



EÖTVÖS LÓRÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK

TANSZÉK

Heurisztikus útvonalkeresés

Témavezetők:

Király Zoltán

egyetemi docens,
ELTE TTK,
Számítógéptudományi Tanszék

Szerző:

Bibók Tamás

Programtervező informatikus BSc,
nappali tagozat

Kovács Péter

programtervező matematikus
ChemAxon kft.
ELTE IK,
Algoritmusok és Alkalmazásaiak Tanszék

Budapest, 2011

Tartalomjegyzék

1. Bevezetés.....	5
1.1. LEMON.....	5
1.2. A szakdolgozat célja.....	6
1.3. Dijkstra.....	6
1.4. Kétirányú Dijkstra.....	7
1.5. A*.....	7
1.6. Kétirányú A*.....	7
2. Felhasználói dokumentáció.....	9
2.1. Előfeltételek, telepítés.....	9
2.1.1. GNU C++ Compiler.....	9
2.1.2. LEMON.....	9
2.2. A LEMON használata.....	10
2.2.1. A LEMON alapvető fogalmai.....	10
2.2.2. A LEMON és a biztonságos kód.....	11
2.2.3. Általánosság, kontrollálhatóság.....	11
2.3. Az algoritmusok.....	12
2.3.1. Az egyirányú Dijkstra algoritmus.....	12
2.3.2. A kétirányú Dijkstra.....	15
2.3.3. Az egyirányú A*.....	17
2.3.4. A kétirányú A*.....	22
2.3.5. Általános konklúzió.....	24
2.4. Az algoritmusok felhasználása.....	25
2.4.1. Az egyirányú Dijkstra használata.....	25
2.4.1.1. Példányosítás.....	25
2.4.1.2. A run metódus.....	25
2.4.1.3. A dist metódus.....	26
2.4.1.4. A processed metódus.....	26
2.4.1.5. A currentDist metódus.....	26
2.4.1.6. A reached metódus.....	26
2.4.1.7. A path, predNode és predArc metódusok.....	27
2.4.1.8. Egyéb metódusok.....	27
2.4.1.9. Belső tárolók lekérdezése, módosítása.....	27

2.4.1.9. Wizardok.....	27
2.4.1.10.Biztonság.....	28
2.4.2. A kétirányú Dijkstra használata.....	28
2.4.2.1. Példányosítás, indítás.....	28
2.4.2.2. Az addTarget metódus.....	28
2.4.2.3. A processNextNodeBi metódus.....	29
2.4.2.4. Az eredeti metódusok kétirányú változatai.....	29
2.4.3. Az egyirányú A* használata.....	29
2.4.3.1. Példányosítás.....	29
2.4.3.2. Az algoritmus indítása.....	30
2.4.3.2. A setEstimateCalculator metódus.....	31
2.4.3.3. A módosított addSource metódus.....	31
2.4.3.4. A processNextNode metódus módosított változata.....	31
2.4.3.5. Belső tárolók.....	31
2.4.4. A kétirányú A* használata.....	32
2.4.4.1. Példányosítás, futtatás.....	32
2.4.4.2. A processNextNodeBi metódus.....	32
2.4.4.3. Az eredeti metódusok kétirányú változatai.....	32
3. Fejlesztői dokumentáció.....	33
3.1. Terminológia.....	33
3.2. Az algoritmusok részletes vizsgálata.....	34
3.2.1. Az egyirányú Dijkstra algoritmus.....	34
3.2.1.1. Az algoritmus menete.....	34
3.2.1.2. Az algoritmus pszeudokódja.....	35
3.2.1.3. Az algoritmus helyessége.....	36
3.2.1.4. Az algoritmus előnyei, hátrányai.....	37
3.2.2. A kétirányú Dijkstra algoritmus.....	38
3.2.2.1. Az algoritmus menete.....	39
3.2.2.2. A megállási feltétel, és a legrövidebb út kiválasztása.....	39
3.2.2.3. Az algoritmus helyessége.....	40
3.2.2.4. Az algoritmus előnyei, hátrányai.....	40
3.2.3. Az egyirányú A* algoritmus.....	42
3.2.3.1. Az algoritmus menete.....	42
3.2.3.2. Az algoritmus pszeudokódja.....	42

3.2.3.3. Az algoritmus helyessége.....	43
3.2.3.4. Az algoritmus előnyei, hátrányai.....	44
3.2.4. A kétirányú A* algoritmus.....	46
3.2.4.1. Az algoritmus menete.....	46
3.2.4.2. Gyorsítások a legrövidebb útra vonatkozó felső becslés segítségével.	46
3.2.4.3. Az algoritmus pszeudokódja.....	47
3.2.4.4. Az algoritmus előnyei, hátrányai.....	48
3.3. Az algoritmusok implementációja.....	50
3.3.1. Az egyirányú Dijkstra algoritmus.....	50
3.3.1.1. A legfontosabb adattagok bemutatása.....	50
3.3.1.2. Az algoritmus fő logikáját végző metódus, a processNextNode.....	50
3.3.2. A kétirányú Dijkstra algoritmus.....	51
3.3.2.1. A legfontosabb adattagok bemutatása.....	51
3.3.2.2. Az algoritmus fő logikáját végző metódus, a processNextNodeBi.....	52
3.3.3. Az egyirányú A* algoritmus.....	53
3.3.3.1. A legfontosabb adattagok bemutatása.....	53
3.3.3.2. Az algoritmus fő logikáját végző metódus, a processNextNode.....	54
3.3.4. A kétirányú A* algoritmus.....	54
3.3.4.1. A legfontosabb adattagok bemutatása.....	54
3.3.4.2. Az algoritmus fő logikáját végző metódus, a processNextNodeBi.....	55
3.4. Tesztelés.....	55
3.4.1. Helyesség.....	56
3.4.1.1. A használt gráfok.....	56
3.4.1.2. Forrás- és célcímsok, útvonalak.....	58
3.4.1.3. Heurisztika.....	58
3.4.2. Eredmények, összehasonlítások.....	59
3.4.2.1. Az „utazási idők” New York gráf.....	60
3.4.2.1.1. Futásidő.....	60
3.4.2.1.2. Vizsgált csúcsok száma.....	62
3.4.2.1.3. Egyéb mérőszámok.....	65
3.4.2.2. A New York úthálózatgráf.....	67
3.4.2.3. A „simított” New York gráf.....	68
3.4.3. Speciális tesztek.....	69
3.4.3.1. Pontos heurisztika.....	69

3.4.3.2. Az „utófázis” javításának elhagyása.....	70
3.4.3.3. Az algoritmus leállítása a két oldali keresés találkozása esetén.....	72
3.4.3.4. Alternálás szabályának változtatása.....	74
3.5. Továbbfejlesztési lehetőségek.....	75
3.5.1. Az algoritmusok általánosítása, közös felület alá hozása.....	75
3.5.2. Az irányválasztás funktorról alakítása.....	75
3.5.3. Két külön heurisztika megadása a kétirányú A* esetén.....	76
3.5.4. A kétirányú A* utófeldolgozásának kiemelése.....	76
3.5.5. Kétirányú algoritmusok többszálúvá tétele.....	76
4. Irodalomjegyzék.....	78

1. Bevezetés

A gráfelmélet története egészen 1736-ig nyúlik vissza, a híres königsbergi hidak problémájára tekintünk a matematika ezen területének kezdőpontjaként. A gráfok leginkább kapcsolatok, például útvonalak, ismeretségek leírására alkalmasak, ám évszázadokon keresztül inkább elméleti jelentőségük volt, hiszen gyakorlati alkalmazásuknak gátat szabott a kezelhetetlenségük, nem lehet nagyméretű gráfokat ésszerű keretek között ábrázolni. Ezen kezelhetetlenség jelentősen korlátozta a gráfokkal való operációkat, így jelentősen korlátozott volt felhasználhatóságuk.

Ezen a helyzeten alapvetően változtatott a számítástechnika megjelenése és gyors fejlődése, melynek köszönhetően a gráfelmélet felhasználása is robbanásszerű növekedésen ment keresztül. Hatalmas út- és kapcsolati hálózatok könnyedén tárolhatóvá váltak, és lehetővé vált ezeken a gráfokon műveletek végrehajtása, például útvonalkeresés.

Ám még a rohamosan növekvő tárhely és számítási kapacitás ellenére is nagy kihívást jelent az igazán nagy gráfok kezelése.

1.1. LEMON

Ezen problémakört célozza meg a LEMON (Library for Efficient Modeling and Optimization in Networks) projekt, mely egy főképp gráfok tárolásával és gráfalgoritmusokkal foglalkozó C++ könyvtár. [2]

A LEMON egy magyar projekt, 2003-ban indította az EGRES (Egerváry Research Group on Combinatorial Optimization) [3], amely egy elméleti kombinatorikus optimalizálással foglalkozó, az ELTE-vel és az MTA-val szoros kapcsolatban álló kutatócsoport. A LEMON szintén főképp ELTE-s projekt, több BME-s részvételével. Legfőbb területe a gráfok, gráfalgoritmusok, nagy hangsúlyt fektetve a hatékonyságra. Teljesen nyílt forráskódú, a benne lévő licensz leírás megtartásával akár kereskedelmi célokra is használható.

A LEMON fő célja a hatékonyság mellett az általánosság, a rugalmasság, tehát a minél szélesebb körű felhasználhatóság. Ezt általános gráfkonceptiókkal, és messzemenően generikus algoritmusokkal valósítja meg. A nagyfokú szabadság, és hatékonyságközpontú szemlélet következményeként a LEMON-ban szereplő algoritmusok, osztá-

lyok kisebb hangsúlyt fektetnek a biztonságos kódra, hiszen a mindenre kiterjedő ellenőrzések aránytalanul lassítanák a kódot, így a könyvtár fő célkitűzése veszne el. Ezért a LEMON-beli kódok feltételezik használójukról, hogy tisztában van a követelményekkel, és ügyel az előfeltételek betartására. Ezen ismereteket a minél részletesebb dokumentáció hivatott szolgáltatni.

1.2. A szakdolgozat célja

Ezen szakdolgozat célja három útvonalkereső gráfalgoritmus (kétirányú Dijkstra, A* és kétirányú A*) implementálása és integrálása a LEMON-ba. Mivel az algoritmusok a LEMON szerves részévé válnak, ezért nagy hangsúlyt fektettem annak követelményeinek teljesítésére, valamint a rá jellemző szemlélet elsajátítására és alkalmazására.

Az implementált algoritmusok, valamint számos teszteset megtalálható [17]-ben.

1.3. Dijkstra

Az útvonalkereső algoritmusok közös célja, hogy megtalálják egy vagy több kiindulópontból egy vagy több célpontba a legrövidebb távolságot, és a legrövidebb utat egyaránt. Gyakorlati alkalmazásban sokszor elegendő közel legrövidebb utat számolni megnövekedett sebességért cserébe.

Mind a három, a szakdolgozat részét képező algoritmus alapjául a Dijkstra nevűvel fémjelzett algoritmus szolgál.

Edsger Wybe Dijkstra [4] dán számítógéptudós volt. Nagy szerepe volt a kontrollált ciklusok (while, for, do-while) előretörésében, 1968-ban megjelent „A Case against the GO TO Statement” című cikke nagyban hozzájárult ezek előterbe kerülésében. Neki tulajdonítják a „two or more, use a for” szólást is.

Munkásságának talán leghíresebb eleme az 1959-ben publikált Dijkstra-algoritmus [5]. Ennek lényege, hogy minden lépésben a forrástól legkisebb távolságban lévő, de még nem vizsgált csúcspontot vizsgálja meg, addig folytatva ezt, míg el nem ér a célcsúcshoz. Ezen folyamat közben minden csúcshoz feljegyzi a forráscsúcstól való pillanatnyilag ismert legrövidebb távolságát, valamint a hozzá vezető eddig megtalált legrövidebb úton az öt megelőző csúcsot. Ha a célcsúcs is meg lett vizsgálva, a megelőzőkön keresztül visszafejthető a legrövidebb út a forráscsúcsig.

1.4. Kétirányú Dijkstra

Kétirányú Dijkstra esetén egyszerre két Dijkstrát futtatunk. Az egyik az eredeti módon fut, a forráscsúcstól a célcímsel felé. A másik viszont a célcímcstól halad a forrás-címsel felé, egy módosított gráfon, amit úgy kapunk az eredetiből, hogy megfordítjuk minden él irányítottságát. A két oldalt valamilyen szabály szerint alternálva futtatjuk. Az algoritmus akkor fejeződik be, ha a két oldal találkozik, lesz olyan csúcs, amelyet már mindkét oldal megvizsgált.

1.5. A*

Az A* algoritmus [6] a Dijkstra általánosítása. 1964-ben Nils Nilsson egy heurisztikus becsléssel bővítette a Dijkstra algoritmust. Ezen bővítést javította tovább 1967-ben Bertram Raphael, ezzel egy az eredeti Dijkstránál sokkal hatékonyabb algoritmust hozva létre (amennyiben csak egy célcímsba keresünk legrövidebb utat, és rendelkezésünkre áll egy alkalmas heurisztikus becslés). Egy évvel később Peter E. Hart bebizonyította ennek helyességét, ekkor kapta meg az A* nevet.

Egy függvényt kell megadni az algoritmus számára, melyet heurisztikus becsléseként fog használni. Ezt a heurisztikus becslést az egyszerűség kedvéért a továbbiakban egyszerűen csak heurisztikának fogom nevezni. Ezen heurisztika segítségével módosítjuk a keresés irányát, így csökkentve a vizsgált csúcsok számát és a futáshoz szükséges időt. A heurisztika célja, hogy a keresés minél jobban a célcíms irányába tolódjon el, így gyorsítva az eljárást. A megadott függvény legfontosabb kritériuma, hogy minden csúcs esetén egy nemnegatív, konzisztens alsó becslést adjon a célcímcstól való távolságra. Minél jobb az alsó becslés, azaz minél közelebb esik az adott csúcs tényleges távolságára a célcímcstól, annál hatékonyabb lesz a heurisztika, kevesebb a vizsgált csúcs, és gyorsabb a futási idő. A heurisztika konzisztenciája azt jelenti, hogy semelyik két csúcs heurisztikájának értéke nem térhet el nagyobb mértékben egymástól, mint a közöttük lévő él súlya.

1.6. Kétirányú A*

A kétirányú A* az első két algoritmus ötletének ötvözése. A kétirányú Dijkstrához hasonlóan két A* fut egyszerre, egy a forráscsúcstól a célcíms felé az eredeti gráfon,

egy pedig a célcímetől a forráscímsúcs felé a fordított gráfon. Itt a kétirányú Dijkstránál használt megállási feltétel nem elegendő, bár gyakorlati alkalmazás szempontjából van előnye, hiszen ugyan nem ad minden teljesen pontos eredményt, de cserébe gyorsabb. Ehelyett a két oldal összeérésé után egy erős felső becslést kapunk a legrövidebb útra, melynek segítségével jelentősen csökken a vizsgálandó csúcsok száma. Ezen felső becslést folyamatosan szűkítjük, míg a tényleges legrövidebb utat meg nem találjuk.

Mivel mind a három algoritmus alapjául a Dijkstra szolgál, ezért helyességüket, valamint időbeli és gráfbejárásbeli hatékonyságukat, továbbá implementációbeli eltéréseiket ezen algoritmushoz viszonyítva fogom bemutatni.

2. Felhasználói dokumentáció

2.1. Előfeltételek, telepítés

Az algoritmusok a LEMON könyvtár részei, ennek alkotó elemeit aktívan használják. Így futtatásukhoz elengedhetetlen a LEMON környezet telepítése.

LEMON elérhető Linuxon, Windowson és Mac OS-en is, valamint számos C++ compiler használható hozzá. Ezen szakdolgozat Ubuntu operációs rendszeren és GNU C++ Compliler-rel készült, így ezen specifikus esetet részletezem.

2.1.1. GNU C++ Compiler

A GNU C++ Compiler (továbbiakban csak g++) telepítéséhez először a

sudo apt-get update

parancssal frissíteni kell a letölthető csomagok listáját. Ezután a

sudo apt-get install build-essential

parancssal telepíthető egy csomag, ami a g++-t is magában foglalja.

2.1.2. LEMON

A LEMON telepítéséhez, mint azt a [7] is leírja, már több lépés elvégzése szükséges:

- **wget http://lemon.cs.elte.hu/pub/sources/lemon-1.2.1.tar.gz**
letölti az 1.2.1. verziójú LEMONT.
- **cd lemon-1.2.1**
megnyitja a lemon könyvtárat
- **./configure**
végrehajt pár ellenőrzést, és létrehozza a makefile-okat
- **make**
lefordítja a LEMON nagy részét egy libmemon.a fájlba
- **make check**
leellenőrzi, hogy minden működőképes-e az adott platformon
- **make install**

a végső helyére másolja a mappaszerkezetet

2.2. A LEMON használata

Mint azt a bevezetőben már említettem, a LEMON két fő célja a hatékonyság és az általánosság. Ezek elérése érdekében számos kompromisszumot kellett kötni.

Az egyik legfontosabb kompromisszum, hogy az osztályok, algoritmusok, adatszerkezetek fejlesztésénél nagy bizalmat fektet a felhasználóba, annak tájékozottságába és körültekintésébe.

2.2.1. A LEMON alapvető fogalmai

Legfőképpen a LEMON alapvető koncepcióit szükséges ismerni. Hűen a céltéma-körhöz, mindennek a központjában a gráf [8] áll. Több gráfosztály is létezik (pl.: ListGraph, ListDigraph, SmartGraph, SmartDigraph, StaticGraph, FullGraph, FullDigraph), ám a lényeg a gráf koncepciókban rejlik.

Két alap gráfkoncepció [9] létezik, a Graph concept és a Digraph concept. Ezek amolyan gyűjtőosztályok rendre az irányítatlan és az irányított gráfoknak. Egy közös interfész biztosítanak, melyen keresztül elérhetők a legfontosabb tulajdonságok, iterátorok, azonosítók. Például minden irányított gráfban [10] lévő ponthoz példányosíthatunk csak a bemenő, vagy épp csak a kimenő éleihez tartozó iterátort, lekérhetjük egy él cél- vagy forráscsúcsát. Mind irányított, mind irányítatlan gráfokhoz létezik iterátor a csúcs-pontokra és az élekre is, minden csúcspontnak vagy élnek le lehet kérdezni az azonosítóját, vagy épp lekérdezhetjük a csúcspontok számát.

A LEMON-ban szereplő algoritmusok túlnyomó többsége ezen gráfkoncepciókkal dolgozik az egyes gráfosztályok helyett, így könnyedén használhatjuk őket akár saját magunk által megírt gráfosztályokkal is, feltéve, ha azok megfelelnek az adott koncepciónak.

A gráfok hatékony tárolása és fent említett hasznos tulajdonságaik létfontosságúak a hatékony algoritmusok szempontjából, ám talán még nagyobb szerep jut a hozzájuk rendelhető mapeknek [11] és adaptoroknak [12]. Mapekben tárolódnak az élsúlyok, vagy épp a speciális csúcscímek, de akár egyedileg kiszámolt értékeket is visszaadhatnak, az eredeti érték módosítása nélkül. Ennél is több potenciál rejlik az adaptorokban.

Ezek segítségével meg lehet fordítani gráfok éleinek irányítását, vagy éppen egy részgráfjukat lehet tekinteni, elhagyva bizonyos csúcsokat vagy éleket az eredeti gráfból, így egy új gráfot képezve. Ez csak csekély plusz számítási igénnyel rendelkezik, majdnem ugyanolyan hatékonyan hajthatók végre műveletek az így kapott gráfon, mint ha az eredeti gráfon végeznénk el őket, ám nem kell egy új gráfot eltárolni, ráadásul dinamikusan, gyorsan változtathatóak ezek az adaptorok, különösebb számolás- vagy tárhelyigény nélkül.

2.2.2. A LEMON és a biztonságos kód

A már említett kompromisszumok közül talán a legkomolyabb a biztonságos kód terén jelenik meg. Általános vélekedés, hogy egy átlagos (biztonságos) program legalább 40%-a ellenőrzés, hibakezelés. Ez nem csak a kód terjedelmében és a fejlesztésére fordított időben jelentkezik, hanem a futási időben is. A LEMON-nál a hatékonyság kulcskérdés, ezért ezen ellenőrzések gyakorlatilag teljes egészében a kódot felhasználóra maradnak. Ez nagy körültekintéssel jár az algoritmusok használása folyamán, cserébe viszont végletekig optimalizált működés az eredmény.

Ezen utóbbi tulajdonság miatt is fontos, hogy a felhasználó alaposan tanulmányozza át a használni kívánt részeket, hiszen csak ekkor lehet elkerülni a nem kívánt hibásokat. Ebben nagy segítségére lehet a részletes dokumentáció.

2.2.3. Általánosság, kontrollálhatóság

A fentebb említett követelmények mellett számos lehetőséget is kap a LEMON-t használó programozó.

A C++ szemléletben általánosan is jellemző generikusságra nagy hangsúlyt fektet a LEMON. Ez megnyilvánul a már tárgyalt gráfkonceptiókon, és egyéb koncepciókon keresztül is, de ezeknél még szélesebb, az algoritmusok többségénél a lehető legtöbb tulajdonságot, típust meg lehet változtatni, például a Dijkstra algoritmusnál a tetszőleges élsúlytípuson kívül akár egyedi összeadás operátort, nullértéket és összehasonlító operátort is meg lehet adni. Ezen lehetőségekkel a legspeciálisabb célokra ugyanúgy alkalmas lesz, mint a hagyományosakra, hatékonyságromlás nélkül. Például a Dijkstra algoritmus segítségével könnyen kereshetünk legszélesebb utat, amennyiben a „kisebb” operátort

„nagyobb”-ra, az összeadás operátort minimumot adó függvényre, a nullértéket pedig plusz végtelenre (vagy kellően nagy értékre) állítjuk.

A másik jelentős előny a kontrollálhatóság nagy mértéke. A legtöbb algoritmust akár egy utasítással is le lehet futtatni, de lehetőség van akár lépésről lépésre szabályozni a működést, így az igényeknek megfelelően időzíteni a futást, vagy éppen futás közben lekérni, sőt megváltoztatni az éppen aktuális belső állapotokat.

Összefoglalva tehát a LEMON használata nagyfokú körültekintést és tanulmányozást igényel, ám ezek figyelembe vételével egy általánosan használható, hatékony és nagy mértékben kontrollálható eszközt nyújt.

2.3. Az algoritmusok

Az algoritmusok megfelelő, helyes és hatékony felhasználásának elengedhetetlen feltétele, hogy tisztában legyünk az adott algoritmus működésével, céljával, előnyeivel, hibáival és előfeltételeivel.

