ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Molnár Krisztofer Szebasztián**

2023 Hallgató törzskönyvi száma: **T/006873/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Molnár Krisztofer Szebasztián**
Törzskönyvi száma: T/006873/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Útvonal generálás futó alkalmazáshoz
Route generation for running application

Intézményi konzulens: Dr. Sergyán Szabolcs
Külső konzulens:

Beadási határidő: 2022. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés specializáció

A feladat:

Tervezzen meg és készítsen el egy útvonaltervező alkalmazást, amely egy interaktív térképen a felhasználó által választott két tetszőleges pont között képes olyan útvonalat generálni, amely gyalogos, illetve futó közlekedésre alkalmas. Az interaktív térkép tudjon kommunikálni az útvonaltervezővel. Az útvonaltervezés elindításához legyen lehetősége a felhasználónak egy erre a célra megjelenített gombot használni.
Az alkalmazás legyen képes a generált útvonalat megjeleníteni az interaktív térképen. Az útvonal generálást valósítsa meg Dijkstra, illetve A* algoritmussal is. Ezen felül a felhasználónak legyen lehetősége arra is, hogy ezeket az útvonalakat oly módon generálja, hogy az olyan gyalogos utak, amelyek nem gépjármű utak melletti járdaként funkcionálnak, előnyben legyenek részesítve. Ezeknek a megvalósításoknak tesztelje le az eredményeit, illetve jellemezze őket hatékonyság szempontjából.
Az alkalmazás tartsa számon, hogy a felhasználónak mikor, és merre kell fordulnia, illetve tartsa számon azt az információt is, hogy milyen hosszú a megtenni kívánt útvonal.
Amennyiben két pont közötti útvonalgenerálás megfelelően működik, vizsgálja meg hogyan lehetne megoldani egy megadott hosszúságú véletlenszerű útvonal generálását, amely ugyanabba a pontba érkezik vissza, amelyből elindult.

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a szakirodalom alapján megismert hasonló módszerek, rendszerek ismertetését és értékelését,
- a választott módszer bemutatását,
- a megvalósítandó feladat tervét,
- a felhasználói leírást,
- a tesztadatokat, feltüntetve azok forrását és a teszteredményeket,
- az elért eredmények értékelését,
- a továbbfejlesztési lehetőségeket,
- a dokumentációt, a programot, a szükséges input adatokat, valamint a rendszert bemutató prezentációt CD mellékleten.

Ph.

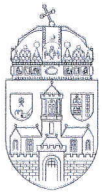
.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. december 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022.12.15

.....

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Molnár Krisztofer Szebasztián

Neptun kód: [redacted]

Tagozat: Nappali

Telefon: [redacted]

Levelezési cím (pl: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Útvonal generálás futó alkalmazáshoz

Szakdolgozat / Diplomamunka[2] címe angolul:

Route generation for running application

Intézményi konzulens: Dr. Sergyán Szabolcs

Külső konzulens:

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.02.14	Félév menetének átbeszélése mit kell implementálni	
2.	2022.03.04	Útvonalgeneráló előrehaladásának bemutatása, és térképszolgáltató kiválasztása	
3.	2022.04.22	Térképpel kapcsolatos előrehaladás bemutatása, Dokumentációk átbeszélése	
4.	2022.05.10	Leadáshoz szükséges dokumentációk bemutatása	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.05.11

intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Molnár Krisztofer Szebasztián .....

Neptun kód: [redacted] .....

Tagozat: Nappali .....

Telefon: [redacted] .....

Levelezési cím (pl: lakcím): [redacted] .....

Szakdolgozat / Diplomamunka[1] címe magyarul:

Útvonal generálás futó alkalmazáshoz .....

Szakdolgozat / Diplomamunka[2] címe angolul:

Route generation for running application .....

Intézményi konzulens: Dr. Sergyán Szabolcs .....

Külső konzulens: .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.09.14	Félév menetének átbeszélése, mit kell javítani	
2.	2022.10.04	Véletlenszerű generálás, és a bővített szakdolgozat dokumentáció bemutatása	
3.	2022.10.21	Szakdolgozat dokumentáció javított verziójának bemutatása, és javított kód bemutatása	
4.	2022.11.23	Végleges kód, és dokumentáció bemutatása	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.11.28 .....

.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

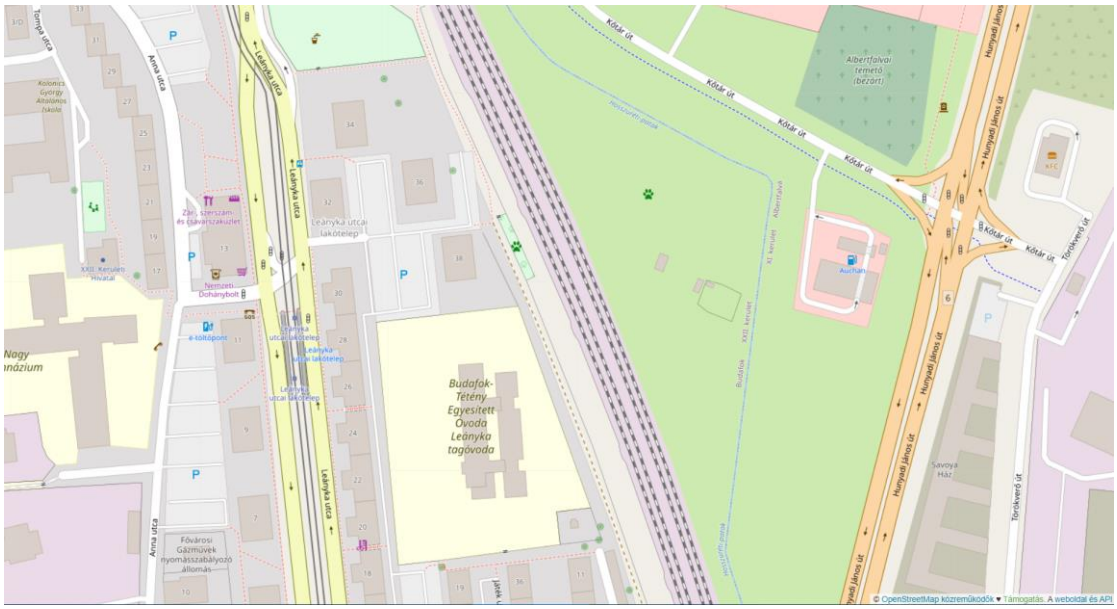
Tartalomjegyzék

1. BEVEZETÉS

Projektmunkámnak egy futó/sétáló alkalmazást választottam. Olyan útvonaltervező alkalmazást szeretnék megvalósítani, amelynek segítségével két kiválasztott pont közötti legrövidebb út legenerálását tudjuk elvégezni, vagy egy megadott pontból kiindulva, egy megadott hosszúsággal, véletlenszerű útvonalat tudjunk generálni. Véletlenszerű generálás esetén, az útvonaltervezőnek úgy kell legenerálnia az útvonalakat, hogy a generált út hossza közelítőleg megegyezzen a felhasználótól előre bekért hosszúságnak az értékével. A véletlenszerűen generált utaknak lehetőség szerint egy körút szerű alakot kell kirajzolniuk, tehát ha van rá lehetőség, akkor a kiinduló ponthoz visszafelé más útvonalon kell eljutnunk, mint amelyik útvonalon távolodtunk tőle. Véletlenszerű generálásnál, a körút alakú generáláson kívül arra is figyelni kell, hogy ha lehetőségünk van rá, akkor egy generáláson belül, egy adott útrészt kétszer ne érintsünk, ezzel biztosítva, hogy a generált utak változatosak legyenek. Ez hasznos lehet azok számára, akik már belefáradtak, hogy ugyanazokat a köröket futják, és szeretnének mindig változatos útvonalakon futni, de nincs kedvük azzal tölteni az időt, hogy mindig előre megtervezzenek maguknak egy új futási útvonalat. Ezen felül online testnevelés órákon is hasznát lehet venni, amikor egy meghatározott távolságot kell minden héten lefutni a diákoknak. Sokkal egyszerűbb és gyorsabb egy ember számára, ha nem magának kell megtervezni azt az utat, amin futni szeretne, hanem ezt elvégzi helyette egy program.

Ahhoz, hogy egy útvonaltervező program működhessen, a megfelelő formában kell neki szolgáltatni azt az úthálózatot, ahol az útvonalakat akarjuk generálni. Gráfokkal [1] egy ilyen úthálózat tökéletesen ábrázolható, ami azt jelenti, hogy az utakat meg tudjuk feleltetni pontoknak, illetve ezeket a pontokat összekötő éleknek. Fontos megemlíteni, hogy egy ilyen szoftver nem tudja vizuálisan értelmezni a térképeket, mint ahogy az emberek. Ezért van szükség egy gráfra, mert annak csúcsait, illetve éleit, és az ezek közötti kapcsolatot már a szoftver is értelmezni tudja.

Ezt az ***1.1 ábrán*** lévő példával demonstrálom.



1.1. ábra: Térképrészlet az OpenStreetMap weboldalról térképelemek megjelenítése nélkül

Ez a térkép vizuálisan van reprezentálva, számunkra átlátható, és értelmezhető módon. Ahhoz viszont, hogy egy útvonaltervező program is rendesen értelmezni tudja, át kell alakítani. Az ***1.2 ábrán*** látható hogyan kell átalakítani egy térképet, hogy azzal dolgozni tudjon egy program. Az utak élekként, a kereszteződések és az érdekesebb pontok pedig csúcsokként vannak megjelenítve.

1.2. ábra: Térképrészlet az OpenStreetMap weboldalról térképelemek megjelenítésével.

2. HASONLÓ MEGVALÓSÍTÁSOK

2.1 Google Maps

A Google Maps a világ egyik legismertebb útvonaltervező alkalmazása. Az interneten nem található pontos dokumentáció a szoftver működéséről, ezért az alábbi adatokat a fejlesztőknek szóló dokumentációs oldalakról tudtam kigyűjteni. A Google a saját útvonaltervező API-ját használja a térképein, és ezt a térképet több mint ezer harmadik féltől származó adat alapján [2] építették fel. A Directions API [3] HTTP kéréseken keresztül generál útvonalakat, amiknek az eredményét json formátumban szolgáltatja vissza. A Google Maps oldaláról útvonal generálást indítva, a hálózati kommunikációknál látszódik, hogy ez az API van meghívva az oldalról, tehát arra lehet következtetni, hogy a Google Maps ezt az útvonaltervező API-t használja. A Google Maps-en egy pontból indított véletlenszerű útvonal generálására nincsen lehetőség, [3] csak két pont közötti generálásra.

2.2 OpenStreetMap

Az OpenStreetMap [4] egy ingyenesen használható, és szerkeszthető térkép az egész világról, amit önkéntesek jelenleg is folyamatosan bővítenek adatokkal, ráadásul ezekhez az adatokhoz teljesen ingyenes a hozzáférés. Az OpenStreetMap csak magát a térkép adatokat szolgáltatja, útvonaltervezéshez nem saját API-t használnak. A ***https://www.openstreetmap.org/directions*** oldalon van lehetőség útvonalak generálására, itt azonban ki lehet választani, hogy a GraphHopper, vagy az OSRM által szolgáltatott útvonaltervezési API-k közül melyiket szeretnénk igénybe venni. Véletlenszerű útvonaltervezésre itt sincs lehetőség, csak két pont között lehet legenerálni az útvonalakat.

2.3 Trail Router

A Trail Router alkalmazás [5] egy olyan útgeneráló alkalmazás, ami a zöldebb utakat részesíti előnyben az útvonalak generálásakor. Az alkalmazás használható két pont közötti útvonal generálásra is, és egy kezdő pontból induló körút generálására is. Ez az alkalmazás OpenStreetMap adatokkal dolgozik, útvonal generáláshoz pedig az Openrouteservice átalakított változatát használja, az oldalon megjelenített interaktív térképhez pedig a Mapbox térkép szolgáltatást használja. Útvonal generáláshoz az A* algoritmust használja az alkalmazás, és a generálás során lehetőséget ad arra, hogy ne repetitíven generálódjon le az útvonal, tehát lehetőleg ugyanarra az útra ne lépjen kétszer a felhasználó. Ezt úgy oldja meg az alkalmazás, hogy a legenerált út részeit eltárolja egy hashelt táblába, és ha az útvonal a generálás során megint egy ilyen eltárolt útrészre lépne rá, akkor büntetve lesz a lépés súlya, hogy másik irányba generáljon. A véletlenszerű körutak generálását úgy közelíti meg az alkalmazás, hogy a térképre lerakott ponthoz képest megkeres egy olyan területet, ami az OpenStreetMap

adatbázisban természeti területként van felvéve, majd, ha egy ilyen pontot talált, akkor lehetőleg annak a területnek a közelében generál egy körutat, majd végül visszavezeti a felhasználót a kiinduló pontjához, mindezt úgy, hogy nagyjából megmaradjon a generálni kívánt út hossza. Amennyiben ilyen természeti terület nincs a közelben akkor a kiinduló pontból generál körutat, oly módon, hogy a kiinduló pontból létrehoz egy akkora kört, aminek a kerülete akkora, mint a generálni kívánt távolság, és a kiinduló pont, illetve a kör középpontja közötti vektor irányszöge 0 fok. Ez azt jelenti, hogy a kezdő ponttól jobbra lesz egy képzeletbeli vektor, aminek hossza a generált kör sugara lesz, maga a körút pedig jobbra fog generálódni a kiinduló ponthoz képest. Ennek a körnek a körvonalán elhelyez a program három egyenletesen elosztott pontot, amikhez megkeresi, hogy melyik térkép utak helyezkednek el a legközelebb, és ezek között az utak között generál le úgy útvonalat, hogy a természeti utak előnyben legyenek részesítve. Több ilyen útvonal [5] is generálódik a térképen, méghozzá olyan módon, hogy a körút meghatározásához használt vektor, aminek irányszöge 0 fok volt, többször is meg lesz növelve 45 fokkal, ezzel biztosítva, hogy több irányba is útvonalat generáljon a program, nagyjából hasonló hosszúságokkal. Tehát az első generálásnál a körút a kezdőponttól jobbra fog generálódni, mivel a vektor irányszöge 0 fok lesz, viszont a harmadik generálásnál már felfele fog haladni a körút a kezdő ponttól, mivel a kör középpontja és a kezdőpont közötti vektor irányszöge meg lesz növelve 90 fokra.

3. ELENGEDHETETLEN FOGALMAK

Mivel az ***1. fejezetben*** megemlítettem, hogy gráffal tud megfelelően dolgozni egy útvonaltervező program, ezért mindenekelőtt szeretnék ismertetni néhány gráf fogalmat, amely ismerete nélkülözhetetlen a továbbiakban.

Gráf: Egy gráf két halmazból épül fel: [6] az egyik halmaz a pontokat/csúcsokat tartalmazza (nevezzük „V" halmaznak), a másik halmaz (nevezzük „E" halmaznak) az éleket tartalmazza, ezek az élek pedig a „V" halmazbeli csúcspárokat tartalmazzák. Ha például egy él összeköti v1 és v2 csúcsokat, akkor azt az élt, (v1, v2) -vel jelezzük.

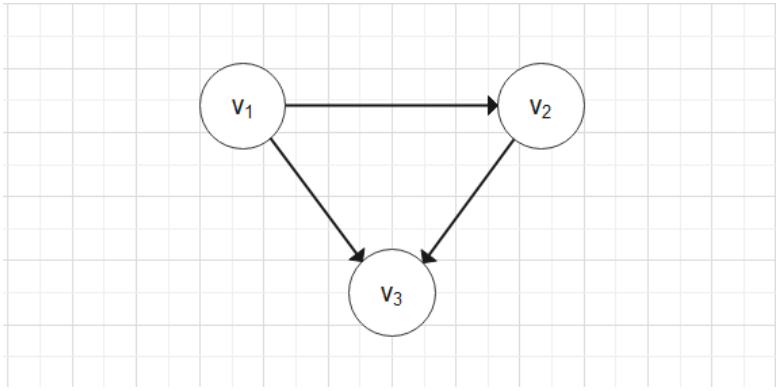
Hurokél: Azt az élt nevezzük hurokélnek,[7], amelyik élnek mind a két csúcspontja ugyanaz a csúcs. Tehát például: (v1, v1) egy hurokél.

Út: Csúcsok olyan sorozata [6] (vi, … vn), ahol minden (vi, vi+1) él szerepel a gráf éleinek halmazában.

Legrövidebb út: Gráfelméletben legrövidebb út alatt [8] egy gráf két csúcsa, vagy éle közötti minimális hosszúságú/költségű utat értünk. Amennyiben súlyozatlan gráffal dolgozunk, akkor ez az út megegyezik a legkevesebb élt tartalmazó úttal. Hogyha súlyozott gráffal dolgozunk, akkor ez az út megfelel a legkevesebb költséggel rendelkező útnak.

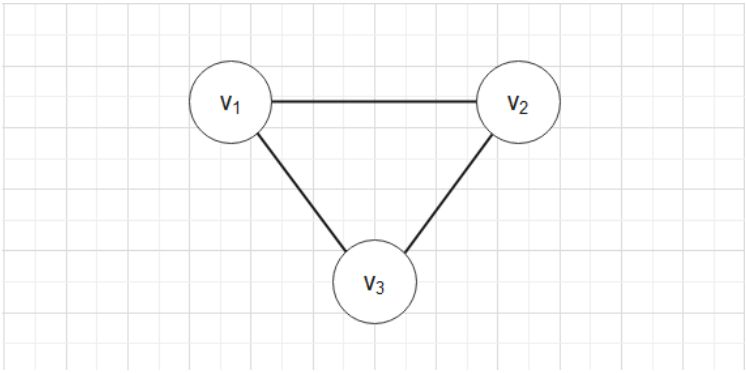
Irányított gráf: Olyan gráf, [9] amelynél két csúcs közötti kapcsolat egyirányú, élhalmaza irányított éleket tartalmaz. Irányított éleknél, a zárójelen belüli csúcspontok sorrendje számít. Azt mondjuk, hogy egy irányított él a (v1, v2) csúcspár első eleméből, mutat a csúcspár második elemébe. Például: [6] A ***3.1 ábrán*** (v1, v2) csúcspárok esetén az irányított él v1 csúcsból mutat v2 csúcsba.

Ezt jelölhetjük így is: v1 → v2.



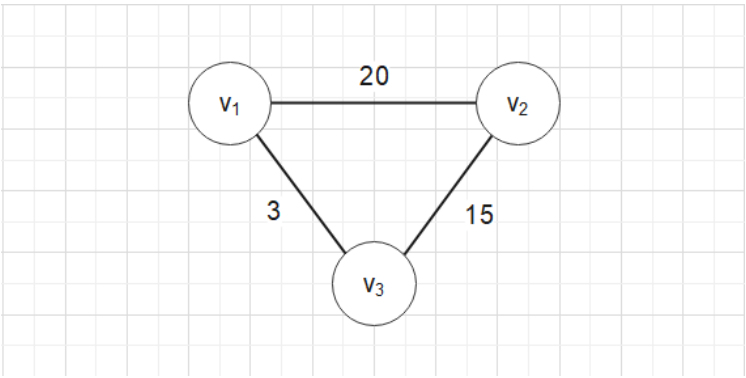
3.1. ábra Példa egy irányított gráfra

Irányítatlan gráf: Olyan gráf, [9] ahol két csúcs közötti kapcsolat mindig kölcsönös, tehát ha v1 és v2 csúcsok egy éllel össze vannak kötve, akkor v1 → v2, és v2 → v1 is teljesül, ahogy az a ***3.2. ábrán*** is látszik.



3.2. ábra: Példa egy irányítatlan gráfra

Súlyozott gráf: Egy olyan gráf, [10] amelyben minden élhez hozzá van rendelve, egy számérték. Ezek a számértékek jelölik az egyes éleknek a költségét (pl. végig haladás ideje), és általában pozitív az értékük. Ez példaként látható a ***3.3. ábrán***.



3.3. ábra: Példa egy súlyozott gráfra

4. ALGORITMUSOK

4.1 Szélességi bejárás

Útvonaltervezéshez a továbbiakban Dijkstra algoritmusát, illetve az A* algoritmust fogom részletezni, viszont mielőtt azokat bemutatnám, ki szeretném fejteni, hogy a szélességi bejárás miként működik, ugyanis ennek ismerete szükséges lesz a továbbiakban, erre épül mindkét másik algoritmus.

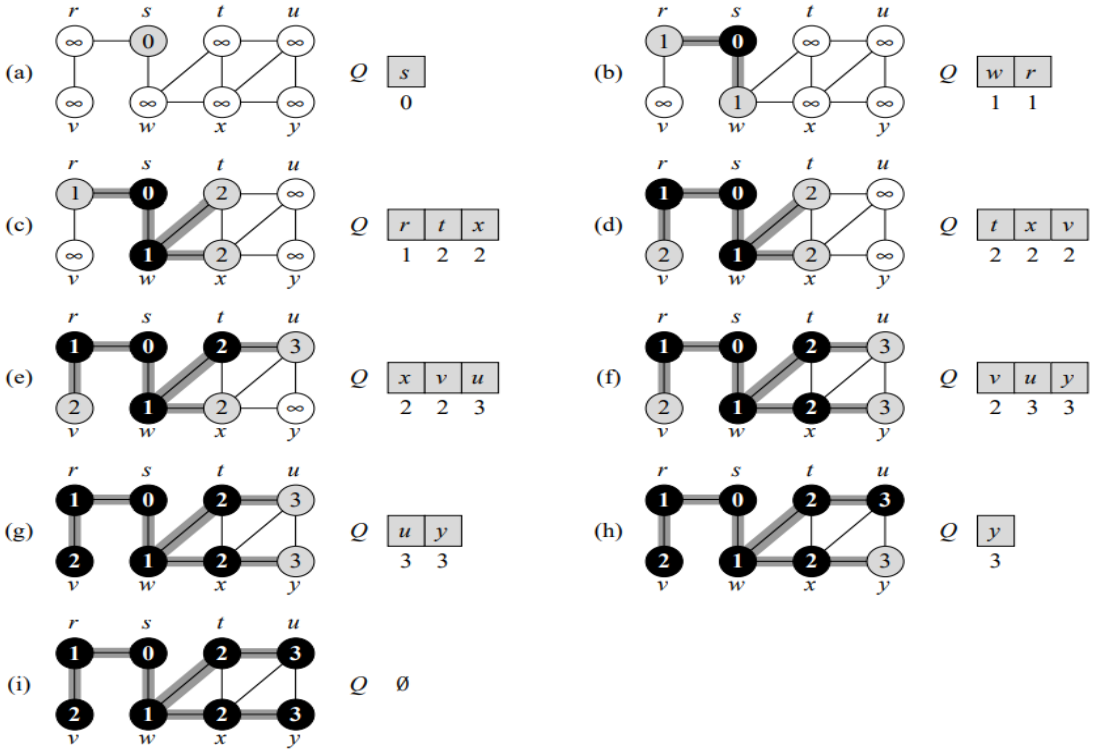
Ezek a gráf keresési algoritmusok [11] mind memóriában tárolják el a generált csúcsokat, méghozzá külön azokat a csúcsokat, amelyek a „keresési határon” (továbbiakban frontier) helyezkednek el, illetve külön azokat a csúcsokat, amelyeket már megvizsgált, és rájuk terjeszkedett. A generált csúcsok tárolásának két oka van. Először is, hogy azokat a csúcsokat, amelyeket már megvizsgált, és elért az algoritmus, ne tárolja el még egyszer, ha máshonnan újra elérte az adott csúcsot; Ezt angolul „duplicate detection” -nek hívják, magyarra fordítva: másolat észlelés.

