra egy esetleges nagyobb méretű, komplexebb térkép esetén), célszerű ezen algoritmusokat a szerveroldalon implementálni. Ilyenkor csupán a szervert futtató gépnek kell a számításokat elvégezni, a kliens eszköz csupán az adatgyűjtés, a kommunikáció és a felhasználói interakció végett használatos.

### 5.2.1. A gráf építése

A navigációhoz, mint már említettem, szükség volt egy olyan adatmodellre, amivel az épület belterét le tudjuk fedni, hogy azon iterálva eljuthassunk egyik pontból a másikba. Ezen szerepet tölti be a több csomópontból álló, összefüggő gráf.

Magára a gráf építésére a szerveroldalon kerül sor, közvetlenül az után, hogy a térkép-adatok az adatbázis tábláiból lekérdezésre kerültek. Habár az adatbázis strukturális alapja a pont, nem minden pontból lesz egy az egyben csomópont. Egy olyan eltárolt szobán belül ugyanis, ami átjáró típusú (például folyosó), szükség volt több pont felvételére is. Az összeköttetések felépítése szintenként történik, melyeket a szintek közötti átjárók (liftek megállóhelyei,

lépcsőknek az emeletekhez való csatlakozási pontjai) kötnek össze. Legelőször tehát a folyosók „gerinchálózatát” kell felépíteni, mely végighúzódik az egy-egy szinten elterülő folyosókon.

A gráf építésének alapja, hogy minden, a szerveren az adatbázisból felépített szobát sorra kell venni, és szobán belül meghatározni a csomópontokat, majd összekötni azokat. Ha ezzel ugyanis megvagyunk, akkor azon oknál fogva, mivel a szobák egymással szorosan érintkeznek (vagyis egy fal, ha nem épülethatároló, akkor legalább két szobához tartozik), a gráfunk már fel is épült. Ez persze mindössze a csomópontok logikus elhelyezésén múlik, aminek a menete a következő:

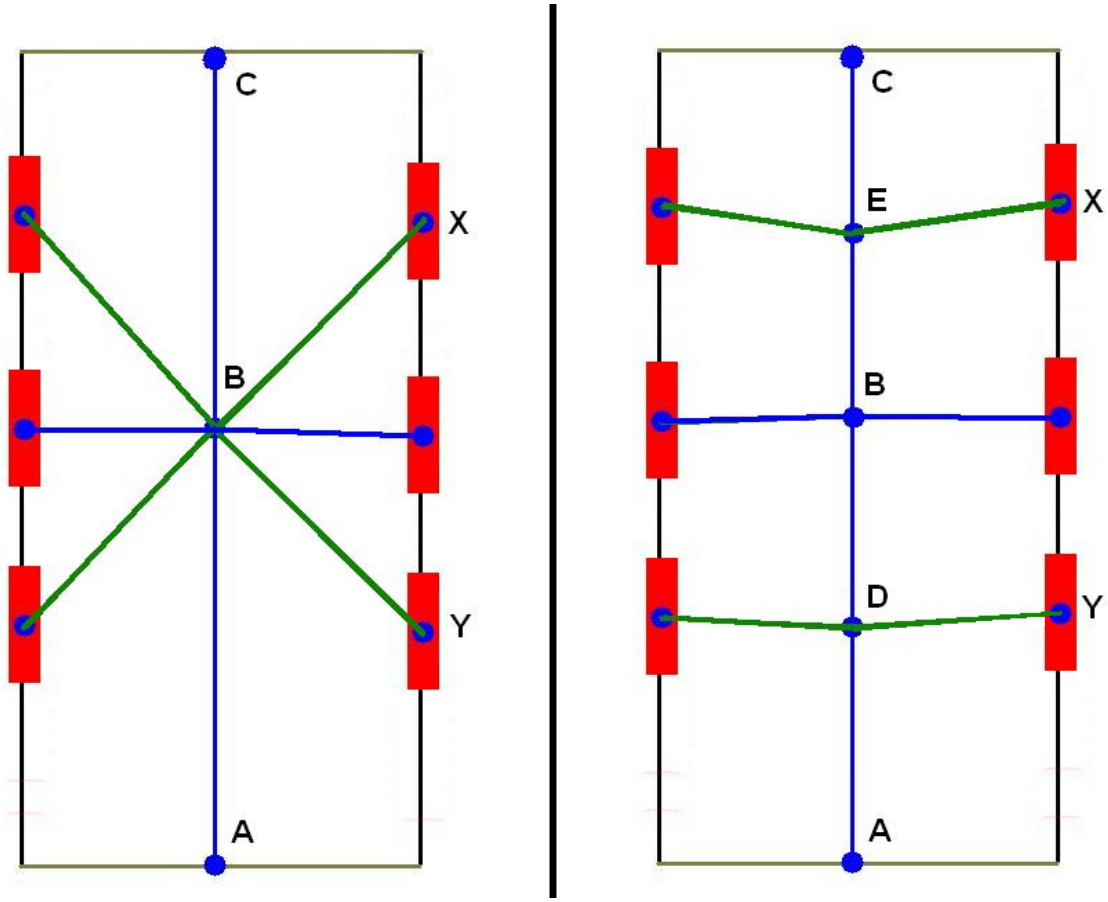
Legelőször, ha megvizsgálunk egy szobát, ismerjük annak a referenciapontját. Ez a pont ugyanis még a szobának a térképszerkesztő program által való elkészítésekor kerül bele az adatbázisba, és meghatározza a szobának úgymond a középpontját. Ez a pont tehát alkalmas arra, hogy gráf csomópont legyen.

Mint már korábban írtam, minden ajtó kétféle lehet. Vannak valódi ajtók, melyeket kinyitva egyik helyiségből átsétálhatunk egy másik helyiségbe. Ezen kívül lehet a szobának azon határoló szakasza, ahol közvetlenül egy másik szobával érintkezik (azaz a valós helyzetben ez a két helység egynek számít, csak a szobák méretének limitálása miatt lett ketté bontva), ez az elem szintén ajtónak számít, csak úgynevezett „virtuális” ajtóként kezeljük őket.

Azon okoknál fogva, hogy a navigáció során célként helyiségek ajtaját adhatjuk meg, minden ilyen elemre kell, hogy egy csomópont illesztve legyen (pontosabban a hosszának középpontjára), hogy azokat a navigációs algoritmus számításba vegye. Ebből adódik, hogy nem csak a valódi ajtókra kerül csomópont, hanem az úgynevezett virtuálisakra is.

Ezzel már majdnem el is készültünk az egy szobán belül elhelyezkedő csomópontok definiálásával. Azonban van még egy fontos tényező, amit figyelembe kell vennünk, mielőtt összekötnénk a csomópontokat. Ha az éleket jelen helyzetben behúznánk, akkor azon a szoba-szakaszon, ami átjáró, és több, tőle eltérő típusú szoba nyílik a belteréből, megvizsgálnánk az egyik végéből két olyan ajtóhoz mért távolságát, melyek a szoba azonos falán helyezkednek el, néhány esetben eltérő adatot kapnánk, ha a gráfon (élek hosszával) mérhető és a valós távolságot hasonlítanánk össze. Ugyanis ha a vizsgált végpont és a hozzá a valóságban legközelebbi, a szobába eső ajtó távolságát vizsgálnánk, akkor az a gráf szempontjából ugyanakkora költségű útra lenne azzal az ajtóval, ami az előzőleg vizsgált ajtóval megegyező falon van, viszont a szoba másik végében helyezkedik el.

Az 6. ábrán egy ilyen esetre láthatunk példát: tegyük fel, hogy az **A** és az **Y** pont távolságát hasonlítjuk össze az **A** és az **X** közötti távolsággal. Értelem szerűen az **Y**-al jelölt pont közelebb van **A**-hoz, mint az **X**-el jelölt pont. Viszont, ha a gráfon mért távolságokat nézzük, a két távolság ezen esetben megegyezik: **A**-ból egy él megy a referenciapontként felhasznált csúcsba (**B** pont), onnan azonos hosszúságú élek futnak **X**-be és **Y**-ba.




6. ábra: a folyosószakasz "gerinchálózatán" elhelyezett csomópontok szerepe

Hogy erre a problémára egy megoldást nyújtsunk, szükség van úgynevezett „köztes” csomópontok elhelyezésére a referenciapontok és a virtuális ajtókon elhelyezett csomópontok között félúton (az ábrán **D**-vel és **E**-vel jelölt pontok). Amint ez a fent megemlített ábra jobboldalán látszik, ez valamelyest megoldást ad a problémára, amit viszont a végtelenségig lehetne fokozni egyre kisebb távolságokra és sokasodó ajtókra. Ez viszont már több szempontból nem jelent problémát: egyrészt az egyben felépíthető, átjárható szoba falhosszának egy ajánlott felső korlátot adtunk, ami értelem szerűen függhet az aktuális elem alakjától, épületen belüli helyzetétől, és attól, hogy hány ajtó található benne (azaz ha például összesen két ajtó van egy, az ajánlott értéknél hosszabb és egyenes folyosószakaszon, akkor természetesen felesleges plusz elemként felosztani több szakaszra). Másrészt nincsen szükségünk egy jelenleg szoba-szinten működő pontosságú helymeghatározó alkalmazás navigációjához aprólékosabb, ezt folytatva már egy idő után centiméter pontosságú távolság meghatározására. Ha ugyanis figyelembe vesszük, hogy egy ajtóméret szélességének minimális értéke 1 méter, és a szoba ajánlott maximális hossza 10 méter, akkor ezen „legrosszabb” esetben a szoba végeitől 5 méterre helyezkedik el a referenciapont, az újonnan felhelyezett köztes csomópontok pedig 2.5 méterre találhatóak a szoba egyik és másik végétől.