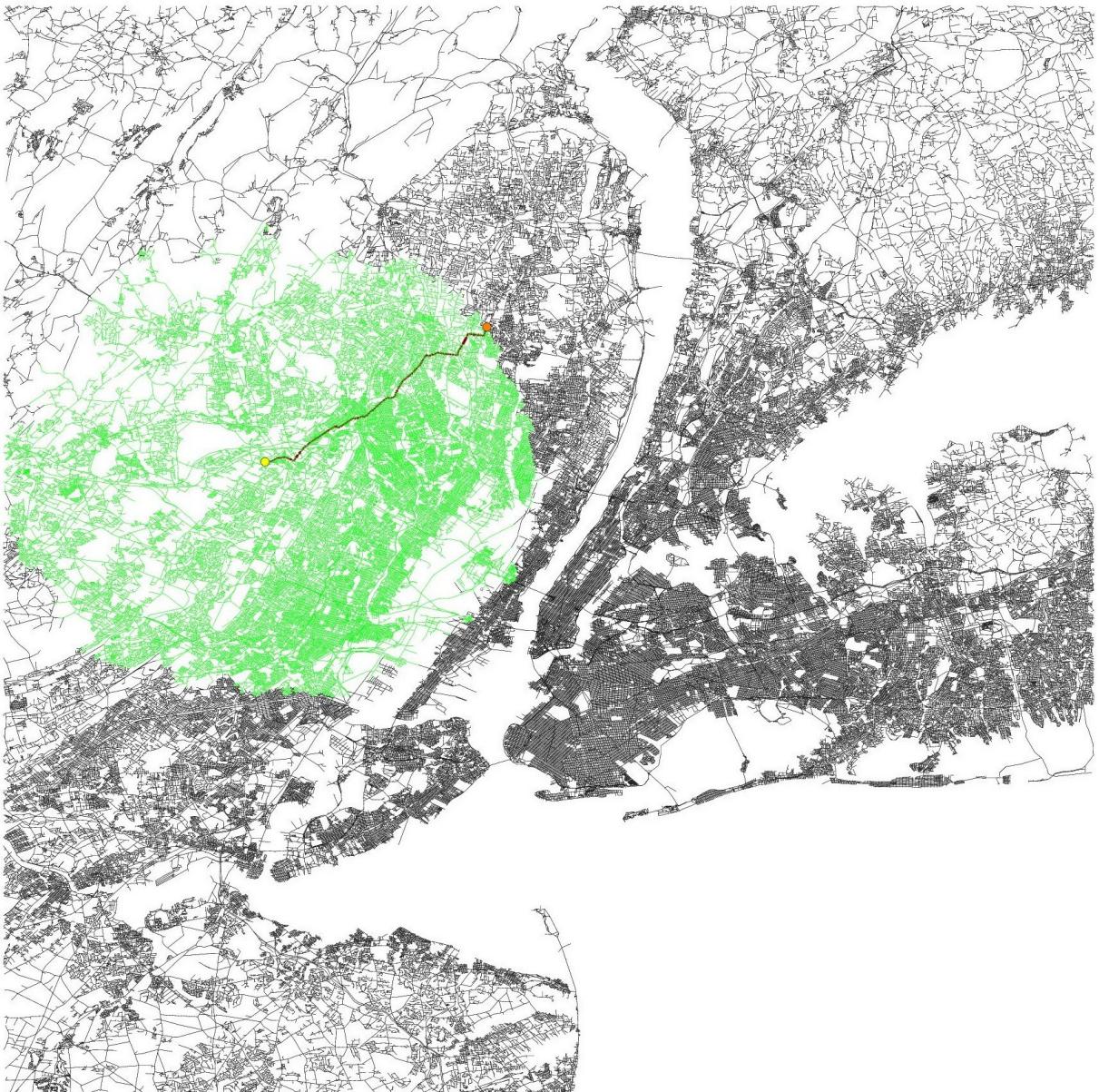
2.3.1. Az egyirányú Dijkstra algoritmus

Ezen szakdolgozatnak nem szerves része az egyirányú Dijkstra algoritmus, mégis szükséges megismerése, ugyanis minden implementált algoritmus nagy mértékben épít rá, minden elméleti alapjaikban, minden pedig a LEMON-beli megvalósításukban. A gyorsaság, helyesség és gráfcsúcs-látogatások számának vizsgálata is az eredeti Dijkstra-val való összehasonlítással nyer értelmet.

A Dijkstra alapvető célja egy adott csúcsból egy adott csúcsba, valamint egy adott csúcsból az összes csúcsba vezető legrövidebb út és távolság megtalálása. Különösen alkalmas az összes csúcsba vezető utak megtalálására, ugyanis amennyiben egy adott csúcshoz megtalálta a legrövidebb utat, akkor ismeri az összes, az adott csúcsnál a forráshoz közelebb eső csúcshoz vezető utat is. Így kellően messze lévő csúcs esetén nincs jelentős futásbeli különbség az adott csúcshoz, valamint az összes csúcshoz vezető legrövidebb út megkeresése között. Speciálisan, ha egy adott csúcsból a töle legmesszebb lévő csúcsba keresünk legrövidebb utat, akkor amint ezt megtaláltuk, ismerjük az összes csúcsba vezető legrövidebb utat.

A Dijkstra ugyanis szemléletesen „körkörösen” halad, mindenkor az éppen legköze-

lebb eső, de még nem vizsgált csúcsot vizsgálva meg. A keresett eredményt akkor kapjuk meg, amikor ennek az egyre bővülő körnek a széle eléri a célcímet, mint ahogyan az az alábbi, 2.1. ábrán is látható.

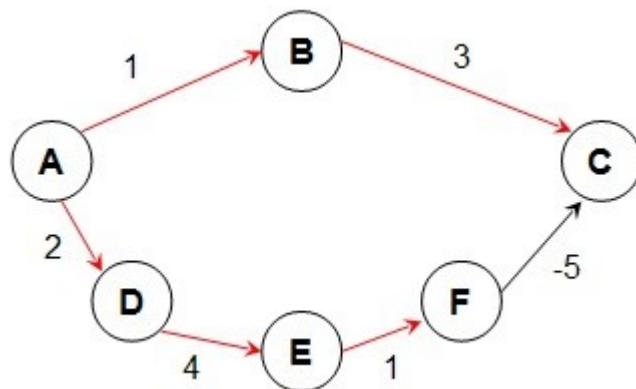


2.1. ábra: Egyirányú Dijkstra

Így általában akkor érdemes az egyirányú Dijkstrát választani, ha egyszerre sok célcíshoz vezető legrövidebb út érdekel minket. Ez a működés egyben a Dijkstra egy nagy hátránya is, ugyanis amennyiben csak egy kívánt végpontunk van, a fentiek szerint számos, számunkra érdektelen eredményünk is születik, jelentős plusz futásidőt eredmé-

nyezve. Az ezen szakdolgozatban bemutatott algoritmusok pont ezt a gyengeséget igyekeznek csökkenteni vagy kiküszöbölni.

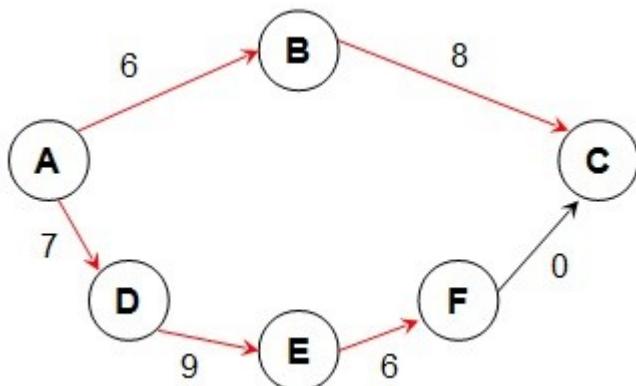
Másik jelentős korlátozás, hogy a vizsgált gráfban nem szerepelhetnek negatív élsúlyok. Ennek oka, hogy az algoritmus működése jelentős mértékben kihasználja a távolságok monoton növekvő természetét.



2.2. ábra: Dijkstra-algoritmus negatív élsúlyjal rendelkező gráfon

A 2.2. ábrán az A-ból C-be vezető legrövidebb útként a Dijkstra-algoritmus a 4 hosszú A-B-C útvonalat találja meg, pedig a legrövidebb út a 2 hosszú A-D-E-F-C útvonal lenne.

Eztazzal az egyszerű trükkkel sem lehet kijátszani, hogy minden él súlyát megnöveljük a legkisebb negatív élsúly értékével.



2.3. ábra: Egységesen növelt élsúlyok a negatív élsúly eltüntetéséért

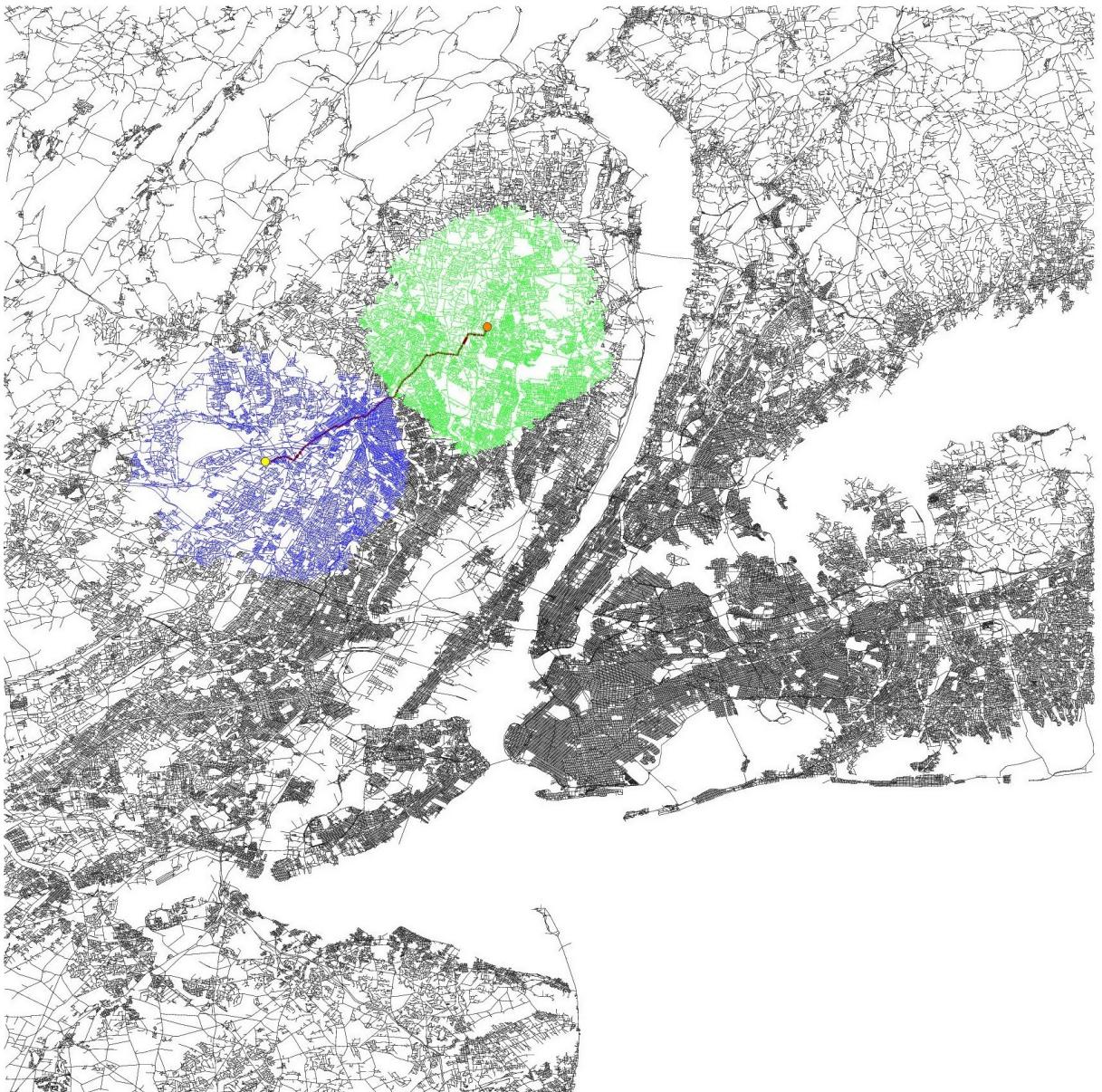
A 2.3. ábrán a Dijkstra algoritmus ismét a fenti, A-B-C útvonalat találja meg leg-rövidebb útként, és ezen esetben igaza is van (hiszen az A-B-C útvonal 14, míg az A-D-E-F-C útvonal 22 hosszú), ám nem az eredeti gráfban legrövidebb utat találja meg. Ennek oka, hogy a fenti naiv javítási próbálkozás eltérő számú élt tartalmazó utak hosszát eltérő mértékben növeli, így felborít(hat)ja az eredeti utak hosszának egymáshoz viszonyított mértékét. Negatív élek kezelésére más algoritmusok, például a Bellman-Ford algoritmus alkalmas (amennyiben a gráf nem tartalmaz negatív kör).

Ezen megszorítást a bemutatott három másik algoritmus sem oldja fel.

2.3.2. A kétirányú Dijkstra

A kétirányú Dijkstra, mint azt a neve is mutatja, lényegében két, ellentétes irányban indított Dijkstra. A kezdőcsúcstól egy hagyományos Dijkstra indul, míg a végcsúcstól egy olyan, ami az eredeti gráf fordítottján halad. A két oldalt valamilyen szabály szerint felváltva érdemes futtatni. A megoldást akkor kapjuk, amikor a két oldal összeér, azaz egy csúcsot minden oldalon megvizsgáltunk. Ekkor az eddig a pontig talált útvonalak közül a legrövidebb adja a kívánt eredményt.

Ezen algoritmus minden oldalról körkörösen egyre nagyobb sugárban vizsgálja meg a csúcsokat, így a megoldást szemléletesen a két kis kör érintkezésénél kapjuk. Mivel a két kör sugarának összege megegyezik az egyirányú Dijkstra által bezárt kör sugarával, de a vizsgált csúcsok száma (például úthálózatot leíró gráf esetén) a sugár hosszával négyzetes arányban nő, ezért a kétirányú Dijkstra esetén lényegesen kevesebb csúcsot kell bezárni azonos eredmény eléréséhez, mint az az alábbi, 2.4. ábrán is látható.



2.4. ábra: Kétirányú Dijkstra

A működés áttekintésből világosan látszik, hogy a kétirányú Dijkstra leghatásosabban olyan esetben használható, amikor egy adott csúcsból egy adott csúcsba keressük legrövidebb utat. Ekkor nagyjából fele annyi csúcsot jár be, mint egyirányú társa. Egy csúcsból az összes csúcsba való legrövidebb út keresésére a kétirányú Dijkstra nem nyújt hatékonyabb módszert egyirányú társával szemben, hiszen minden célcsúcsra külön kellene futtatni az algoritmust, ami nyilvánvalóan sok redundáns számítást eredményezne, ráadásul a kétirányú Dijkstra javítása abból adódik, hogy kevesebb csúcsot jár be, ebben az esetben viszont elkerülhetetlen az összes csúcs megvizsgálása.

Érdekesebb a helyzet, amennyiben a vizsgált gráfban nem létezik a keresett út. Ekkor ugyanis az egyirányú Dijkstra minden, a kezdőcsúcsból elérhető csúcsot megvizsgál. A kétirányú Dijkstra egyik előrefelé haladó oldala ugyanezen csúcsokat járja be, a visszafelé haladó irány azonban azokat a csúcsokat, amelyekből a célcímsúcs érhető el. Amint bármelyik oldal végzett az elérhető csúcsok megvizsgálásával, az algoritmus leáll. Ez természetesen csak akkor előny, ha az azon csúcsokat tartalmazó részgráf, melyekből a célcímsúcs elérhető, jelentősen (a pluszköltségeknél nagyobb mértékben) kisebb a forráscsúcsot tartalmazónál, egyéb esetben hátrányként jelentkezik.

Az eredmény helyességének szempontjából nincs semmi plusz előfeltétele a kétirányú Dijkstrának az egyirányú társával szemben, amennyiben nincsenek negatív élsúlyok, minden algoritmus ugyanazt az eredményt adja. Különbség csak a végrehajtási időben, a felhasznált tárhely méretében, valamint a bejárt csúcsok számában lesz.

Összefoglalva a kétirányú Dijkstra az egy csúcsból egy csúcsba vezető legrövidebb út keresésében lesz a leghatékonyabb egyirányú párral szemben, ekkor körülbelül feleannyi csúcsot jár be. Ennél a többletköltségek miatt kisebb a javulás futásidő tekintetében, a tesztek alapján nagyjából 20-25%-os.

2.3.3. Az egyirányú A*

Mind az egy-, mind a kétirányú Dijkstra a kezdőcsúcsból (illetve kétirányú esetén a végcsúcsból is) körkörösen haladva vizsgálja a csúcsokat. minden lépésben a legközelebbi még nem vizsgált csúcsot vizsgálják meg, függetlenül attól, hogy ezen csúcs az eddig vizsgált csúcsokhoz képest közelebb vagy messzebb volt a célcímcstól. Így minden a két algoritmus a vizsgált csúcsok túlnyomó többségét feleslegesen járja be, ezek az eredmény szempontjából teljesen irrelevánsak lesznek. Amennyiben valaki például egy térképet nézve keresne két pont között legrövidebb utat, akkor a forráscímcstól sokkal inkább a célcímsúcs felé eső pontokat ellenőrizné, mintsem az ellenkező irányba lévőket.

Ezen az ötleten alapul az egyirányú A* [18]. A Dijkstrával ellentétben egy heurisztikát használ, amivel a „célcímsúcs felé eső” csúcsokat vizsgálja előbb a más irányban lévőkkel szemben. Ezen heurisztikára a LEMON szemléletének megfelelően a lehető legkevesebb megszorítás lett téve, így csak annyi megkötés van, hogy minden csúcs esetén a célcímcstól való távolságára kell egy nemnegatív, konzisztens alsó becslést adnia. A konzisztencia itt azt jelenti, hogy bármely két csúcs esetén a csúcsokra vonatkozó

heurisztika értékének különbsége nem haladhatja meg a két csúcs közti él súlyát. Az algoritmus ezen, a heurisztika által adott alsó becslést használja fel futása közben, a csúcsok pillanatnyilag ismert távolságát ennek értékével megnövelte vizsgálja, és minden lépésben az ezen összeg alapján legközelebb eső csúcsot vizsgálja. Így a Dijkstra-féle körkörös haladással szemben egy a célcímsúcs felé elnyújtott ellipszis alakban járja be a csúcsokat. Természetesen minél jobb az alsó becslés, annál elnyújtottabb lesz az ellipszis, annál hatékonyabb lesz a keresés. Az alábbi, 2.5. ábrán az euklideszi távolság szolgál heuristikaként.

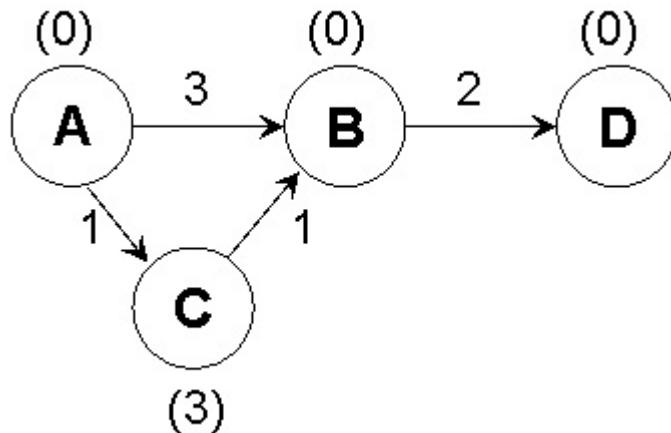


2.5. ábra: Egyirányú A*

Az algoritmus az egyirányú Dijkstrához hasonlóan addig fut, amíg meg nem vizsgálja a célcsúcst (avagy amíg minden a forrásból elérhető csúcst meg nem vizsgált. Ez akkor következhet be előbb, ha a célcíms nem érhető el a kezdőcsúcsból). Alkalmas heurisztika esetén azonban lényegesen kevesebb részét járja be a gráfnak, ezzel jelentős tárthelyet és futásidőt spórolva.

Az A* legfontosabb része a heurisztika, ennek megválasztásától jelentősen függ a hatékonyság, sőt a helyesség is. Ha a heurisztika negatív értéket is adhat, akkor a 2.3.1. részben tárgyalt módon elronthatja a keresést. Amennyiben nem ad minden esetben alsó

becslést, a keresést rossz irányba „irányíthatja”, így nem biztosan a legrövidebb úton fogja elérni a célcímet, ezzel rossz eredményt adva. Hasonlóan elronthatja az eredményt, ha a heurisztika nem konzisztens, ekkor ugyanis nem biztosított a köztes eredmények helyessége.



2.6. ábra: Példa inkonzisztens heurisztikára

A 2.6. ábrán a csúcsok felett (vagy alatt) zárójelben látható az adott csúcsra vonatkozó heurisztika értéke. Ezen példa esetében a nem konzisztens heurisztikának köszönhetően az algoritmus a B csúcs távolságát 3-nak fogja véglegesíteni, pedig látható, hogy az A-C-B útvonal csak 2 hosszú.

Így a heurisztikával szemben támasztott legfontosabb követelmény, hogy minden csúcsra egy nemnegatív, konzisztens alsó becslést adjon.

Ám nagyon sok műlik az alsó becslés minőségén is. Minél alacsonyabb becslést ad, annál inkább a távolság, és nem pedig a heurisztika értéke határozza meg az algoritmus haladásának irányát. Szélsőséges esetben, amennyiben minden csúcshoz 0-t rendel, az eredeti Dijkstra működését kapjuk vissza. Ezért az A*-ot a Dijkstra általánosításának is lehet tekinteni. Amennyiben viszont nagyon közeli az alsó becslés a valós értékhez, akkor ez válik dominánsabbá, és így jelentős gyorsulás észlelhető az algoritmus futásában. A legszélsősegebb esetben, amikor a heurisztika minden csúcsponthoz megadja annak pontos távolságát a célcímtől, az A* gyakorlatilag csak a forrás- és célcíms közti legrövidebb úton iterál végig, csak az úton lévő, és az út 1 élnyi szomszédságában lévő csúcsokat ellenőrizve. Természetesen ilyen jó becslés ritkán áll rendelkezésre, ráadásul tárolása nagy gráfok esetén a memória korlátos mérete miatt nagyon nehezen

oldható meg.

Gyakori, könnyen megadható és viszonylag jó heurisztikának számít távolságok vizsgálatánál az euklideszi távolság, vagy ennek valamilyen felhasználása. Például úthálózat esetén, ha nem legkisebb távolságot, hanem legrövidebb utazási időt keresünk, akkor alkalmas heurisztika az euklideszi távolság leosztása a gráfban szereplő utakon használható legnagyobb sebességgel. Természetesen ez rosszabb heurisztikát ad, mint például a sima euklideszi távolság legrövidebb utak keresése esetén.

Az euklideszi távolság is lehet azonban nagyon rossz heurisztika, amennyiben csak sok kerülővel érhető el a célcímsúcs. Például egy labirintus úthálózatára futtatott A* valószínűleg nagyon rossz hatékonyságú lesz, ugyanis a tényleges távolságokra ebben az esetben nagyon rossz becslést ad az euklideszi távolság.

Ennek oka, hogy az euklideszi távolság nem monoton heurisztika, azaz nem garantált, hogy amennyiben egy A csúcs közelebb van a célcímsúcsnál, akkor az A csúcsra vonatkozó heurisztika értéke is kisebb, vagy legalább nem nagyobb a B csúcsra vonatkozó heurisztika értékénél. Amennyiben rendelkezésre áll egy monoton heurisztika, az lényegesen felgyorsítja az útkeresést.

Ugyan minél jobb az alsó becslés, annál hatékonyabb a keresés, ez csak a bejárt részgráf méretére igaz. A futási idő szempontjából fontos megvizsgálni a heurisztika kiszámításának igényét is. Mivel ezen heurisztika minden egyes csúcspont ellenőrzésénél kiszámolásra kerül, ezért lényeges a műveletigénye. Tehát egy arany középutat kell találni a pontosság és a gyorsan kiszámíthatóság között. Mivel (feltéve, hogy nem nagyon pontos a heurisztika) egy adott csúcsra általában többször is kiszámításra kerül az alsó becslés, elegendő tárhely rendelkezésre állása esetén érdemes megfontolni, hogy előre kiszámoljuk és letároljuk a heurisztika értékét az összes csúcs esetén.