Másodszor pedig biztosítja a megoldási útvonal rekonstrukcióját „traceback method” visszavezetéses módszer segítségével. Minden létrehozott csúcs tárolása lehetővé teszi mindkét funkciónak a működését, azonban, ha csak az egyik funkció működése fontos, akkor nem kell minden legenerált csúcsot eltárolni.

4.1.1 A szélességi bejárás működése

Egy ***s*** csúcsból kiindulva a gráfon, [12] a szélességi bejárás egyesével felfedez minden olyan csúcsot, amit el lehet érni az ***s*** pontból. Az algoritmus minden ***k*** távolságra lévő csúcsot felfedez az ***s*** csúcstól, mielőtt elkezdené a ***k+1*** távolságra lévő csúcsok felfedezését, egy csúcs pedig akkor válik felfedezetté, amikor legelőször találkozik vele a bejáró algoritmus.

A szélességi keresés működése során három állapota lehet az egyes csúcsoknak. [12] Az első állapot, ami a kiinduló csúcs kivételével minden másik csúcsnak a kezdő állapota lesz, az a felfedezetlen állapot. A második állapot az a felfedezett állapot, amikor az algoritmus megtalálta az adott elemet, de még nem dolgozta fel, a harmadik állapot pedig a feldolgozott állapot. A ***4.1.1.1. ábrán*** lévő példában ezek színekkel vannak megkülönböztetve, amik a következőket jelentik: fehér - felfedezetlen, szürke - felfedezett, és fekete - feldolgozott. Egy fehér csúcsból mindig akkor lesz szürke, ha az algoritmus felfedezi a csúcsot, és belerakja a frontier-be. Ezek a frissen felfedezett csúcsok mindig az aktuálisan feldolgozott fekete csúcsnak lesznek a szomszédai, tehát mindig az aktuálisan feldolgozott csúcs szomszédai fognak bekerülnek a frontier-be, az algoritmus bejárása pedig addig tart, amíg van elem a frontier-ben. Erre példa a ***4.1.1.1. ábrán*** látható.



4.1.1.1. ábra: A szélességi bejárás működésének ismertetése. [12]

Az algoritmus kezdetekor minden csúcs fehérrel van jelölve, a kiinduló pont kivételével. Az egyes ábrák mellett a ***4.1.1.1. ábrán***, látható a frontier aktuális állapota, amiben a felfedezett csúcsok vannak eltárolva, az elérési súlyukkal. Az algoritmus minden iterációban kiveszi a legkisebb súllyal rendelkező csúcsot a frontier-ből, feldolgozottá teszi azt, majd a felfedezetlen szomszédaival kibővíti a frontier-t. A frontier bővítésekor, a felvett csúcsokhoz az elérési súlyokat is beállítja. Ez a folyamat egészen addig megy, ameddig a frontier nem lesz üres. Az algoritmus futásának végeztével pedig megkapjuk, hogy milyen csúcsokat érünk el a kezdő csúcsból, és hogy ezeket az elért csúcsokat, milyen súllyal lehet elérni.

4.2 Dijkstra algoritmus

Ez az algoritmus Edsger Wybe Dijkstra nevéhez fűződik, aki 1956-ban dolgozta ki, viszont publikálva csak később, 1959-ben lett. Dijkstra algoritmusa [13,14] egy adott csúcs (kezdő csúcs), és a gráf összes többi csúcsa közötti legrövidebb utakat adja meg. Lehetséges, hogy egy csúcsba több ugyanolyan költségű út is vezet, ilyenkor csak az egyiket adja meg ezek közül. Maga az algoritmus csak akkor [14] működik megfelelően, amikor egy irányított, súlyozott gráffal dolgozik, amiben nincsenek negatív súlyok. Mivel súlyozott gráffal dolgozik, ebből adódóan a legrövidebb út a legkevesebb költséggel rendelkező utat fogja jelenteni.

Az algoritmus az egyik legnépszerűbb gráfbejáró algoritmuson alapszik, a szélességi bejáráson, amit a ***4.1 fejezetben*** már részletesen ismertettem. Szélességi bejárással [13]

a kiinduló csúcsból megkeresi az összes többi elérhető csúcsot, majd minden csúcshoz hozzárendeli a hozzá vezető legrövidebb út hosszát, és ennek az útnak az előző csúcsát, tehát hogy melyik csúcsból érjük el az adott csúcsot, a megfelelő költséggel.

4.2.1 A Dijkstra algoritmus működése

Működéséhez [13] az egyes csúcsoknál meg kell határoznunk, hogy a csúcsot elértük-e, és feldolgoztuk-e, ezen kívül pedig el kell még tárolni az oda vezető út költségét, és egy másik csúcsra való hivatkozást. Ez a hivatkozás lesz az a csúcs, ahonnan az adott költséggel elérhető a vizsgált csúcs. A kiinduló csúcsot elérhetőnek, a költségét pedig 0-nak tekintjük, ezek után pedig egy ciklus indul, amelynek lépései: [13]

1. *Kiválasztja a legkisebb költséggel rendelkező, még feldolgozatlan, de már elért elemet, és feldolgozottra állítja. Nevezzük most ezt a csúcsot „u” -nak. (Az algoritmus indulásakor ez a kiinduló pont lesz.)*

2. *Az „u” csúcs összes szomszédos elemét megvizsgálja. Ezeket a szomszédos csúcsokat most jelöljük „x” -el. Beállítja hozzájuk a költséget, amennyiben az „u” csúcsból vezető út költsége kisebb az „x” csúcs jelenlegi költségénél, vagy ha az „x” csúcs még nem lett elérve. Ha be kell állítani az új költséget, akkor az „x” csúcs költségét lecseréli, az új költségre, illetve az eléréséhez tartozó csúcsnak beállítja az „u” csúcsot.*

3. *Ezeket a lépéseket addig végzi, amíg van elérhető, de fel nem dolgozott elem.*

Az algoritmus működésénél célszerű előre beállítani a kiinduló pont kivételével minden csúcs költségéhez egy nagyon magas (végtelen/strázsa) értéket. Ezután az elemek elérhetősége vizsgálható, miszerint, ha a költség értéke nem egyezik meg ezzel a strázsa értékkel, akkor elérhető. Maga az algoritmus [15] mohó stratégia szerint működik, mivel mindig a legkevesebb költségű elemet választja.

A ciklus lefutása után [13] minden csúcshoz hozzá lesz rendelve, hogy a hozzá vezető út költsége mekkora, illetve, hogy melyik csúcsból juthatunk el oda ilyen költséggel. Ezután visszavezethető az egyes csúcsokból, hogy mi a hozzá vezető legrövidebb útvonal a kiinduló csúcsból.

Ahhoz, hogy ne kelljen az egész gráfot bejárni, lehetőség van arra, [13] hogy csak egy adott csúcshoz vezető útvonal megtalálásáig fusson az algoritmus. Amennyiben az algoritmus a fenti lépésekben említett **„1.”** pontban feldolgozza a keresett „cél” csúcsot, akkor elvégezte a feladatát. Ezután ennél rövidebb utat nem fog találni, mivel a feldogozott csúcsokból ez lesz a legrövidebb oda vezető út, a feldolgozatlan csúcsoknak pedig amikor kiválasztottuk, már akkor is nagyobb volt a költségük.

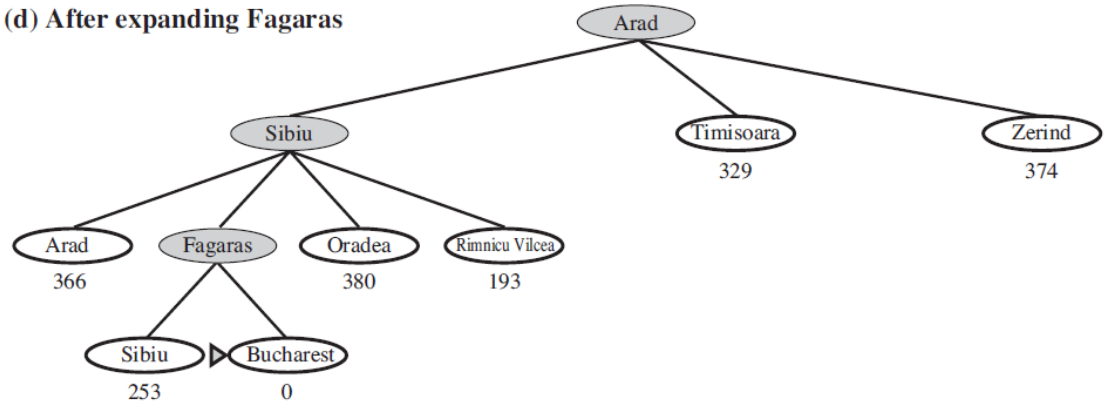
4.3 Mohó legjobbat először algoritmus

A mohó legjobbat először algoritmus mindig egy cél felé halad. Az útvonal építése közben [16] mindig azokat a pontokat választja ki, amik a legközelebb viszik a céljához, ezzel törekedve arra, hogy gyorsan találjon egyik pontból a másikba útvonalat. Mivel

az algoritmus csak azt vizsgálja, hogy milyen közel van a vizsgált pont a célhoz, azt lehet állítani, hogy ez az algoritmus tisztán csak heurisztikán alapszik. A keresés, habár gyors, nem optimális. Előfordulhatnak olyan esetek, hogy a legjobbnak vélt pontok által generált útvonal mégsem a legrövidebb lesz.

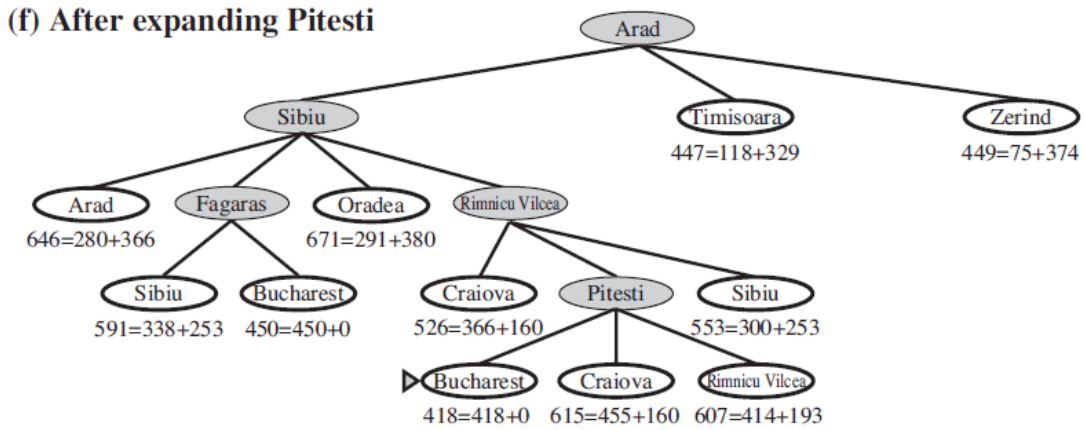
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

4.3.1. ábra: Romániai települések távolsága Bukaresttől [16]



4.3.2. ábra: Legrövidebb útvonal ábrázolása Aradtól Bukarestig. [17]

A ***4.3.1 ábrán*** látható súlyok egy egyenes vonal távolság heurisztikával [16] lettek kiszámolva. A kiszámított súlyok alapján, a mohó legjobbat először algoritmus, először ***Sibiu*** felé fog haladni, mivel a szomszédjai közül, ez a település van a legközelebb Bukaresthez, 253-as súllyal. Az algoritmus ezután ***Faragas*** felé fog menni, mivel ***Sibiu*** által megközelíthető települések közül annak a legkisebb a súlya, ***Faragas***-ból pedig eléri Bukarestet. A heurisztika [16] alapján, úgy tűnhet, hogy az algoritmus megtalálta a legrövidebb útvonalat a két település között, viszont ez az útvonal a valóságban 32 km-rel hosszabb a legrövidebb útvonalnál. A tényleges legrövidebb útvonal a ***4.3.3. ábrán*** látható, amin az A-csillag algoritmus generálta le az útvonalat.



4.3.3. ábra: A legrövidebb útvonal Arad és Bukarest között. [17]

Az algoritmus nem optimális működésére látható egy másik példa is, a ***4.4.4.3 ábrán***.

4.4 A-csillag (A*) algoritmus

Az A-csillag algoritmus, az egyik legelterjedtebb gráf bejáró algoritmus. 1968-ban dolgozták ki, [18] és az volt a célja, hogy úgy terjeszkedjen a célpontja felé, hogy a lehető legkevesebb pontot kelljen megvizsgálnia ehhez a gráfon. Ahhoz, hogy ez megvalósulhasson, az algoritmus [19] a heurisztikus keresés, illetve a legrövidebb útvonalakon alapuló keresés kombinációjaként fogható fel. Az algoritmus fő célja, hogy egy kijelölt cél felé terjeszkedjen. Ha olyan [18] pontok fele terjeszkedne, amelyek feltűnően nem a célhoz vinnék közelebb, akkor csak feleslegesen vizsgálná meg az adott pontokat, tehát az algoritmus megfelelő működéséhez el kell dönteni, hogy melyik pontok viszik a célja felé, és arra kell terjeszkedni.

A ***4.2 fejezetben*** ismertetett Dijkstra algoritmus, szélességi bejárás segítségével dönti el, hogy melyik úton lehet a legrövidebben megközelíteni egy célpontot, viszont ezzel rengeteg pontot vizsgál meg teljesen feleslegesen. A ***4.3 fejezetben*** ismertetett Mohó legjobbat először algoritmus pedig ezzel ellentétben, csak heurisztikákat felhasználva próbálja megtalálni a legrövidebb útvonalat két pont között. Nem vizsgál meg feleslegesen sok pontot, viszont előfordulhat, hogy nem az optimális útvonalat találja meg. Az A* algoritmus ezt a két működést olvasztja egybe. Az egyes pontokhoz tartozó [19,20] súlyokat, a pont elérésének költségéből, illetve egy heurisztikával számolja ki, de ez részletesebben a ***4.4.4 fejezetben*** van kifejtve, mert a működés előtt ismertetni szeretnék pár alkalmazható heurisztikát.

4.4.1 Manhattan távolság

A [21,22] Manhattan távolság értéke két pont között, egy kétdimenziós térben, az egyes dimenziókban lévő távolságoknak az összege. Tehát szükség van az x tengely szerinti különbségre is a két pont között, és az y tengely szerinti különbségre is:

$$d(A,B) = |Ax - Bx| + |Ay - By| \quad (1)$$

A heurisztika azért kapta a nevét Manhattanről, [22] mert Manhattan városában az épületek négyzet alakú blokkokként vannak kiépítve, és egy ilyen városban pontosan ezzel a módszerrel lehet megkapni a legrövidebb útvonalat.

4.4.2 Az Euklideszi távolság

Az Euklideszi távolság, [23,24] egy egyenesen elhelyezkedő két pont távolságát adja meg, méghozzá a Pitagorasz-tétel használatával. A [25] képlethez szükség van a két pont közötti különbségre, amit a ***4.4.1 fejezetben*** hasonlóan lehet megkapni. Ezeket a különbségeket négyzetre kell emelni, majd az így kapott értékek összegéből, gyököt kell vonni. A képlet kétdimenziós pontok esetén a következőképpen néz ki:

$$d(A,B) = \sqrt{(Ax - Bx)^2 + (Ay - By)^2} \qquad (2)$$

4.4.3 A Nagy-kör távolság

A Föld alakja geoid. Ezt jelen esetben tekinthetjük egy gömbnek, [26] ezért két pontja közötti legrövidebb távolságot (gömbi koordináta-rendszert feltételezve) a (3) képlettel tudjuk meghatározni. Ez a távolság egy olyan ívet fog reprezentálni, ami a bolygó két pontját köti össze. A bolygó gömbölyűségét figyelembe véve, bármelyik legrövidebb útvonal a bolygó görbületén fog átívelni. Mivel a térképek egy síkon reprezentálják a bolygót, a rajtuk keletkező egyenes vonalak nem feltétlen adják a legrövidebb utat. A valós legrövidebb utat az alábbi (3) képlet [26] segítségével lehet megkapni.

$$\mathrm{Cos(T)} = (\mathrm{Sin(a)} * \mathrm{Sin(b)}) + (\mathrm{Cos(a)} * \mathrm{Cos(b)} * \mathrm{Cos(|c|)}) \qquad (3)$$

Ahol ***a*** az egyik koordináta szélességi fokát jelenti szögben megadva, ***b*** a másik koordináta szélességi fokát jelenti szögben megadva, ***|c|*** pedig a két koordináta hosszúsági fokának a különbsége.

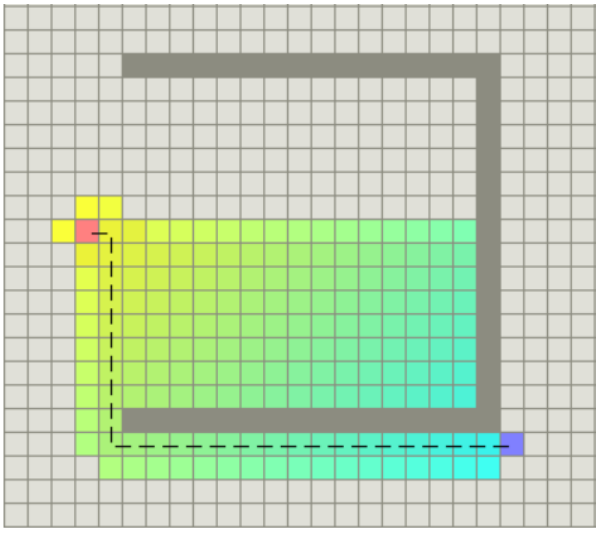
A függvény eredménye fokban lesz megadva, viszont a föld felszínén egy fok nagyjából átváltható 111,32 km-re, [26] szóval az eredményt csak meg kell szorozni 111,32-vel.

4.4.4 Az A* algoritmus működése

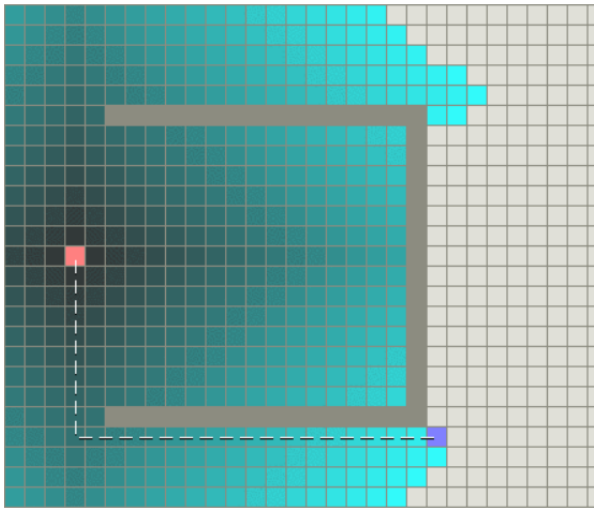
Az algoritmus működéséhez be kell vezetni az alábbi képletet.

$$f(v) = g(v) + h(v) \quad (4)$$

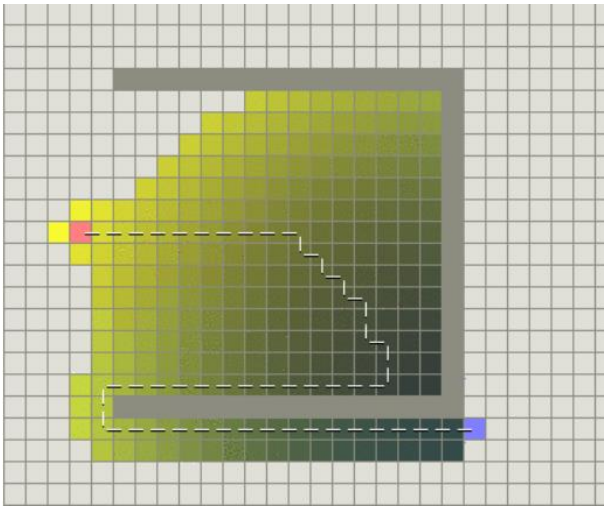
Ahol [19,20] ***g(v)*** azt a pontos költséget jelenti, amely a kiinduló pont és az adott ***v*** csúcs között áll fenn, ***h(v)*** pedig azt a heurisztikus becsült költséget jelenti, amely az adott ***v*** csúcs, illetve a cél csúcs között áll fenn. Tehát ***f(v)*** adott ***v*** csúcshoz adja meg a kezdő csúcstól való pontos távolságának, illetve a célcsúcstól való becsült távolságának az összegét. Az algoritmus futása során, [19] az aktuálisan elért csúcs minden szomszédjához fel lesz véve a (4) képlet által kiszámított súly, illetve, hogy melyik csúcsból lett elérve, hogy vissza lehessen követni az optimális útvonalat. A ***4.2.1 fejezetben*** leírtakhoz hasonlóan, itt is érvényesül, hogy ha egy pont elérhető máshonnan kevesebb súllyal, akkor ez a kevesebb súlyú elérés fog érvényesülni az adott pontnál. Ezekkel a súlyokkal kerülnek be a csúcsok a frontier-be. Minden lépésnél, az elért szomszédok rögzítése után, ki lesz választva a legkisebb ***f(v)*** értékkel rendelkező csúcs a frontier-ből. Ilyenkor a kiválasztott csúcsot el kell távolítani onnan, utána pedig mivel a kiválasztott csúcs lesz az aktuálisan elért csúcs, fel kell venni a szomszédait a frontier-be. Az algoritmus [20] akkor fog végezni, amikor az aktuálisan kiválasztott eleme, a cél csúcs lesz. Az alábbi ***4.4.4.1 ábrán,*** az ***f(v)*** értéke van vizuálisan megjelenítve. Minél sárgább egy pont, annál távolabb helyezkedik el a ***v*** csúcs a cél csúcstól, míg minél kékebbek az egyes csúcsok, annál közelebb vannak a cél csúcshoz.



4.4.4.1. ábra: A algoritmus: [27]*



4.4.4.2. ábra: Dijkstra algoritmusa: [27]



4.4.4.3. ábra: Mohó legjobbat-először keresés: [27]

Ugyanezen a gráfon Dijkstra algoritmusa, illetve a mohó legjobbat-először keresés teljesítménye látszódik a ***4.4.4.2***, és ***4.4.4.3 ábrákon.***

A ***4.4.4.2***, és ***4.4.4.3 ábrákon*** is látszik, hogy Dijkstra algoritmusa olyan pontokat is megvizsgál, amik csak messzebb vezetik a céltól, sok időt eltöltve felesleges vizsgálatokkal. A mohó legjobbat először keresés viszont bele eshet abba a csapdába, hogy arra indul, amerre a leghamarabb jutna a cél felé, viszont akadályba ütközhet *(pl.: Budapesten a Duna)*, így pedig nem a legoptimálisabb útvonalat adja eredményül. Ezek alapján az ábrázolások alapján jól látszik, miért célszerű az A* algoritmus használata, mivel nem vizsgál meg feleslegesen sok pontot, mint a Dijkstra algoritmus, mégis képes az optimális útvonalat megtalálni. A fentebb megemlített heurisztikák konzisztensek, azonban mielőtt tovább lépnék, be szeretném mutatni, hogy az A* algoritmus mikor optimális.