Ezen adatokkal kalkulálva és a felsorolt érvek miatt pedig felesleges az ennél részletesebben történő csomópontokkal való lefedése a területnek.

Miután az összes pont sikeresen meghatározásra került, nincs más dolgunk, mint összekötni őket, azaz éleket behúzni közéjük. A folyosó úgynevezett „gerinchálózata" a belőle álló szobákban található referenciapontok, köztes pontok, és a virtuális ajtókon található pontok összekötéséből áll. A folyosókból nyíló szobákat, vagyis azoknak az ajtaját elég a hozzájuk legközelebb található gerinchálózati pontokkal összekötni, aminek minden, sajátjával megegyező szinten átjárhatóságot biztosító szobában való ismétlésével felépült az épületet lefedő végleges gráf. A szintek közötti átjáróknál ugyanis nincs szükség ezen csomópontok külön-külön történő felhelyezésére, ott elég egy csomópontot az ajtóra helyezni, majd azt a szomszéd szinten lévő, vele megegyező átjáró ajtajára tett ponthoz élekkel összekötni.

Egy dolog van még, amit mindenképp szükséges megemlíteni: az élek értékeinek meghatározását, ebből kapjuk meg ugyanis a navigáció által mért távolságokat. Erre a problémára a következő megoldást adtam: mivel minden térkép alapvetően a szerkesztésnél jön létre, ott (a szerkesztőprogramban) megadjuk, hogy milyen X és Y koordinátájú értékekkel szerepeljenek az elemei. Ezekből adódóan az egész térképnek is van egy, koordinátákban mérhető maximális értéke. Mivel tehát minden csomópontnak vannak X és Y koordinátái, ezért definiálhatjuk az ezzel kapcsolatos mérhető euklideszi távolságukat is: az éleket ugyanis egyenes szakaszok alkotják a csomópontok között. Azonban mi a helyzet akkor, ha a cél helyszínnel nem egyező emeleten tartózkodunk? Ilyenkor elképzelhető, hogy a kívánt érkezési pontunk koordinátái megegyeznek a kiindulási koordinátákkal, csak eltérő térképen találhatóak, azaz igénybe kell vennünk valamilyen szintek közötti átjárót az eljutáshoz. Ám ilyenkor nem szükséges (és valószínűleg a térkép-adatokból nem is tudnánk) pontos értékeket használni, a lényeg csupán annyi, hogy minden átjárónál minden szomszédos szint között húzódó él hossza egységes legyen. Ezen okoknál fogva alkalmaztam alapértékként minden ilyen típusú átjárónál a szintek közötti élek hosszaként 100 egységet. Visszatérve az előző példához, ha a célunk nem ugyanazon az emeleten helyezkedik el, mint ahol mi tartózkodunk, akkor a szintek közötti átjárókig az élsúlyok az X és Y koordinátákból számolt euklideszi távolságból adódnak, ezek után minden egyes emeletváltásnál egy 100 egység-költségű utat teszünk meg, majd pedig a célt tartalmazó szintre érve újból a koordinátákból adódnak az élköltségek.

Két tetszőleges csomópont távolsága a navigáció szempontjából tehát alapvetően ezekből az adatokból, az egész út hossza pedig a kiválasztott csomópontok közötti élek súlyainak összegéből adódik.

### 5.2.2. Navigációs algoritmusok

Ahhoz, hogy a gráf alapú navigáció működhessen, szükség volt olyan algoritmusokra, melyek alapján a megadott helyszínek között az optimális út megtalálható. Erre két, az irodalomban beltéri navigációhoz útkeresésre használt útkereső algoritmust használtam fel: a Dijkstra-algoritmust és az A* algoritmus beltéri navigációra átalakított, háromdimenziós változatát, melyeket a szerveroldalon implementáltam, mivel magára az útvonalválasztásra ott került sor.

#### 5.2.2.1. A Dijkstra-algoritmus

Elsőként azt az útkereső algoritmust valósítottam meg, ami mind kül- és beltéri navigációs használat során a leggyakrabban alkalmazott: a Dijkstra-algoritmust, melynek működési elve a következő:

Adott egy **G(V,E)** súlyozott, irányított gráf, és egy **S**-el jelölt csúcs a gráfon. Ez a csúcs a kiindulási pontunk. A mi feladatunk, hogy ebből a kezdőpontból a legrövidebb utat megtaláljuk az összes többi csomópontba, ezen gráfon belül. Úthossz alatt a csúcsok közötti élek súlyainak összegét értjük. Jelöljük most az éppen aktuálisan vizsgált csomópontot **Q**-val, az **S** és **Q** csúcsok legrövidebbnek ítélt távolságát ***d(S,Q)***-el.

Az algoritmus futása során a **G** gráf minden egyes csúcspontjára nyilvántartja az **S**-el jelölt kiindulási pont és a **Q**-val jelölt csúcs közötti, a futás során eddig a legrövidebbnek talált út költségét. Indulásakor legelőször a kiindulási csúcs saját magától mért távolságát vizsgálja, ami értelem szerűen 0, az összes többi csomópont távolsága végtelen. Ez reprezentálja azt a tényt, hogy a gráfban még egyetlen utat sem ismerünk. Maga az algoritmus két ponthalmazzal dolgozik: az egyikben tárolja azokat a csúcsokat (jelöljük most **A**-val), melyekre ***d(S,Q)*** értéke már a legrövidebb utat adja meg, a másikban (jelöljük **B**-vel) pedig a gráf többi pontját. Kezdetben az **A** halmaz üres, **B** halmaz pedig tartalmazza az összes csúcsot. Az algoritmus során minden egyes iterációnál átkerül egy csomópont a **B**-ből az **A**-val jelölt halmazba. Amikor a **Q**-csomópont átkerül **B**-ből **A**-ba, az algoritmus megvizsgálja, hogy az újonnan bővített halmazban az áthelyezett elemhez vezető, aktuálisan vizsgált út hosszához az eddig arra a csúcsra kapott érték hogyan viszonyul.

Első iterációként tehát megvizsgálja az **S** csúcsból elérhető szomszédos csúcsokat: a vizsgáltak közül a legkisebb költségű úton elérhetőt teszi be az **A** halmazba, és kiértékeli a csúcs többi szomszédjáig vezető költségeket is, amiket értelem szerűen az eddig mért legrövidebb úton találta meg. Ezután folytatja a kiindulási csúcsnál: a második legkisebb költségű úton elérhető szomszédját teszi be az **A** halmazba, a hozzá vezető költséggel, ami jelenleg még a szomszédság miatt nem egy kiszámítandó összeg. Majd megvizsgálja ezen csúcs szomszédait is, kiértékeli a hozzájuk vezető út súlyát, ám ha ezek közé esik egy olyan csúcs is, amit az előző lépésben már megvizsgált, és az a költség, amit az előző lépésben

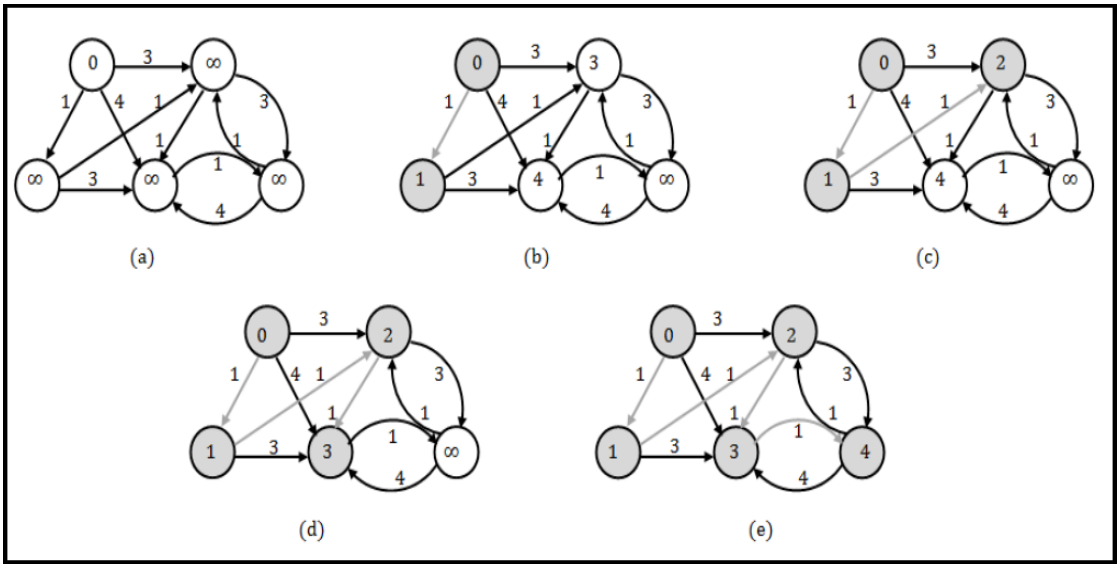
kapott a csúcsig, kevesebb volt a jelenlegi csúcstól mért, és a kiindulási csomóponttól kapott súlynak az összegénél, akkor az összeg nem kerül felülírásra, hiszen az eddig talált odáig vezető út kevesebb költségű volt, mint a jelenlegi iterációban mért.