Általában azonban a gráf mérete ezt nem teszi lehetővé, ráadásul jó heurisztika esetén a csúcsok nagyon nagy részénél nem is kerül felhasználásra ezen érték. Így még ha elegendő tárhely áll is rendelkezésre, akkor is növelteheti ezen módszer a futási idő, amennyiben csak egy keresést hajtunk végre. Sok keresés esetén már több haszna lehet, ám csak akkor, ha a célcímsúcs mindenkor ugyanaz, és csak a kezdőcsúcsok változnak. Ekkor persze érdemes megfontolni hagyományos Dijkstra futtatását a célcímsúcsból a fordított gráfon.

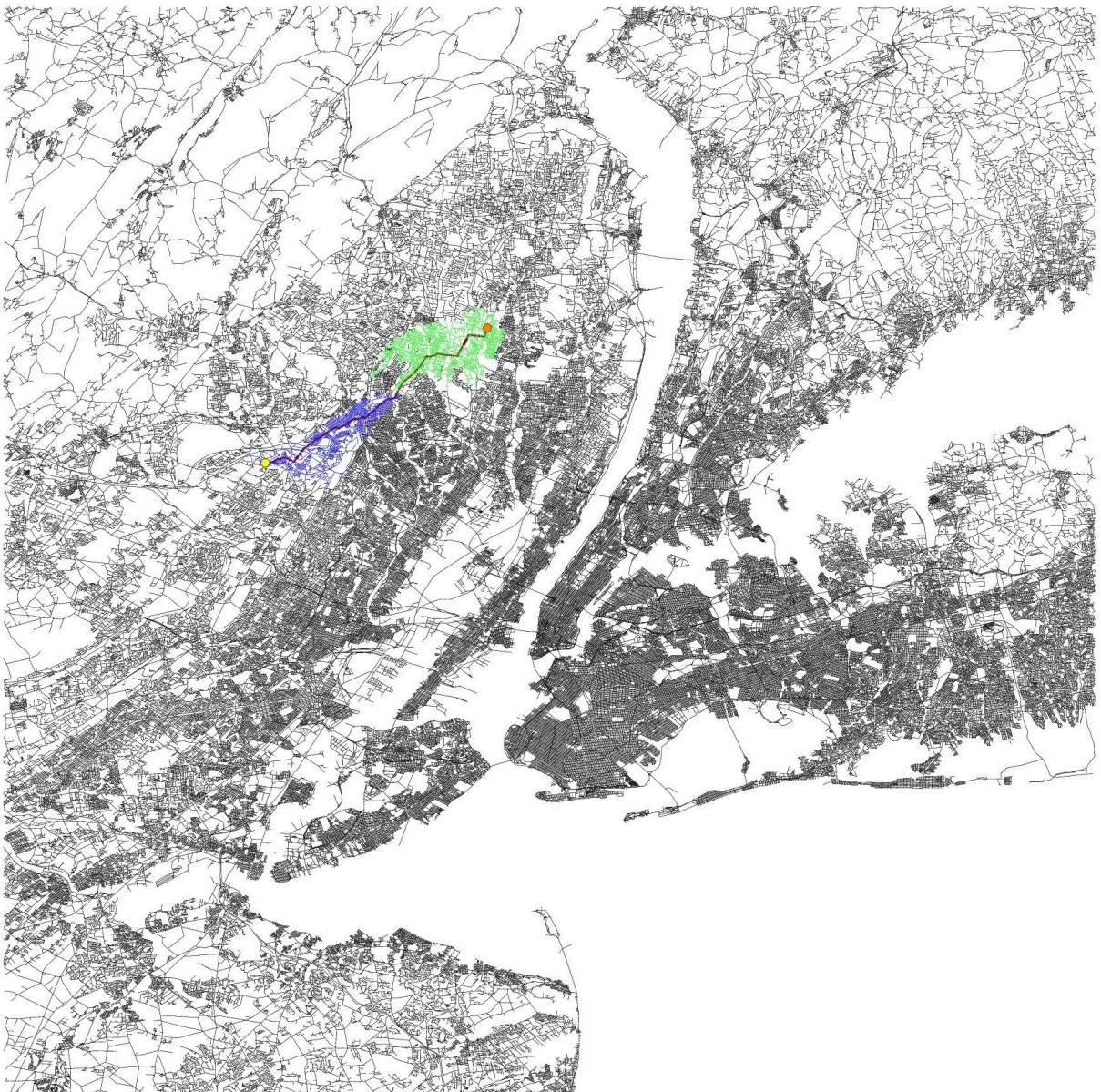
Ehhez hasonló ötlet, hogy az egy adott csúcsnál már egyszer kiszámított heurisz-

tika értéket tároljuk le, így a legközelebbi felhasználásánál nem kell újra kiszámolni. Így sokkal kevesebb értéket kell tárolni, ám nagyon jó heurisztika esetén viszonylag kevés-szer használjuk fel ugyanazt az értéket. Így ezen módszer akkor érheti meg, ha a heurisztika kiszámításának költsége meghaladja a már kiszámított értékek tárolásának költ-ségeit.

2.3.4. A kétirányú A*

A kétirányú Dijkstra után kézenfekvő ötletnek tűnik az A* kétirányúsítása is. Ennek előnyei azonban nem annyira egyértelműek, mint a Dijkstra esetében, és a megvaló-sítás is bonyolultabb. A Dijkstra körkörösen terjedő természetével szemben az A* ugyanis ellipszis alakban járja be a gráfot, így a kétirányúsításból fakadó előnyök általá-ban kisebbek annak hátrányainál.

A kérdés eldöntésében kulcsszerepe van az A* meghatározó elemének, a heurisztikának. Ugyanis amennyiben nagyon jó heurisztikát használunk, a bejárt részgráf egy keskeny ellipszis (szélsőséges esetben egy vonal), amely csak csekély mértékben széle-sedik, így a másik irányból is indítva a keresést, csak kis mértékű területcsökkenést érünk el. Azonban minél rosszabb becslést ad a használt heurisztika, annál jobban széle-sedik a bejárt terület a célcímsel felé haladva, így sokkal nagyobb a kétirányúsításból fa-kadó haszon. Szélsőséges esetben, ha a heurisztika konstans 0-t ad minden csúcsra, ak-kor közel megkapjuk a kétirányú Dijkstra egyirányú párájával szembeni előnyét. Az aláb-bi, 2.7. ábrán látott keresés a 2.5. ábrán látotthoz hasonlóan euklideszi távolságot használ heurisztikaként. Amennyiben rosszabb becslést adó heurisztikát használnánk, még nagyobb lenne a javulás aránya.



2.7. ábra: Kétirányú A*

Az A* algoritmus kétirányúvá tétele azonban több többletköltséggel jár, mint ugyanez a Dijkstra esetében. A két oldal találkozásánál megismert út ugyanis még nem a pontos eredmény, így az algoritmust még nem lehet ekkor leállítani. Gyakorlati szempontból azonban érdekes, hogy egy meglepően pontos eredményt kapunk, ami a lehetséges felhasználások egy részénél elegendő lehet, és a nagyon kis mértékű pontatlanságért cserébe kis mértékű sebesség növekedést is kapunk. Ezen módszer akkor lehet a legjövedelmezőbb, ha csak viszonylag rossz alsó becslést tudunk adni, és a felhasználás szempontjából fontosabb a gyorsaság, mint a 100%-os pontosság.

A LEMON szemléletének természetesen ez korántsem felel meg, így az ezen szakdolgozatban implementált kétirányú A* pontos eredményt ad. Ennek eléréséhez a két oldal találkozása után még némi utómunkálatra van szükség, aminek időigényét az éppen ismert legrövidebb út hosszának segítségével jelentősen redukálni lehet.

A heurisztika alkalmazási módjának egy érdekes következménye, hogy a két külön oldalról indított résznek nem szükséges azonos heurisztikát használnia. Ennek azért lehet előnye, mert míg az egyik iránynál a heurisztika az adott csúcs és a célcímsúcs közti távolságra ad alsó becslést, addig a visszafelé haladó rész heurisztikáját a forráscímsúcs és az adott csúcs közti távolságbecslésre használjuk. Így sokszor előfordulhat, hogy az egyik oldalhoz birtokunkban van egy jó heurisztika, ami a másik oldalnál nem használható fel. Természetesen amennyiben ez a „féloldalas” heurisztika kiemelkedően jó, érdemes elgondolkodni a segítségével futtatott egyirányú A* választásán.

2.3.5. Általános konklúzió

A fejezet korábbi alpontjai alapján világos, hogy nincs egyértelmű preferenciasorrend a tárgyalt négy algoritmus között, azok hatékonysága az adott körülményekhez képest változik.

Az egyik (talán legegyszerűbben előnézhető) kérdés, hogy egy adott pontból sok másikba, vagy csak néhány másikba keresünk legrövidebb utat. Az első esetben ugyanis a négy algoritmus közül az eredeti, egyirányú Dijkstrát érdemes választani.

A másik meghatározó szempont, hogy létezik-e heurisztikánk, valamint ha létezik, milyen pontos becslést ad, mennyire monoton, és milyen gyorsan lehet kiszámolni.

Amennyiben csak egy (vagy csak kevés) célcímsúcsunk van, de nem rendelkezünk semmi használható heurisztikával, akkor a kétirányú Dijkstra fogja (nagy valószínűséggel) a legjobb eredményt szolgáltatni.

Ha egy alacsony minőségű heurisztikánk van (ami nem speciálisan csak a célcímsúchoz kötődik), akkor a kétirányú A* szolgálja legjobban céljainkat, míg ha egy jó minőségű heurisztikánk van, akkor az egyirányú A* az ideális választás. Amennyiben nagyon jó minőségű heurisztikánk van, akár több célcímsúcs esetén is megérheti az A*-ot választani a Dijkstrával szemben, azonban a célcímsúcsok számának növelésével ez az előny drámaian csökken.

Fontos szempont lehet még speciális gráfoknál, hogy mennyire időigényes egy

konkrét csúcs vizsgálata. Általános esetben ez a műveletigény elhanyagolható, ám lehet olyan speciális felhasználás, ahol akár dominánssá is válhat. Ekkor a megvizsgált csúcsok száma szerint kell megfontolni a választást, így alkalmas heurisztika rendelkezésre állása esetén a kétirányú A*, ennek hiányában pedig a kétirányú Dijkstra fogja a legjobb eredményt adni.

2.4. Az algoritmusok felhasználása

Mivel az algoritmusok csupán eszközök a felhasználó kezében, fontos ismerni a felhasználásuk módját.

2.4.1. Az egyirányú Dijkstra használata

Az algoritmusok általános bemutatásánál az egyirányú Dijkstrából indultam ki, mivel nagy mértékben építenek rá. Különösen igaz ez a felhasználás esetében, sőt a LEMON-beli interfészük gyakorlatilag megegyezik vele. Ez természetesen cél is volt, a LEMON-ra általánosságban is jellemző a nagyon hasonló, avagy megegyező interfész, ami megkönnyíti a felhasználó dolgát.

A Dijkstra egy osztály a LEMON-ban, számos metódussal és adattaggal. Ezek nagy része a nagyfokú kontrollálhatóság kedvéért nyilvános, így a felhasználó által is szabadon hozzáférhető. Ezen fejezetben csak a leglényegesebbekre térek ki röviden, részletes ismertetésük a fejlesztői dokumentációban található meg.

2.4.1.1. Példányosítás

Az osztály két legfontosabb sablonparamétere a gráf és az élsúlyokat tartalmazó map típusa. A konstruktorban is ezen típusoknak megfelelő gráfot, valamint élköltség-map-et kell megadni. Például:

Dijkstra<SmartDigraph, SmartDigraph::ArcMap<double> > dijkstra(g,cap);
ahol 'g' SmartDigraph, míg 'cap' egy SmartDigraph::ArcMap<double> típusú objektum.

2.4.1.2. A run metódus

Egy Dijkstra típusú objektumot legegyszerűbben a run(s) vagy a run(s, t) metódussal használhatunk fel, ahol s és t rendre a forrás-, illetve célcímet jelölik. Ekkor

megtörténik az algoritmus során használt tárolók inicializálása, majd az algoritmus lefuttatása (az első esetben minden s-ból elérhető csúcsra, a második esetben t eléréséig).

Példa:

```
dijkstra.run(s,t);
```

2.4.1.3. A *dist* metódus

A run futtatása után a dist(n) metódussal lehetséges az n csúcs s-től való távolságának lekérése. Ez (a LEMON szellemiségből fakadó ellenőrzés hiánya miatt) csak akkor szolgál helyes eredménnyel, amennyiben az algoritmus már megvizsgálta az adott csúcsot, ellenkező esetben az eredmény nem meghatározott.

```
dijkstra.dist(n);
```

Így nagyon egyszerűen és kényelmesen, kizártlag a példányosítással, valamint a run() és a dist() metódusok használatával már meg is kapható a kívánt végeredmény.

2.4.1.4. A *processed* metódus

A dist(n)-nél említett elmaradt ellenőrzést a processed(n) hívással tudjuk elvégezni. Ezen metódus csak akkor tér vissza igazzal, ha az algoritmus már megvizsgálta n-et.

2.4.1.5. A *currentDist* metódus

Az dist(n) metódus kiterjesztettje a currentDist(n), mely amennyiben n-t már megvizsgálta az algoritmus, akkor a dist(n)-nel azonos eredményt ad, ha az algoritmus már elérte, de még nem vizsgálta meg, akkor az éppen aktuális s-től való távolságát (ami nem feltétlenül a minimális), ha pedig még el sem érte, akkor a dist-hez hasonló nem meghatározott értéket adja.

2.4.1.6. A *reached* metódus

Ahogy a dist(n)-nél elmaradt ellenőrzést a processed(n) metódussal pótolhattuk, úgy a currentDist(n)-nél elmaradt ellenőrzés biztosítására használható a reached(n) függvény, mely csak akkor tér vissza hamissal, ha az algoritmus még nem érte el n-t, azaz akkor lesz igaz, ha már elérte, vagy már meg is vizsgálta.

2.4.1.6. A path, predNode és predArc metódusok

Nevéhez híven, a path(n) metódus az s forráscúcsból az n célcúcsba vezető legrövidebb utat adja vissza. Természetesen itt is szükséges, hogy a reached(n) igaz legyen. A predNode(n) valamint predArc(n) metódusok rendre az s-ből n-hez vezető (eddig megtalált) úton az n-et megelőző csúcsot valamint élt adják vissza.

2.4.1.7. Egyéb metódusok

A futtatás finomhangolása érdekében a run(s), run(s, t) metódusok által végrehajtott műveletek egyenként is indíthatók. Erre szolgálnak az init(), az addSource(s) vagy addSource(s, dist), és a start() valamint start(t) metódusok.

A start() és start(t) működésének tovább bontására használhatóak a processNextNode(), a nextNode(), emptyQueue() valamint queueSize() metódusok. Ezek közül a processNextNode() valósítja meg a Dijkstra fő logikáját, a fejlesztői dokumentációban megtalálható működésének részletes leírása.

2.4.1.8. Belső tárolók lekérdezése, módosítása

Az összes fontosabb belső tároló szabadon lekérdezhető, megadható és inicializálható. Természetesen ezen belső tárolók futás közbeni, azaz az algoritmus befejeződése előtti módosítása befolyásolhatja és helytelenné teheti az algoritmus eredményét.

2.4.1.9. Wizardok

A LEMON-ban osztályként vannak implementálva az algoritmusok, ám ezek könnyebb, tömörebb, valamint sokak által jobban megszokott függvényeszerű hívásokkal is végrehajthatók. Erre szolgálnak az úgynevezett Wizard osztályok [13].

Ezen Wizardok wrapper osztályok, melyek lefordítják a függvényeszerű hívásokat az adott osztály szerkezetére. Például a

Dijkstra<ListDigraph> dijkstra(g, length);

dijkstra.run(s, t);

kódrész könnyedén helyettesíthető a

dijkstra(g, length).run(s, t);

sorral.

Ezen megoldás szemmel láthatóan tömörebb kódot eredményez, ám sokkal kevesebb funkciót biztosít, az osztály szemléletű megközelítésnél tapasztalt teljes körű kontrollálhatóságnak csak a töredéke érhető el.

2.4.1.10. Biztonság

A Dijkstra felhasználásának leírásánál nyilvánvalóan jelentkezik a LEMON egészére jellemző jelenség, az ellenőrzések hiánya. Például inicializáció nélkül is meghívhatók az algoritmust vezérlő függvények, de súlyos hibát okozhatnak, vagy például a processNextNode() akkor is meghívható, amikor már nincs következő csúcs, így szegmentálási hibát okozva. Ezen ellenőrzések szándékosa hiányoznak, hiszen a LEMON a felhasználóbiztos kód helyett a hatékony kódot helyezi előtérbe. Ezért szükséges a 2.2.2-ben már említett hozzáértés egy-egy kód használatánál.

2.4.2. A kétirányú Dijkstra használata

A kétirányú Dijkstra a befejező logikától eltekintve gyakorlatilag két külön Dijkstra futtatása, ez a struktúrájában, és metódusaiban is megjelenik.

2.4.2.1. Példányosítás, indítás

A kétirányú Dijkstra példányosítása semmiben nem tér el egyirányú társától, pontosan ugyanúgy használható. Ez megkönnyíti az egyirányút használó kódok triviális módosítását kétirányú Dijkstra használatára.

Az algoritmus futásának indítása szintén nem különbözik név és paraméterezés tekintetében. Azonban fontos megjegyezni, hogy kétirányú Dijkstrát csak a célcímsúcs ismeretében van értelme futtatni, egy csúcsból összes csúcsba való útkeresés esetén nem. Ennek megfelelően a run(s) metódus teljes mértékben az eredeti Dijkstra logikája szerint fog futni, ettől tárhely- és időfelhasználás tekintetében sem fog különbözni.

A run(s, t) futtatása esetén azonban a kétirányú Dijkstra logikája fog lefutni.

2.4.2.2. Az addTarget metódus

Külön említést érdemel az addTarget(t, dst) metódus, ami az addSource(s, dst) metódus kétirányú párja (amellett, nem ahelyett). Ennek működése értelemszerű, cél-

csúcsot lehet az algoritmushoz adni. A külön figyelemre azért jogosult, ugyanis az algoritmus az alapján dönti el, hogy két- vagy egyirányú Dijkstrát futtasson-e, hogy legalább egy célcímcsoport hozzá lett-e adva az algoritmushoz. `run(s, t)` futtatása esetén ez automatikusan megtörténik, ám „kézi vezérlés” esetén így ezen osztály használata mellett is elérhető az egyirányú Dijkstra futtatása. Ennek általánosíthatósági céljai vannak, a nagy mértékű egyezés, és ezen megoldás miatt lehetővé válhat a két Dijkstra egy osztállyal való helyettesítése.

2.3.2.3. A `processNextNodeBi` metódus

Ezen metódus valósítja meg a kétirányú Dijkstra logikáját. A fejlesztői dokumentáció részletesen foglalkozik működésével.

2.4.2.4. Az eredeti metódusok kétirányú változatai

A célcímcsoport felől induló Dijkstra miatt a legtöbb metódusnak kétirányú változata is lett. Ezen metódusok a `revProcessed(n)`, `revReached()`, `currentRevDist(n)`, `emptyRevQueue()`, `revQueueSize()`, `nextRevNode()`, `revPath(t)`, `revDist(n)`, `revPredArc(n)`, `revPredNode(n)`, melyek minden az egyirányú társaik azonos változatai a visszafelé haladó Dijkstrára vonatkozóan. Hasonlóan az összes, korábban lekérdezhető, beállítható és inicializálható belső tárolók új változatainak ezen funkcióik is elérhetők. Természetesen a célcímcsoport vonatkozó `path()` függvény a forráscímcsoporttól a célcímcsoportig vezető, a két oldal eredményét egyesítő utat ad vissza, ez garantáltan igaz az ezen az úton szereplő összes másik csúcsra is.

2.4.3. Az egyirányú A* használata

Az egyirányú A* a Dijkstrához képest egy nagy újítást tartalmaz: a heurisztikát. Ennek megfelelően az interfész is különbözik a Dijkstrától.

2.4.3.1. Példányosítás

Az A* a Dijkstrához képest egy plusz sablonparaméterrel rendelkezik, ami a heurisztikáként használt funktor, melynek „()” operátora két csúcs megadása esetén visszatérít az első csúcs második csíccsúpcsoporttól való távolságára vonatkozó heurisztikus becslés értékét.

tékét. Ezen visszatérési érték típusának kompatibilisnek kell lennie az élsúlyok típusával, pontosabban az azokhoz megadott „kisebb” és „összeadás” operátorokkal. Egyszerű példa euklideszi távolságot adó heurisztikára „double” élsúlytípus esetén (ahol az x és az y változók a csúcsok megfelelő koordinátáját tárolják):

```
template<typename GR>
class EuclideanEstimateCalculator {
    typedef GR Graph;
    typedef typename Graph::Node Node;
    typedef typename Graph::NodeMap<double> NodeMap;

    NodeMap *x,*y;
public:
EuclideanEstimateCalculator(const NodeMap &xv, const NodeMap &yv) :
    x(&xv), y(&yv) {}

double operator()(const Node& u, const Node& v) const {
    double dx = (*x)[u] - (*x)[v];
    double dy = (*y)[u] - (*y)[v];
    return sqrt(dx * dx + dy * dy);
}
};
```

Mivel az eredeti három sablonparaméter közül a harmadik továbbra is csak ritkán kerül megadásra, míg az A* lelkét adó heurisztika gyakran, ezért az A* esetén ezen paraméter a harmadik, és az előbbi az utolsó. Természetesen a konstruktörben átadható a heurisztikául szolgáló funktor egy példánya. Ennek elhagyása esetén az erre mutató pointer null-ra lesz inicializálva. Példa:

```
Astar<ListDigraph, ListDigraph::ArcMap<double>,
EuclideanEstimateCalculator > astar(g, cap, ec);
```

ahol az EuclideanEstimateCalculator egy megfelelő heurisztika típusa.

2.4.3.2. Az algoritmus indítása

Mivel az A* lényege, hogy a célcímtól való távolságot egy heurisztika segítségé-

vel megbecsüljük, ezért az algoritmus nem értelmezett ismert célcímes nélkül. Ezért amennyiben a run(s) metódussal indítjuk, az eredeti Dijkstra logikája fog lefutni. Ha a run(s, t) metódussal indítjuk, akkor attól függően fogja az algoritmus kiválasztani, hogy az A* vagy a Dijkstra logikáját futtassa-e, hogy lett-e beállítva bármilyen heurisztika. Ezen ellenőrzés nagy előnyivel szolgál az általánosság szempontjából, így ugyanis (például egy heurisztika használhatatlan voltának kiderülése esetén) könnyedén, jelentős kódátírás nélkül át lehet váltani az eredeti Dijkstrára.

2.4.3.2. A *setEstimateCalculator* metódus

A heurisztika példányának megadása a konstruktoron kívül is lehetséges. Erre a setEstimateCalculator(ec) metódus szolgál. Ennek segítségével korábban már beállított heurisztikát is le lehet cserélni, természetesen a megfelelő típusúra. Példa:

```
astar.setEstimateCalculator(ec);
```

A heurisztika példány beállításának nagy szerepe van, amennyiben ugyanis null értéke van, az algoritmus az eredeti Dijkstrát fogja futtatni. Ez a kétirányú Dijkstrához hasonlóan kompatibilitási és általánosítási célokat szolgál.