4.4.5 Az A* algoritmus működésének jósága

Ahhoz, hogy az A* algoritmus optimálisan [17] működjön, szükség van egy elfogadható heurisztikára. Az ilyen elfogadható heurisztikákat optimista heurisztikáknak szokták nevezni, mivel mindig kevesebb költséget jósolnak egy probléma megoldására, mint az igazi költség. Egy gráfokhoz használt heurisztika felé a másik elvárás az, hogy konzisztens és monoton legyen.

„Egy ***h(v)*** heurisztika akkor konzisztens, ha minden ***v*** pontra, és egy ***a*** művelet által a ***v*** pontból elért ***v'*** pontra igaz az, hogy a ***v*** pontból a cél eléréséig becsült költség, sosem nagyobb, mint a ***v'*** pont elérésének, és a ***v'*** ponttól a célig becsült költségnek az összege.” [17]

$$h(v) \leq c(v, a, v') + h(v') \tag{5}$$

5. VÉLETLENSZERŰ GENERÁLÁS LEHETŐSÉGEI

Véletlenszerű generáláshoz többféleképpen is hozzá lehet állni. Mivel ehhez nem találtam megfelelő szakirodalmat, az alábbiakban saját megoldási lehetőségeket sorolok fel, ezért nincsenek hivatkozások megjelölve.

5.1 Első próbálkozás

Egyszerű megoldásként, a kiinduló szegmenstől el lehet távolodni a leghosszabb szomszédos szegmenseken keresztül, egészen addig, amíg a generálni kívánt hossznak el nem érjük a felét. Ha ezt a hosszt elértük, akkor a maradék távolság generálását fel lehet osztani két részre. Ehhez szükség lesz három pontra. Szükség van a távolodás által elért pontra, ami most az ***(a)*** jelölést kapja. Szükség van a kiinduló pontra is, ami a ***(b)*** jelölést kapja. ***(a)*** és ***(b)*** pont között véletlenszerűen ki lehet választani egy harmadik ***(c)*** pontot, aminek az ***(a)*** és ***(b)*** ponttól vett távolsága is közelítőleg egyenlő a fennmaradó generálni kívánt hossz felével. Ekkor a ***(b)*** pontból le lehet generálni A* algoritmus segítségével egy utat a ***(c)*** pontba, majd ***(c)*** pontból is le lehet generálni egy útvonalat ***(a)*** pontba. Az így kapott három utat összefűzve pedig megkapunk egy félig véletlenszerűen generált utat, aminek hossza közelítőleg egyezik a generálni kívánt hosszal.

Ez a megoldás, habár működőképes, sajnos nem teljesen véletlenszerűen generálna útvonalakat, ezért nem ezt a megoldást választottam. A két pont között generált véletlenszerű útvonalak első fele mindig ugyanaz lenne, mivel az algoritmus csak a távolodás után választana véletlenszerűen pontot, ahova tovább generálhatna.

5.2 Második próbálkozás

Dijkstra algoritmus segítségével könnyen meg lehet találni az összes olyan pontot, ami valamekkora távolságra van a kezdő ponttól. Ez a módszer ezt használja fel.

A generáláshoz szükségünk van a generálni kívánt hossz valahányadára, ez ***(x)***-szel lesz jelölve. Dijkstra algoritmussal meg tudjuk találni az összes olyan pontot, ami ***(x)*** távolságra van a kezdő pontunktól. Ezek közül a pontok közül véletlenszerűen lehet egyet választani. A kiinduló pont jelölése legyen ***(a)***, a véletlenszerűen választott pont jelölése pedig legyen ***(b)***. Ezután ***(b)*** pontból is megkeresünk minden olyan elemet, ami közeledik vissza az ***(a)*** pont felé, és az elérése ***(x)*** háromnegyedével egyenlő. Ezek közül az elemek közül véletlenszerűen ki lehet választani egy harmadik pontot, ennek legyen ***(c)*** a jelölése. A ***(c)*** pontig már le is van generálva az útvonal, mivel Dijkstra algoritmus segítségével választottuk ki pontokat, ezért már csak a ***(c)*** és ***(a)*** pont közötti részt kell legenerálni. Ezt A* algoritmussal gyorsan meg lehet oldani, és az így kapott útvonalat hozzá lehet fűzni a Dijkstra algoritmus által generált útvonalhoz.

Mivel nincs egy célpont, ahol leállhatna a Dijkstra algoritmus, minden olyan pontot meg kellene találnia, ami ***(x)*** távolságra van a kiinduló ponttól. Ez nagy távolságok

generálásánál sokáig is eltarthat, ezért habár ténylegesen véletlenszerű útvonalakat lehetne generálni ilyen módon, mégsem ezt a megoldást választottam.

5.3 Harmadik próbálkozás

A* algoritmus segítségével gyorsan le lehet generálni két pont közötti útvonalat, ezzel a módszerrel pedig ezt használom ki.

Véletlenszerű útvonal generálásnál egy kör alakú útvonalat szeretnénk kapni, aminek a kiinduló pontja egyben az érkezési pontja is. Ezt a problémát az A* algoritmussal fel tudjuk bontani négy részre. Ehhez szükség van a generálni kívánt hossz valahányad részére, ezt ***(x)***-el jelölöm. Ennek az értéknek a segítségével ki tudunk választani egy olyan pontot, ami ***(x)*** távolságra van a kiinduló pontunktól. Legyen a kiinduló pont jelzése ***(a)***, a másik választott pont jelzése pedig ***(b)***. Az ***(a)*** és ***(b)*** pontot össze lehet kötni egy képzeletbeli vonallal, aminek segítségével el tudjuk dönteni, hogy melyik pontok vannak az ***(a)*** és ***(b)*** ponttól balra, illetve melyik pontok vannak tőlük jobbra. Ezután kiválasztjuk azokat a pontokat, amik ***(a)*** és ***(b)*** ponttól is ***(x)*** távolságra vannak. Ezek közül a pontok közül már véletlenszerűen tudunk választani egy olyan pontot, ami az ***(a)*** és ***(b)*** ponttól balra van, és tudunk egy olyat is választani, ami jobbra van.

Így lesz négy darab pontunk, amelyek között A* algoritmussal egyenként le tudjuk generálni az útvonalakat, ezeket a kisebb generált utakat pedig össze tudjuk fűzni egy nagyobb útvonallá. Az így kapott útvonal pedig kör alakú lesz, véletlenszerűen lesz generálva, és a hossza közelítőleg egyezni fog a generálni kívánt hosszal is.

Ez a megoldás ténylegesen véletlenszerű útvonalakat tud eredményezni, illetve gyors is, ezért ezt a megoldást választottam a véletlenszerű generálás megvalósításához.

6. TÉRKÉP ADATOK

Fentebb már szó esett arról, hogy egy útvonaltervező szoftvernél szükség van egy gráfra, mivel a program ezzel tud dolgozni. Ezt a gráfot pedig térkép adatok alapján kell valahogy felépíteni. Ebben a fejezetben azt fogom ismertetni, hogyan, és milyen adatokkal lehet felépíteni egy gráfot, amivel egy virtuális térkép dolgozni tud.

Sokféle online térkép létezik, ezek közül az OpenStreetMap térkép adatbázisát fogom ismertetni. Az OpenStreetMap [4] egy ingyenesen használható, és szerkeszthető térkép az egész világról, amit önkéntesek jelenleg is folyamatosan bővítenek adatokkal, ráadásul ezekhez az adatokhoz teljesen ingyenes a hozzáférés.

Az OpenStreetMap, egy ***osm*** kiterjesztésű fájllal dolgozik. [28] Sokféleképpen lehet földrajzi adatokat virtuálisan eltárolni, az ***osm*** kiterjesztésű fájlok ezeknek az egyike. Ez a kiterjesztés specifikusan az OpenStreetMap-hez köthető, máshol nem lehet vele találkozni. Az ***osm*** fájlok, azért vannak olyan kiterjesztésben használva, amit senki más nem használ, mert nagyon sok földrajzi adatokkal foglalkozó kiterjesztés létrejötte, jóval megelőzte a modern internetes korszakot. Amíg ezeket a régebbi kiterjesztéseket, gyors hozzáférhetőségre, illetve lekérdezésekre tervezték, addig az ***osm*** adatokat úgy tervezték, hogy az egy általános alakban, könnyen továbbítható, illetve megkapható legyen az interneten keresztül. Éppen emiatt az ***osm*** fájlok XML kódolásban vannak tárolva, és egy jól strukturált, rendezett alakban tartalmaznak földrajzi adatokat. Az ***osm*** fájlokat könnyedén meg lehet nyitni akár a windows platformokon beépített jegyzettömbben is, hogy megtekintsük annak tartalmát, azonban nagyobb mennyiségű adatoknál speciális szoftvert igényelhet az adat megjelenítése.

6.1 Egy OSM fájl felépítése

Példa egy egyszerűbb ***osm*** fájlra az ***6.1.1 ábrán*** látható. Az alábbiakban csak a fontosabb elemeket fogom ismertetni, ami nem annyira érdekes a gráf építés szempontjából, azokat csak említés szinten érintem.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.8.3 (2505482 spike-08.openstreetmap.org)" copyright="OpenStreetMap and
contributors" attribution="http://www.openstreetmap.org/copyright"
license="http://opendatacommons.org/licenses/odbl/1-0/">
 <bounds minlat="47.4319600" minlon="19.0357000" maxlat="47.4337000" maxlon="19.0395800"/>
 <node id="60458391" visible="true" version="7" changeset="70295090" timestamp="2019-05-15T21:57:00Z" user="EXAMPLE"
uid="000000" lat="47.4310516" lon="19.0401763"/>
 <node id="61087603" visible="true" version="5" changeset="60066465" timestamp="2018-06-22T07:44:04Z" user="EXAMPLE"
uid="000000" lat="47.4329730" lon="19.0391089"/>
 <node id="8549755198" visible="true" version="1" changeset="101509419" timestamp="2021-03-22T13:47:33Z"
user="EXAMPLE" uid="000000" lat="47.4299109" lon="19.0377682">
  <tag k="railway" v="tram_crossing"/>
 </node>
...
 <way id="15139312" visible="true" version="20" changeset="101034662" timestamp="2021-03-15T08:40:17Z" user="EXAMPLE"
uid="000000">
  <nd ref="149598381"/>
  <nd ref="8520102693"/>
  <nd ref="1864908862"/>
  <nd ref="1864908861"/>
  <nd ref="680055315"/>
  <tag k="highway" v="secondary"/>
  <tag k="lanes" v="1"/>
  <tag k="lit" v="yes"/>
  <tag k="maxspeed:type" v="HU:urban"/>
  <tag k="name" v="Leányka utca"/>
  <tag k="oneway" v="yes"/>
  <tag k="surface" v="asphalt"/>
 </way>
 <way id="15219821" visible="true" version="17" changeset="93104200" timestamp="2020-10-27T08:20:21Z" user="EXAMPLE"
uid="000000">
  <nd ref="97933834"/>
  <nd ref="1864908845"/>
  <nd ref="1755364428"/>
  <nd ref="150550747"/>
  <tag k="highway" v="tertiary"/>
  <tag k="lanes" v="3"/>
  <tag k="lanes:backward" v="1"/>
  <tag k="lanes:forward" v="2"/>
  <tag k="lit" v="yes"/>
  <tag k="maxspeed" v="50"/>
  <tag k="noname" v="yes"/>
  <tag k="source:maxspeed" v="HU:urban"/>
  <tag k="surface" v="asphalt"/>
  <tag k="turn:lanes:forward" v="slight_left|right"/>
 </way>
...
 <relation id="12477407" visible="true" version="1" changeset="101509419" timestamp="2021-03-22T13:47:33Z"
user="EXAMPLE" uid="000000">
  <member type="way" ref="25581753" role="from"/>
  <member type="node" ref="428853680" role="via"/>
  <member type="way" ref="53864758" role="to"/>
  <tag k="restriction" v="no_right_turn"/>
  <tag k="type" v="restriction"/>
 </relation>
</osm>
```

6.1.1. ábra: Egy osm fájl felépítése.
(Megjegyzés: A fenti ábra csak egy példa, csupán ábrázolásképp hoztam létre.)

Az ***6.1.1 ábrán*** látható fájl legelső sorában [29] egy sima XML jelző látható, illetve a karakterkódolás. Ezt egy ***osm*** elem követi, ahol az API verziószáma található, illetve további, a fájlt jellemző információk. Az ***osm*** elemen belül találhatóak a gráf építés szempontjából fontos elemek, vagyis a pontok, az utak, illetve az ezek közötti összefüggések.

6.1.1 Node – (Pont)

„Point of Interest”-nek (POI) is szokás nevezni, vagyis érdekességi pontnak. [30,31] Az ***osm*** fájlok egyik központi eleme. Egy adott pontot határoznak meg a föld felszínén, a pont szélességi, és hosszúsági fokával, illetve egy ***ID***-t társítanak hozzá, ami minden pontnál egyedi, ezzel könnyen beazonosíthatóvá téve őket. Ezeken kívül más jelzői is

lehetnek egy ***node***-nak, de az előbb említett három tulajdonság mindig szükséges. Választható jelző lehet például egy pont magaslata, viszont gyakori, hogy a pontokhoz nincs magaslat rögzítve. Egy ***node***-ot fel lehet használni arra is, hogy a térképen különálló pontokat határozzunk meg, viszont egy ***way*** alakjának a meghatározásához is használható.

Az adatbázisban [31] lévő pontok különböző utakon, illetve kapcsolatokon kívül is megjelenhetnek. Ezek is ***node*** elemekként vannak tárolva az adatok között, viszont ilyenkor legalább egy ***tag***-gel is rendelkeznie kell, ami meghatározza, hogy mi a szerepe a térképen.

Az ***6.1.1 ábrán*** a harmadik ***node***-nál például látható egy ***tag***, ami azt határozza meg, hogy az a ***node*** egy villamos átkelőhely.

6.1.2 Tag - (Jelző)

Az egyes elemeknek lehetnek ***tag***-jeik. Ezeknek [30,32] a ***tag***-eknek van egy kulcsa, ami ***k*** betűvel van jelölve, és van egy értéke, ami ***v*** betűvel van jelölve, tehát kulcs-érték párokként lehet őket használni. Ezek a térkép elemeinek konkrét funkcióit, jellemzőit írják le, az ***6.1.1 ábrán*** például az első ***way*** elemnél egy ***tag*** mutatja meg hogy mi az adott út neve. Ezek segítségével jelezhető, ha egy adott ***node*** például egy lámpát reprezentál, de olyan dolgokat is lehet vele jelezni, mint az utaknál, hogy rendelkezik-e járdával, vagy hogy egyirányú-e.

6.1.3 Way - (Út)

A ***way***, a térkép egyik alapvető eleme. [30,33] Ezek általában olyan, a Földön elhelyezkedő lineáris függvényeket jelentenek, amik utakat, vagy folyókat, esetleg falakat ábrázolnak. Mindennapi nyelvhasználatban vonalként lehetne rájuk hivatkozni, de térképeknél nem ezt a jelzőt használják. Egy ***way*** technikailag rendezett ***node***-ok listája, és általában van legalább egy ***tag***-je is. Fontos megemlíteni, hogy egy út lehet nyílt, illetve zárt is.

Nyílt út (Open way): Egy nyílt útban [33] az első, illetve az utolsó ***node*** nem egyezik meg egymással. Pár általános példa lehet erre az autóutak többsége, vagy akár a gyalogos utak is, mivel ezek mindegyike egy adott pontból indul, és egy másik pontba érkezik. Az adatbázisban ezek az utak úgy vannak eltárolva, hogy mindig az első ponttól mutat az utolsó felé. Ez akkor is igaz, hogyha az adott út kétirányú, ilyenkor mindkét irányba fel van véve egy út. Tehát egy kétirányú út, két darab egyirányú útból van felépítve. Amennyiben egy út ténylegesen egyirányú, akkor annak rendelkeznie kell egy ***(oneway = yes) tag***-gel. Ha az út egyirányú, akkor ahogy az előbb is említettem, a pontok sorrendisége miatt meg lehet határozni, hogy melyik irányba halad. A ***6.1.1 ábrán*** lehet látni, hogy egy ***way***-nél a ***tag***-eket ***node***-ok előzik meg. Ezek sorrendisége miatt, az elemeken fentről lefelé haladva meg lehet határozni az út irányát.

Zárt út (Closed way): Egy zárt útban, [33] az első pont megegyezik az utolsó ponttal. Ezt általában területként lehet értelmezni, még akkor is, hogyha a ***tag***-ek között nincs ott az ***(area = yes)*** kulcs-érték pár. Lehetséges azonban, hogy egy zárt vonalláncként kell értelmezni. Amennyiben például egy (***barrier = *)*** jelzővel rendelkezik egy ***way***, az egy olyan kerítést, vagy hasonló akadályt jelent, amely valamilyen birtokot vesz körbe. Olyan esetekben amikor egy teret, vagy plázát szeretnénk megjeleníteni, lehet közösen használni a ***(highway = pedestrian)*** és az ***(area = yes)*** jelzőket.

6.2 Térkép adatok összefoglalása

Ezek ismeretében, a fenti elemek használatával fel lehet építeni egy olyan gráfot, amelyben a csúcsok, pontoknak felelnek meg, az élek pedig az utaknak. Mivel az alkalmazás elsősorban futáshoz készül, ezt a nagy halmazt még le kell szűkíteni olyan utakra, amik gyalogos használatra alkalmas jelzőket tartalmaznak. Az ***osm*** adatok között ***relation*** elemek, vagyis kapcsolatok is megtalálhatóak, viszont a gráf felépítéséhez elegendő a ***way***-ek, ***node***-ok, illetve a ***tag***-ek ismerete, ezért ezeket a ***relation*** elemeket nem ismertetem.

7. TÉRKÉP MEGJELENÍTÉSE

A térkép megjelenítéséhez valamilyen térkép szolgáltatóra lesz szükségem. Többféle térképszolgáltató is létezik, amiből hármat vizsgáltam meg. Az egyik ilyen térkép szolgáltató a Google. A Google Maps platform, [34] különféle API-kból és SDK-kból (szoftverfejlesztői csomagokból) épül fel, ezzel biztosítva a fejlesztőknek, hogy beágyazhassák a térképet a weboldalaiknál. Ennek a szolgáltatásnak nincsen ingyenes verziója, havonta 200$-ba [35] kerül a használata. A második térképszolgáltatás, amit megvizsgáltam, az a HERE által szolgáltatott Maps API volt. Ez egy interaktív [36] térkép, aminél lehetőség van saját adatokat megjeleníteni a térképen, és a használata is ingyenes, olyan formában, hogy például a térkép részeinek megjelenítésénél, havi szinten 30 ezer megjelenítésig [37] nem kell fizetni a szolgáltatásért. Ez a térkép viszont sajnos nem működik megfelelően az OpenStreetMap által szolgáltatott adatokkal, mivel egy saját adatbázist [38] használ a térkép megjelenítéséhez, ami miatt az ***osm*** adatok elcsúszhatnak a megjelenített térképen, illetve előfordulhat, hogy egyes utak meg sem jellenek. Harmadik szolgáltatásként pedig egy olyan szolgáltatót kerestem, ami kifejezetten OpenStreetMap adatokkal dolgozik, így a végleges választás a Mapbox térképszolgáltatóra esett. Ez a szolgáltatás havi korlátozással ingyenesen használható, [39] és a mögötte lévő adatok [40] mind az OpenStreetMap adatbázisból származnak.

7.1 Mapbox GL JS

A mapbox, egy olyan kliens oldali javascript könyvtár, [41] aminek a segítségével olyan webes alkalmazásokat lehet csinálni, amik a mapbox modern térképészeti technológiája által készített térképeket használnak. A Mapbox GL JS segítségével olyan interaktív térképeket lehet megjeleníteni egy böngészőben, amit a felhasználó könnyedén tud használni.

A Mapbox GL JS sok mindenre felhasználható, [42] például földrajzi adatok megjelenítésére, vagy egyedi jelzőpontok elhelyezésére. Ezen kívül lehet saját, egyedi kliens adatoknak a dinamikus megjelenítésére, és formázására használni, illetve lehetőség van arra is, hogy az adatokat elhelyezzük egy Mapbox stílus rétegei közé.

Ezek mind nagyon hasznos funkciók, mivel nem a Mapbox által beépített útvonaltervezési algoritmusokkal fogok dolgozni, hanem a saját programomban implementált útvonaltervezővel. Mivel a saját kódom által generált adatokkal fogok dolgozni, a felsorolt funkciók nagy részére szükségem lesz, hogy ezeket az egyedi adatokat meg tudjam jeleníteni a térképen. Ezen felül a felhasználói interakcióra is szükségem lesz, mivel a felhasználónak kell majd a térképen kiválasztania, hogy mely pontok között szeretné legenerálni az útvonalat.

A mapbox-szal kapcsolatos kulcsfontosságú gondolatok:

A mapbox GL JS elnevezésben lévő GL [43] rész, a mapbox által használt grafikus könyvtárra utal, aminek a segítségével pluginok nélkül lehet renderelni 2D-s vagy 3D-s mapbox térképeket a webböngészőben.

A mapbox GL JS kliens oldali renderelésre támaszkodik. Ezek a térképek úgy renderelődnek dinamikusan, hogy a térképek [43] megjelenítésére használt vektor lapkákat, a böngészőben lévő formázási szabályokkal kombinálja, nem pedig szerver oldaliakkal. Emiatt lehetséges, hogy a térképen megjelenített adatok, illetve a térkép stílusa dinamikusan változzon, felhasználói interakció hatására.

A Mapbox GL JS projektek alapja, [43] a ***mapboxgl.Map*** osztály. A térkép használatához, ezt az osztályt kell használni, és a ***7.1.1 ábrán*** lévő kód segítségével lehet inicializálni magát a térképet.