Ezek után az algoritmus halad tovább, minden egyes iterációnál az eddigi értékekhez hasonlít, és csak akkor frissíti őket, ha a jelenlegi iterációban kapott érték kevesebb. Így tehát az élek egyenkénti hozzávételével kapjuk **G** gráf egy **F** részgráfját, melyben minden **Q** ponthoz tartozik egy **F**-beli irányított út, ami a legrövidebb a kiindulási ponttól, a részgráf pedig fokozatosan konvergál az eredeti gráfunkhoz, melyet végül lefed.

Megvalósítás terén az **A**-val jelölt halmazként egy prioritásos sort használtam, ahol a prioritást a kiindulási csúcstól való távolság reprezentálta. Erre azért volt szükség, mert így mindig az első elem kerülhetett kiértékelésre, tehát az az elem, amelyiknek aktuálisan a legkisebb a távolsága a kezdőponttól.

Ezt az algoritmust akkor tudjuk minimális útkeresésre használni, ha a gráfban nem szerepelnek negatív élek, ugyanis a negatív élsúlyok a pozitívakkal összegezve sok esetben kisebb értéket adnának, mint az olyan utak, ahol csak pozitív súlyú élek találhatóak, és ezzel így már nem lehetne optimális úthosszt keresni (ezen oknál fogva kell a negatív élsúlyokkal is rendelkező gráfokra a Bellman-Ford algoritmust alkalmazni, amire viszont jelenleg nincs szükség). Mohó algoritmus, azaz mindig a lokális optimumok választásával próbálja elérni a globális optimumot. Mivel a vizsgálandó gráfnak n csomópontja van, és az algoritmus működésének alapja egy *while* és egy *for* ciklus, az elsőt n-szer, a másodikat legfeljebb (n-1)-szer végezzük el, így a bonyolultságot $O(n^2)$-ben állapíthatjuk meg, mivel n(n-1) másodfokú.[**13**]

A 7. ábrán jól látható az algoritmus működése, mely egy 5 csomópontból álló gráfon került bemutatásra.




**7. ábra: A Dijkstra-algoritmus működése [14]**

#### 5.2.2.2. Az A* algoritmus

Ennél az algoritmusnál szintén az a lényeg, hogy egy kiindulási helyről a lehető legkisebb költségű úton a célba eljussunk. Ám ez a módszer az eddig megtett úthossz figyelembe vétele mellett egy úgynevezett heurisztikus függvényt is használ működése során ahhoz, hogy meghatározza a sorrendet, ami alapján az egyes csomópontokba eljut. A költségfüggvény (ami alapján az algoritmus meghatározza a következő lépését) tehát két függvényből áll: az egyik megmondja kiindulási hely és az aktuális pozíció közötti útvonalköltséget (jelöljük **g(x)-**el), a másik pedig egy „heurisztikus becslést" ad a céltól való távolságunkra (legyen most **h(x)**), ezt nevezzük heurisztikus függvénynek. Maga a költségfüggvény tulajdonképpen az előző két függvény összege, azaz: **f(x)=g(x)+h(x)**. A lényeg, hogy **h(x)** egy elfogadható heurisztika legyen, azaz sosem szabad túlbecsülnie a jelen helyzettől a célig elérhető legrövidebb utat. Sok útvonalkereső alkalmazásnál ez a függvény a két pont közötti egyenes úton mért euklideszi távolságot jelöli, ennél ugyanis ez a fizikailag elérhető legrövidebb út. Jelen helyzetben azonban ennek egy kissé módosított változatára van szükség: egy épületben több szinten találhatóak helyszínek, azonban a tényleges távolságot ezzel a rendszerrel csak két dimenzióban tudjuk mérni. Ahhoz tehát, hogy egy használható heurisztika-függvényt megadhassunk, szükség volt az adatmodellben egy harmadik dimenzió-egység felvételére is. Konkrét információ tehát nem áll rendelkezésünkre az épület belterének magasságáról, azonban annyit tudunk, hogy értelem szerűen egy építményen belül minden szinten azonos ez az érték. A lényeg az, hogy két szomszédos szintnek a harmadik dimenzióban mért távolsága egységes legyen minden, azokon a rétegeken található pont között. Ez azonban magával vonja azt, hogy nem csak a heurisztika függvénynél kell ezeket az értékeket használni ahhoz, hogy használható rendszert kapjunk. A csomópontok között, mint már szó volt róla, élek teremtenek kapcsolatot. Ezek az élek nem csak síkban helyezkednek el, hanem az emeletek közötti átjárók között is. Ezek azonban szintén két szomszédos szint magasságát határozzák meg, és mivel a fent említett élek már rendelkeznek egy 100 egység költségű súllyal, így akkor kapjuk a megfelelő megoldást a heurisztika számításakor, ha a harmadik dimenzióként két egymás alatt elhelyezkedő szint pontjai között is mindenhol ezt az értéket vesszük.

Működését tekintve az A* algoritmus két listával dolgozik: egy „nyílt" és egy „zárt" listával. A nyílt listába teszi azokat a csomópontokat, amelyek a számunkra optimális útba lehet, hogy bevehetőek, azaz ezeket a csomópontokat kell még megvizsgálni. A zárt listába kerülnek azok a pontok, melyeket már megvizsgált és kiértékelt.

Az algoritmus indulásakor a kiindulási csomópontot adjuk a nyílt listához. Ezek után azokat, amelyeket innen közvetlenül el lehet érni: ezen csomópontoknak megjelöljük (és ezek után minden hasonló esetben), hogy melyik pont volt a „szülője", azaz melyik csúcsból jutottunk el közvetlenül, a kiindulásitól legrövidebb úton oda, ami természetesen ilyenkor a kiindulási

csomópont. Ezek után a kezdeti helyszínt kivehetjük a nyílt listából, és beletesszük a zártba. A most következő lépésben tulajdonképpen az előzőt kell ismételni, azaz a nyílt listában szereplő elemek közül választani egyet, és azoknak kell a szomszédjait vizsgálni: ilyenkor vesszük figyelembe a heurisztikát, azaz azt választjuk, amelyikre az **f(x)** értéke a legkisebb. Amikor kiválasztottuk, és kiértékeltük a szomszédjait is, ezt az elemet is hozzáadjuk a zárt listához. A szomszédok kiértékelésénél figyelnünk kell: akkor kell csak a szomszéd elem értékeit felülírnunk, ha az ebből a csomópontból történő megközelítésük rövidebb utat eredményez a kiindulási helyről nézve, mint az eddigi, azaz ha **g(x)** értéke ebből az elemből vizsgálva kisebb lesz. Ha ez az eset áll fenn, az új helyszín **f(x)** és **g(x)** értékeit, és a szülő csomópont jelölést is frissítenünk kell.

Megvalósítást tekintve a nyílt lista egy prioritásos sorként került megvalósításra, ahol a prioritást a minél kisebb **f(x)**-érték jelentette. Ez amiatt hasznos, mert így mindig az a csomópont kerül vizsgálatra, amelynek a kezdőponttól való költségének és a célig becsült távolságának összege a legkisebb (azaz feltételezhetően a legrövidebb út a két pont között).

Sok útkereső megoldás ezt az algoritmust a bejárandó terület négyzethálóval való lefedésével, és az egyes négyzet-egységek közti lépésekkel, mint út-szakaszokkal alkalmazza, ám gráfon történő navigálásnál annyiban könnyebb a dolgunk, mint a fent említett változatú útkeresésnél (ahol gyakorlatilag minden négyzet egy csomópont), hogy míg ott csak olyan elemeket szabad figyelembe venni, amik járhatóak (például lehet elem egy fal is), addig a gráfon csak az összeköttetéseket kell figyelni, tehát gyakorlatilag minden csomópont bejárható.