2.4.3.3. A módosított *addSource* metódus

Amennyiben ismert célcímsuccsal indítjuk az algoritmust, és heurisztikát is megadtunk, az eredeti helyett egy módosított addSource(s, t, dst) metódus fut le. Ebben az A* logikájának megfelelő inicializáció hajtódik végre a forráscímsra vonatkozóan.

2.4.3.4. A *processNextNode* metódus módosított változata

A processNextNode() egy módosított változata, a processNextNode(t) gondoskodik az A* logikájának megvalósításáról. Ennek működésével a fejlesztői dokumentáció foglalkozik részletesen.

2.4.3.5. Belső tárolók

A Dijkstrához képest az A* csak egy új belső tárolóval rendelkezik, így tárigénye közel azonos vele. Ezen új belső tároló természetesen a többihez hasonlóan bármikor szabadon lekérdezhető, módosítható, inicializálható.

2.4.4. A kétirányú A* használata

A kétirányú A* a befejező logikától eltekintve gyakorlatilag két külön A*. Ennek megfelelően az egyirányú A* lekérőmetódus- és belsőtárolószáma is majdnem megduplázódik. Így a négy bemutatott algoritmus közül ez tartalmazza a legtöbb ilyet.

2.4.4.1. Példányosítás, futtatás

A kétirányú A* interfésze teljes mértékben megegyezik egyirányú társáéval, mind sablonparaméterek, mind konstruktor, mind a run(s) és run(s, t) metódusok tekintetében. A setEstimateCalculator metódus is a 2.4.3.2-ben szereplővel azonos módon használható. Az egyedüli különbség a start(t) metódusnál észlelhető, ezt ugyanis felváltotta a start(s, t) metódus. Ennek oka, hogy a visszafelé haladó A*-nak szüksége van a forráscsúcsra a heurisztika számításához. Amennyiben a run(s)-el indítjuk, avagy nem adunk meg heurisztikát, az algoritmus az egyirányú Dijkstra logikáját fogja futtatni. A másik két, ezen szakdolgozat keretei közt tárgyalt algoritmus felé nem biztosít hatékony kompatibilitást.

2.4.4.2. A processNextNodeBi metódus

A processNextNodeBi(s, t) metódus a kétirányú Dijkstrában jelenlévő azonos nevű változatához képest a forráscsúcsot is megkapja paraméterül. Ennek ismerete ellenégedhetetlen feltétele a visszafelé haladó heurisztika számításához. Ezen metódus valósítja meg a kétirányú A* fő logikáját, ezt a fejlesztői dokumentáció írja le részletesen.

2.4.4.3. Az eredeti metódusok kétirányú változatai

Hasonlóan a kétirányú Dijkstrához, a kétirányú A*-ban is elérhetőek a visszafelé irányú belső adattárolókra, valamint a keresés aktuális állapotára vonatkozó metódusok, lekérdezések, beállítások, inicializálások. Ezek funkcionálisan megegyeznek az előre irányú társaikkal.

3. Fejlesztői dokumentáció

3.1. Terminológia

A pontosabb, és egyszerűbb kifejezés érdekében rögzítünk néhány elnevezést, a továbbiakban ezek így értendők

- **kezdő- vagy forráscsúcs:** Azon csúcs, melyből a keresést indítjuk. Speciális esetben több ilyen csúcs is lehet, az algoritmusok szempontjából ezek között nincs különbség, minden csúcshoz a(z egyik) legközelebbi ilyen tekintendő forráscsúcsként. Az egyszerűség kedvéért általában feltételezni fogjuk, hogy csak egy ilyen csúcs van.
- **célcsúcs:** Azon csúcs, amelybe a forráscsúcsból vezető legrövidebb utat keresünk.
- **kupac:** Azon adatszerkezet, melyet az algoritmus fő tárolóként használ, ebben tárolja azokat a csúcsokat, melyek eddig ismert legkisebb távolsága már nem végtelen, de még nem is biztosan a tényleges távolság. Az A* algoritmusok esetén az adott csúcsra vonatkozó heurisztika értékével növelt értékek tárolódnak benne. Ezen adatszerkezet nem feltétlenül kupac, bármilyen más adatszerkezet is lehet, amely ugyanazokat a műveleteket nyújtja, pontosabban megfelel a LEMON kupac interfész koncepciójának.
- **visszakupac:** A kétirányú Dijkstra és a kétirányú A* algoritmusokban a visszafelé haladó rész kupaca.
- **nem elérő csúcs:** Az algoritmus még semmilyen formában nem ellenőrizte ezt a csúcsot, nem került be a kupacba sem. A forráscsúcstól való ismert távolsága végtelennek tekintendő.
- **elérő csúcs:** A kupacba már bekerült csúcs. Ennek az ismert távolsága már nem végtelen, de még nem is biztosan a tényleges távolság.
- **vizsgált csúcs:** A kupacból már kikerült csúcs. Ennek ismert távolsága már biztosan a forráscsúcstól való tényleges távolság.
- **konzisztenz heurisztika:** Egy heurisztikát [1] meghatározása alapján konzisztensek nevezünk, ha minden u, v csúcsra teljesül, hogy az u-ra és v-re kiszá-

molt heurisztika értékének különbsége nem haladja meg az u és v közti legrövidebb él hosszát, azaz:

$$\forall(u,v): \text{heuristic}(u) - \text{heuristic}(v) \leq \text{length}(u,v),$$

ahol **heuristic** a heurisztikát számoló függvény, **length** pedig az u-t v-vel összekötő él hosszát adja meg. Természetesen amennyiben ez teljesül minden szomszédos csúcsra, akkor minden nem szomszédos csúcsra is, ez esetben a köztük lévő legrövidebb út hossza szerepel a köztük lévő él hossza helyett.

- **ciklusmag:** A továbbiakban ciklusmag alatt az algoritmus egy lépését értem, azaz csúcs megvizsgálását, és az ehhez kapcsolódó műveletek elvégzését. A forráskód alapján nézve a processNextNode (vagy processNextNodeBi) függvény jelenti a ciklusmagot.

3.2. Az algoritmusok részletes vizsgálata

Mivel a szakdolgozat témája a kétirányú Dijkstra, az egyirányú A* és a kétirányú A* algoritmusok implementálása volt, ezért fontos részletesen is megvizsgálni ezen algoritmusok működését. Mivel ezen algoritmusok mind erősen építenek az egyirányú Dijkstrára, ezért ennek leírása is szükséges.

3.2.1. Az egyirányú Dijkstra algoritmus

Ez az alap algoritmus, mind a három másik algoritmus ezen alapszik, ezt fejleszti tovább. A LEMON legfrissebb 1.2.1-es verziója is csak ezt tartalmazza.

3.2.1.1. Az algoritmus menete

Inicializáláskor a forráscsúcs távolsága 0, míg minden más csúcs távolsága végtelen, és csak a forráscsúcs szerepel a kupacban. Ezután minden lépésben kivesszük a kupacból azt a csúcsot, amelynek a legkisebb a kezdőcsúcstól való eddig ismert távolsága (kezdetben ez maga a forráscsúcs lesz). Az egyszerűség kedvéért nevezzük ezen csúcsot u-nak.

Ekkor válik u távolsága véglegessé, biztosan ez a legkisebb távolsága a forráscsúcstól, ekkor lesz vizsgált csúcs. Ezután u összes kimenő élén végig kell iterálni. Az

ezben egy ilyen szomszédot v-nek nevezünk:

- Amennyiben v már vizsgált csúcs, semmi teendő nincs.
- Ha v még nem elért csúcs, be kell helyezni a kupacba, pillanatnyi távolságként u távolságának, valamint az $u \rightarrow v$ él hosszának összegét kell beállítani.
- Ha v már elért, de még nem vizsgált csúcs, akkor meg kell vizsgálni az eddig ismert legrövidebb távolságát. Amennyiben az előző pontban számolt összeg ezen korábbi távolságnál kisebb, le kell rá cserélni.

Ezen folyamat addig folytatódik, amíg a kupac ki nem ürül. Amennyiben nem minden elérhető csúcs távolságának meghatározása volt a cél, az algoritmus leállítható, amint a célcímsúcs (több célcímsúcs esetén az összes) vizsgálttá válik. Ha a kupac kiürült, minden olyan csúcs, ami nem vált sem elérhetővé, sem vizsgálttá, nem elérhető a kezdőcsúcsból.

3.2.1.2. Az algoritmus pszeudokódja

Az alábbi pszeudokód teljes egészében megtalálható [5] -ben, itteni elhelyezése a későbbi hivatkozások egyszerűsítése miatt szükséges.

1. function Dijkstra(Graph, source):
2. for each vertex v in Graph:
3. dist[v] := infinity;
4. previous[v] := undefined;
5. end for;
6. dist[source] := 0;
7. Q := the set of all nodes in Graph
8. while Q is not empty:
9. u:= vertex in Q with smallest dist[];
10. if dist[u] = infinity:
11. break;
12. end if;
13. remove u from Q;
14. for each neighbor v of u:
15. alt := dist[u] + length(u,v);

```

16.           if alt < dist[v]:
17.               dist[v] := alt;
18.               previous[v] := u;
19.           end if;
20.       end for;
21.   end while;
22.   return dist[];
23. end Dijkstra.

```

3.2.1.3. Az algoritmus helyessége

Az algoritmus csak akkor ad helyes eredményt, ha minden élsúly nemnegatív. Ennek teljesülése esetén helyessége az alábbiakon nyugszik:

1. Egy adott csúcs pillanatnyi legkisebb távolsága monoton csökken.
2. A kupacban lévő legkisebb távolság monoton nő. (ciklusmag lefutásonként tekintve)
3. minden, a forráscsúcsból elérhető csúcs feldolgozásra kerül.
4. Egy megvizsgált csúcs pillanatnyi távolsága megegyezik a tényleges távolságával.

A 4. pont helyessége ekvivalens az algoritmus helyességével, így ha teljesül, akkor az algoritmus helyes. Ehhez felhasználjuk az első három pontot, így kezdjük ezek helyességének belátásával.

Az 1. pont triviálisan teljesül, hiszen mint az a pszeudokód 16. sorában is látható, a pillanatnyi legkisebb távolságot csak akkor változtatjuk, ha találtunk annál rövidebbet.

Egy ciklusmag lefutása folyamán kivesszük a legkisebb távolsággal rendelkező csúcsot a kupacból és módosítunk néhány (akár 0) bent lévő értéket. A módosított értékek minden az épp kivett csúcs távolságának és egy nemnegatív élhossznak az összegeként állnak elő, így 2. is minden teljesül (ezen pont miatt előfeltétele a Dijkstrának, hogy minden él hossza nemnegatív legyen).

Az algoritmus működéséből egyenesen következik 3., hiszen a forráscsúcs feldolgozásra kerül, és minden feldolgozásra került csúcs minden szomszédja feldolgozásra kerül.

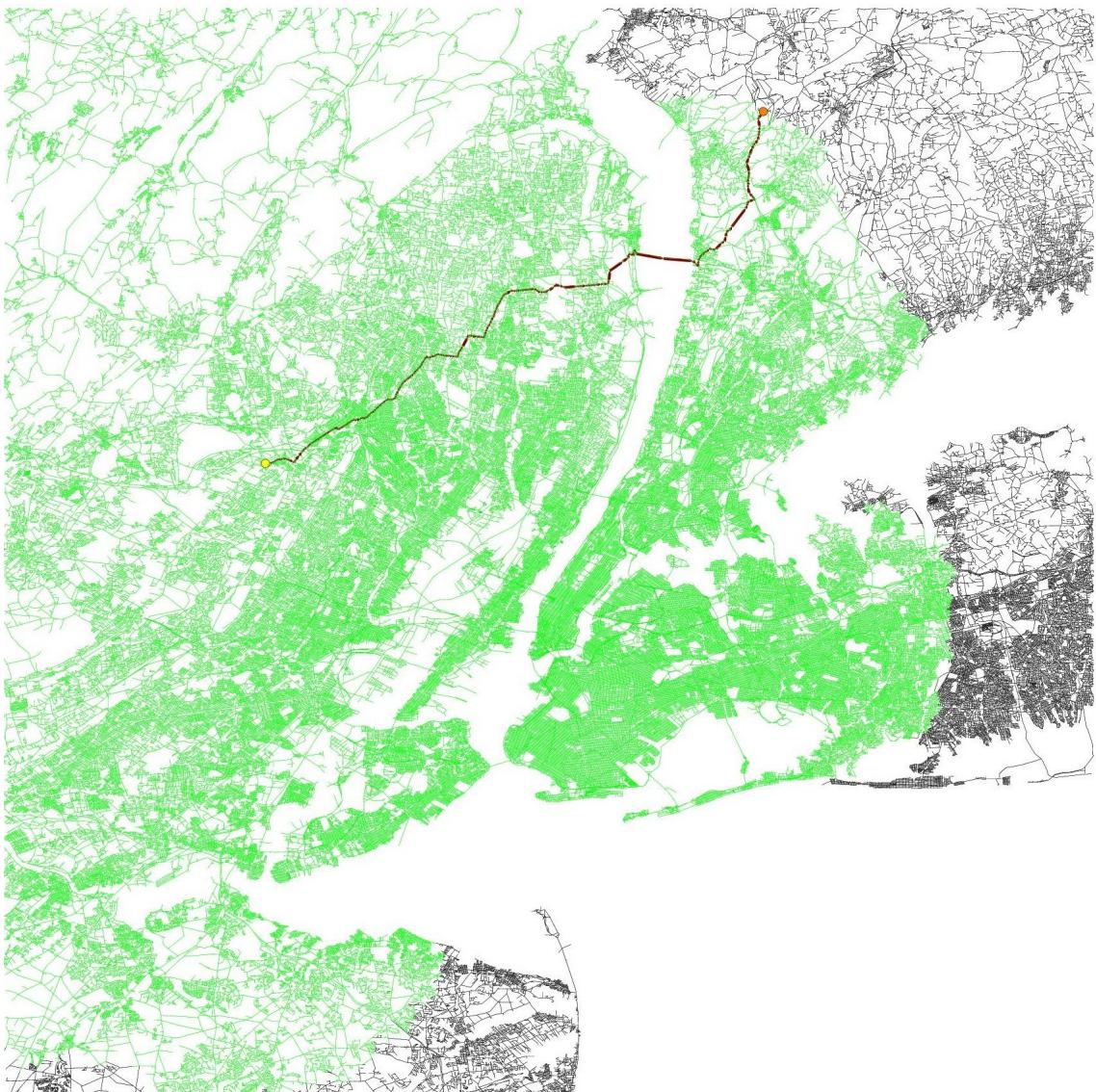
Mivel minden ciklusmag lefutás során az újonnan kiszámított távolságértékek na-

gyobbak vagy egyenlőek a kupacban szereplő távolságok minimumánál, ezért ha egy csúcs pillanatnyilag ismert távolsága megegyezik ezen minimummal, akkor ez a minimális távolsága a kezdőcsúcstól, hiszen 2. miatt a későbbiekben ez a minimum sosem lesz kisebb. Mivel 3. és 2. miatt minden elérhető csúcs távolságát megtaláltuk, ezért 4., és így az algoritmus helyességét is bizonyítottuk.

Fontos megjegyezni, hogy 2-ből, 3-ból és 4-ből együtt következik, hogy minden megvizsgált csúcsra igaz, hogy minden, a forráscsúcshoz nála közelebb lévő csúcs már megvizsgált.

3.2.1.4. Az algoritmus előnyei, hátrányai

Mint az a 2.3.1. pontban is szerepel, az egyirányú Dijkstrának a legnagyobb előnye és egyben legnagyobb hátránya is, hogy egy adott csúcshoz való legrövidebb út megtalálása esetén már minden, az adott csúcsnál a kezdőcsúcshoz közelebb eső csúcs legrövidebb távolságát ismerjük, azaz „körkörösen” halad. Mint az a 3.1. ábrán is látszik, ez különösen nagy gráfokban, távoli pontok esetén jelent nagy hátrányt, feltéve persze, hogy csak kevés csúcs távolságának kiszámítása volt a cél. Ellenkező esetben ez jelenti az előnyt.



3.1. ábra: Egyirányú Dijkstra két távoli pont esetén

3.2.2. A kétirányú Dijkstra algoritmus

A kétirányú Dijkstra gyakorlatilag két, egymással ellentétes irányba haladó egyirányú Dijkstrá. Ennek megvalósítási nehézségei inkább gyakorlati jellegűek, elméleti szempontból a megállási feltétel, és a legrövidebb út kiválasztása lényeges.

3.2.2.1. Az algoritmus menete

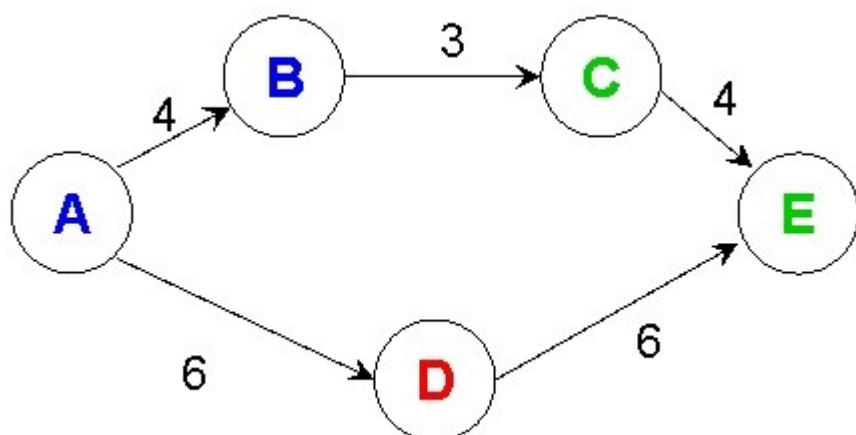
Inicializáláskor minden oldalt inicializálni kell. Az algoritmus folyamán valamelyen (többnyire a használt adatszerkezet tulajdonságaitól függő) logika szerint felváltva léptetjük a két oldalt. Ezen léptetés minden az adott oldalban az egyirányú Dijkstra szerinti logika szerint történik. A visszafelé irányú rész természetesen a fordított gráfban fut, hiszen irányított gráf esetén csak így lesz a végeredmény helyes.

Amint a megállási feltétel teljesül, a közben jegyzett legrövidebb utak közül a legkisebbet kiválasztva megkapjuk a végeredményt.

3.2.2.2. A megállási feltétel, és a legrövidebb út kiválasztása

Szemléletesen az a helyes megállási pont, amikor a két oldal „összeér”. Azonban lehetőséges, hogy nem elég, ha van egy csúcs, amit minden oldal elérte, hiszen ennek távolsága még egyik irányból sem garantáltan a legrövidebb, így ezek összege, azaz a kezdő- és célcíms távolsága sem.

Vegyük észre azonban, hogy ha egy csúcs minden oldalról megvizsgálttá válik, akkor leállíthatjuk a keresést, hiszen ezen csúcson keresztül biztosan van út a forrás- és célcíms között, és mivel ezen csúcs minden oldalon a kupac minimuma volt az adott pillanatban, a 3.2.1.3. miatt ennél már csak nagyobb távolságot találhatunk. Azonban koránt sem biztos, hogy ezen csúcson keresztül vezet a legrövidebb út, csak azt tudjuk, hogy már megtaláltuk a legrövidebb utat. Ezt jól szemlélteti az alábbi, 3.2. ábra.



3.2. ábra: Legrövidebb út a kétirányú Dijkstrában

Itt ugyanis (feltéve, hogy alternálva futtattuk a két oldalt) a D csúcs válik először minden két oldal által vizsgálttá, ám a legrövidebb út mégsem rajta keresztül, hanem az A-B-C-E útvonalon halad.

Ennek az az oka, hogy ugyan a C az A-tól, és a B az E-től is 7 távolságra van az A-D és E-D 6-6 távolságával szemben, azonban az A-E távolságot a fenti útvonalon nem ezen $7 + 7 = 14$ adja, hiszen így a B-C él kétszer lett számolva.

Ezen jelenség miatt az algoritmus futása során fel kell jegyezni a lehetséges utak közül (legalább) a legkisebbet. Az ilyen lehetséges utakat úgy tudjukelfedezni, hogy minden egyes alkalommal, amikor egy csúcs pillanatnyi távolságát bármelyik oldalon módosítjuk, leellenőrizzük, hogy a másik oldal megvizsgálta-e már ezt a csúcsot. Amennyiben igen, egy lehetséges utat találtunk. Ennek (össz)távolsága könnyen összehasonlítható az eddig talált legrövidebb út távolságával, így minden számon tudjuk tartani a pillanatnyilag talált legrövidebb utat. A megállási feltétel teljesülése esetén ezen út lesz a tényleges legrövidebb út a forrás- és a célcímszöveg között.

Amennyiben a másik oldal már elérte, de még nem vizsgálta meg az adott csúcsot, akkor is találtunk egy utat, ám ennek hosszát nem kell ellenőrizni, mert a másik oldalbeli értéke még csökkenhet. Amikor a másik oldal véglegesíteni fogja ezen (vagy az addigra már csökkentett) értéket, akkor ennek az útnak a hosszát is ellenőrizni fogja az algoritmus, így amennyiben ez lesz a legrövidebb út, számon fogjuk tartani.

3.2.2.3. Az algoritmus helyessége

A két oldal külön-külön a 3.2.1.3. miatt helyesen működik. A 3.2.2.2-ben leírtak alapján pedig az ott leírt megállási feltétel teljesülése esetén biztosan ismerjük a legrövidebb utat. És mivel a forrás- és célcímszöveg között csak úgy lehet út, ha ebben van legalább egy olyan él, melynek egyik végpontját (legalább) az egyik, másik végpontját (legalább) a másik algoritmus megvizsgálta, ezeket pedig számon tartjuk, ezért ezek minimuma biztosan a tényleges legrövidebb utat adja. Így ezen algoritmus is helyesen működik.