```
mapboxgl.accessToken = '<your access token here>';
const map = new mapboxgl.Map({
    container: 'map', // container ID
    style: 'mapbox://styles/mapbox/streets-v11', // style URL
    center: [-74.5, 40], // starting position [lng, lat]
    zoom: 9 // starting zoom
});
```

7.1.1 ábra: Kódrészlet a térkép inicializálásához kapcsolódóan. [43]

A [43] térképnek szüksége van egy ***access token***-re, amit ingyenesen lehet generálni egy regisztrált Mapbox fiókkal. A ***container*** annak a HTML elemnek az azonosítóját tartalmazza, amiben meg szeretnénk jeleníteni a térképet. A ***style*** részen meg tudjuk adni, hogy melyik mapbox stílust szeretnénk használni, a ***center***, illetve a ***zoom*** részeken pedig a megjelenített térkép kezdő pozícióját, illetve a kezdeti nagyítás mértékét lehet megadni.

Egy mapbox GL JS térkép [43] rengeteg rétegből, vagyis ***layer***-ből állhat, ezek a rétegek pedig különféle vizuális elemeket, vagy térkép adatokat tartalmazhatnak. Minden egyes réteg szabályokat tartalmaz arról, hogy a renderelőnek hogyan kell az egyes elemeket kirajzolnia a böngészőben, a renderelő pedig ezen layer-ek által rajzolja ki a térképet. Lehetőség van új layer-eket hozzáadni a térképhez, amiket az ***addLayer*** funkcióval lehet megtenni. Ezek a layer-ek aszinkron használatra is alkalmasak, ezért a Mapbox GL JS alkalmazásoknál gyakran eseményekhez kötik a térkép változását. Ilyen esemény lehet például a térkép ***load***, vagy ***click*** eseménye.

8. TERVEZÉS

8.1 Specifikáció

A szakdolgozatomnak egy Frontend-del, illetve egy Backend résszel is rendelkeznie kell. Frontendre azért lesz szükségem, hogy a felhasználó kommunikálni tudjon az alkalmazással, illetve, hogy a generált útvonalakat meg tudjam valahol jeleníteni. Mivel Budapest tértképén szeretnék útvonalakat generálni, ezért egyből itt kellene megjelennie a térképnek. Itt a felhasználónak lehetősége lesz arra, hogy pontokat rakjon le a térképre, vagy véletlenszerű útvonal generálás esetén értéket írjon be egy mezőbe, és egy gomb segítségével elkezdje legenerálni a kívánt útvonalat. A gomb megnyomásakor REST API híváson keresztül fognak eljutni az adatok a Backend felé, ahol a megkapott adatok alapján megtörténik az útvonal generálás. A Backend feladata felépíteni a gráfot, illetve legenerálni a két pont közötti legrövidebb útvonalat, vagy egy pontból véletlenszerűen generálni, adott hosszúságú útvonalat. A gráf felépítésének külön kell működnie az útvonal generálástól, mivel ez egy hosszú folyamat, ami több órát is igénybe vehet. Ha a gráf felépítése megtörtént, akkor azt el kell tárolni, hogy útvonal generálásnál gyorsan be lehessen tölteni a már felépített gráfot. Útvonal generáláshoz összesen három darab algoritmust fogok implementálni, amihez még tartozni fog két alternatív verzió is. Az első algoritmus a Dijkstra algoritmus, a második az A* algoritmus, a harmadik pedig egy véletlenszerű útvonal generáló algoritmus lesz. Az alternatív algoritmusokra azért lesz szükség, hogy előnyben részesítsenek olyan útvonalakat, amelyek nem autóutak melletti járdákból épülnek fel, még akkor sem, ha az lenne a legrövidebb út. Alternatív verziója csak az A* algoritmusnak, illetve a Dijsktra algoritmusnak lesz. E közül az öt algoritmus közül majd Frontenden választhat a felhasználó, hogy melyiket szeretné használni a generáláshoz. Az útvonal generáláson kívül az is feladata lesz a program Backend részének, hogy a generált út hosszát kiszámítsa, illetve az egyes pontok közötti fordulási irányokat meghatározza. Amennyiben ezek mindegyikével elkészült, és előállt a kész útvonal, akkor REST API-n keresztül elküldi az útvonal adatait a Frontend-nek. Ha az útvonal generálás nem sikerül, arról értesíteni kell a felhasználót.

8.2 Térkép adatbázis

Térkép adatbázisnak az OpenStreetMap adatbázisát választottam, azért, mert ingyenes a hozzáférés, és elérem Budapest teljes térképét. Ez egy ***osm*** kiterjesztésű fájl, aminek a felépítése lényegében egy xml dokumentumhoz hasonlít. Mivel egy ilyen térkép adatbázisban rengeteg elem található, az épületektől kezdve, a jelzőlámpákig, nem lesz minden elemre szükségem. Emiatt szűkíteni kell, hogy milyen adatokat szükséges feldolgozni a fájlból. Csak az utakat, illetve az ezeket felépítő pontokat kell eltárolni, de ezeket az elemeket is tovább kell majd szűkíteni, hogy csak gyalogos használatra alkalmas adatok legyenek eltárolva a memóriában.

A térkép adatoknál két fontos kifejezést szeretnék ismertetni, amiket a továbbiakban használni fogok.

Junction: Útkereszteződés [44] amivel két utat lehet egymással összekapcsolni. A szakdolgozatom további részében, a **Junction** megnevezés az egyes gráf pontokat fogja jelenteni.

Segment: Régebben két pontot kötött össze, [45] ezzel egy utat reprezentálva. 2007-ben el lettek távolítva, hogy az adatmodell-t egyszerűsítsék. A szakdolgozatom további részében, a **Segment** megnevezést két **Junction** közötti út leírásaként fogom használni.

8.3 Útvonaltervezés

Két pont közötti útvonal generáláshoz a Dijkstra algoritmust, illetve az A* algoritmust fogom implementálni. Ezeket az algoritmusokat módosítanom kell olyan formában, hogy ne ***Junction***-ök között történjen meg az útvonal generálás, hanem ***Segment***-ek között. Erre azért van szükség, mert az alternatív algoritmusok működésénél a ***Segment***-nél eltárolt adatok alapján lehet manipulálni az egyes elemek súlyát. Két pont közötti generáláson kívül véletlenszerű útvonalak generálására is lehetőség lesz, ami külön algoritmussal fog működni.

Sok olyan út lesz, ahol a felhasználó egy járdával rendelkező autóúton tud közlekedni, ezért egy alternatív verziót is készítek a két pont között generáló algoritmusokhoz. Ez a verzió előnyben fogja részesíteni azokat az utakat, amik nem járdával rendelkező autóutakon haladnak.

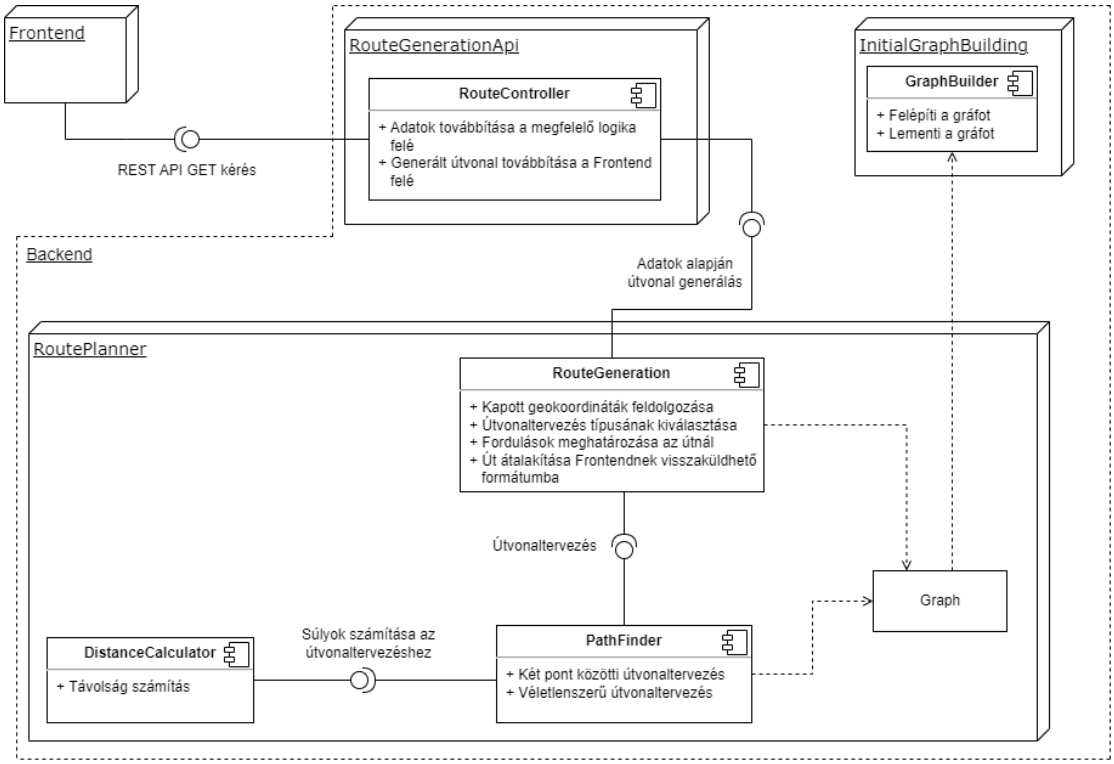
Az útvonal generáláshoz a súlyok számítására is szükség lesz. Mivel a legrövidebb utakat szeretném legenerálni, ezért az algoritmusokhoz használt súlyszámításokhoz az elemek közti távolságot fogom alapul venni.

8.4 Programterv

A program több részből fog állni. Szükségem lesz egy Frontend részre, ahol a térképpel interakcióba tud lépni a felhasználó. Szükségem lesz egy Backend részre, ahol a gráf felépítése, és maga az útvonaltervezés történik. A program Backend részét további részekre kell osztani, egy gráf építést megvalósító részre, egy útvonal generálást megvalósító részre, és egy kommunikációs részre. A Backend, és a Frontend közötti kommunikációt is ki kell alakítani, amit REST API hívások segítségével fogok biztosítani.

8.5 Backend

A Backend C# programnyelven lesz megvalósítva, és három részből fog állni. Kell egy projekt, ami a gráf felépítéséért felel, kell egy másik projekt, ami az útvonaltervezésért felel, és kell egy harmadik projekt, ami képes kommunikálni REST API hívásokon keresztül a szoftver Frontend részével.



8.5.1. ábra: A rendszer komponens diagramja

A gráf felépítéséért felelő ***InitialGraphBuilding*** alkalmazás, egy konzol alkalmazás lesz, amit elindítva el lehet kezdeni a gráf felépítését, és nyomon lehet követni annak folyamatát. Ehhez szükség lesz egy ***Way*** osztályra, ahol az ***osm*** adatok közül kinyert ***way*** elemek adatait lehet tárolni. Ezen az objektumon kívül szükség lesz egy **Segment** osztályra, ahol a szomszédos ***node***-ok közti utakat tárolom el, illetve egy ***Junction*** osztályra, ahol a ***node*** elemek értékeit tárolom el. Ezekből az elemekből fog felépülni a gráf, amit el is kell menteni az alkalmazás futásának végén. A gráf felépítésekor ügyelni kell arra, hogy a megfelelő kapcsolatok kiépüljenek az egyes ***Junction*** és ***Segment*** elemek között. A gráf felépítését azért kell külön venni az útvonaltervezéstől, mert sokáig tarthat egy nagyobb térkép adatbázis feldolgozása és gráffá alakítása. A gráfra szükség van az útvonaltervezéshez, ezért a gráf felépítésére egy előfeldolgozásként lehet tekinteni. A gráf felépítésének nem kell minden útvonaltervezésnél futnia, elég csak egyszer felépíteni egy térképet, és onnantól a felépített térképpel már tud dolgozni a ***RoutePlanner***.

Az útvonal generálásért a ***RoutePlanner*** fog felelni, ami egy ***ClassLibrary*** lesz. Ennek a működéséhez el kell tárolni a memóriában a gráfot felépítő elemeket. Ehhez szükségem lesz egy ***Graph*** osztályra, amibe be lehet tölteni a gráfépítő alkalmazás által készített ***Segment***, illetve ***Junction*** elemeket. A ***Graph*** osztályban ezért ***Segment***, illetve ***Junction*** elemeket tároló listák lesznek. A ***RoutePlanner***-ben a ***RouteGeneration*** osztály fog kommunikálni a ***RouteController***-rel. A ***RouteGeneration*** feladata, hogy a Frontend-től kapott adatokat feldolgozza, és a

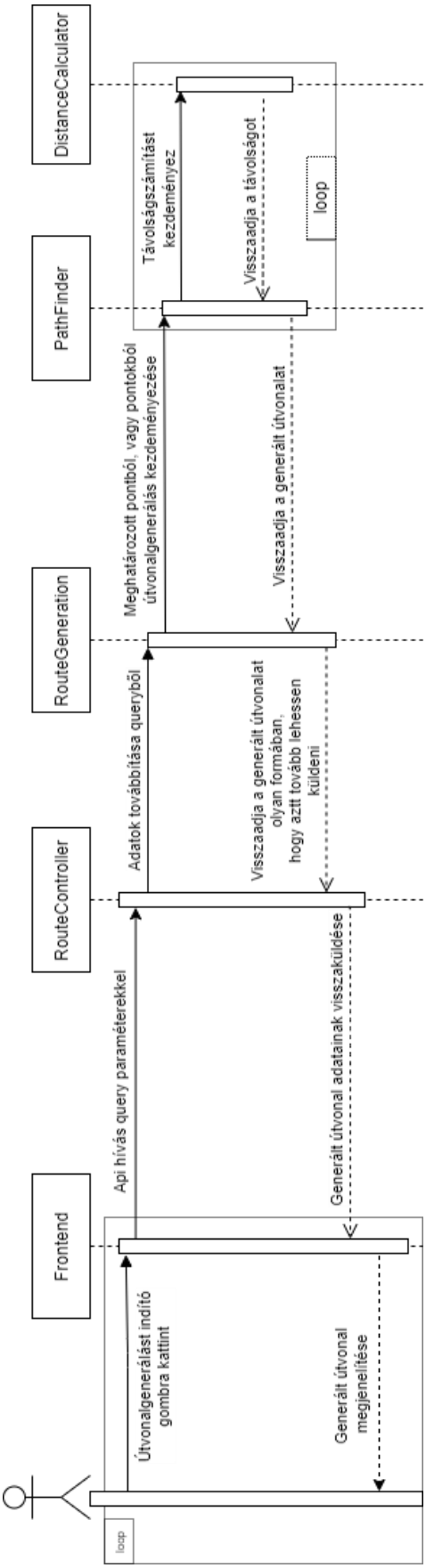
megfelelő átalakítások, illetve számítások után elindítsa az útvonaltervezést. Az útvonaltervezés eredménye egy ***Path*** objektum lesz, amiben az útvonalat reprezentáló ***Segment***-ek listája lesz benne, illetve az útvonal teljes hossza. Ezt a ***Path*** objektumot a ***RouteGeneration*** feladata átalakítani olyan formában, hogy azt a ***RouteController*** json objektumként vissza tudja küldeni a Frontend számára. Az így átalakított adatok tárolásáért pedig a ***RouteDto*** fog felelni. Az útvonaltervezésért a ***PathFinder*** osztály lesz felelős. Itt lesznek eltárolva az útvonaltervező algoritmusok, amik a gráf adatainak segítségével le tudják generálni az egyes útvonalakat. Az útvonaltervezéshez szükséges frontier használatához távolságokat kell majd számítani a ***Segment*** elemek között, ezért szükség lesz a ***DistanceCalculator*** osztályra is. Ez ***Segment***-ek, illetve ***Junction***-ök közötti távolságok számításáért felel. Az alternatív útvonaltervezés miatt szükségem lesz egy ***PathPart*** osztályra is, amiben az egyes ***Segment***-ek közötti tényleges távolságot tudom majd eltárolni. Erre azért lesz szükség, mert az alternatív útvonaltervezésnél nem mindig a ***Segment***-ek közötti tényleges távolságot fogom használni, viszont a felhasználónak az útvonal tényleges hosszát kell visszaadnom.

Mivel Frontend-től kapja az adatokat a Backend, szükség lesz egy ***RouteGenerationApi*** elnevezésű ASP.NET projektre. Itt a Controller segítségével tud kommunikálni a Backend a Frontend-del. A ***RouteController*** lesz az a része a programnak, ami a REST API hívásokat fogja megkapni, és a GET kérésekben lévő adatokat továbbítja a ***RoutePlanner*** felé, illetve a ***RoutePlanner***-től kapott generált út adatait továbbítja a Frontend felé.

8.6 Frontend

A Frontend a felhasználóval történő kommunikációt fogja biztosítani. Szükséges, hogy a felhasználó le tudjon rakni pontokat a térképre. Ezekkel a pontokkal határozza meg, hogy honnan hova szeretne útvonalat generálni. Mivel valamekkora távolsággal véletlenszerű útvonal generálására is szükség van, kell egy input mező is, amiben a felhasználó meg tudja adni mekkora véletlenszerűen generált távot szeretne futni. Amennyiben a térképre két pont van lerakva, viszont a generálni kívánt út hossza is meg van adva, akkor a Backend dolga lesz kezelni, hogy a generálni kívánt út értékét ne vegye figyelembe. Az útvonal generálást egy gomb segítségével lehet majd elindítani. A Backend-től válaszul kapott adatokat meg kell jeleníteni. A generált út távolságát, illetve a fordulások értékeit a lap tetején fogom megjeleníteni, amíg nem történt útvonal generálás, addig ezek alapértelmezetten üres értékek lesznek. A generált út megjelenítését egy színátmenetes vonallal fogom reprezentálni, amin a fordulatok helyei szürke pontokkal lesznek jelölve. Ha a felhasználó új útvonalat generál, akkor a már megjelenített útvonalnak, a fordulási pontokkal együtt el kell tűnnie, hogy csak az új generált útvonal legyen látható. Az útvonal generálás sikerességéről, vagy esetleges hibák előfordulásáról is értesülni fog a Frontend, amit a lap felső részének közepén fogok megjeleníteni, hogy jól látható legyen, ha valami baj történt a generálás során. A ***7. fejezetben*** említettem a MapBox-ot, mint térképszolgáltatást. Rengeteg React

nyelven írt MapBox dokumentáció található az interneten, amik segítségével a fentebb leírt elvárások mindegyikét meg lehet oldani, ezért a Frontend megvalósítását React keretrendszer segítségével fogom megoldani.



8.1. ábra: A program szekvencia diagramja

9. MEGVALÓSÍTÁS

9.1 Gráf felépítése

A gráf felépítése az ***InitialGraphBuilding*** projektben történik. Az útvonaltervezőhöz szükséges gráf felépítéséhez egy OpenStreetMap által szolgáltatott ***osm*** kiterjesztésű fájlt használtam, amiben egész Budapest térképe el van tárolva. Ahogy azt a ***8.3 fejezetben*** leírtam, ez a fájl önmagában még nem használható gyalogos, illetve futó útvonalak tervezésére. Le kell szűkíteni a feldolgozandó adatok halmazát olyan elemekre, amik ténylegesen relevánsak az útvonaltervezőhöz, és csak az ezek közötti kapcsolatokat kell felépíteni.

Ehhez a szűréshez az ***1. kódrészleten*** látható feltételeket alkalmaztam:

```
var osm_ways = from way in xdoc.Root.Descendants("way")
               where way.Elements("tag").Attributes("v").All(x => x.Value != "private")
                     && (way.Elements("tag").Attributes("k").All(x => x.Value != "psv") ||
                         way.Elements("tag").Where(e => e.Attribute("k").Value == "psv")
                                            .Single().Attribute("v").Value == "no")
                     && (way.Elements("tag").Attributes("k").All(x => x.Value != "access") ||
                         way.Elements("tag").Where(e => e.Attribute("k").Value == "access")
                                            .Single().Attribute("v").Value != "no")
                     && (way.Elements("tag").Attributes("v").Any(x => x.Value == "footway")
                     || way.Elements("tag").Attributes("k").Any(x => x.Value.Contains("footway"))
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "steps")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "service")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "path")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "track")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "cycleway")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "pedestrian")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "unclassified")
                     || (way.Elements("tag").Attributes("k").Any(x => x.Value.Contains("sidewalk")) &&
                         way.Elements("tag").Where(e => e.Attribute("k").Value.Contains("sidewalk"))
                                            .Any(x => x.Attribute("v").Value != "no"))
                     || (way.Elements("tag").Attributes("v").Any(x => x.Value == "residential") &&
                         way.Elements("tag").Where(e => e.Attribute("v").Value == "residential")
                                            .All(e => e.Attribute("k").Value == "highway") &&
                        (way.Elements("tag").Attributes("k").All(x => !x.Value.Contains("sidewalk")) ||
                         way.Elements("tag").Where(e => e.Attribute("k").Value.Contains("sidewalk"))
                                            .Any(x => x.Attribute("v").Value != "no")))
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "platform")
                     || (way.Elements("tag").Attributes("v").Any(x => x.Value.Contains("tertiary")) &&
                        (way.Elements("tag").Attributes("k").All(x => !x.Value.Contains("sidewalk")) ||
                         way.Elements("tag").Where(e => e.Attribute("k").Value.Contains("sidewalk"))
                                            .Any(x => x.Attribute("v").Value != "no")))
                     || (way.Elements("tag").Attributes("k").Any(x => x.Value == "foot") &&
                         way.Elements("tag").Where(e => e.Attribute("k").Value == "foot")
                                            .Single().Attribute("v").Value != "no")
                     || way.Elements("tag").Attributes("v").Any(x => x.Value == "bridleway"))
               select new Way()
               {
                   ID = way.Attribute("id").Value,
                   node_id_reference = way.Elements("nd").Attributes("ref").Select(x => x.Value).ToList(),
                   Footway = way.Elements("tag").Attributes("v").Any(x => x.Value == "footway"
                                                                       || x.Value == "steps"
                                                                       || x.Value == "path"
                                                                       || x.Value == "track"
                                                                       || x.Value == "cycleway"
                                                                       || x.Value == "pedestrian")
                             ||
                             way.Elements("tag").Attributes("k").Any(x => x.Value.Contains("footway")
                                                                       || (x.Value == "foot"
                                                                           && way.Elements("tag")
                                                                                 .Where(e => e.Attribute("k").Value == "foot")
                                                                                 .Single().Attribute("v").Value != "no"))
               };
```

1. kódrészlet: Megfelelő utak szűrése az osm adathalmazból

A térképadatok beolvasásához az ***XDocument*** osztályt használom, mivel ennek segítségével nagyon gyorsan fel lehet dolgozni xml felépítésű fájlokat. A ***8.3 fejezetben***

leírtam, hogy az ***osm*** fájlban a ***way*** elemek tartalmazzák az őket felépítő ***node*** elemek azonosítóit, ezért elég volt ezeket az elemeket kinyernem a fájlból. Az utak további szűkítéséhez pedig ***tag***-eket használtam, hogy csak a gyalogos használatra alkalmas adatokat kapjam meg. A szűrési feltételeknél van, ahol elegendő a ***tag***-ek kulcs-érték párjai közül csak a kulcsot, vagy csak az értékét megvizsgálni. Olyan eset is van, amikor a kulcs-érték párok együttes állapota határozta meg, hogy használható-e futáshoz vagy gyaloglóshoz az adott út, például a ***sidewalk*** értékű ***tag***-nél, ahol nem elég a kulcs létezése az elemnél, mert annak értéke ***no*** is lehet, ami pont azt jelzi, hogy nincs járda az adott úthoz. A ***10.1.1. ábrán*** látható, hogy a privát utak, a tömegközlekedési utak, illetve a nem használatba vehető utak azonnal szűrésre kerülnek. Ha a kezdeti szűrésen túl jut egy ***way*** elem, akkor viszont a ***10.1.1. ábra*** további részén látható szűrések alapján is meg kell vizsgálni, hogy alkalmas-e gyalogos közlekedésre. Itt olyan szűrések láthatóak, hogy például járda, ösvény, bicikli út, esetleg parkoló részhez tartozó út-e az adott ***way*** elem, de ezeken a példákon kívül több is leolvasható a ***10.1.1 ábráról***.

Minden egyes releváns elemhez készítek egy új ***Way*** objektumot, amiben eltárolom az út azonosítóját, az úthoz tartozó ***node***-ok azonosítóit, illetve egy logikai értéket. A ***way*** elemekhez tartozó ***node***-okat egy listában tárolom el, hogy megmaradjon a ***node*** elemek közötti kapcsolatok sorrendje. A logikai értéket pedig a ***tag***-ek segítségével határozom meg, hogy útvonal generálás közben előnyben tudjam részesíteni azokat az utakat, amik nem járdák.

Miután megvannak azok a ***way***-ek, amik eleget tettek a szűrésnek, végig iterálok rajtuk, és minden olyan ***node*** azonosítót eltárolok egy hash tömbbe, ami megjelenik ezeknek az utaknak a pont azonosító listájában. Ehhez ***HashSet***-et használok, ahol le van kezelve az egyediség, ezért ugyanaz az azonosító nem kerülhet be kétszer a tömbbe.