Komplexitását tekintve az algoritmus függ a megadott heurisztikától, az aktuális gráf felépítésétől és méretétől is: O(log(n)) és O(exp(n)) lehet legjobb és legrosszabb esetet tekintve, de a legtöbbször ezek közötti értéket kapunk. Legjobb esetnek azt tekintjük, amikor a heurisztika pontosan eltalálja a hátralévő bejárandó legrövidebb utat, ám a jelenlegi ismereteim szerint nincs olyan heurisztika, ami egy tetszőleges gráfban tetszőleges csomópontok között már előre a legrövidebb utat adja a még megteendő távolságra.

Mindent összevetve ez a módszer és az előzőleg bemutatott Dijkstra-algoritmus lényegében abban tér el egymástól, hogy míg ott egy kiválasztott csomópontból a legrövidebb út szempontjából minden pont vizsgálatra kerül (a célt ezek után választjuk ki a már kiértékelésre került, összes csomópontot tartalmazó halmazból), addig itt egyszerre adunk meg kiindulási és cél node-ot, és azok között a heurisztika segítségével vizsgálódik az algoritmus. Ez annyit jelent, hogy bár nem csak közvetlenül a két helyszín közötti csomópontokat értékeli ki, de korántsem az összeset. Ugyanakkor azt is figyelembe kell vennünk, hogy az algoritmus erősen függ a megadott heurisztikától, ahol a kikötés mindössze annyi, hogy nem szabad túlbecsülnie a biztosan megtételre kerülő út hosszát.

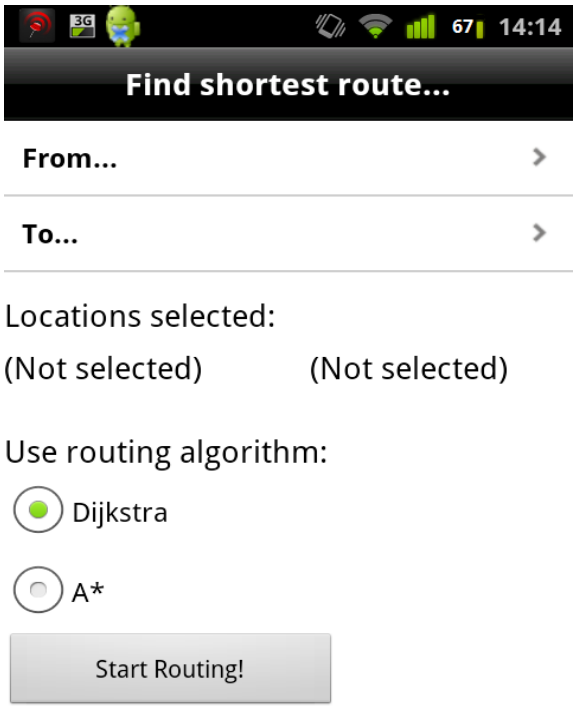
## 5.3. Módosítások a kliens oldalon

Mint ahogy sok navigációs szolgáltatás, a jelenlegi is elkülöníthető kliens- és szerveroldali elemekre, ahol a kliensoldal az, ami kommunikál a felhasználóval. Ez tehát az a része a programnak, ahol a legkönnyebb vizualizálni az eddig elkészített részben ejtett különféle hibákat. Mivel még nem teljesen végleges navigációs alkalmazásról van szó, hanem a módszernek a rendszerbe történő beépítéséről és teszteléséről, ráadásul a projektmunka legvégén még a különböző fejlesztési ágak egymásba integrálása és az azok után felmerülő hibák javítása is hátra van, ezért legelőször olyan kliensoldali átalakításokra van szükség, ami megmutatja a felmerülő esetleges hibákat az elkészített rendszerben, ugyanakkor ne legyen szükséges az összes módosítást az integrációkor eldobni. Ezen okoknál fogva a következő fejlesztéseket hajtottam végre az alapprogram kliensoldalának Android operációs rendszeren futtatható alkalmazásán:

Először is felvettem egy új, navigációs menüpontot, amit a program indulásakor megjeleníthető főmenüből kiválasztva egy új felületet kapunk, melynek segítségével kiválaszthatjuk, hogy melyik helyszínről hova akarunk eljutni, melyik algoritmus felhasználásával. A kiindulási („From” nevű) és a cél („To”-ként elnevezett) mezőt egy-egy, a szervertől egy listát lekérő menüelem felhasználásával adhatjuk meg: amint rákattintunk például a „From”-mal jelölt felületelemre, az egy kérést indít a szerver felé, amiben megjelöli, hogy ő az adatbázisban található összes helyszínt le akarja kérni, hogy azok közül kiindulási helyszínt válasszon. Válaszként a szervertől megkapja az összes helyszínt, vagyis egy olyan listát, ami gráf csomópontokat tartalmaz. Minden csomópontnak van azonosítója, emberi szemmel is értelmezhető neve (pl. Bejárati ajtó), és tartalmazza azt az információt, ami magát helyszínt az épületen belül azonosítja (X és Y koordináták, szint-azonosító). Ezen információk segítségével tehát pontosan definiálhatunk egy épületen belüli, tetszőleges pontot.

Miután sikeresen kiválasztottuk a kiindulási helyünket és az utunk céljaként megjelölt helyszínt, a navigációs felületen úgynevezett „rádiógombok” segítségével választhatjuk ki, hogy a szerver milyen algoritmus szerint határozza meg számunkra a bejárandó útvonalat. A két, fent megjelölt algoritmus közül választhatunk: Dijkstra-algoritmust, vagy A* algoritmust. Azért rádiógombokat alkalmaztam erre a célra, mert számomra az volt a lényeg, hogy viszonylag kevés választási lehetőségből gyorsan választhasson az ember, míg a kiindulási és célhelyszínnél indokoltabb volt egy új felületen megjelenítendő lista használata, mert ott az egy épületen belül kiválasztható helyszínek száma jelentősen több, mint amit a képernyőn egyszerre meg tudunk jeleníteni.

Amint kiválasztottuk a számunkra érdekes kiindulási, és célhelyszínt, majd az algoritmust is, ezt a három dolgot továbbítjuk a szerver felé; azaz ilyenkor a szerveroldal fogad tőlünk két helyszín-azonosítót, és egy algoritmus-azonosítót.




8. ábra: a navigáció teszteléséhez elkészített fő felület

Miután a szerver kiszámította a számunkra szükséges adatokat, visszaküldi a kliensoldalnak. Azaz a válasz, a bejárandó út, vagyis azon helyszínek listája, amelyeknek az egymás utáni meglátogatásával eljuthatunk a célba, mindezt a lehető legrövidebb út megtételével. Amit kapunk a szervertől tehát, az egy olyan lista, ami sorrendben tartalmazza az általunk bejárandó helyszíneket (vagyis azokat a gráf-csomópont objektumokat, amikre nekünk szükségünk van), nekünk csak annyi a dolgunk, hogy az elejétől a végéig végigiterálunk rajta.

Amikor tehát a kliens oldalon a navigációs felületen megnyomjuk a legrövidebb út keresése („Start routing”-al jelölt) gombot, egy új felületre visz minket el a program, amin láthatjuk listaként felsorolva az általunk bejárandó útvonalat, melyet a szervertől újonnan fogadott csomópont-lista alapján fog nekünk a kliens a képernyőre kiírni. Mivel minden listaelem egy csomópont-objektum, ezért tartalmaz minden szükséges információt, ami arra szolgál, hogy egy épületen belül merre található. Végül tehát ezek azok az elemek, amelyeket a később integrálásra kerülő megjelenítési réteg megkap, csupán annyi lesz a dolga, hogy kinyeri a szükséges információkat az objektumokból, és azok alapján rajzolja ki az utat a térképre.

## 5.4. A kliens-szerver kapcsolat

Ahhoz, hogy a navigációs megoldást a meglévő programba beépítve tesztelni tudjam, szükség volt arra, hogy a navigációs kérés-válaszokat a szerver és a kliens kezelni tudja. Ehhez azonban nélkülözhetetlen volt az eddigi kapcsolat vizsgálata, hogy annak segítségével az én programrészem is kommunikálni tudjon a szerverrel, és annak a navigációt működtető részével.

Maga az alapprogram többféle mobilplatformon (Android, iOS, Symbian, J2ME) futtatható. Ehhez azonban egy olyan módszert kellett implementálni a kliens-szerver közötti kommunikációra, melyet mindegyik platform támogat. Tehát ugyanúgy kell tudnia egy JAVA programozási nyelven implementált szerver-egységnek fogadnia és kezelnie több kliens-alkalmazás adatait, még akkor is, ha ezek a példányok egyidőben különböző platformokon futnak.