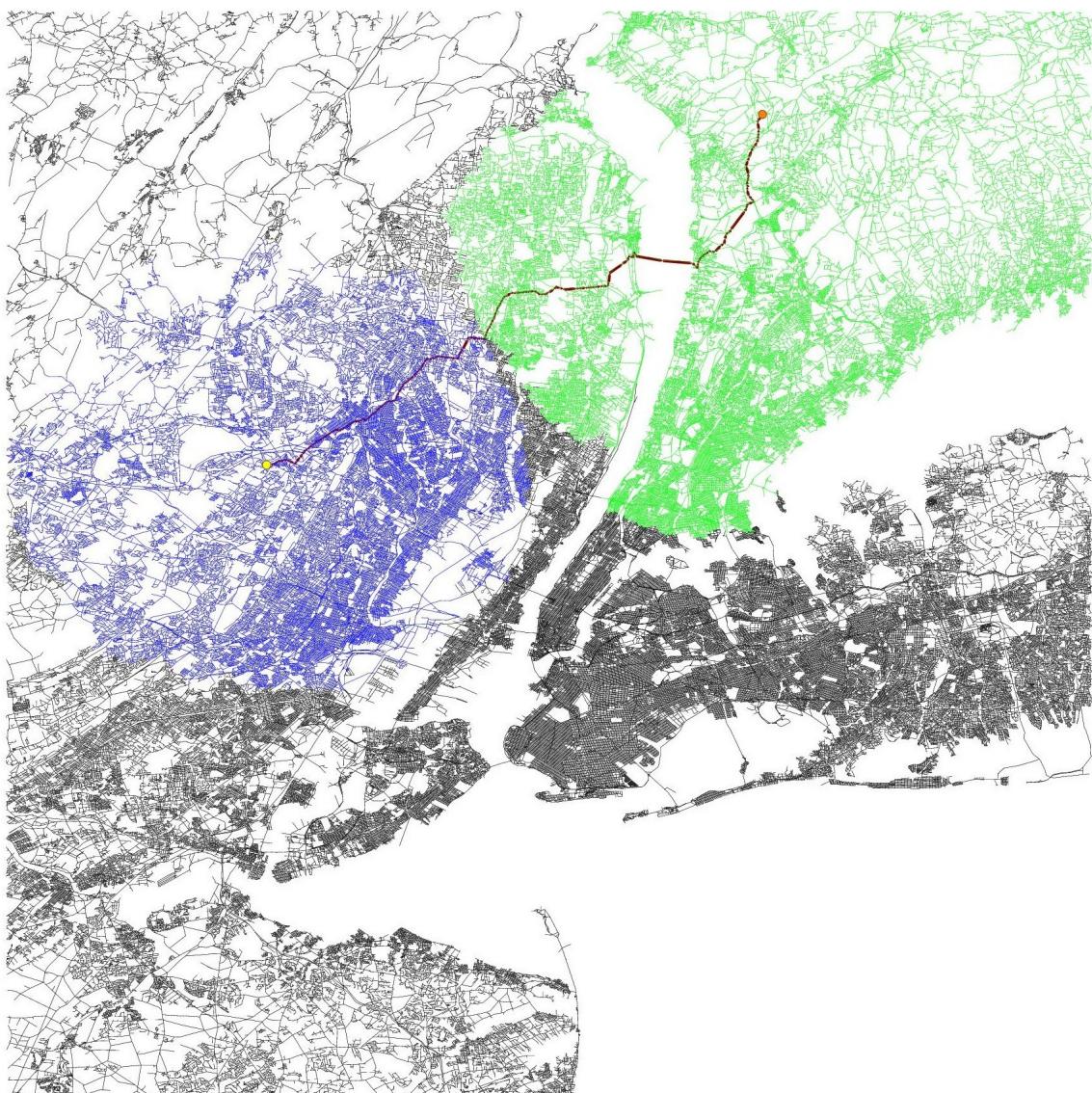
3.2.2.4. Az algoritmus előnyei, hátrányai

Mint az a 3.2.1.4-ből is kiderül, a Dijkstra „körkörösen” halad. A 3.2.2.1. alapján a kétirányú Dijkstra két pontból, a kezdő- és a célcímszövegből halad, szintén körkörösen. A 3.2.2.2. alapján az algoritmus akkor ér véget, amikor ez a két kör „összeér”, amikor van

olyan csúcs, amit már minden oldal megvizsgált. Ezen két kör metszete azon csúcsok összessége, melyeket minden oldal elérte, vagy megvizsgált.

Minden elérte csúcs egy már megvizsgált csúcstól csak egy élnyi távolságra van, azaz az elérte, de még nem megvizsgált csúcsok halmaza ezen körök legszélső sávja. Mivel a metszetben a megállási feltétel miatt csak egy olyan csúcs lehet, amelyet minden oldal megvizsgált, így a metszet csak elhanyagolhatóan kevés csúcsot tartalmazhat a körök területéhez képest (eltekintve a szélsőségesen kis sugarú köröktől, azaz ha egyik vagy minden oldal csak nagyon kevés csúcsot vizsgált meg).

Emiatt, és a kör területének a sugárhoz viszonyított négyzetes növekedése miatt a kétirányú Dijkstra két része összesen körülbelül feleannyi csúcsot érint, mint egyirányú társa. Ezt szemlélteti az alábbi, 3.3. ábra is.



3.3. ábra: Kétirányú Dijkstra két távoli pont esetén

Emiatt a kétirányú Dijkstra az egyirányú Dijkstrával szemben annál hatékonyabb lesz, minél messzebb van a kezdő- és a célcímsúcs egymástól a legrövidebb út élszámát tekintve. Természetesen ha sok célcímcshoz keresünk egyszerre legrövidebb utat, akkor a 3.2.1.3-nek megfelelően az egyirányú Dijkstra lesz a hatékonyabb.

3.2.3. Az egyirányú A* algoritmus

Az egyirányú A* a Dijkstrával szemben egy heurisztika alkalmazásával a célcímsúcs felé tolja el a keresés irányát [19], így csökkentve a vizsgálandó csúcsok számát.

3.2.3.1. Az algoritmus menete

Az algoritmus menete nagyon hasonló az egyirányú Dijkstráéhoz. A legfőbb különbség, hogy a kupac nem a Dijkstrában látott értékeket tárolja.

A ciklusmagban a Dijkstrában használt értékeket hasonlítja össze, ám ezeket egy másik tárolóban tárolja. A kupacba már az ezen értékek az adott csúcsra vonatkozó heurisztika értékével megnövelt értékét teszi. Ezen heurisztika minden csúcschoz egy nem-negatív alsó becslést ad az adott csúcsból a célcímcshoz vezető legrövidebb út távolságára. Mivel az ezen heurisztikával növelte értékeket közül választja ki a minimumot minden lépésben, ez szemléletesen azt jelenti, hogy a célcímsúcs felé eső csúcsokat próbálja előnyben részesíteni.

Az algoritmusnak hasonlóan a Dijkstrához akkor lesz vége, amikor a célcímsúcs vizsgálttá válik, avagy ha a kupac kiürül. Ezen utóbbi eset csak akkor fordulhat elő, ha a célcímsúcs nem elérhető a forráscímcshóból.

3.2.3.2. Az algoritmus pszeudokódja

Az itt bemutatott pszeudokód a jelölésekkel teljes egészében megtalálható [1]-ben.

1. function AStar(Graph, source, target):
2. for each vertex v in V:
3. dist[v] := infinity;

```

4.           previous[v] := undefined;
5.       end for;
6.       dist[source] := 0;
7.       Q := the set of all nodes in Graph
8.       while target is in Q:
9.           u := vertex in Q with the smallest dist[u,target] +
              heuristic[u,target];
10.          if dist[u] = infinity:
11.              break;
12.          end if;
13.          remove u from Q;
14.          for each neighbor v of u, v not in S:
15.              alt := dist[u] + length(u,v);
16.              if alt < dist[v]:
17.                  dist[v] := alt;
18.                  previous[v] := u;
19.              end if;
20.          end for;
21.      end while;
22.      return dist[];
23. end AStar.

```

3.2.3.3. Az algoritmus helyessége

A bizonyításhoz vezessünk be pár jelölést. Jelölje $g(x)$ az x csúcs kezdőcsúcstól való pillanatnyilag ismert legrövidebb távolságát, $h(x)$ pedig a rá vonatkozó heurisztika értékét. Jelöljük továbbá $dist(x,y)$ -nal az x és y közti (tényleges) távolságot.

Amennyiben teljesülnek az előfeltételek, azaz a nemnegatív élhosszak, valamint az, hogy a heurisztika minden csúcsra egy nemnegatív, konzisztens alsó becslést ad a célcúcstól való távolságára, az algoritmus helyessége két állításon nyugszik:

1. Amennyiben egy v csúcsot már elérte az algoritmus, de még nem vizsgált meg, $g(v)$ a legrövidebb út azon utak közül, melyek csak a már megvizsgált csúcson keresztül vezetnek hozzá a kezdőcsúcsból.

2. minden, már megvizsgált v csúcsra $g(v)$ megegyezik a ténylegesen legrövidebb távolsággal a forráscsúcstól nézve, azaz $\text{dist}(\text{kezdőcsúcs}, v)$ -vel.

Mivel az algoritmus akkor áll le, amikor a célcímsúcs is vizsgálta válik, a 2-es pont teljesülése esetén helyes az algoritmus.

Az 1-es pont egyenesen következik az algoritmus működéséből, ugyanis amikor elérhetővé válik egy csúcs, akkor addig még nem létezett hozzá vezető út az eddig már megvizsgált csúcsokon keresztül, így triviálisan ezen új út lesz a legrövidebb ilyen. Innen elő kezdve pedig amíg az adott csúcs megvizsgálta nem válik, minden egyes új csúcs megvizsgálásakor ellenőrizve és karban lesz tartva ezen távolság.

A 2. pont bizonyítása már összetettebb. Tegyük fel, hogy épp a v csúcsot vizsgáljuk meg (azaz épp a v csúcsot vesszük ki a kupacból). Nevezzük a forráscsúcsból v-be vezető tényleges legrövidebb úton az első, még nem megvizsgált csúcsot p-nek (p akár v is lehet). Az 1-es pont miatt $g(p)$ megegyezik $\text{dist}(\text{kezdőcsúcs}, p)$ -vel, hiszen csak úgy lehet a v-be vezető legrövidebb út első, még nem vizsgált eleme, ha a hozzá vezető legrövidebb út minden eleme már vizsgált. Mivel $h()$ konzisztens, $h(p) - h(v) \leq \text{dist}(p, v)$. Mivel v a kupac jelenlegi minimuma, és p még nem vizsgált csúcs, ezért $g(v) + h(v) \leq g(p) + h(p)$. Ezek, valamint a távolságokra vonatkozó háromszög-egyenlőtlenség felhasználásával:

$$\begin{aligned} g(p) + h(p) &= \text{dist}(\text{kezdőcsúcs}, p) + h(p) \leq \text{dist}(\text{kezdőcsúcs}, p) + \text{dist}(p, v) + h(v) \\ &\leq g(v) + h(v) \leq g(p) + h(p) \end{aligned}$$

Tehát p megegyezik v-vel, így valóban $g(v)$ a legrövidebb út v-be a forráscsúcsból, azaz 2., és így az algoritmus helyessége is bizonyított.

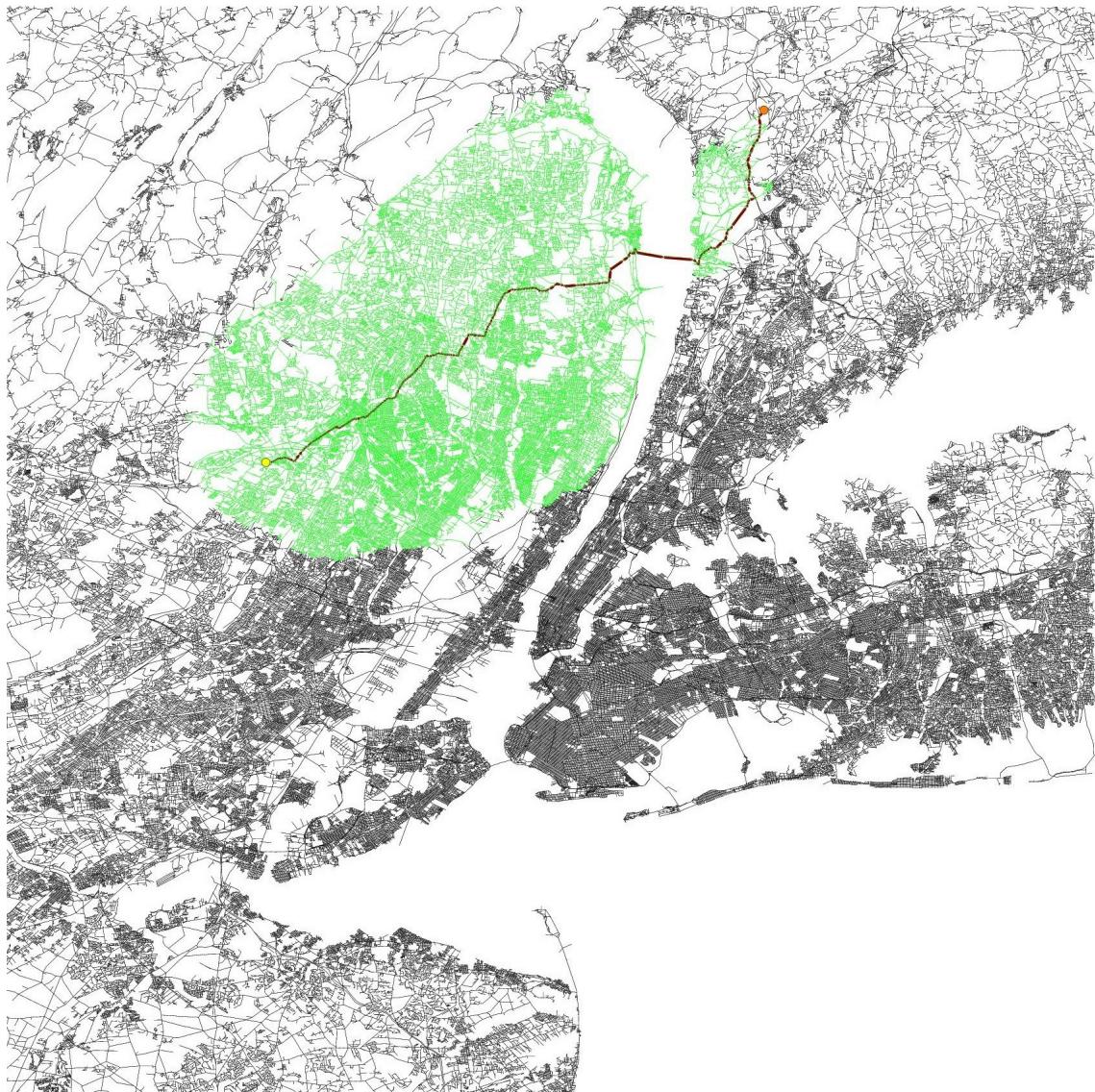
Ezen bemutatott bizonyítás megtalálható [1]-ben.

3.2.3.4. Az algoritmus előnyei, hátrányai

Az egyirányú A* hatékonyisége főképp a megadott heurisztikán múlik. Nagyon jó heurisztika esetén csak a legrövidebb úton lévő csúcsokat vizsgálja meg, nagyon rossz heurisztika esetén akár az összes csúcsot. Így használatához elengedhetetlen a heurisztika adott gráfon való alkalmazhatósága.

Ugyan ez sokszor nehéz lehet, általában rendelkezésre áll néhány átlagos hatékonyiságú heurisztika, úthálózatok esetén például általánosan és könnyen használható

heurisztika az euklideszi távolság, amennyiben adottak a koordináták. Ugyan közel sem a lehető leghatékonyabb, de sokszor rendelkezésre áll, és mint az az alábbi, 3.4-es ábrán is látszik, általában jelentősen kisebb részét járja be a gráfnak az egyirányú, de még a kétirányú Dijkstrával szemben is.



3.4. ábra: Egyirányú A* két távoli pont esetén

3.2.4. A kétirányú A* algoritmus

Az egyirányú A* algoritmus egyre szélesedő ellipszis alakban kereste a legrövidebb utat, így a kétirányú változata csökkentheti a bejárt területet, bár érezhető, hogy a nyereség csak töredéke lesz a Dijkstra kétirányúsításánál tapasztalhatónak.

3.2.4.1. Az algoritmus menete

A kétirányú A* algoritmus két, egymással szemben futó A*, ahol a visszafelé haladó irány a fordított gráfon halad. Az algoritmus akkor áll le, amikor valamelyik oldal kupaca kiürül. Amíg nem ér össze a két oldal, addig hagyományos A*-ként haladnak, utána azonban már lesz egy felső becslés a legrövidebb útra, melynek segítségével jelentősen szűkíthető a vizsgálandó csúcsok halmaza, így érve el gyorsítást az egyirányú A*-hoz képest.

3.2.4.2. Gyorsítások a legrövidebb útra vonatkozó felső becslés segítségével

A két oldal összeérésével kapott felső becslés segítségével [1] alapján többféleképpen is csökkenthetjük a vizsgálandó csúcsok számát.

Az egyik ilyen csökkentés triviális, miszerint ha az éppen vizsgálandó csúcs kupacbeli értéke nagyobb az eddig talált legrövidebb útnál, akkor ezen csúcs biztosan nem szerepelhet a legrövidebb úton, hiszen ezen érték ráadásul csak alsó becslése a tényleges értéknek.

A másik csökkentés azon az ötletek alapszik, hogy amennyiben az ellenkező oldal még nem vizsgálta az éppen vizsgálandó v csúcsot, akkor adjunk egy alsó becslést az ellenkező oldali pillanatnyilag legrövidebb távolságára. Ezt legkönyebben úgy lehetjük meg, ha számon tartjuk a kupacok legutóbbi minimumát. Ez ugyanis egy alsó becslés az adott oldal által még nem vizsgált csúcsok kupacbeli értékére. Ha ebből kivonjuk a v csúcsra vonatkozó ellenkező oldali heurisztika értékét (a kupac a pillanatnyilag ismert távolság és a heurisztika értékének összegét tárolja), akkor meg is kaptuk a kívánt alsó becslést. Amennyiben ennek és az aktuális oldali pillanatnyilag legrövidebb távolságnak az összege nagyobb az eddigi legrövidebb útnál, ezen csúcsot is biztosan ki lehet hagyni. (Ezen logika a 3.2.4.3-ban szereplő pszeudokód 14. sorában jelenik meg.)

Ezen csökkentések, valamint a kétirányú A* helyességének bizonyítása megtalálható [1]-ben.

3.2.4.3. Az algoritmus pszeudokódja

Jelölje L a talált legrövidebb utat, valamint * az adott halmaz, vagy függvény el- lenkező oldali megfelelőjét. Ekkor az algoritmus pszeudokódja [1] alapján:

```

1. function ABiStar(Graph, source, target):
2.     for each vertex v in V:
3.         dist[v] := infinity;
4.         previous[v] := undefined;
5.     end for;
6.     S := {}
7.     L, F := infinity;
8.     dist[source] := 0;
9.     Q := the set of all nodes in Graph;
10.    while Q is not empty:
11.        while Q is not empty:
12.            u := vertex in C with the smallest dist[u,target] +
13.                heuristic[u,target];
14.            if u is not in S* and dist[u]+F*-heuristic*[source,u] >= L:
15.                remove u from Q;
16.            else:
17.                break;
18.            end if;
19.        end while;
20.        if Q is empty OR dist[u] = infinity:
21.            break;
22.        end if;
23.        add u to S;
24.        F := dist[u] + heuristic[u,target];
25.        if u is in S*:
26.            L := min(L,dist[v] + dist*[v]);

```

```

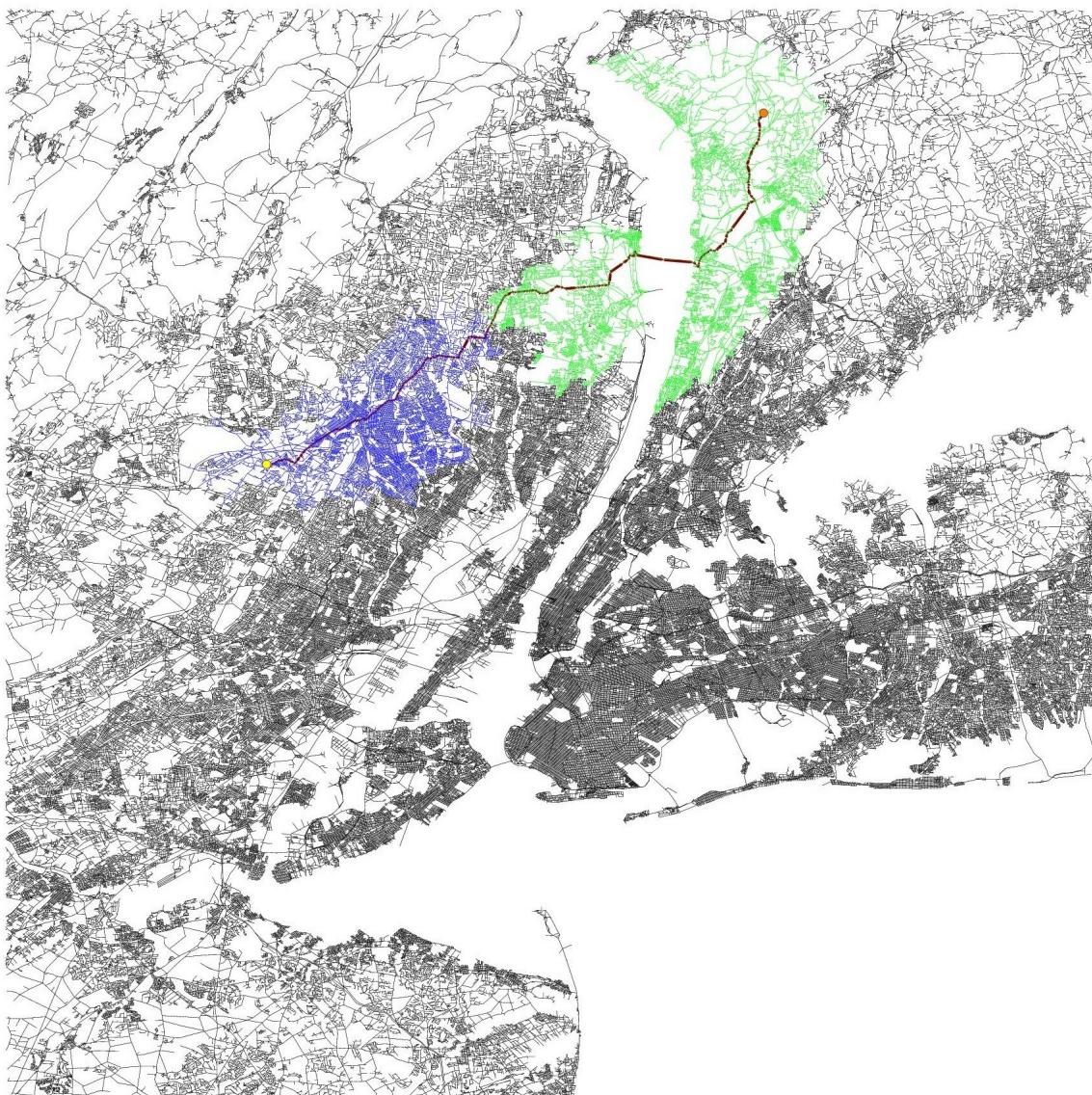
21.           else:
22.               for each neighbor v of u, v is in Q:
23.                   alt := dist[u] + length(u,v);
24.                   if dist[v] > alt:
25.                       dist[v] := alt;
26.                       previous[v] := u;
27.                   end if;
28.               end for;
29.           end if;
30.       end while;
31.   return dist[];
32. end ABiStar.

```

A fenti pszeudokód nem írja le a két oldal közti váltást, csak egy oldalt ír le. Amennyiben épp a visszafelé oldalon fut, úgy a source target-nek, a target pedig source-nak értendő.

3.2.4.4. Az algoritmus előnyei, hátrányai

A kétirányú A* algoritmusnak egyirányú párrával szemben akkor lehet jelentős előnye, ha viszonylag rossz becslést ad a heurisztika, avagy ha egy átlagos heurisztika van, de gráf sűrűségéhez képest a forrás- és célcímsúcs kellően messze van egymástól, mint például az alábbi, 3.5. ábrán is:



3.5. ábra: Kétirányú A* két távoli pont esetén

Közeli pontok esetén is kevesebb csúcsot jár be egyirányú társához képest, ám ez esetben a pluszköltségek még jelentősen felülmúlják a kevesebb csúcsból származó előnyöket.

3.3. Az algoritmusok implementációja

Ezen fejezet az algoritmusok implementációinak legfontosabb részleteit mutatja be. A felhasználói megközelítéssel a 2.4. fejezet foglalkozik.

3.3.1. Az egyirányú Dijkstra algoritmus

Az egyirányú Dijkstra algoritmus az egyetlen az ezen szakdolgozatban szereplő négy algoritmus közül, mely már most (1.2.1. verzió) is a LEMON része, ám a többi három algoritmus nagyon nagy mértékben épít rá mind logikai, mind implementációs szempontból. Ezért szükséges részletes bemutatása.