```
var osm_nodes = from node in xdoc.Root.Descendants("node")
                where hashNodes.Contains(node.Attribute("id").Value)
                select new Junction()
                {
                    ID = node.Attribute("id").Value,
                    myNumId = juncidx++,
                    Lattitude = node.Attribute("lat").Value,
                    Longitude = node.Attribute("lon").Value
                };
```

2. kódrészlet: Megfelelő pontok szűrése az osm adathalmazból, a már leszűrt utak segítségével

Ezután a ***2. kódrészlet*** alapján, csak azokat a ***node***-okat veszem fel a ***Junction***-jeim közé, amik megtalálhatóak valamelyik releváns ***way***-ben. Itt nagyon hasznos a ***HashSet*** használata, mivel gyorsan meg lehet keresni a hashelt indexek alapján, hogy benne van-e az adott ***node*** a tárolóban.

Minden ponthoz létrehozok egy ***Junction*** objektumot, amiben az azonosítóját, és a geo koordináta adatait tárolom el, illetve egy saját index értéket is, ami a gráf betöltésekor lesz hasznos.

A gráf felépítésénél, a kigyűjtött ***Junction***-ökön iterálok végig. Minden ***Junction***-höz megnézem, hogy melyek azok a ***Way***-ek, ahol az azt felépítő ***Junction***-ök azonosítói között, szerepel az éppen vizsgált ***Junction*** azonosítója. Amennyiben egy ***Way***-t felépítő ***Junction*** azonosítók listájában megtalálható a vizsgált ***Junction*** azonosítója, akkor az indexek alapján meg kell vizsgálni, hogy hol helyezkedik el ebben a listában. Amennyiben a ***Junction***, a lista legelső eleme, nem lehet előtte más ***Junction***, amivel kapcsolatban állna, ugyanakkor, ha az utolsó elem lenne ebben a listában, akkor pedig utána nem lenne már más elem. A vizsgált ***Junction***, és szomszédai között ***Segment*** elemeket hozok létre, amiket egy listába rakok bele. ***Segment***-ek létrehozásakor, mindig a vizsgált ***Junction*** lesz a ***Segment*** első pontja, és a szomszédos ***Junction*** lesz a második pontja. Itt megemlíteném, hogy az OpenStreetMap adatbázisban előfordulnak olyan utak, amelyek nincsenek kétirányúként rögzítve, gyalogos útvonal generálásnál viszont ez nem okozhat problémát, tehát azokat a ***Segment***-eket, amiket itt legenerálok, fel kell vennem fordítva is. Emiatt, amikor létrehozok egy ***Segment*** elemet, annak a fordítottját is fel kell vennem egy másik listába.

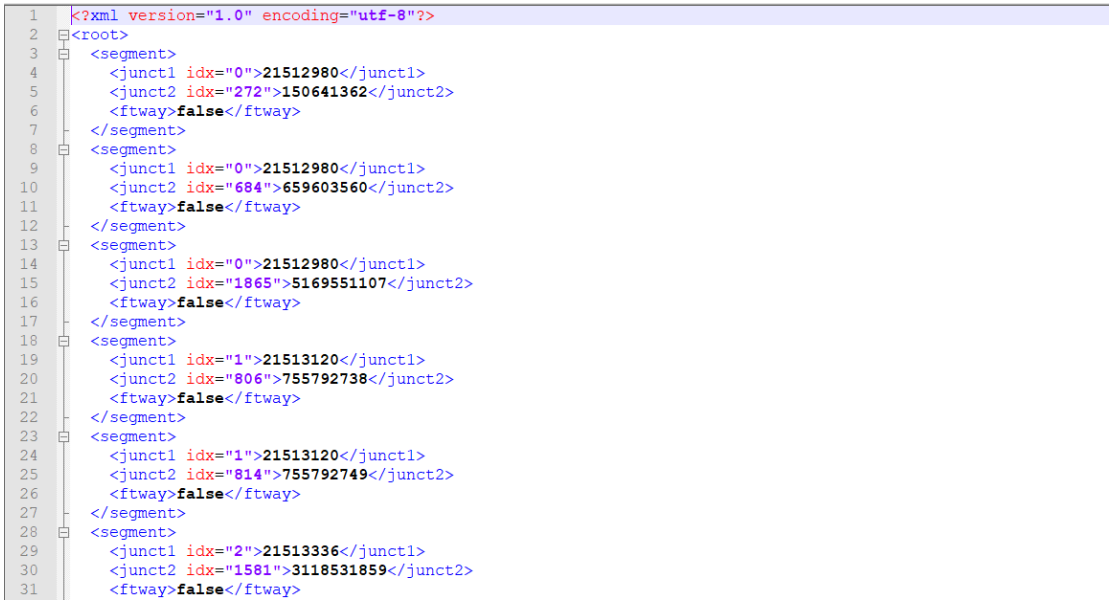
A generálás lefutásának végeztével két darab ***Segment*** lista lesz. Ebben nagyon sok lesz a duplikált adat, amire az összefésülésnél ügyelni kell, hogy csak egyszer szerepeljen a végleges gráfban. Mivel fontos a sorrendiség a gráf felépítésénél, az eredetileg létrehozott ***Segment*** listának az elemeit átmásolom egy ***origSegments*** nevezetű tömbbe, majd az eredeti lista tartalmát kitörlöm. Az eredeti lista tartalmát azért törlöm ki, mert a ***Junction***-öknél el fogom tárolni, hogy a ***Segment***-ek listájában hányadik indextől, hány darab ***Segment*** tartozik a szomszédai közé. Itt fontos a sorrendiség, ezért az egyes ***Junction***-öket megvizsgálva töltöm fel újra az eredeti ***Segment*** listát.

Az ***origSegments,*** illetve a fordítva felvett ***Segment*** listák összefésülésénél újra a felvett ***junction***-ökön kell végig iterálni. Az egyes ***Junction***-ökhöz először be kell állítani, hogy hányadik indextől kezdődnek a hozzá tartozó ***Segment***-ek a listában. Ez az érték az első ***Junction***-nél 0 lesz, utána ez az érték mindig növelve lesz az aktuális ***Junction***-höz tartozó ***Segment***-ek a számával. Az index beállítása után, meg kell keresni azokat a ***Segment***-eket, amiknek az első pontja az éppen vizsgált ***Junction***. Ezeket a ***Segment***-eket az ***origSegments*** tömbből is meg kell keresni, és a visszafelé generált ***Segment*** listából is meg kell keresni. Az így kapott ***Segment***-ek között nagy lesz az átfedés, ezért az uniójuk eredményét kell felvenni az eredeti ***Segment*** listába. Amikor az unió által megkapott ***Segment***-ekkel újra feltöltöm az eredeti listát, megnövelem a ***Junction WayCount*** tulajdonságának értékét is annyival, amennyi ***Segment*** elemet felveszek a listába. Ha a ***Segment***-ek felvétele megtörtént a listába, akkor ennek a ***WayCount***-nak az értékét hozzáadom egy változóhoz, ami alapján meg lehet határozni, hogy a következő ***Junction***-höz hányadik indextől lesznek eltárolva a ***Segment***-ek a listában.

A gráf felépítésekor fontos a ***way***-eken belüli ***node***-ok sorrendisége. Ehhez nem tudok Hash-t használni, mivel ott ez a sorrendiség nem fog megmaradni, ezért sima listát kellett használnom. Két listában való keresés és ezeknek az uniója is egy lassabb futásidővel rendelkező művelet ekkora mennyiségű adatnál. A gráf felépítésének ideje a ***10.1 fejezetben*** látható. Elfogadhatatlan, hogy a program használata közben minden út generálásánál több óráig épüljön a gráf, ezért a gráf felépítését külön vettem az útvonaltervezéstől. A fenti módszer alapján, miután le lett generálva a gráf, elmentem egy xml felépítésű fájlba, a ***SaveBuiltGraph*** metódus segítségével. Mivel a ***Junction***-ökre is külön szükségem van, és a ***Segment***-ekre is, ezért két darab ilyen fájlt generálok, ***GraphJunctions.xml***, és ***GraphSegments.xml*** néven. Ezeknek a fájloknak a felépítése a ***9.1.1, és 9.1.2 ábrákon*** látható.

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <junction>
    <idx>0</idx>
    <id>21512980</id>
    <lat>47.4290607</lat>
    <lon>19.0413709</lon>
    <start>0</start>
    <wcount>3</wcount>
  </junction>
  <junction>
    <idx>1</idx>
    <id>21513120</id>
    <lat>47.4302900</lat>
    <lon>19.0410608</lon>
    <start>3</start>
    <wcount>2</wcount>
  </junction>
  <junction>
    <idx>2</idx>
    <id>21513336</id>
    <lat>47.4246201</lat>
    <lon>19.0418634</lon>
    <start>5</start>
    <wcount>2</wcount>
  </junction>
```

9.1.1. ábra: A lementett Junction-öket tartalmazó file felépítése


<?xml version="1.0" encoding="utf-8"?>
<root>

<junct1 idx="0">21512980</junct1>
<junct2 idx="272">150641362</junct2>
<ftway>false</ftway>

<junct1 idx="0">21512980</junct1>
<junct2 idx="684">659603560</junct2>
<ftway>false</ftway>

<junct1 idx="0">21512980</junct1>
<junct2 idx="1865">5169551107</junct2>
<ftway>false</ftway>

<junct1 idx="1">21513120</junct1>
<junct2 idx="806">755792738</junct2>
<ftway>false</ftway>

<junct1 idx="1">21513120</junct1>
<junct2 idx="814">755792749</junct2>
<ftway>false</ftway>

<junct1 idx="2">21513336</junct1>
<junct2 idx="1581">3118531859</junct2>
<ftway>false</ftway>

9.1.2. ábra: A lementett Segment-eket tartalmazó file felépítése

Az általam generált indexek, amiket a pontokhoz vettem fel, itt lesznek hasznosak. Azzal, hogy a ***Segment***-ekhez el tudom menteni, hogy a ***Junction*** listán belül hányadik elem tartozik hozzá index szerint, nagyban gyorsítja a gráf felépítését. Így, hogy a felépített gráfot lementem egy fájlba, és onnan töltöm be az adatokat, egész Budapest térképét pár másodpercig tart betölteni.

9.2 Frontend

Az alkalmazás frontend részét React-tel készítettem el, a Mapbox által szolgáltatott ingyenes interaktív térkép segítségével, amiről a ***7. és 8.6 fejezetben*** írtam.

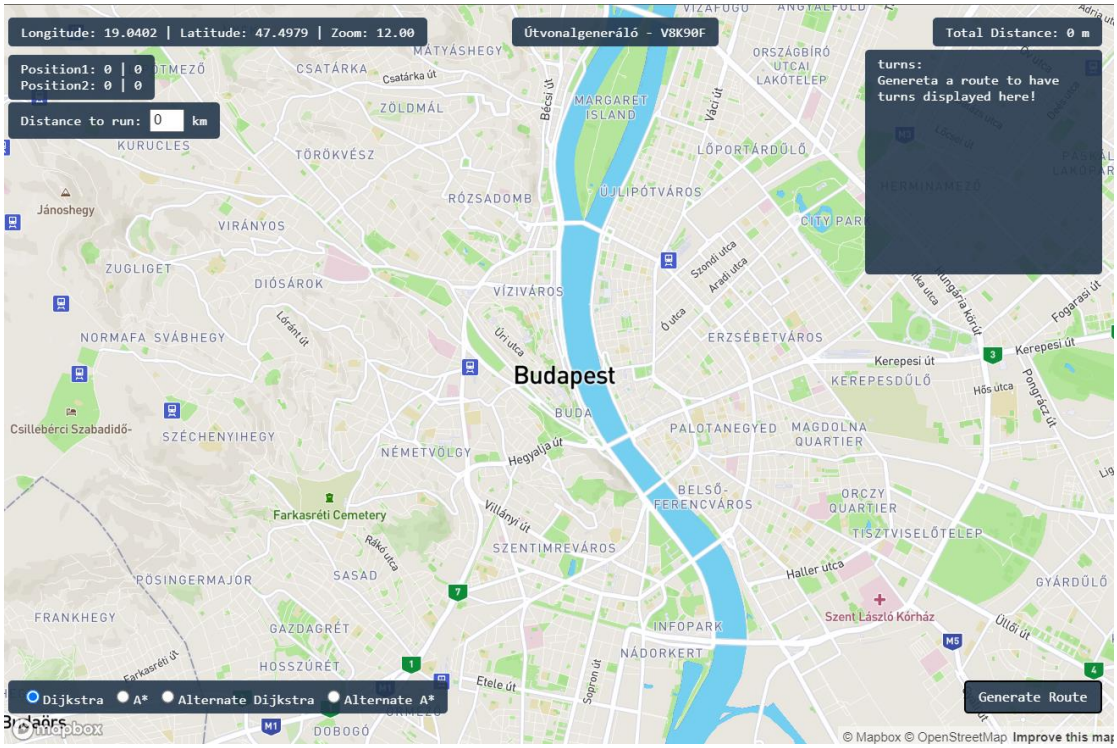
A frontend oldal megnyitásakor, a ***9.2.1 ábrán*** látható térkép jelenik meg. Az oldal bal felső sarkában a megjelenített térkép aktuális középpontjának a szélességi és a hosszúsági fokai láthatóak, illetve a nagyítás mértéke. Ezek alatt az értékek alatt találhatóak a felhasználó által elhelyezett pontok koordinátái. Ezeknek a kezdő értéke 0, de ha a felhasználó valahol kattint a térképen, akkor a kattintás helyének geo koordinátái lesznek betöltve a pozíciókhoz. Ezeken kívül az oldal bal alsó sarkában elhelyeztem az egyes algoritmusokhoz tartozó radio button-okat is, illetve az oldal jobb felső részénél helyeztem el az útvonal hosszához és az úton történő fordulásokhoz tartozó elemeket. A pontok lerakását, illetve a felület kezelését a ***12. fejezetben*** lévő felhasználói segédletnél részletezem.

A térkép jobb alsó sarkában, egy gomb helyezkedik el, amivel REST API-n keresztül GET kérést lehet kezdeményezni az alkalmazás Backend része felé. A választ megkapva pedig azonnal meg lesz jelenítve a generált útvonal a térképen.

Az oldalon ***hook***-ok segítségével minden adat változásakor frissülnek a megfelelő elemek. ***useEffect*** funkciók segítségével a térkép egyes eseményeire rá tudtam csatolni

a megfelelő működéseket. Így oldottam meg például, hogy kattintáskor megjelenjenek a pontok a megfelelő koordinátákon, vagy a térkép mozgatásakor frissüljenek a koordináta értékek az oldal tetején.

A kommunikációhoz REST API GET kéréseket használtam, aminél a Backend „query string”-en keresztül kapja meg a kérés paramétereit, visszafelé pedig ***json*** formátumban küldi az adatokat. A térkép ***load*** eseményében ki lehet rajzolni bizonyos útvonalakat, amik koordináta párok sorozatát várják. Ennek a kezdeti értéke egy üres tömb, ezért a térkép betöltéskor nem jelenít meg semmit, azonban az API hívás után ez feltöltődik elemekkel, majd a ***hook*** felépítés miatt, a változást érzékelve a ***load*** esemény lefut, és kirajzolódik a térképen az útvonal.



9.2.1. ábra: A Frontend oldal megjelenése

9.3 Kommunikáció Frontend és Backend között

Ahhoz, hogy a Frontend kommunikálni tudjon a program Backend részével, valamilyen kommunikációt kellett létrehozni a kettő között. Ehhez REST API hívásokat használtam. Ahhoz, hogy a REST API hívás megtörténhessen, egy ASP.NET projektet hoztam létre, aminek ***RouteGenerationApi*** lett a neve. Mivel a Frontend-et magát Reactben írtam meg, az ASP.NET projektet csak a kommunikáció kialakítása miatt hoztam létre, ezért az MVC modell felépítéséből csak a Controller és a Models részeket használom, az ASP.NET projekt View része helyett a React-ben írt oldalt használom.

A kontroller REST API elérési címe a ***„/generatedroute”***. A kontroller két pont geo koordinátáit, egy távolságot, és egy útvonaltervező indexét várja. Ez összesen hat

értéket jelent, mivel a geo koordinátáknál a szélességi és hosszúsági fokot is meg kell adni.

9.4 RouteController

A ***RouteGenerationApi*** projekten belüli kontroller kapja meg a Frontend-ről küldött GET kéréseket, amik az útvonaltervezéshez szükséges adatokat tartalmazzák. Ezek az értékek nem kéréshez tartozó body-ban lesznek, hanem a „query string”-ben. Ezeket az adatokat a Kontroller továbbítja a ***RouteGeneration***-nek.

Itt lesz eldöntve a kontroller által, hogy a kapott adatok alapján két pont közötti legrövidebb generálást kell-e indítani, vagy egy pontból induló véletlenszerű generálást.

Ha kész van az útvonaltervezés, akkor a generált adatok egy ***RouteDto*** osztályba lesznek becsomagolva, amit a kontroller egyszerűen vissza tud adni ***json*** formátumban a Frontend-nek.

9.5 RouteGeneration

A ***RouteGeneration*** osztályon belül két megvalósítása van a ***GenerateTheRoute*** metódusnak. Az egyik két pont közötti útvonal generálását kezdeményezi, a másik pedig egy pontból kezdeményez bizonyos hosszúságú útvonal generálást. A két pont közötti generálást megvalósító metódusnak öt darab bemeneti paramétere van, amik a Frontendtől kapott geo koordinátáknak felelnek meg, és a generáláshoz használt algoritmus típusának. A legelső lépés beazonosítani azt a két pontot, amik a legközelebb vannak a kapott koordinátákhoz. Ezt a felépített gráfban lévő ***Junction***-ök koordinátáinak, és a kapott koordináták különbségének kiszámolásával meg lehet határozni. Ahol ez a különbség legkisebb, az a ***Junction*** lesz a legközelebb a kapott koordinátához.

Mivel az útvonaltervező ***Segment***-ekkel dolgozik, a legközelebbi pont meghatározása után, el kell dönteni, hogy melyik ***Segment***-en helyezkedik el a lerakott koordináta. Ehhez szükség van a lerakott koordináta pont, és a legközelebbi pont közötti vektorra. Egy vektor irányszögét a (6) képlet segítségével lehet kiszámítani.

$$\theta = \arctan\left(\frac{y}{x}\right) \tag{6}$$

A (6) függvény eredménye radiánban lesz megadva, ezért ahhoz, hogy az irányszöget megkapjuk, a radián értéket meg kell szorozni ***(180/π)*** -vel. Az így kapott szöget pedig korrigálni kell aszerint, hogy a vektor iránya hányadik síknegyedbe mutat. Miután ez a korrigálás is megtörtént, akkor megkapjuk az irányszögét annak a vektornak, ami a felhasználó által lerakott koordináta, és a hozzá eső legközelebbi ***Junction*** között van. Ezután már csak össze kell hasonlítani a legközelebbi ***Junction***-höz tartozó ***Segment***-

eket, hogy melyik ***Segment*** irányszögének különbsége a legkisebb az egyenlet által kiszámított irányszöggel. Ha ezt megkapjuk, akkor meghatároztuk, hogy a pont melyik ***Segment***-en fekszik. A vektor irányszögének számításakor ügyelni kell arra is, hogy ha a két pont hosszúsági foka nem különbözik, akkor egy 0-val való osztás történne. Ilyenkor a vektor irányszöge 90 vagy 270 fok lehet, amit a szélességi fokok szerint vett különbség alapján lehet meghatározni.

Ha a felhasználó véletlenszerű utat szeretne generálni, akkor a ***GenerateTheRoute*** metódus három paraméteres változata lesz meghívva a kontrollerből. A bemeneti paraméterek közül kettő a lerakott pont geo koordinátái lesznek, a harmadik pedig a generálni kívánt út hossza. Itt a ***Segment*** meghatározása ugyanúgy működik, mint a két pont közötti generálásnál.

A ***Segment***-ek meghatározása után el lehet közöttük indítani az útvonal generálás megfelelő változatát, az alapján, hogy a felhasználó hogyan akarta generálni az útvonalat.

Itt történik az út két szélének rendbetétele is az útvonaltervezés megfelelő lefutása után. Előfordulhat, hogy egy generált útvonalnál, az a ***Segment***, ahova lerakta a felhasználó a pontot, éppen az ellenkező irányba mutat, mint amerre az útvonalnak mennie kellene. Ilyenkor a generált útvonal első két eleme ugyanaz a ***Segment*** lesz, mivel a felhasználónak vissza kell fordulnia. Ilyenkor a generált útvonal első két eleme helyett egy új ***Segment***-et rakok be a listába, aminek az első pontja a felhasználó által megadott kezdőpont lesz, a második pontja pedig a generált útvonal következő ***Segment***-jének az első ***Junction***-je lesz. Ugyanez a helyzet előfordulhat a generált út végén is, ezért ott is hasonlóan kicserélem az utolsó két elemet egy módosított ***Segment***-re, aminek a végpontja a felhasználó által lerakott második pont. Amennyiben a listának valamelyik szélén nem kell a két elemet lecserélni, akkor elég csak a legszélső elem megfelelő ***Junction***-jének koordinátáit átírni a felhasználó által lerakott koordinátára, ezzel biztosítva, hogy ténylegesen az általa lerakott pontok között legyen az útvonal, és ne a ***Segment***-ek pontjaitól. Olyan eset is előfordulhat, hogy a lerakott pontok, ugyanazon a ***Segment***-en helyezkednek el, ilyenkor pedig egyetlen ***Segment***-et kell visszaadnia a programnak, aminek mindkét pontja megfelel a felhasználó által lerakott pontoknak.

Ha minden adat feldolgozása sikerült, akkor egy új ***RouteDto*** elemet ad vissza a ***RouteGeneration***, amiben el lesz tárolva a generált útvonal olyan formában, hogy azzal egyszerűen dolgozni tudjon a Frontend. Ezt a ***MakeMapBoxPolyLines*** metódus biztosítja, ami az útvonal egyes ***Junction***-jeinek koordinátáit egy # jellel elválasztva szöveggé alakítja, ezt Frontend-en könnyen fel lehet dolgozni. Ezen kívül a ***RouteDto*** elemben el lesz tárolva a fordulások listája is, a generált útvonal hossza is, és az is, hogy mennyi idő alatt lett legenerálva az útvonal.

9.6 Útvonaltervező

Az útvonaltervező algoritmusok mindegyike úgy lett felépítve, hogy megfelelően tudjanak működni a szegmensek közötti generálással. A ***4. fejezetben*** bevezetett frontier megnevezés helyett a megvalósításnál inkább a „priority queue” megnevezést használom, ami magyarul az elsődleges sort jelenti. Ez maga a frontier, viszont útvonaltervezésnél beszédesebb a priority queue elnevezése, az adatok kiválasztása miatt. A priority queue feltöltéséhez mindenhol ***PathPart*** példányokat alkalmaztam, aminek három tulajdonsága van. Két ***Segment***. ami azt jelöli, hogy mi lett elérve, és honnan, illetve egy ***TrueWeight*** érték, ami az A* algoritmusoknál lesz hasznos a heurisztika nélküli súlyok eltárolására.

9.6.1 Dijkstra

A metódusnak három bemeneti paramétere van, két ***Segment***, és a ***Segment***-ek listája. A két ***Segment*** közül az egyik a kiinduló ***Segment***, ahonnan generálni szeretné az útvonalat a felhasználó, a másik pedig az a ***Segment*** lesz, ahova érkezni szeretne. A ***Segment***-ek listája pedig minden ***Segment***-et tartalmaz, ami benne van a gráfban.

A metódus egy ***Path*** osztállyal tér vissza, amely két tulajdonságot tartalmaz. Egy ***Segment*** listát ***ThePath*** néven, ami a legenerált útvonalat fogja tartalmazni, és egy double típusú ***OnTheFlyTotalDistance*** nevezetű tulajdonságot, ami futási időben számolja ki, hogy mekkora a ***ThePath*** listában lévő útvonal hossza.