Erre a program készítői egy elég jól használható megoldást találtak: a kliens-szerver közti kommunikációt JSON (JavaScipt Object Notation) objektumok segítségével oldották meg. A JSON lényege a következő: van egy tetszőleges (típusú, mennyiségű, méretű) adatunk, amit el szeretnénk juttatni a klienstől a szervernek. Ezt az adatot először is JSON-formátumra kell átalakítanunk, vagyis szerializálnunk kell. A formátum tulajdonképpen emberi szemmel is olvasható név-érték párokat takar, amelyek egy listába vannak belesűrítve. A lista kezdetét és végét egy-egy kapcsos zárójel jelzi, és ezen belül helyezkedik el az átküldeni kívánt adatunk. Miután az adatunkat szerializáltuk a megfelelő módon, átküldjük a szerver számára, mely az adatunkat fogadja, és deszerializálja. Miután tehát a szerializált objektumunk megérkezett, legelőször szükség van annak a megállapítására, hogy átalakítás előtt milyen típusú volt az objektumunk.

A legegyszerűbben primitív típusokat tudunk ilyen módon átküldeni, hiszen ezek szinte kivétel nélkül minden programozási nyelvben megtalálhatóak. Ám visszatérve a jelen helyzetünkhöz, az alapprogram olyan információkat akarhat átjuttatni a szerverhez, melyre természetesen a megfelelő típusú választ várja. Ahhoz, hogy ez megtörténhessen, olyan segéd-osztályokat kell mind szerver- és kliensoldalon implementálni, melyek segítségével megtörténhet a sikeres át- és visszaalakítás.

Hogy a programmal a navigációs módszereimet tesztelni tudjam, szükségem volt legelőször arra, hogy általam átküldendő adatok át- és visszaalakítását támogató segédosztályokat mindkét oldalon implementáljam.

A kliens-szerver közti kommunikációra a láncolt lista adatszerkezetet használtam fel: ez a JAVA nyelvben egy már előre implementált osztály, használata könnyű és lehetőséget biztosít arra, hogy ha az ember egyszerre több, azonos osztályhoz tartozó objektumot akar egyben kezelni, ezt könnyen megtehesse. Ez az én szempontomból amiatt hasznos, mert mint ahogy

már szó volt róla, a kommunikáció négy fő navigációs fázisra különíthető el: összes helyszín lekérdezése, ezek fogadása, helyszínek és algoritmus választása (és egyben ezeknek a szerver felé továbbítása), majd a legrövidebb út fogadása a szervertől csomópont sorozat formájában; ezen lépések mindegyike olyan információ küldéséről és fogadásáról szól, amely több rész-információt több objektumot tartalmaz.

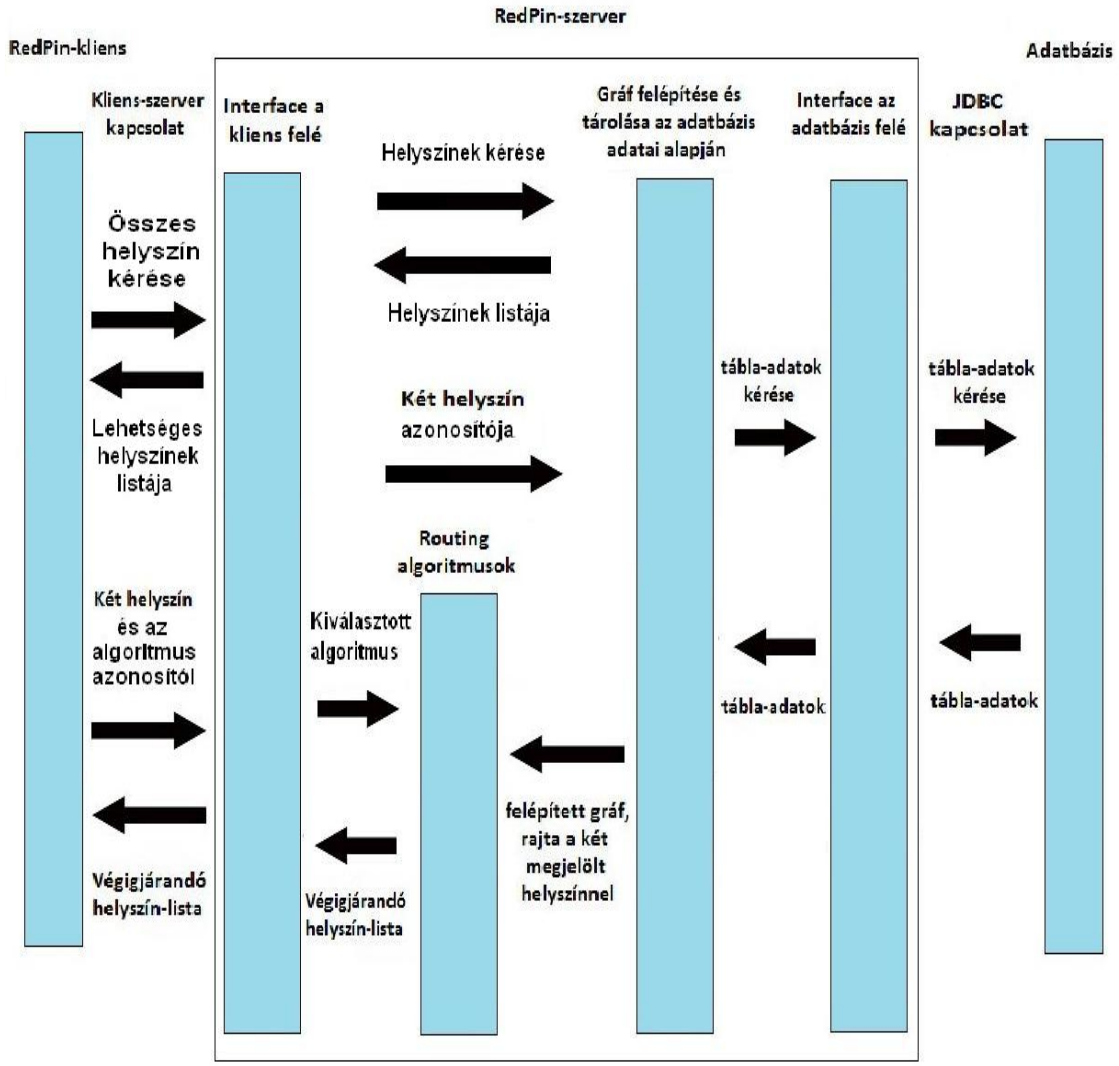
Visszatérve tehát a szerializációhoz, a legcélszerűbb az volt, hogy ezeket a küldendő és fogadandó információkat fázisonként egyetlen objektumba illesszem, amelyet egy lépésben továbbítani lehet a másik fél felé. Emiatt mindkét oldalon implementálni kellett a láncolt lista JSON-objektummá átalakító, majd abból visszaalakításra képes segédosztályát.

A kommunikáció tehát a következőképpen zajlott a kliens és a szerver között: a navigációs lépés első fázisaként szükség volt az összes helyszín lekérdezésére. Itt meg kell említenem, hogy saját konvenciók miatt, mivel a későbbiekben háromtagú láncolt listákkal kommunikálok a szerver felé, az összes helyszín lekérdezésénél is egy háromtagú láncolt listát küldtem. Ez lényegében egy saját magam részére meghatározott konvenció: a szervernek mindig egy háromtagú, Integereket magába foglaló listát küldök, míg válaszként egy, a szerver által meghatározott hosszú, csomópont objektumokat tartalmazó listát kapok. Ez azért is lehet hasznos, mert így egyetlen objektumot hozok létre, amit a szerver számára elküldök, és ennek mindig csak az értékeit változtatom. Ennek a legelső kérésnek viszont, amikor az összes helyszínre szükségem van, a lista tartalma teljesen tetszőleges lehetne, hiszen csak annyi a lényeg, hogy a szerver meg tudja különböztetni a többi kéréstől: „Nekem most minden csomópontra szükségem van!”.

Mivel a válaszként kapott, csomópontokat tartalmazó lista elemeire a kommunikáció során a minden csomópontot egyénileg meghatározó ID-je alapján hivatkozok (ami nulla, vagy egy pozitív egész szám), az összes helyszín lekérésekor egy negatív számot adok értékül a háromtagú lista mindhárom elemének, mivel az biztos, hogy nem lehet csomópont ID. Végeredményként tehát az első fázisban a szerver felé küldött elem információtartalmát három darab, -5 értékű Integerként határoztam meg, amely értékeket a túloldalon megvizsgál deszerializálás után, és a kapott értékek alapján dönti el, hogy mi legyen a válaszreakció.

Amikor tehát értékvizsgálat során egy háromelemű, elemenként -5 értékeket tartalmazó listát kap, lekérdezi az adatbázis navigációs tábláinak a tartalmát, felépíti belőle az objektumokat, és a csomópontokat beleteszi szintén egy láncolt listába, amit szerializálva visszaküld a kliens felé. A kliens ezt az információt fogadja, deszerializálja, és kilistázza a képernyőre a választható helyszíneket, azaz a csomópontok neveit. Ezekből választhatunk egy kiindulási és egy célhelyszínt, és a megadott módon egy algoritmust, amiket ezek után eljuttat a szervernek: a két kiválasztott helyszínnek a csomópont ID-ját, az algoritmus kiválasztásánál pedig egy 1-est, vagy 2-est tesz a listába a választott módszertől függően. Miután ezeket visszaküldte a szerver számára, az a meghatározott algoritmust lefuttatja a gráfon a két pont között, az

eredményt pedig visszaküldi a kliensnek, ahol megtekinthetjük az eredményt. A 9. ábrán látható az egész általam készített rendszer adatfolyam modellje, mely mutatja, hogy milyen főbb állomásokon halad az információ a kliens, a szerver és az adatbázis között.