3.3.1.1. A legfontosabb adattagok bemutatása

Az algoritmus megvalósításában a legfontosabb szerep az alábbi adattagoknak jut:

- G: Konstans pointer arra a gráfra, amelyen az algoritmus fut. Bármiel, a gráf-koncepciónak megfelelő gráftípus lehet.
- _length: Konstans pointer az élsúlyokat tároló map-re.
- heap: Pointer az algoritmus fő tárolójaként funkcionáló adatszerkezetre. Alapértelmezetten bináris kupac, de tetszőleges rendezett, vagy azt szimuláló adatszerkezet megadható.
- _pred: A megelőzőket tároló map-re mutató pointer.
- _dist: A távolságokat tároló map-re mutató pointer.
- _processed: Pointer a véglegesített csúcsokat, és azok véglegesített távolságukat tároló map-re.

3.3.1.2. Az algoritmus fő logikáját végző metódus, a processNextNode

Az algoritmus lényege egy metódusban van, ez hajtja végre az algoritmus ciklusmagját, vagyis feldolgozza a következő csúcsot.

Kiveszi a _heap legnagyobb prioritású v elemét, és véglegesíti ennek távolságát a finalizeNodeData(v) függvény meghívásával. Ezzel válik vizsgálttá az adott csúcs.

Ezután végigiterál v összes kimenő élén, és megvizsgálja azok w végpontjait.

Ezen w _heap-hez tartozó állapota szerint az alábbiakat hajtja végre:

- ha w még nem volt a _heap-ben, beleteszi a v távolságának és a v -> w él hosszának összegével, valamint beállítja v-t megelőzőjének.
- ha w már benne volt a _heap-ben, megvizsgálja, hogy az új értéke kisebb-e a korábbinál. Ha igen, akkor javítja, valamint beállítja v-t megelőzőjeként.
- ha w már ki lett véve a _heap-ből, akkor nem csinál vele semmit.

Végül visszatér v-vel.

Mivel ezen metódusban van az algoritmus műveletvégzésének lényegi része, gyakorlatilag ez adja az algoritmus művelet- és időigényét. Így nagyon fontos, hogy a lehető legjobban legyen optimalizálva. Ebben nagy segítséget jelentenek a LEMON-beli osztályok, így például a különböző kupacok hatékony implementációja, valamint a map-ek konstans idejű elérése, módosítása.

3.3.2. A kétirányú Dijkstra algoritmus

A kétirányú Dijkstra algoritmus interfésze teljesen változatlan egyirányú társához viszonyítva, ám belső megvalósításában gyakorlatilag két egyirányú Dijkstrát használ.

3.3.2.1. A legfontosabb adattagok bemutatása

A kétirányú Dijkstra minden adattagot tartalmaz, amit egyirányú társa, sőt, majdnem mindegyikból kettőt is. Ezekben felül szükséges pár adattag a két oldal összehangolása céljából. Itt most csak azon fontosabb adattagokat mutatom be, melyek nem szerepeltek az egyirányú Dijkstrában.

- _rev_heap: Pointer a visszafelé haladó rész fő adattárolójára. Típusa megegyezik a _heap típusával.
- _rev_pred: A visszafelé haladó rész megelőzőit tartalmazó map-re mutat.
- _rev_dist: Pointer a visszafelé haladó rész távolságait tartalmazó map-re.
- _rev_processed: A visszafelé haladó rész véglegesített csúcsait tároló map-re mutat.
- reverse_is_next: Ezen logikai változó alapján dönti el az algoritmus, hogy a következő ciklusmagban melyik oldal lépjen.
- bi_path_found: Ezen logikai változó jelzi, hogy összeért-e már a két oldal.

- `con_node_id`: Az eddigi legrövidebb út azon csúcsának az azonosítóját tárolja, amelynél a két oldal találkozott.
- `shortest_dist`: Az eddigi legrövidebb utat tárolja.
- `bi_dist_found`: Ezen logikai változó jelzi, hogy teljesült-e már a megállási feltétel.

3.3.2.2. Az algoritmus fő logikáját végző metódus, a `processNextNodeBi`

A `processNextNodeBi()` metódus az algoritmus lényegi részét, a ciklusmagot tartalmazza.

A metódus legelején meghívja `chooseDirection()` függvényt, mely beállítja a `reverse_is_next` adattagba, hogy melyik oldal lesz léptetve ebben a ciklusmagban. Ezt a két oldal kiegyszúlyozásának elvén dönti el, vagyis figyelembe veszi a kupacokba való beszúrásokat, módosításokat és kétszeres súllyal a kivételeket. Ez a logika a bináris kupac műveletigényei alapján dönt, más adatszerkezet használata esetén kiegyszúlyozott-lansághoz vezethet. Így későbbi fejlesztés esetén hasznos lehet biztosítani egy funktor megadásának lehetőségét erre a célra.

A megfelelő oldal kiválasztása után végrehajt egy ciklusmagot az adott oldalon. Hala a gráfok LEMON-beli ábrázolásának, sem végrehajtás, sem tárhelyigény szempontjából nem jár semmilyen plusz teherrel az, hogy a visszafelé irány a fordított gráffal dolgozik. Annyi ugyanis csak a teendő, hogy a kimenő élek helyett a bejövő éleken kell iterálni, és azok célcsúcsai helyett a forráscsúcsait kell vizsgálni.

Egy ciklusmag lefutása az elején csak annyiban különbözik az egyirányú Dijkstrában látottól, hogy leellenőrzi, hogy az éppen vizsgált v csúcs a másik oldal által meg lett-e már vizsgálva. Amennyiben igen, akkor teljesül a megállási feltétel, ismert a legrövidebb út, és a metódus vissza is tér.

Ellenkező esetben az egyirányú Dijkstrában látott hasonlóan végigiterál v kimenő élein. Azonban minden w szomszéd esetén ellenőrizni kell, hogy nem létezik-e rajta keresztül egy út a forráscsúcsból a célcsúcsba. Ez akkor igaz, ha az adott csúcs a másik irány kupacában szerepel, avagy már ki is lett onnan véve. Ekkor le kell ellenőrizni, hogy az eddig talált legrövidebb útnál nem rövidebb-e ez az út. Amennyiben igen, akkor mind a `shortest_path`, mind a `con_node_id` változókat karban kell tartani. Természetesen ezen ellenőrzésre csak akkor van szükség, ha az adott w csúcshoz tárolt

távolság változott, azaz most értük el először, avagy találtunk hozzá egy rövidebb utat. Ellenkező esetben ugyanis ugyanezzel az értékkel már korábban elvégeztük az ellenőrzést (vagy esetleg a másik oldalról, ha onnan értük el).

Ám az egyirányú Dijkstrával ellentétben a processNextNodeBi befejeződése után még némi utómunkálatra van szükség. A megállási feltétel esetén ugyanis, ha találtunk legrövidebb utat, az csak két félként tevődik össze, így a csatlakozási ponttól a visszafelé oldalon egészen a célcísig végig kell iterálni, és bejegyezni az előrefelé oldalon a megfelelő távolságokat, valamint a legrövidebb úton az adott csúcs előtt közvetlenül szereplő csúcsot. Amint így elérünk a célcísig, kész az eredmény, lekérdezhető a legrövidebb út a forrás- és a célcíms között.

3.3.3. Az egyirányú A* algoritmus

Az implementáció szempontjából az egyirányú A* az egyirányú Dijkstrától a heurisztika megadásában, számolásában és az ezzel kapcsolatos adattagok jelenlétében, karbantartásában tér el.

A heurisztika megadásának módjaival a 2.4.3.1-2 rész foglalkozik.

3.3.3.1. A legfontosabb adattagok bemutatása

Az egyirányú A*-ban az egyirányú Dijkstrához képest csak néhány új adattag van, melyek mindegyikére a heurisztika bevezetése miatt van szükség. Ezek a következők:

- est_calc: A heurisztikát számoló funkcióra mutat. Ennek típusát a példányosítás-kor megadott sablonparaméter határozza meg.
- heap: Ugyan nevét tekintve nem új adattag, de tartalmát tekintve igen, ugyanis a csúcsokat egy másik kulccsal tárolja, a 3.2.3.1. részben bemutatott módon a heurisztikával növelt értékekkel.
- current_dist: A kupac megváltozott funkciója következtében ezen változó tartja karban az eredeti értelemben vett pillanatnyilag ismert távolságokat.

3.3.3.2. Az algoritmus fő logikáját végző metódus, a processNextNode

Hasonlóan a két Dijkstrához, ezen algoritmus legfontosabb működése is a processNextNode metódusban található. Fontos különbség azonban, hogy vár egy paramétert, a célcímcsoportot, hiszen a heurisztika számításához erre szükség lesz.

A legfontosabb különbség, hogy más értékeket, a távolságok heurisztikával növelt értékét tárolja. Ennek ellenére továbbra is ez a fő tároló, amely szerint az algoritmus fut, ennek az aktuálisan legkisebb értékű v eleme lesz a következő vizsgálandó csúcs.

Ezzel párhuzamosan minden karban tartjuk a pillanatnyilag ismert távolságokat is, de egy másik tárolóban. A már elérte, de még nem vizsgált csúcsok esetén az új távolság összehasonlítása természetesen továbbra is ezen tároló adatai alapján történik.

Az egyirányú Dijkstrához képest semmi különbség nincs sem a megállási feltétel, sem az utómunkálatok terén.

3.3.4. A kétirányú A* algoritmus

A kétirányú A* tartalmazza a 4 algoritmus közül a legtöbb adattagot, minden szempontból ez tekinthető a legbonyolultabbnak.

3.3.4.1. A legfontosabb adattagok bemutatása

Ugyan ezen algoritmus implementációja tartalmazza a legtöbb adattagot, ám ezek túlnyomó többsége csak az egyirányú A* adattagjainak kétirányúsítása, mint ahogyan az a 3.3.2.1. pontban a kétirányú Dijkstra esetében is látható.

Így ezek részletezésére nem térek ki, értelmezésük magától értetődő.

Ez alól egyedül a laststar, és lastrevstar változók képeznek kivételt. Ezen változók a megfelelő oldali kupac legutóbbi minimumát tárolják.

3.3.4.2. Az algoritmus fő logikáját végző metódus, a processNextNodeBi

A kétirányú A* esetében is a processNextNodeBi metódusban valósul meg az algoritmus logikája. Ahogyan az egyirányú A* esetében szükség volt a célcímszám paraméterként való átadására, úgy ezen algoritmusban már mind a forrás-, mind a célcímszám paraméterként szerepel, hiszen az előrefelé haladó rész a cél-, míg a visszafelé haladó rész a forráscímszámot használja a heurisztika számításához.

Hasonlóan a kétirányú Dijkstrához, ebben az esetben is az irányválasztással kezdődik a metódus, és ennek megfelelően vagy az előre-, vagy a visszafelé haladó irány hajt végre egy ciklusmagot. A használt tárolók, valamint a szomszédos csúcsok bejárása közti, 3.3.2.2-ben leírt különbségétől eltekintve minden két irány azonos műveleteket hajt végre, így elegendő az egyik irányt megvizsgálni.

A vizsgálandó csúcs, valamint a szomszédos csúcsok kezelése teljesen megegyezik az A*-nál látottnál. A különbség a megállási feltételben, valamint a két oldal találkozása utáni optimalizációban van.

A megállási feltétel nagyon egyszerű, mindaddig fut az algoritmus, amíg valamelyik oldali kupac ki nem ürül. Ez persze csak akkor hatékony az egyirányú A*-gal szemben, ha a két irány segítségével lehetőség nyílik a vizsgálandó részgráf jelentős leszűkítésére.

Ezen javítások egyrészt a laststar valamint lastrevstar változók karbantartásában, másrészt a 3.2.4.2-ben tárgyalt, a vizsgálandó csúcsra vonatkozó feltételekben valósulnak meg. Amennyiben a két oldal már összeér, minden egyes csúcs kupacból való kivétele esetén csak akkor fut le a ciklusmag műveletigényes része, ha az adott csúcs eleget tesz a feltételeknek. Mint az a 3.4.3.2. részben látszik, ezzel jelentősen csökkenhető a vizsgálandó csúcsok száma.

Miután valamelyik kupac kiürült, a kétirányú Dijkstránál látott utófeldolgozásra van szükség, hogy a talált legrövidebb út két külön álló feléből összeálljon a teljes út.

3.4. Tesztelés

A programozás három legfőbb fázisa a tervezés, implementálás és a tesztelés. Ezek mindegyike elengedhetetlen minden fejlesztésnél. A tesztelés során ellenőrizzük, hogy a céloknak megfelelően működik-e a program, ennek során gyűjtünk tapasztalatot

a működésről, de a tesztelés keretein belül javíthatjuk is programunkat. Így a tesztelés feladata hármas: ellenőrzés, megismerés, javítás.

3.4.1. Helyesség

A helyesség ellenőrzésénél több szempontot is figyelembe véve kombináltam a teszteseteket. Így változtattam a használt gráfot és algoritmust, a forrás- és célcímsokat, valamint az A*-ok esetében a használt heurisztikát is. Természetesen minden esetben minden a három algoritmust ellenőriztem, eredményüket az eredeti egyirányú Dijkstra eredményével egyeztetve. Mivel ez utóbbi már a LEMON része, teljes körűen le lett tesztelve, eredményei biztosan helyesek.

3.4.1.1. A használt gráfok

Az alábbi gráfokat használtam, ezekre a fejezet hátralévő részében csak az itt felsorolt számukkal fogok hivatkozni:

1. **New York úthálózata:** egy 264 346 csúcstól és 733 846 élt tartalmazó gráf, mely New York teljes úthálózatát tartalmazza. Ezt a gráfot a [14]-ben található fájlból készítettem átkonvertálva a LEMON által használt „lgf” formátumba. [15]
2. **New York „simított” úthálózata:** Ezen gráfot az előzőből készítettem oly módon, hogy minden él súlyát kicseréltem a kezdő-, és végpontja közti euklideszi távolságra, így „kisimítva” a gráfot.
3. **New York utazási idő gráfja:** Az 1-es gráffal megegyező szerkezetű gráf, de az elsúlyok a távolságok helyett az utazási időket jelentik.
4. **New York „irányított” gráfja:** Ugyan a többi New Yorkot leíró gráf is irányított, ezek csak szimuláltan irányítottak, azaz minden két csúcs között minden irányba létezik él, és ezen élek súlya megegyezik. Így tulajdonképpen irányítatlannak tekinthetők. Ezen gráf esetében azonban minden szomszédos csúcspár esetén 27,75% eséllyel töröltem az egyik élt. (Minden élpárnál 15% eséllyel töröltem az első élt. Amennyiben nem töröltem, úgy az ellentétes irányú élt töröltem 15% eséllyel. Így annak a valószínűsége, hogy egy élpár esetén valamelyik él törlődött, $0,15 + 0,85 * 0,15 = 0,2775 = 27,75\%$.) Így ez a gráf már valóban irányított, ráadásul nem is erősen összefüggő, számos keresett út nem létezik (de

gyengén összefüggő maradt).

5. **Észak-nyugat Egyesült Államok úthálózata:** egy 1 207 945 csúcsot és 2 840 208 élt tartalmazó gráf, megy az Amerikai Egyesült Államok észak-nyugati részének teljes úthálózatát tartalmazza. Hasonlóan az 1. gráfhoz, ezt is [14]-ből konvertáltam. Ez volt a legnagyobb gráf, ami belefért a használt számítógép memóriájába. Mérete miatt csak pár tesztet végeztem ezen a gráfon.
6. **Különböző sűrűségű, „négyzet alakú” generált gráfok:** Készítettem gráfokat különböző sűrűséggel és mérettel. Ezek csúcspontjai síkban rajzolva négyzet alakban, szabályos rácssonterkezetben helyezkednek el. Az élhosszakat véletlenszerűen választottam meg, figyelve arra, hogy az euklideszi távolság alsó becslés maradjon. Néhány példa:
 - a) 961 csúcsot tartalmazó teljes gráf (azaz 923 521 élt tartalmazott, mert minden csúcshoz tartozott egy hurokél is)
 - b) 1369 csúcsot és hozzávetőlegesen 937 080 élt tartalmazó „50%-os” gráf, amit a teljes gráftól annyiban készítettem eltérően, hogy bármelyik élt csak 50% valószínűséggel hoztam létre
 - c) Több olyan, 18 769 csúcsot tartalmazó gráf, melyeknél megszabtam, hogy egy csúcsból maximum milyen „messze” lévő csúcsba mehet él. A „messze” fogalma itt azt jelentette, hogy a szabályos rácssonterkezeten elhelyezkedő csúcspontoknál a rácsonalakon haladva legkevesebb hány lépésből lehet az adott csúcsba eljutni. Ezen gráfokat is több sűrűséggel készítettem (pl.: 10%-os gráf 7 megengedett távolsággal, 100%-os gráf 1 megengedett távolsággal. A százalék itt úgy értendő, hogy ekkora volt az esélye, hogy egy élt megpróbáljak létrehozni, és csak ezután ellenőriztem, hogy megfelelő távolságra van-e).
7. **Speciális példagráfok:** Ismert speciális helyzetet szimuláló példagráfok, mint például amilyen a 3.2. ábrán is szerepel.

Ezen fenti gráfok közül a 4-es, 5-ös gráfot, valamint a 6-os és 7-es gráfokat kizárával helyesség ellenőrzésére használtam (azaz nagy elemszámú teszteket futtattam mindegyik algoritmusra, az eredményeket pedig a Dijkstra eredményeivel ellenőriztem), így ezekről bővebben nem lesz szó.

3.4.1.2. Forrás- és célcímszövegek, útvonalak

A speciális példagráfok esetén természetesen a speciálisságot megtestesítő útvonalat kerestem. minden más gráf esetén a forrás- és célcímszövegeket, tehát a keresendő útvonalakat véletlenszerűen választottam ki. Ezen tesztelésekben a véletlenszerű útvonalak száma 10-18 ezer volt, a kevés csúcsot tartalmazó gráfok esetén (6. típusúak) pedig 200 vagy 500, hiszen ezeknél több teszteset esetén ugyanazon útvonalakat ellenőriztem volna újra és újra.

3.4.1.3. Heurisztika

Az A* algoritmusok tesztelése esetén az alábbi heurisztikákat vizsgáltam, a fejezet hátralévő részében ezekre az itt megadott számukkal fogok hivatkozni:

1. **Alapértelmezett heurisztika:** Ezen heurisztika bármely két csúcs esetén 0-t ad vissza, így használatával az A* az eredeti Dijkstra működésével teljesen megegyezik (csak kevésbé hatékony).
2. **Euklideszi távolságot előre kiszámító heurisztika:** Ezen heurisztika egy megadott célcímszöveg előre kiszámítja az összes ponttól vett euklideszi távolságát, és meghívásakor ezeket adja meg.
3. **Euklideszi távolságot valós időben kiszámító heurisztika:** Ezen heurisztika 2-vel ellentétben csak a koordinátákat inicializálja példányosításkor, és bármely két kért csúcs esetén futási időben számolja ki az euklideszi távolságukat.
4. **Utazási idő heurisztika:** Ez a 3. gráfhoz használt heurisztika, bármely két pontra euklideszi távolságuk és a gráf élein kiszámolt legnagyobb sebesség hányadosát adja.
5. **Tényleges távolságot adó heurisztika:** Ezen speciális heurisztika minden csúcs esetén tárolta a kezdőcsúcsból, valamint a célcímszöveg vezető legrövidebb út hosszát. Ezen előre kiszámított értékeket egy előre generált fájlból olvasta be. Így mind az egyirányú A*, mind pedig a kétirányú A* minkét oldala esetén a megfelelő értéket adta vissza minden csúcsnak.

Mivel általában egy adott csúcs heurisztikáját sokszor felhasználjuk egy útkeresés folyamán, hasznos ötlet lehet, hogy ezen értékeket csak egyszer számoljuk ki, letároljuk, és későbbi hívások esetén ezen letárolt értéket adjuk vissza. Ezzel az ötlettel minden

egyik heurisztika módosítható. Teszteléseim alapján azonban nem volt kíméletlenül futásidőbeli különbség az eredeti és az ily módon „javított” heurisztikák használata között, így erre nem térek ki külön. Természetesen ezen megoldás előnye nagyban függ a kiszámoló függvény műveletigényétől. Az általam használt heurisztikáknál azonban ezen műveletigény nagyon alacsony volt, és nem haladta meg a tárolással kapcsolatos plusz műveletek költségeit.

3.4.2. Eredmények, összehasonlítások

Ebben a fejezetben három gráfon keresztül bemutatom az algoritmusok közti hatékonyágbeli különbségeket. Az eredmények könnyebb áttekinthetőségének érdekében a grafikonokon nem szerepelnek a tengelyek mértékegységei, az X tengely mindenkorral a megtalált utat alkotó élek számát, az Y tengely pedig az idő szemléltetése esetén másodpercen belül, minden más esetben csúcosszámot jelöl.

Kétféle diagramot használunk. A „halmozott golyó” diagram minden egyes eleme egy útkeresés valamely adatának (pl.: futásidő, vizsgált csúcsok, stb.) felélel meg. A vonaldiagram elkészítésénél az adatokat az utat alkotó élek száma szerint edényekbe rendeztem, 50 élszámonként (tehát külön edényt képeznek a 0-49 élhosszú utak, az 50-99 élhosszú utak, és így tovább). Az ábrán látható csomópontok az ezen edényekben lévő adatok átlagai.

A használt elnevezések jelentései:

- Dijkstra: Egyirányú Dijkstra.
- Bijkstra: Kétirányú Dijkstra.
- AStar: Egyirányú A*.
- ABiStar: Kétirányú A*.
- Elért, de nem vizsgált csúcsok: A kupac mérete az algoritmus futásának befejeződésekor.
- Értékcsökkentések száma: Azt jelzi, hogy az algoritmus futása során összesen hányszor volt szükség egy már beállított pillanatnyilag ismert távolság értékének csökkentésére.

Számos adatot közlök az egyirányú Dijkstrához százalékosan viszonyítva. Ezeket az alábbiak szerint kell értelmezni.

Az alsó és felső kvartilisek, továbbá az „átlag”, „legrosszabb” és „legjobb” eredmények az adatok egyenkénti arányainak mutatói, tehát például az „átlag futásidő”-t úgy kapjuk, hogy egyenként meghatározzuk az útkeresések futásidéjének az egyirányú Dijkstra-beli párokhoz viszonyított arányát, és ezen eredmények átlagát számoljuk ki.