A Dijkstra algoritmus futásának elején létrehozok egy ***Segment*** listát, amiben a generált útvonalat fogom eltárolni, illetve létrehozom az elsődleges sort is, vagyis a ***priorityQueue***-t. Ehhez egy ***SortedList***-et használtam, ami double értékek alapján tárol ***PathPart*** példányokat. A ***SortedList***-nél azonban ügyelnem kellett arra, hogy azonos kulccsal nem vesz fel két elemet, és nem is írja felül azt, ami már benne van. Nekem szükségem volt arra, hogy ha két ***Segment***-hez vezető út hossza megegyezik, akkor mindkettőt vizsgálja meg, ha szükséges. Ezt úgy tudtam megoldani, hogy egy ***DuplicateComparer*** nevezetű összehasonlító osztályt adtam át a lista konstruktorában. Ebben az osztályban felülírtam a ***Compare*** metódust olyan módon, hogy engedélyezem az azonos kulcsok használatát. Ez az osztály működésileg úgy viselkedik, hogy ha két érték megegyezik, akkor pozitív értékkel tér vissza, ami azt jelenti, hogy az összehasonlított két paraméter közül, a második értéke nagyobb. A listánál ez úgy működik, hogy egy új elem beszúrásakor, ha már van ilyen értékkel kulcs, a beszúrandó értéket az eredeti ilyen súlyú elem után helyezi el.

Szükségem van még három másik listára is. Ebből kettő ***Segment***-eket fog eltárolni, egy pedig súlyokat. A ***Segment*** listák közül az egyik az ***investigated,*** ami azokat a ***Segment***-eket tárolja magában, amiket már elért a legrövidebb úton az algoritmus. Ennek az első eleme a kiinduló ***Segment***. A másik ***Segment*** lista az ***investigated_from***, ami azt mondja meg hogy az egyes elemeket az ***investigated*** listában honnan lehet

elérni. Ez azért kell, hogy az algoritmus végén vissza tudjam vezetni melyik ***Segment*** honnan lett elérve, és ennek a legelső eleme egy null érték lesz. A harmadik lista pedig az elérési súlyát tartalmazza minden egyes ***Segment***-nek, ami az ***investigated*** listában van. Ennek az első értéke, a legelső ***Segment*** hossza lesz, mivel a legelső lépést úgy veszem, hogy végig is megy a ***Segment***-en az algoritmus.

Kell egy ***currentSeg*** változó, ami mindig az aktuálisan vizsgált ***Segment***-et fogja tárolni. Ennek kezdeti értékként a kiinduló ***Segment*** lesz beállítva. Ezen kívül pedig kell egy ***currentJunct*** változó is, az aktuálisan vizsgált pontnak. Ennek pedig mindig az aktuális ***currentSeg*** második pontja lesz az értéke.

A függvény elején megvizsgálom, hogy az induló ***Segment*** megegyezik-e az érkezési ***Segment***-tel, mert ha igen, akkor csak vissza kell adnom a ***Segment***-et, egy egyelemű útvonalként. Ha nem egyenlőek ezek az elemek, akkor egyesével megvizsgálom a ***currentJunct*** szomszéd ***Segment***-jeit. Ha az ***investigated*** listában még nem szerepel az éppen vizsgált szomszéd, akkor ki kell számolni, hogy a szomszéd ***Segment*** második pontját milyen súllyal lehet elérni a ***currentJunct*** pontból. Itt a súlyok listájának legelső eleméhez adom hozzá a két ***Junction*** közötti távolságot, mivel ez még csak a legelső elem vizsgálatánál fut le. A súly kiszámítása után, a ***priorityQueue***-hoz hozzáadok a kiszámolt súllyal, egy új ***PathPart*** példányt a megfelelő értékekkel. Ilyenkor a ***PathPart*** példányban be kell állítani, hogy melyik ***Segment*** lett elérve, és hogy melyik ***Segment***-től lett elérve. A ***TrueWeight*** tulajdonságra itt nincs szükség.

Miután minden szomszédot megvizsgáltam az első ***Segment***-nél, kiválasztom a ***priorityQueue***-ból a legkisebb súllyal rendelkező ***PathPart*** elemet, és benne található elért ***Segment***-et az ***investigated*** listába rakom. Az ***investigated_from*** listához hozzáadom a ***PathPart*** elem másik ***Segment***-jét, ami azt mutatja, hogy honnan lett elérve az ***investigated*** listába helyezett ***Segment***. A súlyokat tartalmazó listába, a ***priorityQueue***-ból kiválasztott elem kulcsát szúrom be, mivel az mutatja meg az elért ***Segment*** elérésének távolságát. Ezek után be kell állítani, hogy a ***currentSeg*** a ***priorityQueue***-ból kiválasztott legelső elem ***Segment***-jével legyen egyenlő, a ***currentJunct*** pedig a most megváltozott ***currentSeg*** második pontjával legyen egyenlő.

Miután a legelső ***Segment*** feldolgozása megtörtént, elindul egy előltesztelős ciklus, ami addig fut, amíg az utoljára megvizsgált elem nem egyenlő a cél ***Segment***-tel, vagy amíg van elem a ***priorityQueue***-ban.

A ciklus elején mindig törölni kell a ***priorityQueue*** legelső elemét. Ezután ugyanúgy megvizsgálok minden szomszédot, mint ahogy az első ***Segment***-nél, és feltöltöm a megfelelő értékekkel a ***priorityQueue***-t. Itt viszont a súly számításánál már nem a ***weights*** lista legelső elemét adom hozzá a két pont közötti távolsághoz, hanem a ***CurrentSeg***-hez tartozó súlyt, amit index alapján meg lehet kapni a ***weights*** listából. Miután megvizsgáltam minden szomszédot, megvizsgálom, hogy a ***priorityQueue***-ban

van-e még vizsgálható elem. Ha van még benne elem, akkor megvizsgálom, hogy az első elem szerepel-e már az ***investigated*** listában, mert ha igen, akkor csak nagyobb súllyal érném el, tehát törölhető az elsődleges sorból az adott elem. Addig törlöm az első elemeket, amíg el nem jutok egy olyan ***Segment***-ig, ami még nem szerepelt az ***investigated*** listában, vagy ameddig van elem a ***priorityQueue***-ban. Amikor olyan elem van a ***priorityQueue*** elején, ami még nem szerepelt az ***investigated*** listában, akkor az alapján az elem alapján, be kell állítani a megfelelő értékeket minden listában, illetve változóban.

Azért törlöm mindig a ciklus elején az első elemet, mert ha nem generálható valamerre útvonal, előfordulhat, hogy a ciklus további részében a ***priorityQueue***-ba nem kerül bele új elem. Ha egy ***Segment*** már bekerült az ***investigated*** listába, akkor annál kevesebb súllyal már biztos, hogy nem fogom tudni elérni, mivel negatív súlyok nem szerepelnek a térképen. Mivel ezt megvizsgálom a ***priorityQueue*** első elemének kivételekor, lehetséges, hogy minden elemet törölni fogok, mert már kevesebb súllyal eljutottunk oda. Ha ez bekövetkezik, akkor nem generálható le útvonal a felhasználó által kiválasztott ***Segment***-ek között.

Az útvonal generálás végén megvizsgálom, hogy a ***currentSeg*** tényleg egyezik-e a cél ***Segment***-tel, mert ha nem, akkor csak egy üres listát adok vissza.

Amennyiben sikeresen le lehetett generálni az útvonalat, akkor lekérem egy ***curr_idx*** változóba a ***currentSeg*** indexét az ***investigated*** listából. Ezután egy előltesztelős ciklussal addig dolgozom fel az adatokat, amíg az ***investigated_from*** lista ***curr_idx*** helyen lévő eleme nem null. Azért megyek eddig, mert a legelső ***Segment*** eléréséhez egy null értéket adtam meg. A cikluson belül a következő index értékének mindig az ***investigated*** lista azon elemének indexét adom meg, amelyik elem egyezik az ***investigated_from*** lista ***curr_idx*** helyen lévő elemével. Ha megvan a következő index, hozzáadom az útvonal listájához az ***investigated*** lista ***curr_idx*** helyen lévő elemét, ezután pedig a ***curr_idx*** értékét beállítom a következő index értékére. Ha ez lefutott, akkor az utolsó elemet még hozzá kell adnom az útvonalhoz, majd az így generált útvonal már visszaadható a ***RouteGeneration*** felé, egy ***Path*** osztályba becsomagolva.

9.6.2 A*

Ahogy azt a ***4.4 fejezetben*** is írtam, az A* algoritmus abban tér el a Dijkstra algoritmustól, hogy míg az utóbbi szélességi bejárás szerint vizsgálja a szomszédos elemeket, egy egész nagy területet lefedve, addig az A* algoritmus céltudatosan elindul az elérni kívánt csúcs irányába. Ezen kívül viszont a működése hasonlít a Dijkstra algoritmuséhoz, éppen emiatt a ***9.6.1 fejezetben*** leírt Dijkstra algoritmus működését módosítottam úgy, hogy az megfeleljen az A* algoritmus működésének.

Ugyanazokkal az objektumokkal dolgozok itt is, tehát a ***9.6.1 fejezetben*** leírtak alapján, itt is vannak megfelelő listák a ***Segment***-eknek, a súlynak, a ***priorityQueue***-nak, illetve

az útvonalnak. Működésileg a súlyok számításán kívül nincs eltérés. A súlyok számításához itt a ***4.4.4 fejezetben*** ismertetett (4) képletet használom. Ehhez létrehoztam egy ***HeuristicWeightCalculator*** metódust, ami úgy működik, hogy a paraméterként kapott két ***Junction*** között kiszámolja a távolságot a ***DistanceCalculator*** osztály segítségével, majd ehhez az értékhez hozzáadja a megkapott súlyt is. Ennek a metódusnak az eredményeként, minden ***Segment***-hez egy olyan súlyt kapok, ami összesíti a vizsgált ***Segment*** elérésének pontos hosszát, illetve a cél ***Segment*** eléréséhez becsült heurisztikus távolságot, az aktuálisan vizsgált szomszéd ***Segment***-től. Ahogy azt a ***4.4.4 fejezetben*** is ismertettem, amíg egy ***Segment*** elérésének a súlyát pontosan meg lehet határozni, addig azt a tényleges távolságot, ami az éppen vizsgált ***Segment***, és az elérési cél ***Segment*** között van, nem lehet pontosan meghatározni. Erre csak egy becslést lehet adni, amit a ***Haversine formula*** alkalmazásával számoltam ki. Ez a formula a ***9.8 fejezetben*** van kifejtve.

A ***priorityQueue*** feltöltésénél a kulcs értékének a (4) képlet által megkapott súlyt adom meg, viszont ekkor a ***priorityQueue*** kulcsainak értéke nem fog megegyezni a ***Segment*** tényleges súlyával. A ***Segment***-ek pontos elérésének értékét az elem ***TrueWeight*** tulajdonságába tárolom el, és amikor kiveszek egy elemet a ***priorityQueue***-ból, ezt a ***TrueWeight*** értéket szúrom be a ***weights*** lista végére.

9.6.3 Átalakított Dijsktra algoritmus

A sima Dijkstra algoritmus nem tesz különbséget futásra alkalmas utak, és sima, például járdaként működő gyalogos utak között. Készítettem egy alternatív generálási lehetőséget is, ahol a Dijkstra algoritmust olyan módon módosítottam, hogy előnyben részesítse útvonal generálásnál azokat az utakat, amelyek kifejezetten gyalogos, illetve futási közlekedésre használhatóak.

A súlyok kezelésével tudtam elérni, hogy a gyalogos utak előnyben legyenek részesítve. Ehhez, a súlyokat tartalmazó listában, módosított súlyok lesznek eltárolva az egyes ***Segment***-ekhez. Ha az aktuálisan vizsgált szomszéd ***Segment Footway*** tulajdonsága igaz, tehát kifejezetten gyalogos vagy futó út, akkor a súlyt megszorzom egy egynél kisebb tört számmal. Az optimális érték a ***0,7*** lett. Ezzel azt érem el, hogy ha ***Footway*** az adott ***Segment***, akkor kevesebb lesz az elérési súlya a ***priorityQueue***-ban, emiatt előnyben fogja őt részesíteni az algoritmus, de ha túlságosan is hosszabb lenne az útvonal, akkor mégsem arra fogja generálni végeredményt.

9.6.4 Átalakított A* algoritmus

Az előző ***9.6.3 fejezetben*** leírtak az A* algoritmusnál is érvényesek, tehát magától nem képes különbséget tenni az útvonalak típusai között, ezért itt is módosított súlyokkal kell dolgoznia az algoritmusnak. Az átalakított Dijkstra algoritmusnál leírtak mellett annyi módosítást kell még itt tenni, hogy a súlyok módosításakor nem csak a ***CurrentJunct*** aktuálisan vizsgált szomszédjának súlyát kell csökkenteni, hanem a

heurisztikával számított becslést is csökkenteni kell. Ahhoz, hogy ez a módosítás megvalósulhasson, a ***HeuristicWeightCalculator*** metódus paramétereit kiegészítettem egy opcionális paraméterrel, amivel meg lehet adni, hogy ***Footway*** úthoz kell-e heurisztikát számolni, mert ha igen, akkor ezt a heurisztikát megszorozza ***0,7***-tel. A ***PathPart*** elemek ***TrueWeight*** tulajdonságába ugyanúgy a módosított érték fog a heurisztika nélkül bekerülni.

9.7 Fordulás

Azt, hogy a felhasználónak éppen merre kell fordulnia, a magassági és a szélességi fokokból számoltam ki. Ezért a számításért a ***RouteGenerator***-ban lévő ***TurnHandler*** metódus felel. Ez öt darab bemeneti paramétert vár. Az előző fordulás értékét, amit referenciaként kap, az eredeti adatok szerinti fordulás értékét, a változtatott, felhasználó szempontjához igazított fordulás értékét, és azt a két ***Junction***-t, amik között meg kell határozni, a fordulás irányát.

A fordulás értékeit egy ***Enum***-ban tárolom. Ez a következő értékeket tartalmazza:

0. Előre, ***1. Jobbra***, ***2. Hátra***, ***3. Balra***

Kezdeti értékként a felhasználó szemszögéből lévő fordulásnak, és az adatok alapján vett fordulásnak is ***Előre*** értéket állítok be. Az előző fordulás értéke kezdetben egy **-1** lesz, ami a legelső lépés miatt szükséges. Ahhoz, hogy a fordulásokat rendesen meg lehessen határozni a felhasználó szemszögéből, vezetni kell a térkép adatok alapján történő fordulásokat is.

A generált útvonalon végig iterálva, minden két ***Junction*** között meghívom a metódust. A metódusban felveszek egy lebegőpontos számokat tároló kételemű tömböt, amiben a két ***Junction*** koordinátái közti különbséget mentem le. Ebből a két értékből meg tudom határozni, hogy az út, az előző forduláshoz képest, éppen merre fog menni. Ha a szélességi kör alapján vett koordináta változás lesz lényegesen nagyobb, mint a hosszúsági körön vett változás, akkor a felhasználó valószínűleg lefele, vagy felfele haladt a térképen. Ha ez fordítva van, és a hosszúsági kör szerint vett különbség nagyobb a szélességi kör alapján vett távolságtól, akkor a felhasználó vagy jobbra, vagy balra mozgott a térképen. Ezután már csak azt kell megvizsgálni, hogy a nagyobb érték mínuszos előjellel rendelkezik-e. Ha szélességi kör alapján nézzük a változást, és az érték pozitív, akkor az adott pontok között, az út észak felé vezet, viszont, ha ez az érték negatív, akkor dél felé vezet. Ha a hosszúsági kör alapján nézzük a változást, akkor a pozitív érték egy jobbra fordulást fog jelenteni, a negatív érték pedig balra fordulást fog jelenteni. Így megkapjuk, hogy a térképen lévő adatok alapján, a valódi irány merre van. Felfele, lefele, jobbra, vagy balra. Ha a térképen lévő adatok alapján, meghatároztuk, hogy merre kell fordulni, abból egy kivonással el lehet dönteni, hogy a felhasználó szempontjából, ez milyen fordulási értéknek felel meg. Összesen négy darab lehetőség van a fordulásra, ezért az utolsó fordulás, és a jelenlegi térkép szerinti

fordulásnak a különbségét ki kell vonni négyből. Ezt az értéket maradékosan elosztva néggyel, megkapjuk, hogy a felhasználó szempontjából merre kell fordulnunk. Itt figyelni kell arra, hogy mikor elindulunk egy úton, még nem volt előzetes fordulásunk, ezért az első fordulásnál a felhasználónak mindig előre kell mennie, és majd ezután lesz lényeges a további fordulatok számítása. A felhasználó fordulásának kiszámítása után már csak be kell állítani az előző fordulást, a jelenlegi térkép adatok alapján kiszámított fordulási értékként, majd ugyanez le fog futni minden egyes pont között az útvonalon. Itt fontos, hogy ne a futó szemszögéből számított fordulás legyen megadva az előző fordulásnak, hanem a térkép adatok alapján vett fordulás.

9.8 Távolság és heurisztika számítása

A távolság kiszámításához a Haversine formulát használtam. A Haversine formula [46] a Föld két pontja között adja meg a Nagy-kör távolságukat, amit a pontok szélességi, és a hosszúsági foka alapján lehet kiszámolni. A képlet a Föld magaslati különbségeit figyelmen kívül hagyja.

Mivel a térképet felépítő ***Segment***-ek ***Junction***-jei között kicsi a távolság, és a ***Segment***-ek mindig egy egyenes vonalat írnak le, a Haversine formulát fel tudtam használni a ***Segment***-ek hosszának pontos meghatározására. Ezen kívül a heurisztikát is ki tudtam vele számolni, mivel két ***Junction*** között csak akkor kapok pontos távolságot, ha a ***Junction***-ök között elhelyezkedik egy út, vagyis van olyan ***Segment***, aminek a két pontja, az említett két ***Junction***. Amennyiben két ***Junction*** között nincs út, akkor ez a számítás csak egy becsült legrövidebb elérési távolságot fog adni a két elem között, ami biztosan kisebb lesz a tényleges elérés távolságánál. Emiatt a Haversine formulát fel tudom használni optimális heurisztikaként is.

A Haversine formula a ***9.8.1 ábrán*** látható:

$$
\begin{aligned}
\Delta lat &= lat2 - lat1 \\
\Delta long &= long2 - long1 \\
a &= \sin^2(\Delta lat/2) + \cos(lat1)\cos \\
&\quad (lat2)\ \sin^2(\Delta long/2) \\
c &= 2\,\mathrm{atan2}(\sqrt{a}, \sqrt{1-a}) \\
d &= R.c
\end{aligned}
$$

9.8.1. ábra: A távolság számításához felhasznált képlet [46]

Ahol [46] ***Δlat*** a szélességi fokok különbségét, ***Δlong*** a hosszúsági fokok különbségét, ***R*** pedig a Föld sugarát jelenti, ami 6.371 km-nek felel meg.

Mivel az alkalmazásomhoz gyalogosan megtehető távolságokon szeretnék útvonalat generálni, méterre pontosan fogom meghatározni az útvonalak hosszát, ezért az ***R*** értéke az alkalmazásomban ***6.371.000*** lesz.

9.9 Véletlenszerű útvonal generálás

A véletlenszerű generálás végleges verziójához a ***5.3 fejezetben*** felvezetett A* alapú megoldást választottam, mivel ez bizonyult a leggyorsabbnak és leghatékonyabbnak.

A véletlenszerű generálás négy paramétert vár. Egy egész számot, ami azt jelzi, hogy mekkora útvonalat kell generálni, egy ***Segment*** elemet, ami az útvonal generálás kezdőpontja lesz, illetve a gráfot felépítő ***Segment***-ek és **Junction**-ök listáját. Ennél a megoldásnál először venni kell a generálni kívánt távolság valahányad részét. Én ezt az értéket a generálni kívánt hossz ***3,5***-szeresének választottam, és a továbbiakban ***(x)***-el jelölöm. Meg kell vizsgálni, hogy a kiválasztott kiinduló ***Segment*** első pontjától melyek azok a ***Junction***-ök, amelyek ***(x)*** távolságra vannak. Nem biztos, hogy pontosan ***(x)*** távolságra lesznek ilyen elemek a kiinduló ***Junction***-től, ezért érdemes valamekkora hibaarányt alkalmazni. Én 10%-os hibaarányt engedtem meg, vagyis azok közül a ***Junction***-ök közül választ a program, amelyek távolsága az [(x)*0.9; (x)*1.1] zárt intervallumon belül vannak. Ezek közül a ***Junction***-ök közül véletlenszerűen ki kell egyet választani. A kiinduló ***Junction*** jelölése a továbbiakban legyen ***(a)***, a most választott ***Junction*** jelzése pedig legyen ***(b)***. Ezután meg kell találni, hogy melyek azok a ***Junction***-ök, amik ***(a)*** és ***(b)*** elemektől nagyjából ***(x)*** távolságra vannak. Ha ezek a pontok is megvannak, akkor már csak azt kell eldönteni, hogy az ***(a)*** és ***(b)*** elemek közti képzeletbeli vektor bal, vagy jobb oldalán helyezkednek el. Az így kiválasztott ***Junction***-ök közül, legyen a bal oldali jelölése ***(c)***, a jobb oldali jelölése pedig ***(d)***. Ahhoz, hogy el lehessen dönteni ***(c)*** és ***(d)*** elem az ***(a)*** és ***(b)*** elem közti vektor melyik oldalán helyezkednek el, a (8) képletet használtam, ami a vektoriális szorzat, csak két dimenzióban.

$$\overrightarrow{ab} \times \overrightarrow{ac} = i\left(\overrightarrow{ab}_y * 0 - 0 * \overrightarrow{ac}_y\right) - j\left(\overrightarrow{ab}_x * 0 - 0 * \overrightarrow{ac}_x\right) + k(\overrightarrow{ab}_x * \overrightarrow{ac}_y - \overrightarrow{ab}_y * \overrightarrow{ac}_x) \quad (7)$$

$$\mathrm{d} = (\overrightarrow{ab}_x * \overrightarrow{ac}_y - \overrightarrow{ab}_y * \overrightarrow{ac}_x) \quad (8)$$

A (7) képletben a 0-val jelzett értékek a vektoriális szorzat harmadik dimenziójában lévő értékek lennének, viszont a ***Junction***-ök közti vektorjaim csak kétdimenziósak, ezért a ***z*** dimenzió értékeit nullának kell venni.

A (8) képletben lévő példánál ***(c)*** pont van vizsgálva. A képlet az ***(a)*** és ***(b)***, illetve az ***(a)*** és ***(c)*** ***Junction***-ök közti vektorok vektoriális szorzatát adja meg. Ennek a képletnek, ha 0 lesz az eredmény, akkor ***(c)*** ***Junction*** az ***(a)*** és ***(b)*** elemek közötti vektoron helyezkedik el. Az eredmény pozitív lesz, ha a ***(c)*** ***Junction*** az ***(a)*** és ***(b)*** elemek közötti

vektor bal oldalán helyezkedik el, és negatív lesz, ha a jobb oldalán. Ügyelni kell arra, hogy kisebb távolságok generálása esetén, lehetséges, hogy nem lesz ***Junction*** az ***(a)*** és ***(b)*** elemek közti vektor valamelyik oldalán nagyjából ***(x)*** távolságra. Ekkor be kell érni a legközelebbi ***Junction***-nel az adott oldalról.