9. ábra: a navigációt érintő adatfolyam modellje

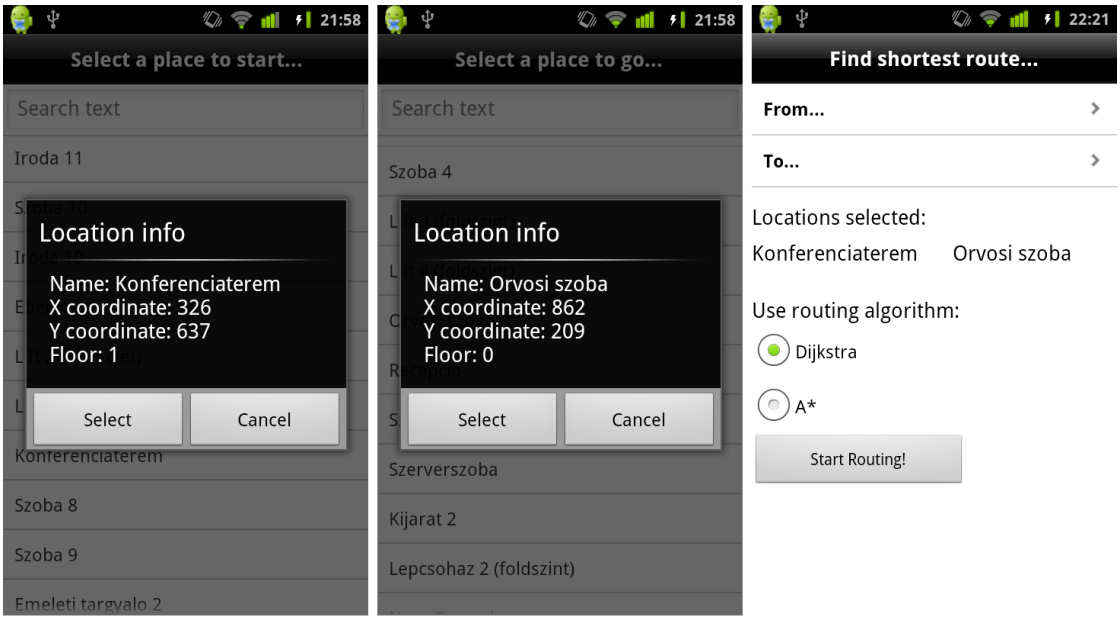
# 6. Tesztelés és eredmények

Miután minden elem elkészítésre került, szükséges volt tesztelni a rendszer működését. Ehhez egy kétszintes tesztépületet töltöttem fel az adatbázisba, mely alaprajza a 10. ábrán látható. A felső képen a földszint, az alsó képen pedig az első emelet található. Az adatbázisban ezek az adatok a fent említett szerkezeti módon (pontok, szobák, ajtók, stb…) kerültek tárolásra.


**10. ábra: a kétszintes tesztépület alaprajza**

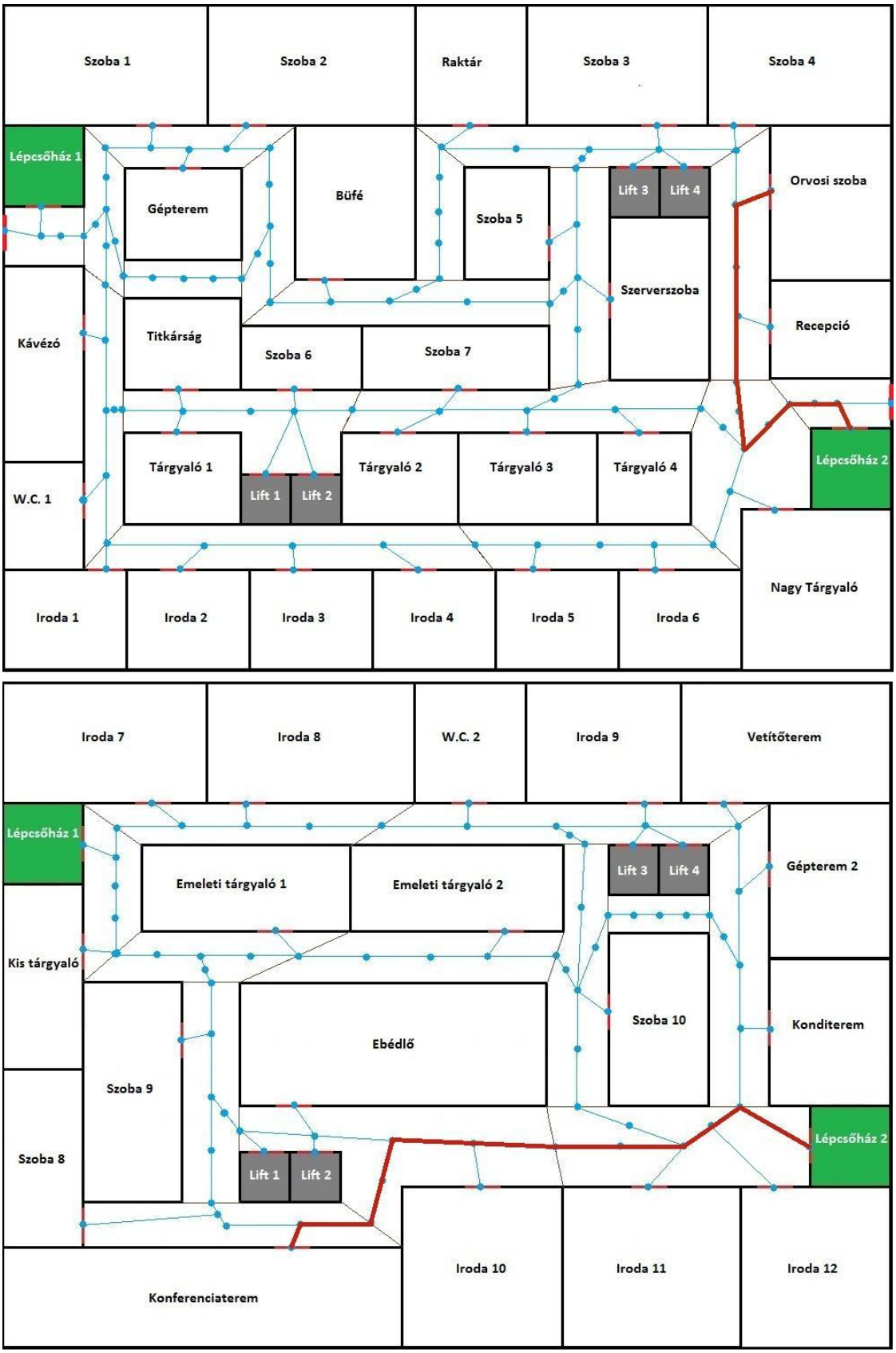
A működés legteljesebb tesztjeként kiválasztottam két, különböző emeleten található helyszínt kiindulási és célpontként, majd ezek között végeztettem útkeresést mind a Dijkstra, mind pedig az A* algoritmussal.

Tegyük fel tehát, hogy az emeleti Konferenciateremből szeretnénk eljutni a földszinten található Orvosi szobába. Ehhez a navigációs főfelületen ki kell válasszuk ezen két pontot, majd a használni kívánt útkereső algoritmust.