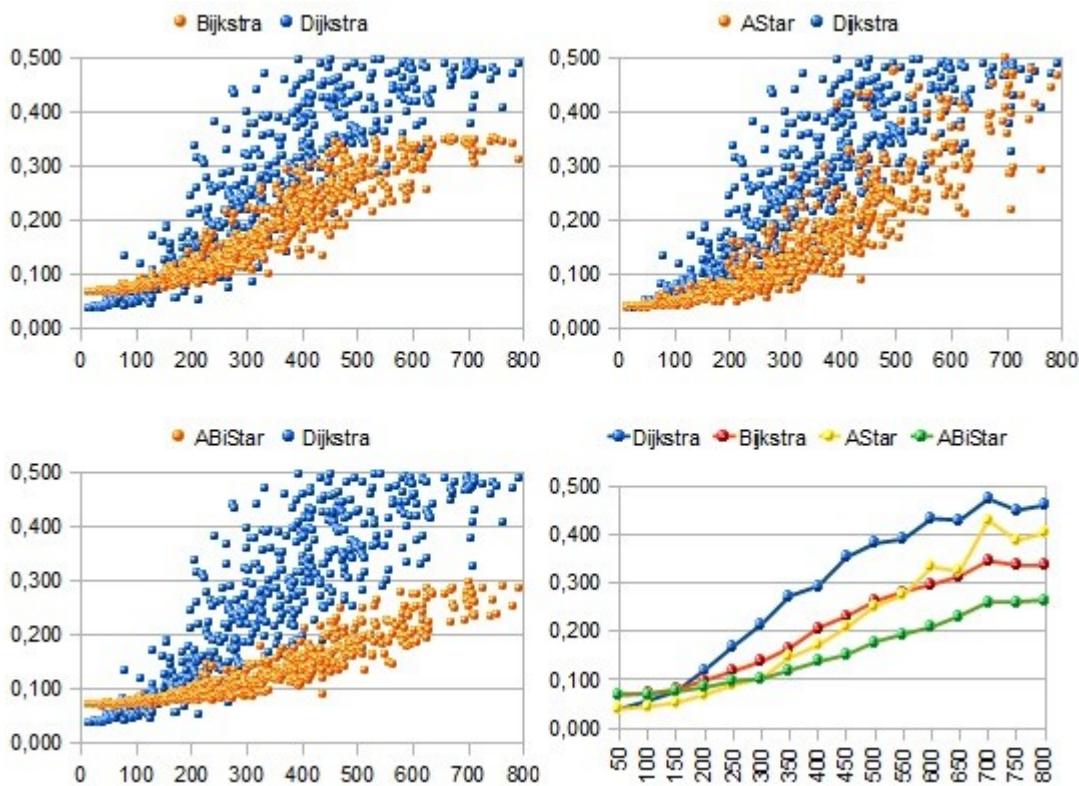
Ezekkel ellentétben az „összesített” eredmény az adatok összegének arányát mutatja az egyirányú Dijkstra összesített eredményeihez képest. Tehát futásidő esetében annak arányát kapjuk meg, hogy az adott algoritmus mennyivel gyorsabban (vagy lassabban) futott le az összes tesztesetre együttvéve.

3.4.2.1. Az „utazási idők” New York gráf

Ezen fejezetben az algoritmusok összehasonlításához a 3.4.1.1. fejezet 3-as gráfját, valamint az ebben keresett 528 db véletlenszerű útvonal keresésének adatait használom. Az A* algoritmusok heurisztikaként a 4-es heurisztikát használják.

3.4.2.1.1. Futásidő

Egy algoritmus lefutásánál általában nagyon fontos tulajdonság, hogy mennyi időt vesz igénybe. A bemutatott adatok közül ez az egyetlen, ami az adott géptől is erősen függ. Az itt szereplő eredmények egy 2 magos 2 GH-zes processzorral és 3 GB DDR2 memóriával rendelkező gép segítségével, Ubuntu 9.04 operációs rendszeren készültek.



3.6. ábra: Futáridők az „utazási idők” New York gráfon

Mint az a 3.6. ábrán is látszik, a tesztelések a legtöbb szempont szerint a várt eredményeket adták. Az egyik ilyen, hogy nagyon közeli csúcsok esetén a kétirányú algoritmusok rosszabb teljesítményt nyújtanak egyirányú társaiknál. Ennek oka, hogy ezek sok plusz költséggel járnak, és közeli csúcsok esetén még nem tudják érvényesíteni „területcsökkentő” képességüket, míg távolabb lévő csúcsok esetén már nagyobb arányban tudják szűkíteni a bejárt „területet”.

A szakdolgozat célja tekintetében fontos eredmény, hogy a közeli csúcsoktól eltekintve mind a három új algoritmus, de különösen a kétirányú változatok jobb eredményt nyújtanak az eredeti Dijkstránál.

Az egyes eredményeket külön szemléltető grafikonokon tisztán látszik, hogy a kétirányú algoritmusok esetén sokkal kisebb a szórás egyirányú társaikhoz képest, stabilabb eredményt nyújtanak.

A (a legrövidebb utat alkotó élek számát tekintve) közelebb eső csúcspárok esetén egyértelműen az egyirányú A* a leghatékonyabb, ám mivel egy viszonylag rossz heurisztikát használ, az élek számának növekedésével a kétirányú A* lesz a hatékonyabb.

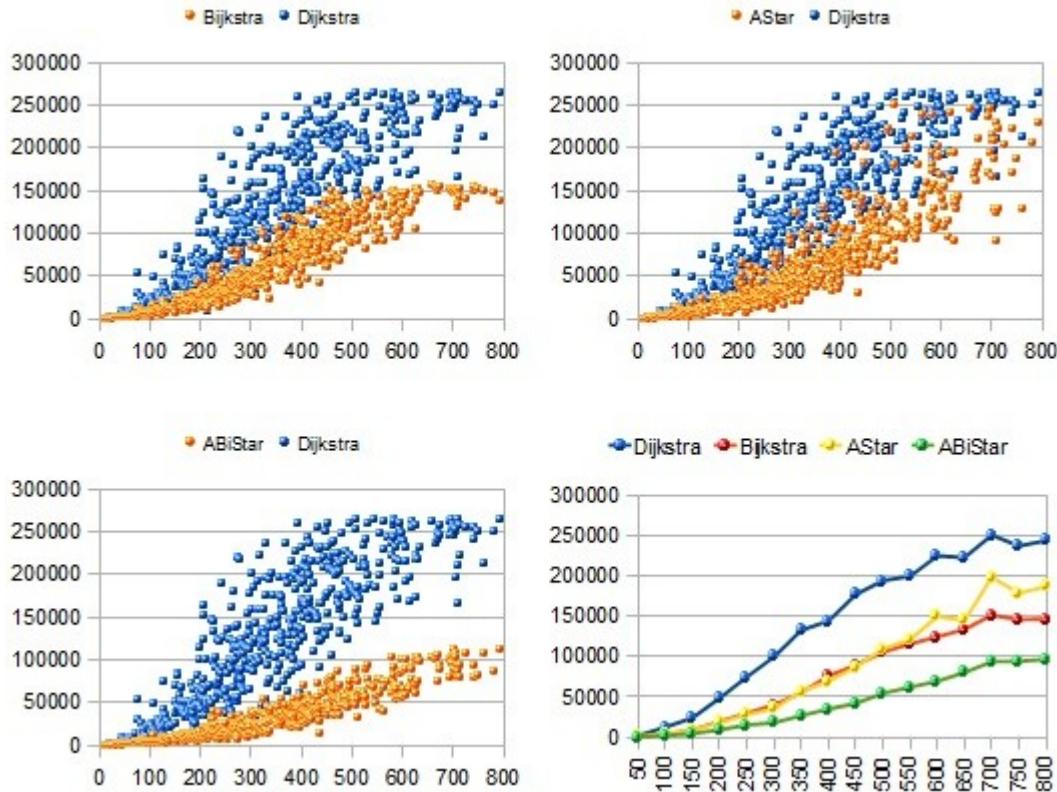
Sőt, egészen távol lévő csúcspárokra még a kétirányú Dijkstra is gyorsabb az egyirányú A*-nál.

Az összesített eredményeket az egyirányú Dijkstra azonos mérőszámaihoz száralékosan viszonyítva:

- alsó és felső kvartilisek
 - kétirányú Dijkstra: alsó: 62%, felső: 96%
 - egyirányú A*: alsó: 53%, felső: 75%
 - kétirányú A*: alsó: 43%, felső: 76%
- összesített és átlagos futásidő
 - kétirányú Dijkstra: összesített: 71%, átlagos: 83%
 - egyirányú A*: összesített: 64%, átlagos: 64%
 - kétirányú A*: összesített: 52%, átlagos: 67%
- szélsőértékek
 - kétirányú Dijkstra: legrosszabb: 186%, legjobb: 26%
 - egyirányú A*: legrosszabb: 105%, legjobb: 29%
 - kétirányú A*: legrosszabb: 192%, legjobb: 20%

3.4.2.1.2. Vizsgált csúcsok száma

Míg a futásidő sok külső paramétertől (pl.: számítógép erőssége, leterheltsége), addig a vizsgált csúcsok száma kizárolag az algoritmus működésétől függ. Emiatt fontos jellemző, hiszen segítségével más gráfstruktúrák használata esetén is könnyen megbeszélhető a futásidő. Fontos szempont lehet akkor is, ha a használt tároló adatszerkezet, vagy épp a használt gráftípus miatt egy-egy csúcs megvizsgálása nagyon költséges, ekkor ugyanis ez fog dominánssá válni a futásidőben is, akár megváltoztatva a 3.4.2.1.1-ben bemutatott arányokat is. A többi csúcosszámmal szemben azért van nagyobb jelentősége a vizsgált csúcsok számának, mert az algoritmusok működésének következtében ezzel megegyező számú iteráció hajtódik végre, a ciklusmag lefutásainak számával egyezik meg.



3.7. ábra: Vizsgált csúcsok száma az „utazási idő” New York gráfon

Mint az a 3.7. ábrán is látszik, a futásidőkkel ellentétben a vizsgált csúcsok számát tekintve kevésbé befolyásolja a távolság az algoritmusok egymáshoz viszonyított hatékonyságát. Az egyirányú Dijkstra vizsgálja meg messze a legtöbb csúcsot, a kétirányú A* pedig a legkevesebbet. Nagyjából ezen két eredmény közötti hatékonysággal rendelkezik az egyirányú A* és a kétirányú Dijkstra, kisebb távolságok esetén valamivel gyorsabb az előbbi, nagyobb távolságok esetén egyre inkább az utóbbi a hatékonyabb.

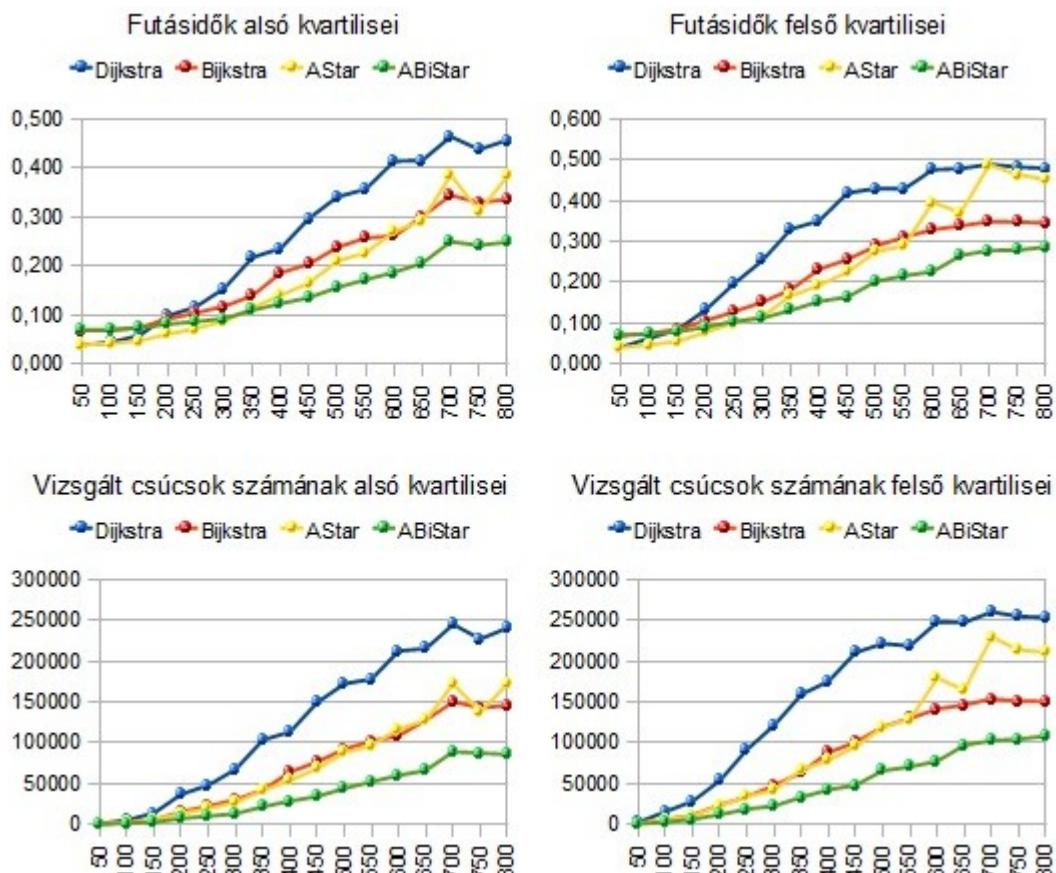
Az arányok változásának kisebb mértéke abból adódik, hogy míg futásidő tekintetében a kétirányú algoritmusoknál kis távolságokon még sokkal nagyobb a pluszköltség, mint a kétirányú keresésből fakadó haszon, addig a vizsgált csúcsok számát tekintve nem jelentkezik ilyen pluszköltség.

Az összesített eredményeket az egyirányú Dijkstra azonos mérőszámaihoz szállékosan viszonyítva:

- alsó és felső kvartilisek
 - kétirányú Dijkstra: alsó: 38%, felső: 62%
 - egyirányú A*: alsó: 38%, felső: 58%

- kétirányú A*: alsó: 43%, felső: 76%
- összes és átlagosan vizsgált csúcsszám
 - kétirányú Dijkstra: összes: 51%, átlagos: 50%
 - egyirányú A*: összes: 53%, átlagos: 48%
 - kétirányú A*: összes: 27%, átlagos: 26%
- szélsőértékek
 - kétirányú Dijkstra: legrosszabb: 108%, legjobb: 11%
 - egyirányú A*: legrosszabb: 95%, legjobb: 16%
 - kétirányú A*: legrosszabb: 90%, legjobb: 5%

Mind futásidő, mind pedig a vizsgált csúcsok számának tekintetében pontosabb képet kaphatunk az algoritmusok hatékonyságának alakulásáról, ha bizonyos intervallumokra külön is kiszámoljuk az alsó és felső kvartilisek értékét. Ezen eredményeket ábrázolja a 3.8. ábra.

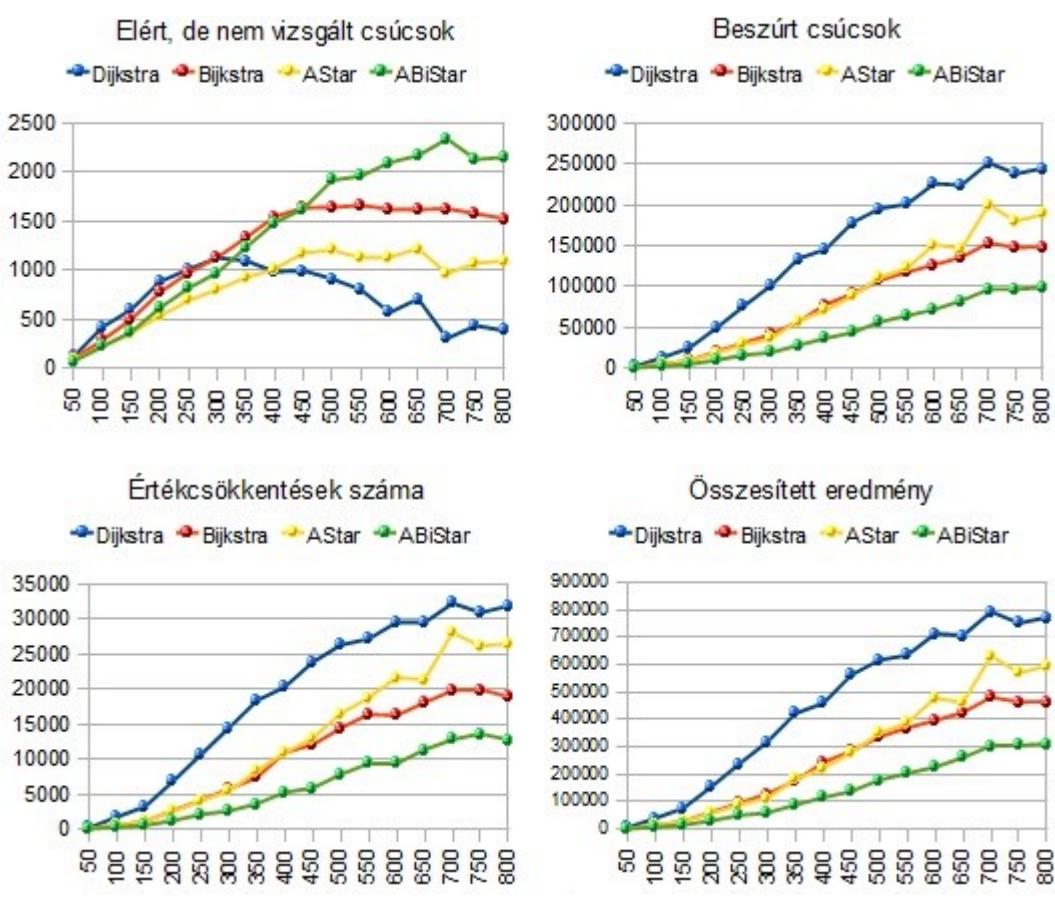


3.8. ábra: Alsó és felső kvartilisek értékei intervallumokon az „utazási idők”
New York gráfon

3.4.2.1.3. Egyéb mérőszámok

A vizsgált csúcsok száma ugyan nagyon fontos, hiszen ez megadja az algoritmus lépéseinak számát, ám ezen kívül van még néhány lényeges mérőszám. Ebben az alfejezetben négy ilyen mérőszám arányait mutatom be: az elért, de még nem vizsgált, a beszűrt csúcsok számát, a már elért csúcsok esetén történt értékcsökkentések számát, valamint egy összesítő számot, amely a kupac működésének megfelelően kétszeresen veszi figyelembe a kivételeket, egyszeresen a beszúrásokat és az értékcsökkentéseket. Mivel a vizsgált csúcsok szomszédjain való végigiterálások száma meglehetősen magas, így ezt is fontos figyelembe venni, azonban ezen költség jóval alacsonyabb a kupacműveleteknél, így ezen számot csak 1/100-ad súllyal vettetem figyelembe.

Ez az összesített mérőszám elég jól jellemzi az adott algoritmus műveletigényét (bináris kupac használata esetén).



3.9. ábra: Egyéb mérőszámok az „utazási idők” New York gráfon

A 3.9. ábrát tekintve általánosságban elmondható, hogy fennáll a 3.4.2.1.2-ben látott sorrend, sőt az arányok is nagyon hasonlóak a vizsgált csúcsok számánál tapasztalhatókhoz. Ez alól csak az elért, de még nem vizsgált csúcsok számát ábrázoló (bal felső) grafikon kivétel, ahol fordított a sorrend, főleg a távoli csúcsok esetén. Ez első ránézésre meglepő lehet, azonban jobban jobban belegondolva minél „keskenyebb sáv” a vizsgált csúcsok alkotta részgráf, annál nagyobb az elért, de nem vizsgált csúcsok és a vizsgált csúcsok aránya, hiszen szemléletesen a vizsgált csúcsok körüli „egy élnyi széles sáv” jelenti ezen csúcsokat. Ezen kívül a gráf átmérője is fontos szerephez jut, ugyanis távol levő csúcsok esetén a Dijkstrák könnyedén elérik a gráf „széleit”, így ezen irányokban megszűnik ez a sáv. Ezzel szemben az A*-ok ellipszis alakban haladnak, így általában csak a gráf „belsőbb területeit” vizsgálják.

A beszúrt csúcsok számát ábrázoló (jobb felső) diagram azért lényeges, mert ez a szám adja meg, hogy összesen hány csúcs került a kupacba, tehát a megvizsgált és az elérő, de nem megvizsgált csúcsok összege.

Az értékcsökkentett csúcsokat megjelenítő (bal alsó) diagram azt mutatja, hogy az algoritmus lefutása során hányszor kellett egy korábban már beállított értéket módosítani. Ez azért lényeges, mert amennyiben ez az érték alacsony, az azt jelenti, hogy az algoritmus már kevés „látogatás” segítségével megtalálja a csúcsok tényleges távolságát.

Az összesített eredményt ábrázoló diagram, és így az algoritmusok műveletigénye is a 3.4.2.1.2-ben látott aránnyal közel megegyező mértékben alakul.

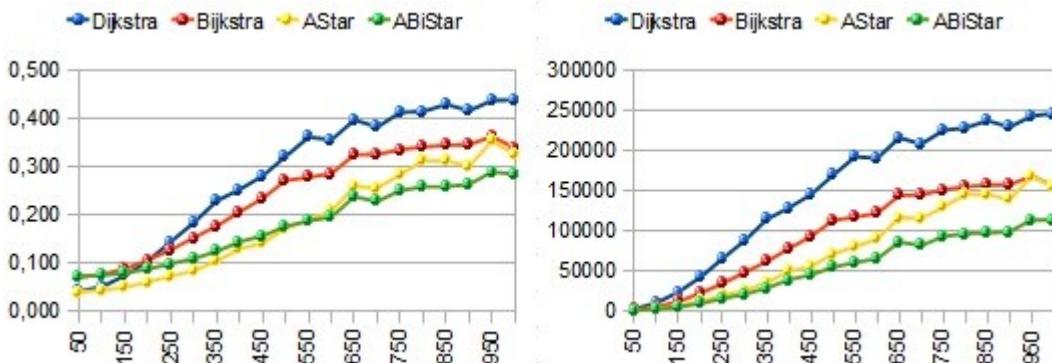
A felsoroltak közül az összesített eredmények részletezését emelném ki a már említett szerepe miatt. Így az egyirányú Dijkstra azonos mérőszámaihoz százalékosan viszonyítva:

- alsó és felső kvartilisek
 - kétirányú Dijkstra: alsó: 38%, felső: 62%
 - egyirányú A*: alsó: 38%, felső: 59%
 - kétirányú A*: alsó: 19%, felső: 34%
- összes és átlagos összesített csúcssqlám
 - kétirányú Dijkstra: összes: 51%, átlagos: 50%
 - egyirányú A*: összes: 54%, átlagos: 49%
 - kétirányú A*: összes: 27%, átlagos: 27%
- szélsőértékek

- kétirányú Dijkstra: legrosszabb: 109%, legjobb: 11%
- egyirányú A*: legrosszabb: 16%, legjobb: 96%
- kétirányú A*: legrosszabb: 91%, legjobb: 5%

3.4.2.2. A New York úthálózatgráf

A New York úthálózatgráf az előző fejezetben szereplő „utazási idők” gráftól csak annyiban tér el, hogy az élek súlya nem a rajtuk való végighaladáshoz szükséges idő, hanem a hosszuk. A használt heurisztika a 3-as heurisztika. Az előző fejezetben szereplő 4-es heurisztika az euklideszi távolságot leosztotta az úthálózatban használható legnagyobb sebességgel. Ezzel szemben a 3-as heurisztika a módosítatlan euklideszi távolságot adja vissza. Ez a becslés érezhetően pontosabb a távolságra nézve, mint a 4-es heurisztika az utazási időkre nézve, így arra számítunk, hogy az A* algoritmusok hatékonyabbak lesznek, mint az előző fejezetben.