Ezzel a függvénnyel két halmazra lehet osztani a kiválasztott pontokat, és mindkét halmazból véletlenszerűen ki lehet választani egy elemet. Ezzel 4 ***Junction*** lesz kiválasztva. A kiinduló ***(a) Junction***, az ***(x)*** távolságra véletlenszerűen kiválasztott ***(b) Junction***, és az ezektől balra fekvő ***(c)***, illetve jobbra fekvő ***(d) Junction***. Az ***(a)*** elem kivételével, mindegyik ***Junction***-höz ki kell választani véletlenszerűen egy olyan ***Segment***-et, ahol az adott ***Junction*** a ***Segment*** első pontja. Az ***(a) Junction***-höz tartozó ***Segment***, a felhasználó által térképre lerakott pont segítségével meghatározható, ami részletezve van a ***9.5 fejezetben.*** Az így kapott négy ***Segment*** között már csak le kell generálni az egyes utakat A* algoritmus segítségével, ilyen sorrendben:

(a) → (c) → (b) → (d)

Ezután lesz négy darab útvonal, amiket össze kell fűzni egymással. Összefűzéskor ügyelni kell arra, hogy a két útvonal közül, vagy az előbbi utolsó elemét, vagy utóbbi első elemét ki kell törölni, mivel ezek a ***Segment***-ek meg fognak egyezni. Összefűzés után viszont készen van a véletlenszerűen generált útvonal.

Ilyen véletlenszerű útvonal generálásnál négy különböző A* generálás fog futni, ami miatt lehetséges, hogy a generált útvonal többször is tartalmazni fogja ugyanazokat a ***Segment***-eket. Ahhoz, hogy ez ne történhessen meg, módosítani kellett az átalakított A* algoritmus működését. Ez lett az ***AStarRandom*** metódus, ami egy negyedik bemeneti paramétert is vár, ami az előző A-csillag generálások által felépített útvonal ***Segment*** elemeit tárolja el. Ez a lista a ***Segment***-ek súly számításánál van használva. Ha egy ***CurrentJunct*** aktuálisan vizsgált szomszédos ***Segment***-je benne van ebben a listában, akkor a súly számításakor a ***Segment*** súlyát meg kell szorozni egy egynél nagyobb számmal, hogy büntetve legyen a ***Segment*** ismétlődése, és csak akkor generáljon arra az algoritmus, ha még a büntetéssel is arra vezet az optimális út. Ez az érték ***2,5*** lett. A heurisztikánál is büntetni kell ezeket a ***Segment***-eket, ezért a ***HeuristicWeightCalculator*** metódus egy újabb logikai bemeneti paraméterrel bővült, aminek, ha igaz az értéke, akkor megszorozza a heurisztika által becsült súlyt ***2,5***-tel.

10.TESZTELÉS ÉS KIÉRTÉKELÉS

10.1 Gráf generálása

A gráf generálása nagyon lassú folyamat, ezért külön szedtem a program többi részétől. Budapest térképét ***35 óra, 23 perc, és 42 másodpercig*** építette a program, és **1.283.850 darab Segment** elemet hoz létre.

10.2 Gráf betöltése

A gráf betöltése egy gyorsabb folyamat, mivel a felépített gráfot elmentettem egy xml formátumú fájlba, és onnan töltöm be a már felépített gráfot. A gráf betöltése ***6,22 század másodpercig*** tart, a memóriában pedig 1.3 gigabájtnyi helyet foglal.

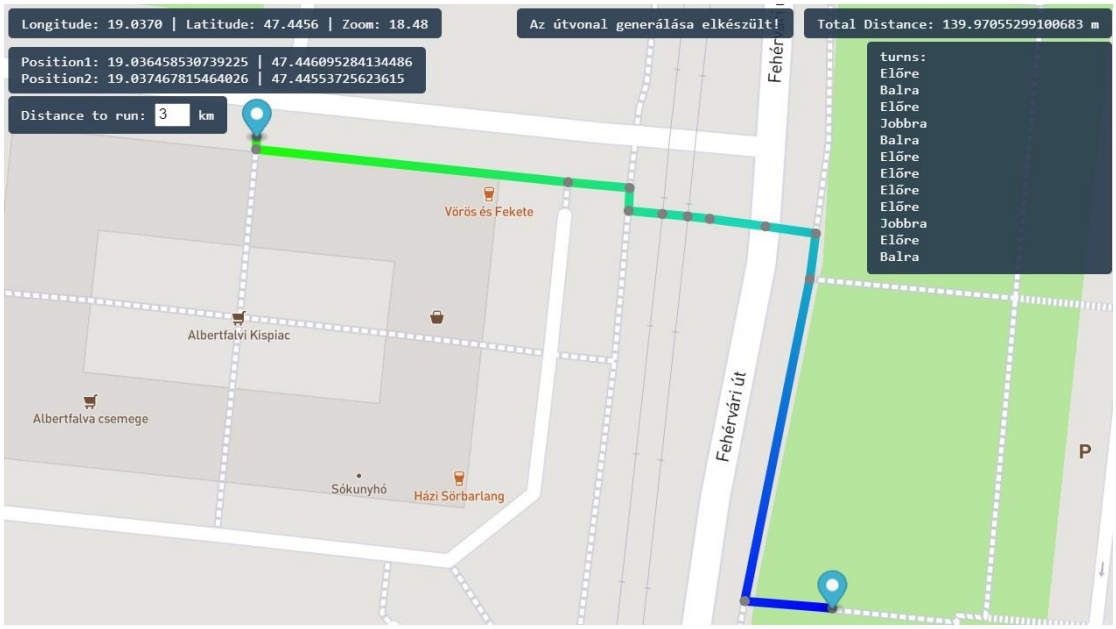
10.3 Távolság, és fordulás a térképen

Az útvonalak hosszának, illetve a fordulások teszteléséhez az OpenStreetMap szolgáltatást használtam, ahol ugyanazon pontok között generáltam le az útvonalat, mint a szakdolgozatomnál. A ***10.3.1. ábrán*** látható az OpenStreetMap által generált útvonal. Az OpenStreetMap által generált útvonal hossza 139 méter, ami az ábra bal felső sarkában látható, az útvonalhoz tartozó fordulások pedig alatta láthatóak.

10.3.1. ábra: Az OSRM szolgáltatás által generált útvonal távolsága és fordulásai

A szakdolgozatom által generált útvonal a ***10.3.2. ábrán*** látható. Az ábra jobb felső sarkában lehet látni a generált útvonal hosszát, ami 139 méter, tehát megegyezik az OpenStreetMap által generált útvonal hosszával. A fordulásokat az úton elhelyezkedő szürke pontok jelzik. Az ábra jobb oldalán, az útvonal távolsága alatt látható rész tartalmazza az egyes fordulásokat. Ha az egyes fordulásokat a kiinduló ponttól nézzük,

tehát az út zöld részétől, akkor látható, hogy az egyes fordulási pontokhoz tartozó fordulás megfelel a gyalogos, vagy futó szemszögéből megtett fordulásoknak.



10.3.2. ábra: A szakdolgozatom által generált útvonal távolsága és fordulásai

Az OpenStreetMap által használt útvonal generáló nem sorolja fel az adatbázis minden pontjához tartozó fordulást, amíg a szakdolgozatom ezt megteszi. Ehelyett az OpenStreetMap megadja, hogy az egyes fordulások után milyen hosszú utat kell megtenni a következő fordulásig.

10.4 Útvonal generálás

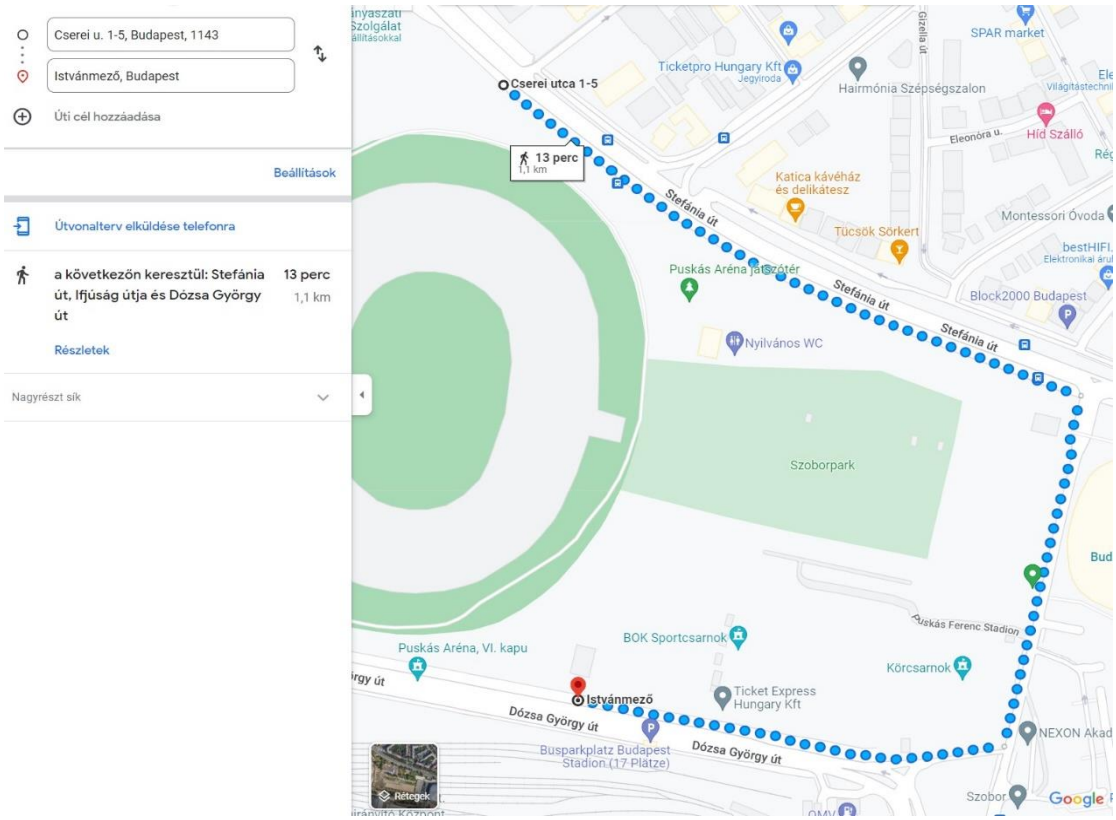
	100 m	1 km	5 km
A*			
Generálás ideje:	00:00:00.0005	00:00:00.052	00:00:04.95
Kifejezetten gyalogos utak százaléka	14,29%	86,96%	36,36%
Út valódi hossza	128 m	1052 m	5044 m
Átalakított A*			
Generálás ideje:	00:00:00.0001	00:00:00.002	00:00:00.61
Kifejezetten gyalogos utak százaléka	14,29%	86,96 %	98,16 %
Út valódi hossza	128 m	1052 m	5248 m
Dijkstra			
Generálás ideje:	00:00:00.0005	00:00:00.81	00:07:37.5
Kifejezetten gyalogos utak százaléka	14,29%	86,96%	36,36 %
Út valódi hossza	128 m	1052 m	5044 m
Átalakított Dijkstra			
Generálás ideje:	00:00:00.0008	00:00:00.56	00:07:11.34
Kifejezetten gyalogos utak százaléka	14,29%	86,96%	98,16 %
Út valódi hossza	128 m	1052 m	5248 m

10.4.1. táblázat: Útvonal generálás sebessége, és adatai két pont között

A generálások ideje egy átlagos érték, amit 20 darab generálás átlagaként számítottam ki. A ***10.4.1 tábla*** alapján leolvasható, hogy minél nagyobb a két pont közötti távolság, annál tovább tart a generálás ideje. Ez amiatt történik, mert a távolság növekedésével, a potenciálisan megvizsgált gráf elemek száma is növekszik. Ugyanakkor amíg kisebb útvonalak generálásánál nem feltűnően nagy a különbség a sima, és az átalakított algoritmusok között, a nagyobb távolságoknál már kitűnik, hogy az átalakított algoritmus inkább előnyben részesíti azokat az utakat, amik nem járdaként vannak rögzítve a térkép adatbázisában. Az értékelésekből az is látszódik, hogy az A* algoritmusok generálási idő tekintetében sokkal jobban teljesítenek, ami várható volt, mivel említettem az ***4.4 fejezetben,*** hogy a Dijkstra algoritmus feleslegesen sok elemet megvizsgál, amíg eljut a céljához.

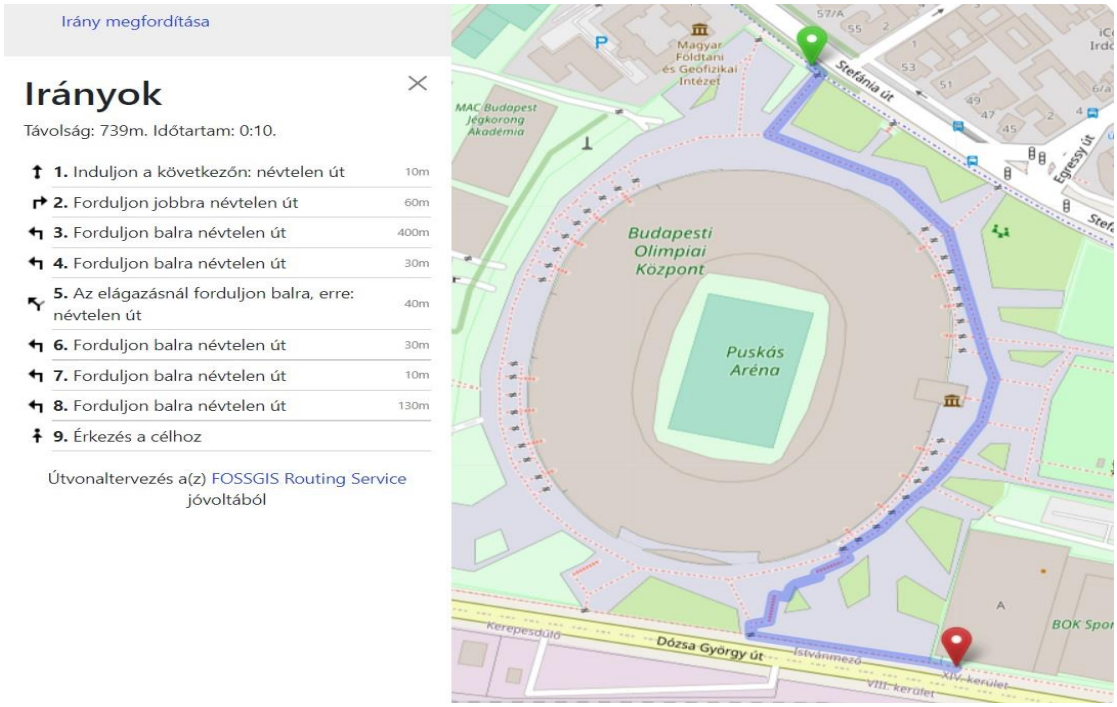
Az útvonaltervezés jóságának teszteléshez kiválasztottam két pontot a térképen, és a szakdolgozatomon kívül az OpenStreetMap, illetve a Google Maps szolgáltatásokon is legeneráltam ugyanazon pontok között az útvonalat.

Google:



10.4.1. ábra: A Google Maps által generált útvonal

OpenStreetMap (OSRM generálással):



10.4.2. ábra: Az OSRM szolgáltatás által generált útvonal

Saját:

10.4.3. ábra: A szakdolgozatom által generált útvonal

Mindhárom alkalmazás más irányba generált. A Google Maps adatbázisában látható, hogy nincs minden gyalogosan járható útvonal eltárolva, ezért gyalogos generálásnál is megkerüli a stadiont. Az OpenStreetMap adatbázisban ezek az adatok megtalálhatóak, ezért az OpenStreetMap által használt OSRM útvonaltervező, és az általam fejlesztett útvonaltervező is fel tudja használni a stadionon belüli gyalogos útvonalakat. A saját útvonaltervezőm egy kicsivel másabb utat talált meg a két pont között, ami ***5 méterrel*** rövidebb, mint az OSRM által megtalált útvonal. Ehhez az útvonal generáláshoz a programomban a sima A* algoritmust használtam.

10.5 Véletlenszerű útvonal generálás

	1 km	5 km	10 km
Saját szakdolgozat			
átlagos generálási idő	00:00:04,6	00:00:07,55	00:00:11,94
átlagos generált távolság	1155 m	5240 m	9835 m
átlagos hiba távolság	169,75 m	523,2 m	1069,1 m
generált távolság jósága	**86,03 %**	**89,54 %**	**89,31 %**
generált távolság hibaaránya	**16,98 %**	**10,46 %**	**10,69 %**
Kifejezetten gyalogos utak százaléka	91,65%	83,50%	81,14%
Trail Router			
átlagos generált távolság	1127 m	5689 m	12038 m
átlagos hiba távolság	256,5 m	832 m	2560 m
generált távolság jósága	74,35 %	83,36 %	74,40 %
generált távolság hibaaránya	25,65 %	16,64 %	25,60 %

10.5.1. táblázat: a Véletlenszerű útvonal generálás kiértékelése

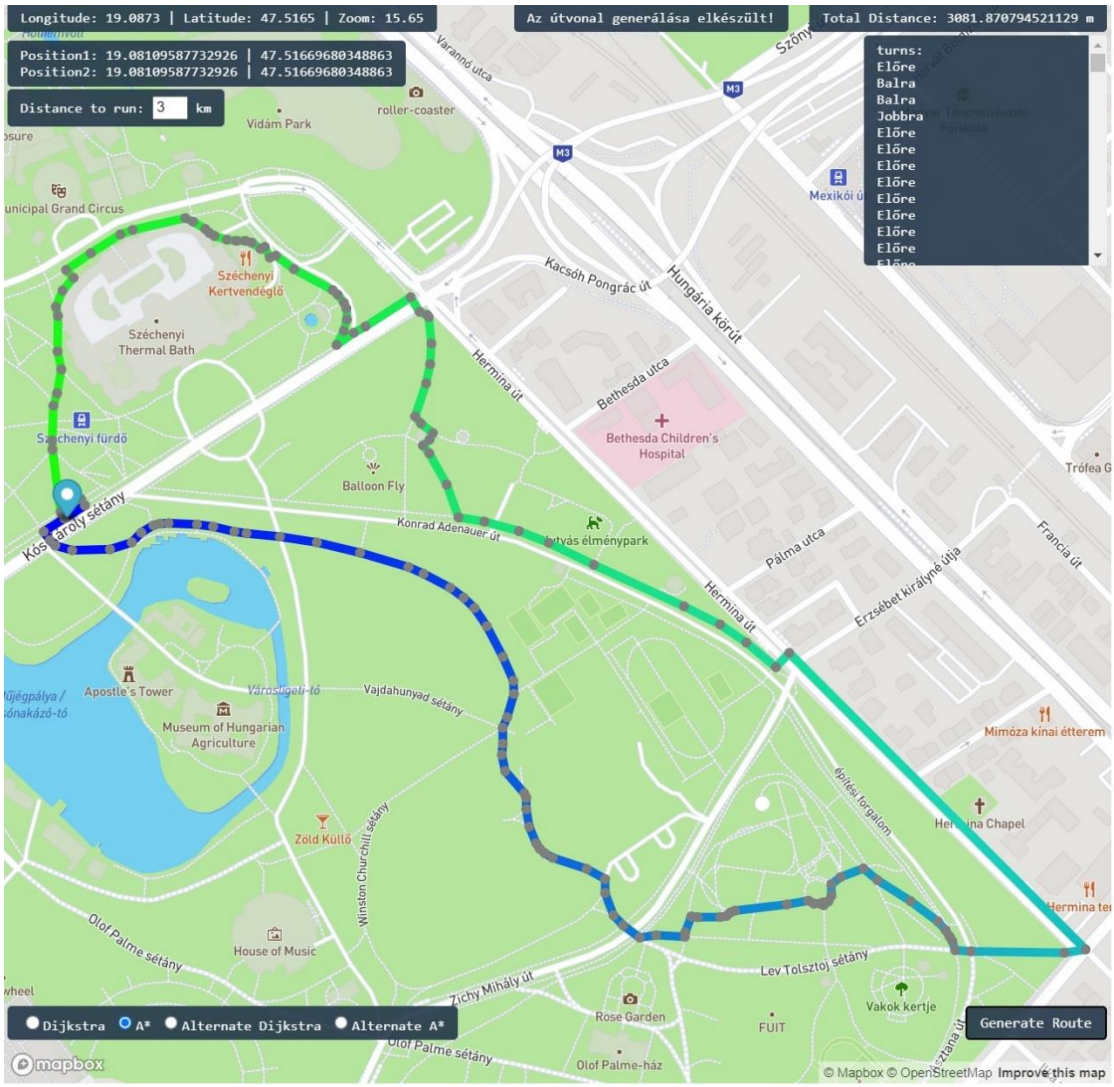
A kiértékeléshez mindkét alkalmazást ugyanarról a pontról indítottam, a Városliget belső részéből. A ***10.5.1 táblán*** látható kiértékelések alapján, egy kilométeres generálásnál ***11,68%***-kal, öt kilométeres generálásnál ***6,18%***, tíz kilométeres generálás esetén pedig ***14,91%***-kal teljesít jobban a programom a Trail Router alkalmazásnál.

A nem járda menti útvonalak generálása a ***10.5.1 tábla*** alapján egy csökkenő tendenciát mutat, ami Budapesten érthető, főleg a Városligetből indulva, mivel amíg ott egy 1 kilométeres utat viszonylag könnyen le lehet generálni a Városliget területén, egy 10 kilométeres útnál már nagy esély van rá, hogy az útvonal generáló el fogja hagyni a Városligetet, és a belvárosban kezdi el generálni az út többi részét.

Sajnos az is látható a ***10.5.1 táblán***, hogy bár az utak hosszúságának jósága mindig ***86%*** felett van, viszont a generáláshoz szükséges idő monoton növekszik az út hosszával.

A táblázat elkészítéséhez a felsorolt hosszúságok mindegyikéhez húsz darab útvonalat generáltam mindkét programmal, tehát szoftverenként hatvan generálás, összesen pedig százhúsz generálás eredményeinek átlagai láthatóak a táblán.

A véletlenszerű generálás alakjának teszteléséhez a ***15.5.1. ábrán*** látható, hogy tényleg képes olyan útvonalakat generálni, amik nagyjából egy körutat írnak le, és ha lehetséges, akkor nem lépnek kétszer ugyanarra az útra, amin már jártak.



10.5.1 ábra: A véletlenszerű generálás alakjának bemutatása

11. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

Egyelőre csak Budapest térképén működik az útvonal generátor, ezért a továbbfejlesztési lehetőségek közé sorolnám több településnek a felépítését is, ezzel biztosítva, hogy bármilyen két pont között lehessen útvonalat generálni. Több adattal való dolgozás esetén azonban a memóriakezelést is át kell alakítani, hogy mindig dinamikusan csak egy olyan térség legyen benne a memóriában, ami éppen szükséges az útvonal generáló működéséhez.