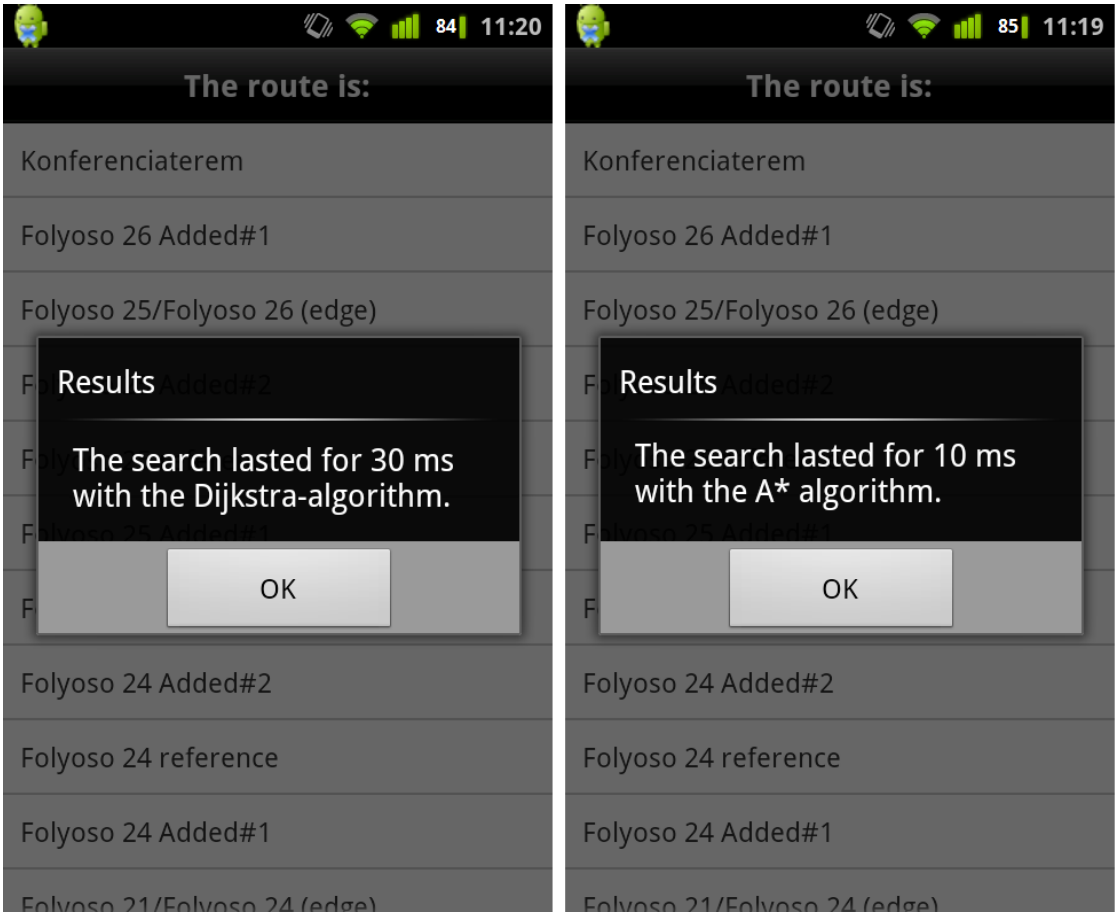



11. ábra: az útkeresés során kiválasztott helyszínek

A szerveren futó gráfépítő algoritmusok az adatbázisban található adatokból a kliens navigációs kérésére felépítettek egy gráfot, melyen az alkalmazott gráfalgoritmus a két kiválasztott csomópont között útkeresést végzett. A felépített gráf és az algoritmusok által talált legrövidebb út szemléltetése a 12.ábrán látható.




**12. ábra: a gráf, és a helyszínek között az algoritmusok által kiválasztott legrövidebb út**




13. ábra: az algoritmusok átlagos futási ideje a jelen épületben

Futási időt tekintve az A* minden esetben gyorsabb volt, mint a Dijkstra, ami viszont várható volt, hiszen míg az utóbbi algoritmus először minden csomópont között meghatározza a legrövidebb utat, addig az előbbi csupán a gráf egy részét járja végig az eredmény érdekében. Ez a tény ennél az épületnél átlagosan 3-szoros futási idő különbséget eredményezett. Ám hozzá kell tegyem, hogy ezek ezredmásodperces értékek voltak, és ezen térképen való navigálás során egyik esetben sem volt egyik algoritmus futási ideje sem több, mint 70 ms. Lényegét tekintve pont ez az az ok, ami miatt a szerver végzi a navigációs számításokat, ugyanúgy, mint ahogy a már elkészítésre került kül- és beltéri navigációs rendszerek esetében: mivel ez egy, a klienseknél sokkal nagyobb számítási kapacitású egység, akármilyen ésszerű esetről is legyen szó, az útkeresés legrosszabb esetben sem tart 2-3 másodpercnél tovább (és ezzel egy elég magas felső korlátot adtunk). Ez tehát az ok, ami miatt előszeretettel alkalmazzák a navigáció során a Dijkstra-algoritmust: általa biztos, hogy az adatmodellen elérhető legrövidebb utat kapjuk, felhasználói szemmel nézve pedig a válasz információ is gyorsan került kiszámításra.

Teszteléskor mindkét algoritmus ugyanazt az útvonalat adta, ami azt jelenti, hogy az A* -hoz használt heurisztika (ami a csomópontok között található háromdimenziós légvonalbeli

távolságot alkalmazza, harmadik dimenzióként egységes szintek közötti távolsággal) ezen esetben jónak bizonyult. Mivel pedig, ahogy említettem, a Dijkstra-algoritmus által biztos, hogy mindig a legrövidebb utat kapjuk, az A*-al pedig ugyanezzel az útvonallal tért vissza, ezért látszik, hogy minkét esetben a jó megoldást kaptuk meg.

Magáról az útkeresésről elmondható, hogy az épületet tekintve tényleg a legrövidebb utat találta meg, még ha az nagyon kis értékkel tér is el a többi, kiindulási ponttól a célpontig megtehető, esetlegesen legrövidebbnek vélhető utaktól. Mivel a távolságok erősen függenek az aktuálisan vizsgált épületben elhelyezett csomópontok helyzetétől, előfordulhat, hogy a kiadott út során a már előttünk lévő (esetleg köztes) célpont esetén is egy kisebb kanyart ír le a meghatározott út az útvonal (mint jelen példánál az emeleten található, „Lépcsőház 2”-vel jelölt helyiség előtti kanyar). Azonban az eredmény még így is tökéletes, mivel nekünk nem centiméter pontossággal bejárandó útvonalra van szükségünk, hanem (lévén a helymeghatározás is szoba szinten pontos) az általa kiadott útvonalnak a szobán áthaladó irányát és a szobából kivezető csatlakozási pontjait kell figyelembe vennünk. Más szavakkal élve: akkor működik jól a rendszer, ha úgy ad legrövidebb utat, hogy minden szobára igaz, hogy egy helység területéről sem lép ki a meghatározott útvonal úgy, hogy kilépés után ugyanazon helység területére visszatérve haladjon tovább. Ezen feltételt pedig a rendszer teljesíti, így bátran mondhatjuk, hogy az alapprogram koncepcióinak rendkívül megfelel a navigációs rendszer.

Végül tehát kijelenthetjük, hogy mindkét módszer tökéletesen alkalmas az útvonalválasztásra, nem hiába alkalmazzák előszeretettel ez eddig elkészült implementációk során. Mivel az A* algoritmus függ a heurisztikától, ezért értelem szerűen a függvény célszerű megválasztásával javulhat, vagy éppen romolhat a teljesítmény, bár ez a gráf felépítésétől és a rajta kiválasztott két csomópont helyzetétől is függ. A program azon irányú továbbfejlesztése esetén például, amikor egy egymáshoz közel elhelyezkedő, több épületből álló épületkomplexumra akarjuk alkalmazni a navigációt, a jelenleg használt heurisztika már nem alkalmazható, hiszen ott számításba kell venni az épületek közti távolságot is. Ezzel szemben a Dijkstra-algoritmus nem használ heurisztikus függvényt, tehát biztos, hogy mindig optimális utat ad. A nagy előnye viszont, mint már említettem, az az A*-nak, hogy nem kell az összes csomóponton végigiterálnia ahhoz, hogy kiadja az útvonalat. Egy rendkívül nagy belterületű épületnél (esetleg épületkomplexumnál, mint például egy egyetemi campus) ez igen hasznos, ugyanis lecsökkenhet a futási idő.

Mindent összevetve tehát sikerült egy olyan rendszert alkotni a szakirodalomból szerzett tapasztalatok alapján, ami tökéletesen alkalmas a beltéri navigációra. Habár több, részben eltérő megvalósítási javaslat született a témával kapcsolatban, az én megoldásom minden általam ismert, beltéri navigációval kapcsolatos specifikáció kielégítése mellett teljesíti a kiindulási program létrejöttének és működésének feltételeit is. Használat szempontjából

pontos, ugyanis a fent leírt szempontokat teljesíti, az épületben célként megadható helységeket mind figyelembe veszi, és képes az implementált algoritmusok segítségével a kiválasztott helyszínek között egy, a felhasználó által bejárható legrövidebb utat megadni. Ezen programot tekintve tehát megvalósításra került egy rendkívül jól alkalmazható, az átlagos felhasználó birtokában lévő mobil eszközzel használható beltéri navigációs rendszer, mely a klienst futtató eszköznek egy bizonyos kereteken belül mért hardveres felszereltségétől függetlenül képes a gyors útvonalválasztásra.

# 7. A rendszer továbbfejlesztésének lehetőségei

Mint láthattuk, az eddig implementált rendszer tökéletesen alkalmazható a kitűzött célra. Ám emellett, mint minden kész programnál, jelen esetben is felmerülhetnek olyan igények, melyekkel egy valamivel tágabb specifikációjú navigációs alkalmazás is létrehozható, ezáltal megfontolandóak az esetleges továbbfejlesztések irányában.