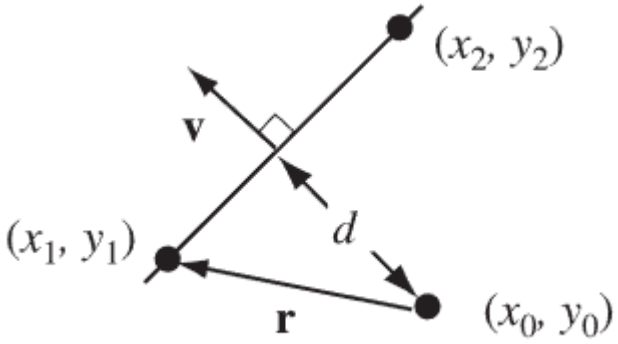
Az útvonal generálás elég lassú tud lenni, amennyiben a város egyik széléről szeretnék útvonalat generálni a másik felébe, még A-csillag algoritmus használatával is. Ezt fel lehetne gyorsítani olyan módon, hogy a kijelölt két pont közötti részt kisebb részekre osztom fel, mivel kisebb útvonalak generálásakor nagyon gyorsan teljesít a megvalósított A-csillag algoritmus.

Bővíteni lehetne azzal is a programot, hogy két pont közötti útvonal generálásnál, ha egy távolságot is megad a felhasználó, ami hosszabb a két pont közötti távolságnál, akkor egy ekkora hosszúságú útvonalat generáljon a két pont között, a legrövidebb út helyett.

Egyelőre a program futása során, többnyire véletlenszerű generálásoknál lehetséges, hogy olyan pontok között akar útvonalat generálni az algoritmus, amik között nincsen kapcsolat. Ez sajnos abból adódik, hogy gyalogos utakat dolgozok fel az adatbázisból, viszont az OpenStreetMap adatbázisa folyamatosan frissül, ezért elképzelhető, hogy egy olyan útvonal, ahol egyébként van járda a valóságban, úgy van rögzítve az adatbázisban, hogy semmi jele nincs a gyalogos használat lehetőségének. Emiatt létrejöhetnek olyan különálló kisebb gráf részek, amik nem kapcsolódnak hozzá a Budapestet felépítő teljes gráfhoz. Ilyenkor előfordulhat, hogy az útvonaltervező algoritmusok elkezdenek addig futni, amíg a teljes gráfot át nem vizsgálják egy lehetséges útvonal után kutatva. Ez nagyon sokáig tart, és ilyenkor be lehetne vezetni egy megszakítást, ami felfüggeszti bizonyos időn belül a generálást. Ehhez lehetne szálkezelést alkalmazni, és bizonyos idő után felfüggeszteni a szálat, viszont ennek is megvannak a veszélyei, [47] ami miatt még további kutatásokat kell végeznem.

Az útvonaltervezés elkezdése előtt, meg kell találnom azt a ***Segment*** elemet, amin a felhasználó által lerakott pont elhelyezkedik. Azért kell a ***Segment***-et megtalálnom, mert az algoritmusok ***Segment***-ek között generálják le az útvonalat, illetve a felhasználó sem tudja pontosan kiválasztani az ***osm*** adatok között eltárolt pontokat a térképen. Ehhez a jelenlegi működés megkeresi a lerakott ponthoz legközelebbi ***Junction***-t, majd az ehhez tartozó ***Segment*** elemek közül választja ki, hogy melyik irányszöge hasonlít a legjobban a ***Junction***, és a lerakott pont között felírható vektor irányszögéhez. Ez nem mindig az optimális ***Segment*** elemet találja meg, mert vannak olyan esetek, amikor a ***Segment*** két ***Junction*** pontja túl távol van a lerakott ponttól, ezért egy másik ***Segment***-hez tartozó ***Junction***-t talál meg a program. Emiatt lehet, hogy a kezdeti útvonal

máshonnan fog indulni, mint ahonnan a felhasználó generálni akarta az útvonalat. Ehelyett a megoldás helyett azt kellene eldönteni, melyik ***Segment*** helyezkedik el a legközelebb a felhasználó által lerakott ponthoz.



11.1. ábra: A pont egyenestől való távolságának ábrázolása [48]

$$d = \frac{|\det(x_2 - x_1 \;\; x_1 - x_0)|}{|x_2 - x_1|} \tag{9}$$

A (9) képletnél a ***det()*** egy mátrix determinánsát adja eredményül, a mátrix elemei pedig a zárójelek között felsorolt elemek.

A (9) képlet által [48] meg lehet határozni, hogy a felhasználó által lerakott pont, ami x_0 és y_0 koordinátákkal rendelkezik, milyen ***d*** távolságra van egy egyenestől. Ezzel a képlettel minden a ponthoz közeli ***Segment***-et megvizsgálva el lehetne dönteni, hogy melyik ***Segment***-hez van a legközelebb a lerakott pont. Ezt megvalósítva, a ***Segment*** megtalálása lassabb lett az irányszöges verziónál, illetve nem jó ***Segment*** elemek lettek megtalálva, ami miatt nem tudom használni ezt a megoldást. Hasonló módszerrel viszont meg lehetne határozni, melyik ***Segment***-hez van a legközelebb a lerakott pont.

12. FELHASZNÁLÓI LEÍRÁS

Az alkalmazást számítógépen lehet használni. Az alkalmazás használatához szükséges a Visual Studio szoftver. Az alkalmazást tartalmazó mappában két almappa található. Az Utvonaltervezes mappa tartalmazza a program Backend részét, ezen belül az Utvonaltervezes.sln fájlt megnyitva, majd a RouteGenerationApi projektet ISS Express módon elindítva lehet futtatni (Az alkalmazás tetején lévő zöld lejátszás jellel). A program Frontend részét a Terkep mappán belülről parancssor segítségével lehet elindítani, az „npm start” parancsot kiadva. Ha a program Backend része elindult, akkor automatikusan megnyílik egy swagger lap a böngészőben. A program Frontend része a parancs kiadás után elkezd felépülni, ami eltarthat egy kis ideig, ehhez a felhasználó türelmét szeretném kérni. Amint felépült az alkalmazás, akkor a böngészőben megjelenik egy újabb lap, ami a localhost:3000 címet tölti be. Ezen a címen használható az alkalmazáshoz szükséges interaktív térkép. A szoftver használatához a Google Chrome böngésző ajánlott.

A Frontend betöltése után, a bal alsó részen radio gombok segítségével kiválasztható milyen típusú generálást szeretnénk használni az útvonaltervezéshez, a felső részen pedig egy szövegdobozba beírható mekkora távolságban szeretnénk véletlenszerű útvonalat generálni. Ez a két funkció együtt nem működik, tehát a térképre lerakott pontok alapján dől el, hogy melyik generálás fog lefutni, illetve véletlenszerű generálásnál 0 km hosszú utakat nem lehet generálni, és a maximálisan generálható út hossza 20 km-re lett lekorlátozva. Az útvonaltervezés elindításhoz a térképre kattintva ki lehet választani pontokat, amiből egyszerre kettő lehet a térképen. Egy pont a véletlenszerű generálást, két pont pedig a pontok közötti legrövidebb út generálását fogják jelenteni. Az első kattintás a térképen ugyanoda helyezi a két pontot, és ezután minden második kattintás fogja lerakni a második pontot a térképre. Ha már lent van mindkét pont a térképen és újra kattintunk valamerre, akkor a lerakott két pont újra egy helyre fog kerülni, és csak egy ezutáni kattintással lehet megint máshova helyezni a második pontot. Tehát minden páratlan kattintás ugyanoda helyezi a pontokat, és minden páros kattintás csak a második pontot helyezi el máshova.

A pontok lerakása után, el lehet kezdeni az útvonal generálását. Az útvonal generálását a lap jobb alsó sarkában lévő gombbal lehet kezdeményezni. Amint elkészült a szoftver által készített útvonal, az megjelenik a térképen szürke pontokkal megjelölve. A generált út színátmenetes vonalként lesz megjelenítve. Az útvonal zöld része jelenti az elejét, és a kék része jelenti a végét. Az elhelyezett szürke pontok a fordulásokat jelentik az úton, és az ezekhez társított értékeket az oldal jobb felső sarkában, a távolság alatti területen lévő területen lehet leolvasni. Ezek az értékek sorrendben az útvonal elejétől, a zöld résztől jelentik egyesével az úton elhelyezett szürke pontokat az út végéig.

A jobb felső sarokban látható milyen hosszúságú a generált útvonal. A biztonságos generálás érdekében, a program lehetséges, hogy túl sokáig generálna útvonalakat. Ha a generálás időtartama tovább tart, mint öt perc, akkor véletlenszerű generálás esetén érdemes újra próbálkozni a gomb megnyomásával, ezzel egy másik véletlenszerű próbálkozást kezdeményezni. Két pont közötti generálás esetén, ha túl sokáig tart az útvonal generálása, az azt jelenti, hogy vagy nincs útvonal a két lerakott pont között, amit a szoftver csak akkor fog felfedezni, ha az egész térképet bejárta, vagy túl nagy a távolság a két pont között. A szoftver egyelőre nem támogatja a nagyobb távolságok közti generálásokat, ezért arra kérném a felhasználót, hogy vagy kisebb területek között generálja az adott útvonalakat, vagy ha meggyőződött arról, hogy a lerakott két pont között biztosan generálható út, (ezt az ellenőrzést kisebb generálásokkal le tudja ellenőrizni) akkor türelmét szeretném kérni az útvonal generálása megtörténik. Egy 15 kilométeres út generálása például 3 percig tart az átalakított A* algoritmussal.

Lehet, hogy előfordulnak olyan pontok a térképen, amik között nem történik meg az útvonal generálás. Ez az adatbázisban lévő adatok miatt van, mivel önkéntesek által bővül az OpenStreetMap által szolgáltatott térkép adatbázis. Az ilyen sikertelen útvonalak generálása miatt a felhasználó megértését szeretném kérni, mivel nem szerettem volna olyan szoftvert írni, ami nem biztonságos, vagy megközelíthetetlen helyekre vezetné a felhasználót.

Az útvonaltervezéshez használt gráf előre fel lett építve, és megtalálható a ***RouteGenerationApi*** projekten belül, a ***GraphElements*** mappában. A gráf generálását bármikor el lehet indítani az ***InitialGraphBuilding*** projekt segítségével, ha esetleg más területen szeretne útvonalakat generálni a felhasználó. Ehhez az ***OSMFiles*** mappába kell behelyezni egy ***osm*** kiterjesztésű fájlt, illetve a kódban is át kell írni a beolvasott fájl nevét, a ***GraphBuilder*** osztályon belül, a 25. sornál. A generált xml fájlokat, ilyenkor be kell másolni a ***GraphElements*** mappába, a megfelelő elnevezéssel.

Lehetőség van a kiértékeléshez használt ***Evaluations*** projekt futtatására is, amit szintén a ***VisualStudio***-ból lehet elindítani. Az ***Evaluations*** projekt indításakor, egy menü várja a felhasználót, ahol számok megadásával ki tudja választani melyik fajta algoritmus működését szeretné kiértékelni.

13.ÖSSZEFOGLALÁS

A szakdolgozatomhoz kitűzött célok mindegyikét sikerült megvalósítanom. Budapest térképének felépítését elkülönítettem a szoftver többi részétől, mivel egy ekkora méretű gráfnál ügyelnem kellett a kétirányúság biztosítására is, több mint 35 óráig tartott Budapest szűrt térképének a felépítése. A gráf felépítése után generált fájljaim betöltésével sokkal gyorsabban, csupán 6 másodperc alatt be tudtam tölteni Budapest teljes, megfelelően szűrt és felépített térképét a memóriába.

Az útvonal generáló algoritmusok közül mind az A*, és a Dijkstra algoritmust is megvalósítottam, illetve ezeknek egy átalakított, nem járdákat előnyben részesítő verzióit is sikerült megcsinálnom. Ezek két pont közötti útvonal generálásához megfelelően működnek, ráadásul a tesztelés, illetve a kiértékelés során bemutatott esetekben, bizonyos szemszögekből még más alkalmazásokkal szemben is jobban teljesít. A véletlenszerű generáláshoz is sikerült elkészítenem egy külön algoritmust, amihez az A* algoritmus már átalakított alternatív verzióját módosítottam tovább. A véletlenszerű generálás, habár nagy távolságoknál lassú, nagyon pontos. A generáló algoritmusok kiértékelését elvégeztem, amiből gyakorlatban is látható, hogy az A* algoritmus mennyivel jobban teljesít a Dijkstra algoritmusnál. Az útvonaltervezéshez használt súlyszámításokhoz a Haversine formula pontos súlyok megadására is használható volt, viszont optimális heurisztikaként is alkalmazni tudtam.

A generált útvonalak megjelenítéséhez, illetve a generálás kezdeményezéséhez egy Frontend részt is megvalósítottam. Ez REST API kérések segítségével kommunikál a program Backend részével, ahonnan ***json*** formában kapja a megfelelően generált adatokat. A Frontend oldalon a felhasználó kiválaszthatja, hogy milyen útvonaltervezési algoritmus szerint szeretne utakat generálni két pont között, a pontokat pedig egy interaktív térkép segítségével tudja meghatározni, amihez a Mapbox által szolgáltatott térképet használtam. Ezen kívül pedig lehetősége van egy távolság megadásával egyetlen pontból útvonalat generálni, ami oda fog visszaérkezni, ahonnan indult, alakja pedig egy körútra fog hasonlítani.

Az útvonal generálása során meg tudom határozni, hogy a térkép melyik pontja fekszik legközelebb a felhasználó által lerakott ponthoz, majd a két pont közötti vektor irányszöge által azt is el tudom dönteni, melyik ***Segment*** elemen fekszik a felhasználó által lerakott pont. Ez nem az optimális működése a programnak, de a ***11. fejezetben*** kifejtettem miért nem működik optimálisan máshogy a kezdő és célpont kiválasztása. Ezen kívül az útvonaltervezés által generált út széleinek rendbe rakása is működik a felhasználó által lerakott pontok szerint, hogy ténylegesen a lerakott két pontja között jelenjen meg az útvonal a térképen, illetve az utat felépítő útszakaszok, vagyis ***Segment*** elemek között is meg tudtam határozni, mikor merre kell fordulnia a felhasználónak. A tervezett útvonal hosszának számításakor mindig a már módosított útvonalat adom vissza, ezért ez is a megfelelő értéket adja vissza.

14.SUMMARY

I have achieved all the goals of my thesis project. The building of the graph of Budapest was separated from the software's other parts, because it took over 35 hours to build since I wanted to assure the two-way properties of each path, because the software is designated to people who want to walk or run. Loading the already built graph is much faster, it takes only 6 seconds.

I implemented both A* and Dijkstra algorithms, furthermore I also implemented an alternative version for both algorithms, that prioritizes footways over paths that are sidewalks. These algorithms can be used to generate routes between two points on the map, but for the randomly generated route another implementation was needed. The random route generation is generating a route that has the same starting point as it's ending point. I modified the alternative A* algorithm to implement this random generation. This random generation might be slower, the longer the route's required length is, but it is more precise in staying close to the required length. The evaluation and comparison of these algorithms had been done between them, and between other applications as well. It can be seen from these comparisons that the A* algorithm is much faster than the Dijkstra algorithm. The Haversine formula could be used for both the exact distance calculation, and the estimated distance calculation, that could be used for an optimal heuristic.

To display the generated routes, and to be able to initiate these generations, I used an incteractive map provided by Mapbox, at the Frontend part of the application. The communication between the Frontend and the Backend is done via REST API calls. The user can select which route generating algorithm they want to use, or if they want to do a random generation, then to determine what the length of that generation should be. The shape of these random generations is close to a round route.

The determination of which point in the graph is the closest to the placed point on the map can was done, and after that, using the angles of vector I could decide on which segment the said point lies on, but this is not perfect. I also modified the geocoordinates at the two ends of the generated route, so the software would really generate a route between the placed two points. I also determined where the user would have to turn at certain points of the generated route, and the total length of the route was also determined, by including the modifications at the two ends of the route.

15.IRODALOMJEGYZÉK

[1] D. Schultes, Route Planning in Road Networks, „1.1.1 The Shortest-Path Problem”, 2008

[2] A. Lookingbill, E. Russel, Google, „Google Maps 101: how we map the world”, „Then you add data”, 2019, Utoljára megtekintve: 2022.11.12

https://blog.google/products/maps/google-maps-101-how-we-map-world/

[3] Google, Google Maps Platform, Directions API overview, utoljára megtekintve: 2022.11.01

https://developers.google.com/maps/documentation/directions/overview

[4] OpenStreetMap, 2021, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/About_OpenStreetMap

[5] S. Crawford, D. Westergren, „How Trail Router works”, 2020, Utoljára megtekintve: 2022.11.12

https://trailrouter.com/blog/how-trail-router-works/

[6] K. Podobni, A. Nagy, Legrövidebb útkereső algoritmusok, „2 Gráfelméleti fogalmak, jelölések”, 2009

[7] R. Sedgewick, K. Wayne, Algorithms, 4th edition, „4.1 Undirected Graphs”, page 518, 2020

[8] K. Podobni, A. Nagy, Legrövidebb útkereső algoritmusok, „3 Legrövidebb út? Mit jelent ez?”, 2009

[9] L. Metcalf, W. Casey, Cybersecurity and Applied Mathematics, „Chapter 5 - Graph theory”, Pages 67-94, 2016, ISBN 9780128044520

[10] Weisstein, Eric W. „Weighted Graph” From MathWorld – A Wolfram Web Resource. Utoljára megtekintve: 2022.11.18.

https://mathworld.wolfram.com/WeightedGraph.html

[11] R. Zhou, E. A. Hansen, Artificial Intelligence, „Breadth-first heuristic search”, Volume 170, Issues 4–5, Pages 385-408, 2006, ISSN 0004-3702

[12] Cormen, Thomas H. Introduction to algorithms, "22.2 Breadth-first search", 2009, ISBN 978-81-203-4007-7

[13] Dr. S. Szénási, Dr. Z. Vámossy, Algoritmusok, adatszerkezetek II., „7.4 Legrövidebb út keresése (Dijkstra algoritmus)”, Budapest, 2018, ISBN 978-615-5460-43-2

[14] K. Podobni, A. Nagy, Legrövidebb útkereső algoritmusok, „4 Az algoritmusok; 4.1.2 A Dijkstra algoritmus”, 2009

[15] Cormen, Thomas H., Introduction to algorithms, „24.3 Dijkstra’s algorithm”, page 659., 2009, ISBN 978-81-203-4007-7

[16] S. J. Russell, P. Norvig, Artificial Intelligence A Modern Approach Third Edition, „3.5.1 Greedy best-first search”, 2010, New Jersey

[17] S. J. Russell, P. Norvig, Artificial Intelligence A Modern Approach Third Edition, „3.5.2 A* search: Minimizing the total estimated solution cost”, 2010, New Jersey

[18] P. E. Hart, N. J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, II. AN ADMISSIBLE SEARCHING ALGORITHM, (1968), IEEE Transactions of Systems Science and Cybernetics, SSC-4, 2

[19] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, L. Jurišica, Procedia Engineering, „Path Planning with Modified a Star Algorithm for a Mobile Robot”, „2. A* algorithm”, Volume 96, Pages 59-69, 2014, ISSN 1877-7058

[20] D. Andiwijayakusuma, A Comparative Study of the Algorithms for Path finding to Determine the Adversary Path in Physical Protection System of Nuclear Facilities, A* algorithm, 2019 J. Phys.: Conf. Ser. 1198 092002

[21] R Suwanda, Analysis of Euclidean Distance and Manhattan Distance in the K-Means Algorithm for Variations Number of Centroid K, *„3.4. Manhattan Distance”*, 2020 J. Phys.: Conf. Ser. 1566 012058

[22] Craw, S. (2016). Manhattan Distance. In: Sammut, C., Webb, G. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-7502-7_511-1 [utoljára megtekintve: 2022.10.28]

[23] S. Karki, H. S. Ranjitkar, Comparison of A*, Euclidean and Manhattan distance using Influence Map in Ms. Pac-Man, „1.3 Euclidean, Manhattan distances and A* search algorithm”, 2016, Faculty of Computing Blekinge Institute of Technology, SE-371 79, Karlskrona Sweden

[24] R Suwanda, Analysis of Euclidean Distance and Manhattan Distance in the K-Means Algorithm for Variations Number of Centroid K, *„3.3. Euclidean Distance”*, 2020 J. Phys.: Conf. Ser. 1566 012058

[25] B. O'Neill, Elementary Differential Geometry (Second Edition), Pages 43-99, *„Chapter 2 - Frame Fields”, „2.1 Dot Product”, „1.2 Definition”*, 2006, ISBN 9780120887354

[26] J.-P. Rodrigue, The Geography of Transport Systems, „The Great Circle Distance”, 2020, New York: Routledge, ISBN 978-0-367-36463-2

[27] A. Patel, Introduction to A*, „The A* Algorithm”, 1997, Utoljára megtekintve: 2022.11.20.
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[28] OpenStreetMap, 2016, Utoljára megtekintve: 2021.05.09,

https://learnosm.org/en/osm-data/file-formats/

[29] OpenStreetMap, 2020, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/OSM_XML

[30] OpenStreetMap, 2021, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Elements

[31] OpenStreetMap, 2021, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Node

[32] OpenStreetMap, 2020, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Tags

[33] OpenStreetMap, 2021, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Way

[34] Google Maps platform, Documentation, „What is the Google Maps Platform?” 2022, utoljára megtekintve: 2022.10.29

https://developers.google.com/maps/faq#whatis

[35] Google Maps platform, Documentation, „Getting started with Google Maps Platform”, „Billing Account Credits”, 2022, utoljára megtekintve: 2022.10.29

https://developers.google.com/maps/get-started

[36] HERE, HERE Maps API for JavaScript, „Introduction”, 2022, utoljára megtekintve: 2022.10.29

https://developer.here.com/documentation/maps/3.1.35.1/dev_guide/index.html

[37] HERE, HERE Base Plan Pricing, „HERE Map Rendering”, 2022, utoljára megtekintve: 2022.10.29

https://www.here.com/get-started/pricing

[38] HERE, HERE Vector Tile API, „Introduction”, 2022, utoljára megtekintve: 2022.10.29

https://developer.here.com/documentation/vector-tiles-api/dev_guide/index.html

[39] Mapbox, Mapbox pricing, „Maps”, utoljára megtekintve: 2022.10.29

https://www.mapbox.com/pricing/#maploads

[40] Mapbox, Mapbox data, „Use our data”, utoljára megtekintve: 2022.10.29

https://docs.mapbox.com/help/getting-started/mapbox-data/#use-our-data

[41] Mapbox, Mapbox GL JS, v2.8.2, Utoljára ellenőrizve: 2022.04.30,

https://docs.mapbox.com/mapbox-gl-js/guides/

[42] Mapbox, Use cases, v2.8.2, Utoljára ellenőrizve: 2022.04.30,

https://docs.mapbox.com/mapbox-gl-js/guides/#use-cases

[43] Mapbox, Key concepts, v2.8.2, Utoljára ellenőrizve: 2022.04.30,

https://docs.mapbox.com/mapbox-gl-js/guides/#key-concepts

[44] OpenStreetMap, 2020, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Junctions

[45] OpenStreetMap, 2021, Utoljára ellenőrizve: 2021.05.09,

https://wiki.openstreetmap.org/wiki/Segment

[46] Z. Arifin, M. R. Ibrahim and H. R. Hatta, "Nearest tourism site searching using Haversine method,", „II. Haversine Method”, *2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE),* 2016, pp. 293-296, doi: 10.1109/ICITACEE.2016.7892458.

[47] Microsoft, .NET fundamentals, Managed threading, *„Destroying threads”,* 2022, Utoljára megtekintve: 2022.11.20

https://learn.microsoft.com/en-us/dotnet/standard/threading/destroying-threads

[48] Weisstein, Eric W. "Point-Line Distance--2-Dimensional." From MathWorld--A Wolfram Web Resource. Utoljára megtekintve: 2022.11.20.

https://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html