A kiindulási programnak a kliensoldalon alapvetően két rétege volt: egy kép, ami a térképet reprezentálta, és ezen egy helyszín, ami tartalmazta magukat az adatokat, melyek alapján megjelenítésre került a térképen a helyzetünk, valamint annak a lenyomatnak az azonosítóját, ami által a rendszer ezt a pontot választotta ki pozícióul a kliens szenzorai által mért adatok alapján. Ahhoz, hogy egy olyan alkalmazást kapjunk, ami képes egyszerre a hely-meghatározásra és a navigációra is, szükség van arra, hogy a navigációs rendszer által felvett csomópontokhoz is egy-egy lenyomatot megfeleltessünk. Ez annyit jelent, hogy miután sikeresen elkészült és az adatbázisba bekerült annak az épületnek a térképe, ahol a rendszert használni kívánjuk, és meghatároztuk a csomópontok elhelyezkedését, szükség van azokon a helyeken méréseket végezni az érzékelhető vezeték nélküli hálózatok jelerősségei miatt. Ebből adódóan olyan lenyomatok kerülnek be az adatbázisba, amelyek azokról a helyekről szolgálnak információval, ahova a szerver a szobákon belül csomópontokat helyezett el. Mivel ezek alapján az eredeti alkalmazás működésének is eleget teszünk, azaz a helymeghatározás a mérések alapján a legközelebbi csomópont helyére azonosítaná a pozíciónkat, emellett már az elkészült gráf miatt a lehetséges összeköttetéseket is meghatároztuk a különböző helyszínek között, így megoldottuk az eredeti program azon hiányosságát, hogy ugyan definiálhattunk különböző helyszíneket, de azok között lehetetlen volt meghatározni az elérhetőség fogalmát a puszta lenyomatok alapján.

Ezzel a módszerrel tehát nem szükséges a navigáció során külön kiválasztanunk azt a helyet, ahová a kiindulási pozíciónkat meghatározzuk, a rendszer alapvető működéséből adódik, hogy képes magától meghatározni ezt a helyet. Innentől fogva csak annyit kell megadnunk, hogy hova akarunk eljutni, a rendszer kiszámolja és visszaadja számunkra az útvonalat, megjeleníti azt a kliensen, sőt, folyamatos mintavételezés, lenyomatképzés és szerverrel való

kommunikáció során arra is képes, hogy bizonyos időközönként frissítse a pozíciónkat, megjelenítse a térképen a még hátralévő utat, és azt is jelezze, ha letértünk az általa meghatározott útvonalról, majd újraszámolja azt.

Egy másik lehetőség, hogy felkészítsük a rendszert olyan helyzetekre, hogy ha esetleg egy olyan személy kívánja igénybe venni, aki (mint a helyzetfelvetésben említettem) valamilyen oknál fogva képtelen minden típusú átjárót igénybe venni. Más szavakkal, hiába adta meg a rendszer a számára, hogy a legrövidebb út egy lépcsőn keresztül vezet a céljához, ő bizonyos egészségügyi problémák miatt képtelen a lépcsőhasználatra.

Ezen probléma megoldásakor nagy előnyként tekinthetjük, hogy jelen program már fel lett készítve arra, hogy amikor az éleket a csomópontok között húzzuk, megadhatjuk, hogy ez csak bizonyos típusú szobákban történjen meg, röviden fogalmazva kihagyhatjuk a lépcsőházakat az élhúzó függvény feltételéből. Ilyenkor egy olyan gráf fog felépülni, ami csak olyan útvonalakat tartalmaz, ami a fent említett személy számára is járható, ezek után már csak a kliens oldalon kell megadni egy olyan opciót, ami ezt szabályozza, azaz egy olyan plusz információt küld útvonaltervezésnél a szerver számára, ami jelzi az aktuális felépítendő gráf milyenségét.

Szintén hasznos dolog, ha a program segítségével célként megadhatunk akár bizonyos helytípusokat is. Azaz ha éppen a legközelebbi mosdót szándékozzuk megtalálni, akkor megeshet, hogy annak a pontos helyéről egyáltalán nincs információnk, csupán annyit tudunk, hogy valahol az épületben biztosan fellelhető néhány ilyen hely.

Ezt a felvetést szintén annak segítségével tudjuk megválaszolni, hogy már minden, az adatbázisból lekérdezett és az adatokból a szerveren objektumként felépített szobának ismerjük a típusát. Ha tehát azt a plusz információt megadjuk a csomópontoknak, hogy milyen típusú szobákban találhatóak, fel tudunk építeni úgy egy gráfot, hogy elemeiként csak ezeket a pontokat tartalmazza (az aktuális pozíciónk mellett), a többi csomópontnál csupán az élt hosszabbítsa meg. Végeredményként egy olyan gráfot kapunk, aminek egyik pontja a mi pozíciónkat jelöli, a többi él pedig egy-egy, célként kiválasztott típusú helyre vezet. Nekünk csupán annyi a dolgunk, hogy a legkisebb költségű él másik végét kérdezzük le, amivel megoldottuk a feladatot: ismerjük a saját helyzetünk, és a legközelebbi, célként azonosított típusú hely hollétét, az eredeti gráfban csupán ezen két pont között kell útvonalat keresni.

Végül pedig, ami az utolsó lépés, hogy a rendszer minden összetevőjének elkészültét követően megkezdődhet az egyes fejlesztési ágak egymással történő egybeintegrálása. Az eredeti célja a projektmunkálatoknak ugyanis, mint már említettem, egy olyan rendszer kialakítása volt, ami a helymeghatározás és a navigáció mellett képes különböző helyfüggő szolgáltatások kezelésére, különféle authentikációs lépések végrehajtásával.

# 8. Köszönetnyilvánítás

Köszönet illeti konzulenseimet, Nagy Istvánt és Tihanyi Attilát, akik szakértelmükkel, szabadidejükkel és rendszeresen biztosított konzultációs lehetőségekkel támogatták a munkám sikeres elkészültét.

# 9. Irodalomjegyzék

**[1]:** NovAtel Inc: GPS+ Reference Manual, 2005, Chapter 1: GPS Overview

[**2**]**:** John Lucas Sikking: The Development of an Indoor Navigation Algorithm for an Autonomous Mobile Robot, Chapter 2.1: Navigation Overview. The University Of Waikato, Hamilton, New Zealand, 2004.

**[3]:** Wooyong Lee, Minkyu Kim, Wonhee Yee, Dooseop Eom: Indoor Navigation System for Mobile Robot using Wireless Sensor Network. Department of Electronics and Computer Engineering, Korea University, 2005.

**[4]:** Soo Yeong Yi, Byoung-Wook Choi: Autonomous Navigation of Indoor Mobile Robot Using Global Ultrasonic System. Chapter 2: A global ultrasonic system. Dept. of Electrical Engineering, Seoul National University of Technology, Republic of Korea, 2007.

**[5]:** Cisco: Wi-Fi Location-Based Services 4.1 Design Guide, Chapter 2: Location Tracking Approaches, Calibration Phase. Fellelés éve: 2011.

**[6]:** Manh Hung V. Le, Dimitris Saragas, Nathan Webb: Indoor Navigation System for Handheld Devices – A major qualifying project report, Worcester, Massachusetts 2009.

**[7]:** Wenjie Yuan, Markus Schneider: iNav: An Indoor Navigation Model Supporting Length-Dependent Optimal Routing Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL 32611, USA. Fellelés éve: 2011.

**[8]:** Nicolas Tissot: Indoor navigation for visually impaired people: The navigation layer. Swiss Federal Institute of Technology, Switzerland, 2003.

**[9]:** Pierre-Yves Gilliéron, Daniela Büchel, Ivan Spassov, Bertrand Merminod: Indoor Navigation Performance Analysis. Swiss Federal Institute of Technology (EPFL), Geodetic Engineering Lab. Lausann, Switzerland, 2004.

**[10]:** Frank Kargl, Sascha Geßler, Florian Flerlage: The iNAV Indoor Navigation System. Ulm University, Institute of Media Informatics, Germany. Fellelés éve: 2011.

**[11]:** RedPin munkacsoport: Redpin - Indoor Positioning for the Rest of Us. Fellelés éve: 2011. http://www.redpin.org

**[12]:** Philipp Bolliger: Redpin - Adaptive, Zero-Configuration Indoor Localization through User Collaboration. Institute for Pervasive Computing ETH Zurich, Switzerland, 2008.

**[13]:** Kása Zoltán: Gráfalgoritmusok, 3.2.4.2. fejezet: A legrövidebb utak algoritmusainak bonyolultsága. Babes-Bolyai Tudományegyetem, Kolozsvár, 2007.

**[14]:** Podobni Katalin: Legrövidebb útkereső algoritmusok, Diplomamunka. 4.1.2.2: Dijkstra algoritmusa éllistával. Eötvös Lóránt Tudományegyetem, Természettudományi kar, Budapest, 2009.