3.10. ábra: A futásidők és vizsgált csúcsok számának összehasonlítása a New York úthálózatgráfpon

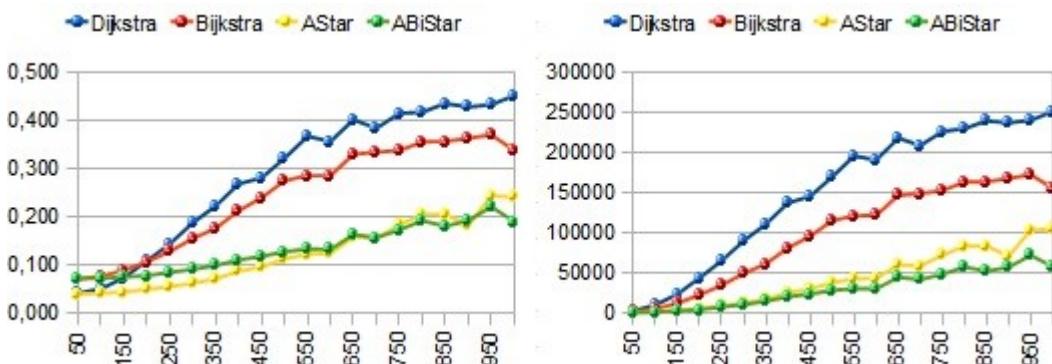
A 3.10. és a 3.6-7. ábrák összehasonlításánál a legszembetűnöbb különbség, hogy megváltozott az A* algoritmusok hatékonysága Dijkstrákhöz, sőt egymáshoz viszonyítva is. Mivel jobb becslést ad a heurisztika, ezért az egyirányú A* sokkal hatékonyabb lesz a Dijkstrákhöz viszonyítva, mint az a 3.4.2.1. fejezetben volt látható. Ezen gráf és heurisztika esetén már stabilan hatékonyabb a kétirányú Dijkstránál. Ugyanígy, a heurisztika javulása miatt az A* kétirányúsítása már kevesebb előnnyel jár, így a kétirányú A* csak kis mértékben hatékonyabb egyirányú pájánál.

Egy különlegesség, hogy noha az 1-es és a 3-as gráfnak mind a csúcs- mind az él-

halmaza megegyezik, és csak az élsúlyokban térnek el egymástól, a megtalált utak elszáma jelentősen (akár 25%-kal is) eltér, pedig a minták alapjául szolgáló útvonalkeresések kiinduló- és végpontjai megegyeznek. Ennek az oka a valós úthálózatok tulajdon-ságaiban keresendő. Ugyanis egymástól nagy távolságra lévő csúcspontok közti élek (pl. autópályák, gyorsforgalmi utak) gyakran sokkal kisebb súlyúak az utazási idő gráfban, mint ha sok kisebb éssel helyettesítenénk őket. Ezen élek nem „vonzóak”, amikor csak távolságokat veszünk figyelembe, azonban utazási idők vizsgálatánál gyakrabban használjuk őket.

3.4.2.3. A „simított” New York gráf

Mint azt a 3.4.1.1-ben említettem, készítettem egy olyan gráfot, melyben a New York úthálózatgráfot módosítottam oly módon, hogy minden él költségét lecseréltem az általa összekötött két csúcs euklideszi távolságára. Ezzel tulajdonképpen „kisimítottam” a gráfot. Mivel heurisztikaként a 3-as heurisztikát használom az algoritmusokban, ez az eddigieknel sokkal jobb becslést ad (szomszédos csúcsok esetén például pontos becslést), hiszen tulajdonképpen a heurisztikához igazítottam a gráfot. Így be tudom mutatni, hogy milyen hatása van egy viszonylag jó heurisztikának (bár még közel sem optimális, lásd 3.4.3.1.).



3.11. ábra: A futásidők és a vizsgált csúcsok számának összehasonlítása a „simított” New York gráf esetén

Mint az a 3.11. ábrán is látható, a jobb heurisztika miatt az egyirányú A* a 3.4.2.2-ben látottnál is hatékonyabb lett, míg a kétirányú A* már csak alig tud hatékonnyabb lenni egyirányú társánál. Sőt, egészen nagy távolságok kivételével nagyobb futás-

időt eredményez. Természetesen a vizsgált csúcsok számának tekintetében még mindig ez a leghatékonyabb a négy algoritmus közül.

A már a 3.6-3.10. diagramok vége felé látható hirtelen időbeli és vizsgált csúcs-számbeli csökkenés azzal magyarázható, hogy nagyon távol levő, a gráf átmérőjéhez hasonló távolságú csúcsok közti útkeresés esetén az algoritmusok előrehaladásuk során elérik a gráf „széleit”, ahol már nem dolgoznak fel több (esetleg felesleges) csúcsot. Ez különösen a Dijkstrák szempontjából fontos, ugyanis csökken a „körkörös” haladásuk-ból fakadó hátrány. Speciális esetben, ha a gráf nagyon keskeny (pl. egy alagutat vagy egy hidat ábrázol), és a célcíccsal ellentétes irányba is „vége” van a gráfnak, akkor a Dijkstra algoritmusok az A*-okhoz hasonló, vagy akár azzal megegyező hatékonyságúak lesznek. (Sőt, a pluszköltségek miatt futásidő tekintetében akár hatékonyabbak is.)

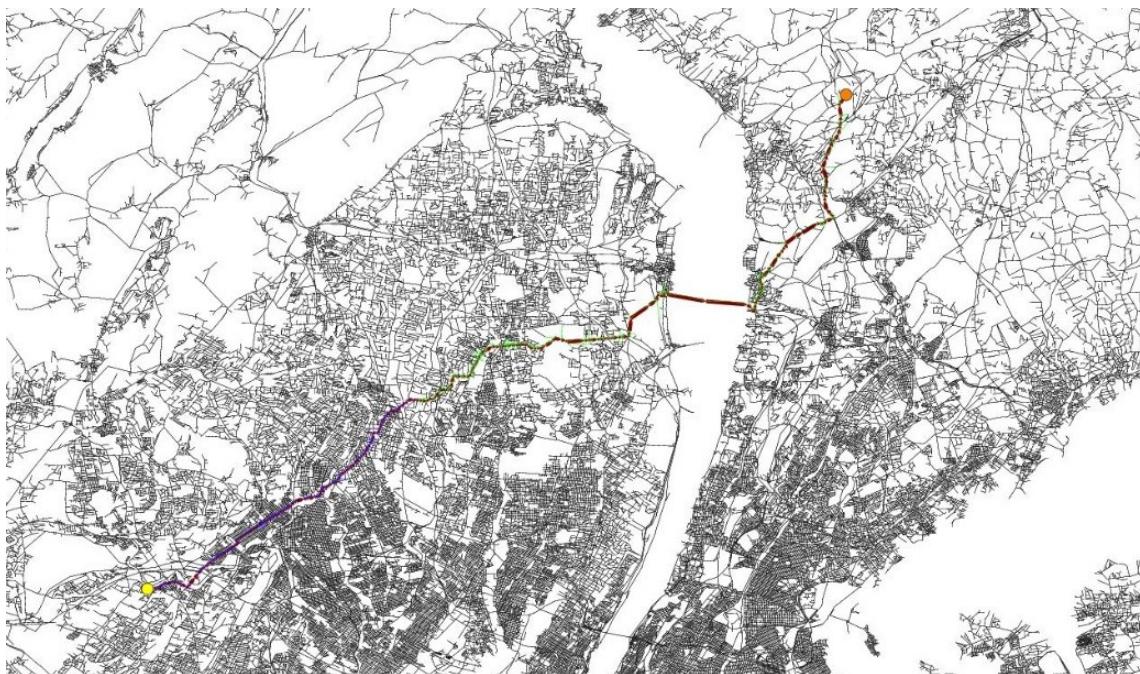
3.4.3. Speciális tesztek

Ezen fejezetben olyan tesztesetek szerepelnek, melyek valamilyen módon meg vannak említve a szakdolgozat más részein, itt a már korábban megállapított konklúziókat támasztom alá.

3.4.3.1. Pontos heurisztika

Mint azt már sok helyen hangoztattam, az A* algoritmusok működését nagyon nagy mértékben befolyásolja a heurisztika minősége, tulajdonságai. Az egyik szélsőséges eset, hogy minden 0-t ad, ekkor (eltekintve a pluszköltségektől) az egy- valamint kétirányú Dijkstra működését kapjuk.

Ezzel szemben az ideális heurisztika minden csúcsnál pontos alsó becslést ad. Így az egyirányú A*, valamint a kétirányú A* előrefelé haladó ágában megadja az adott csúcs célcícestől való távolságát, míg a kétirányú A* visszafelé haladó ágában a forráscsúcestől az adott csúcs távolságát adja meg. Fontos kiemelni, hogy nem az adott csúcestől a forráscsúcs távolságát, hiszen figyelni kell rá, hogy valójában az előrefelé haladó legrövidebb utat keressük, így bizonyos heurisztikák esetén ez a két kifejezés eltérő eredménnyel szolgálhat (de pl. az euklideszi távolság esetén nem).



3.12. ábra: Kétirányú A* ideális heurisztika esetén

Mint az a 3.12-es ábrán is látható, egy ilyen heurisztika használatával az algoritmus kizárolag a legrövidebb utakon (azaz minden olyan csúcson, ami szerepel bármely, az s-ből t-be vezető legrövidebb út egyikén) halad végig, csak ennek csúcsait vizsgálja meg. (Természetesen az úttól egy élnyi távolságban lévő összes csúcsot ellenőrzi, ám ezeket nem fogja soha megvizsgálni). A példaként bemutatott útvonalkeresés megegyezik a már korábban, például a 3.5. ábrán látottal, de a könnyebb láthatóság kedvéért most a térképnek csak a releváns részét jelenítettem meg.

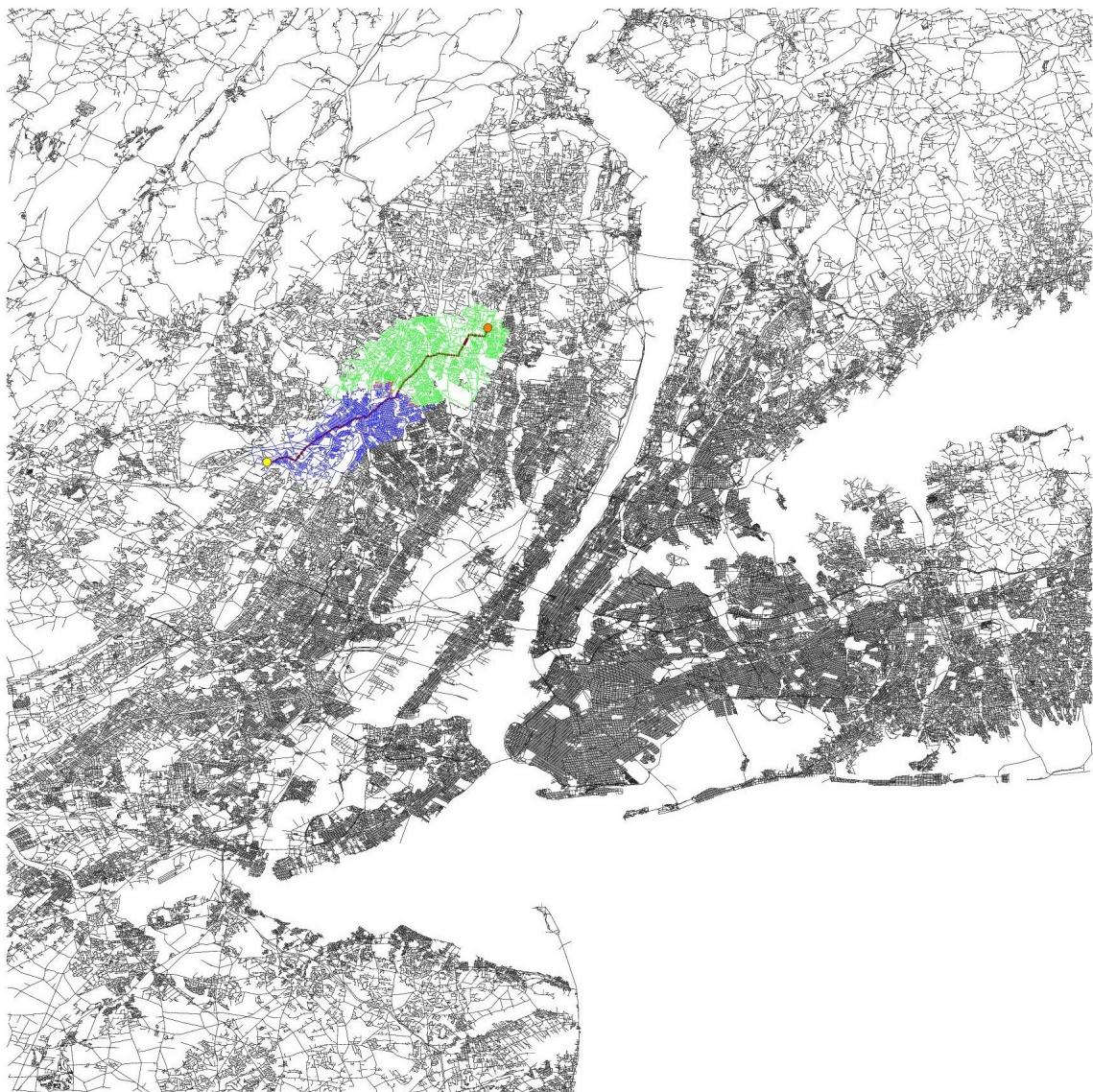
Az egyirányú A* ugyanezt az eredményt adja, természetesen csak egy színnel színezve az éleket.

Ilyen heurisztika használatával nyilvánvalóan nagyon hatékony futást kapunk, mind a futásidő, mind a vizsgált csúcsok tekintetében. Sajnos gyakorlatilag sosem áll rendelkezésre ennyire jó heurisztika, ráadásul több útvonal keresése esetén a tárigény is óriási lenne.

3.4.3.2. Az „utófázis” javításának elhagyása

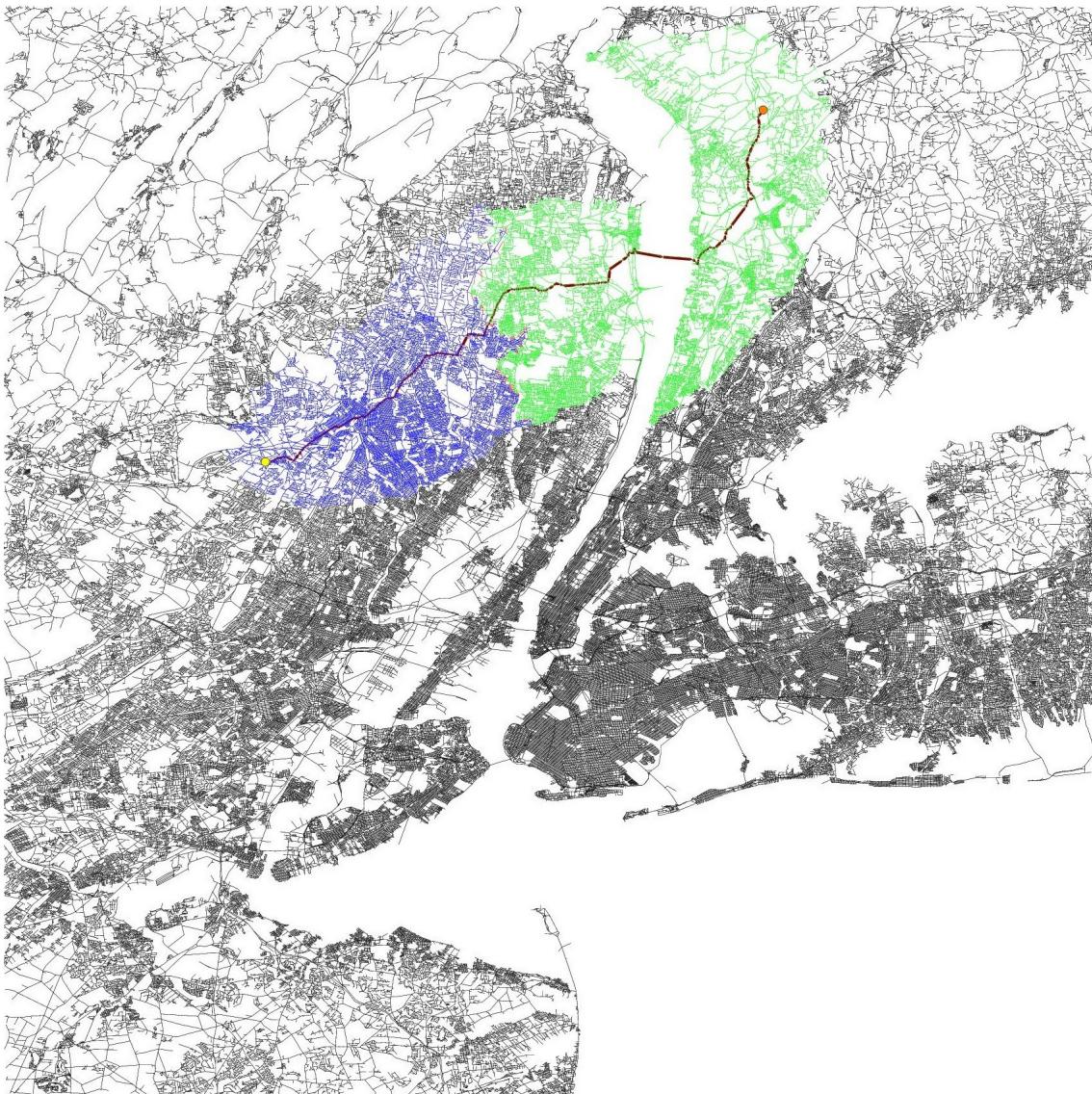
A 3.2.4.2. részben említettem, hogy az „utófázisban” használók egy ([1]-ből származó) javítást, mely a bejárandó csúcsok számát csökkenti. Itt ezen javítás hatását szetrétném szemléltetni. Az „utófázis” azért szerepel idézőjelben, mert az implementáció-

ban nem válik külön az elő- és utófázis, ám ezen rész csak akkor aktív, ha már találtunk legalább egy utat, így szemléletesen utófázisként tekinthető.



3.13. ábra: Két közel eső pont közti útvonalkeresés utófázisbeli javítás nélkül

A 3.13. ábrát összehasonlítva a 2.7. ábrán látottal, jelentős különbség észlelhető a bejárt csúcsok számát tekintve. Ezen két ábra között az egyetlen különbség, hogy a 3.13. ábra által szemléltetett útvonalkeresésnél nem használtam az utófázisbeli javítást. Még szembetűnőbb a javulás, ha az alábbi, 3.14. ábrát hasonlítjuk össze javítást használó keresést ábrázoló párraval, a 3.5. ábrával.



3.14. ábra: Két távol eső pont közti útvonalkeresés útóbázisbeli javítás nélkül

Így mint az a képek alapján is nyilvánvaló, ezen javítás előnyei sokkal nagyobbak, mint az abból fakadó hátrányok, hogy minden vizsgált csúcs esetén plusz ellenőrzést jelent. A konkrét eredményeket nézve közel eső pontok esetében 9%-kal volt lassabb a teljes futásidő, és 82%-kal több csúcsot vizsgált meg. A távol eső pontok esetén a futásidő 28%-kal volt rosszabb, a vizsgált csúcsok száma pedig 48%-kal volt több.

3.4.3.3. Az algoritmus leállítása a két oldali keresés találkozása esetén

A kétirányú A*-gal kapcsolatban külön hangsúlyoztam, hogy a kétirányú A* esetén nem elegendő akkor leállítani a keresést, amikor a két oldal összeér, ekkor ugyanis

még nem ismerjük garantáltan a legrövidebb utat. Viszont azt is megemlítettem, hogy gyakorlati alkalmazás szempontjából érdekes, hogy egy meglepően pontos eredményt kapunk már a keresés ezen pontján is, természetesen – ha nem is nagyon, de – kevesebb idő alatt.

Az itt szereplő adatok a 3-as gráfon, 4-es heurisztika használatával, 2000 véletlenszerűen választott útvonalkeresés eredményeiből származnak. Amennyiben a kétirányú A* algoritmust azon a ponton állítottam le, amikor először „összeért” a két oldal, vagyis a kétirányú Dijkstránál használt megállási feltételet használtam, akkor a helyes algoritmushoz viszonyított futásidő nyereség felső kvartilise 12%, alsó kvartilise pedig 6% volt. Ez persze nem meglepő, hiszen garantáltan kevesebbet dolgozott az algoritmus.

Ami viszont meglepő, az a pontosságban jelentkezik: szintén a helyes algoritmus-hoz viszonyított pontatlanság arányának felső (és így természetesen alsó) kvartilise is 0 lett. Ugyan néhány útvonal jelentősebb mértékben eltért (a maximum 14% volt), de az utak döntő többsége, 82%-a teljesen helyes eredményt szolgáltatott.

Az adatok részletesen (a helyes algoritmus azonos adatainak százalékában):

- **pontatlanság:** felső (és alsó) kvartilis: 0,00%; összesítve: 0,24%; átlagosan: 0,25%, maximálisan: 13,55% (és így talált távolságok 81,95%-a volt helyes eredmény)
- **futásidő-nyereség:** alsó kvartilis: 5,56%; felső kvartilis: 11,84%; összesítve: 10,14%; átlagosan: 9,04%; maximálisan: 32,5%
- **vizsgált csúcsok száma:** alsó kvartilis: 95,12%; felső kvartilis: 97,67%; összesítve: 95,78%; átlagosan: 96,34%; minimálisan: 91,03%

Ugyan csak átlagosan 10%-ot gyorsít a futásidőn, de a 0,24%-os pontatlanság sokszor elegendő lehet. Egy egyszerű példa: amennyiben autóval egy olyan városrészben hajtunk viszonylag nagy sebességgel, ahol nagyon sok kereszteződés van rövid, egyirányú utcákkal, melyek irányítottsága „nem teljesen logikus”. (Ilyen sajnos sokszor előfordul). Ez esetben sokkal fontosabb, hogy a GPS minden irányváltoztatás után a lehető leggyorsabban adjon egy viszonylag jó útvonalat célunkhoz, és így az egyirányú utcákba jó irányból hajtsunk be, mint az, hogy esetleg 0,25%-al hosszabb úton megyünk.

Természetesen a LEMON esetén a pontosság elsődleges, így az ebbe való implementálásnál ezen egyszerűsítést nem alkalmaztuk.

3.4.3.4. Alternálás szabályának változtatása

A 3.3.2.2. részben szerepelt, hogy a kétirányú algoritmusok esetén egy adott ciklusmag lefutásakor az irányválasztás az eddig az egy-egy oldalon végrehajtott műveletek számán, és ezeknek a tárolóként szolgáló kupacbeli műveletigényén alapszik.

Ez elméleti szempontból helyes is (ha nem vesszük figyelembe a kupac típusának megváltoztathatóságát), ám a gyakorlatban egyáltalán nem biztos, hogy ez a leghatékonyabb. A számítógép ugyanis számos információt tárol a rendkívül gyors elérésű cache-ben [16], így például a legutóbb, vagy gyakrabban meghívott számítások eredményeit.

Emiatt érdemes kipróbálni, hogy milyen eredménnyel jár, ha egy-egy oldalt egymás után többször futtatunk.

A kétirányú Dijkstra esetén tesztjeim alapján ezzel a módszerrel akkor érhetjük el a legjobb eredményt, ha nagyból 100-asával váltogatjuk a két oldalt, ám így is alig több mint 0,5% javulást értem el (alsó kvartilis: 95,24%; felső kvartilis: 103,78%; összesített eredmény: 99,34%). Mind kisebb, mind nagyobb szám megadása esetén egyre jobban romlik a hatékonyság. Ez a csekély javulás azonban messze nem ellensúlyozza azt a bizonytalanságot, hogy ez a „bűvös szám” vajon a csúcsok vagy az élek számától, esetleg valami teljesen másról függ-e.

A kétirányú A* esetén 75 körül volt az optimum, ám ez esetben nem kaptam jobb eredményt az eredeti logikánál (alsó kvartilis: 94,19%; felső kvartilis: 105,6%; összesített eredmény: 99,67%).

Mindkét esetben az optimumhoz közel, de azt el nem érő eredményt kaptam, ha egyesével felváltva futtattam a két oldalt. Mivel ezen teszt célja a két oldal váltakozásának optimálissá tétele volt, az időmérésbe nem tartozott bele az oldalválasztást végrehajtó függvény futása, így annak futásideje nem befolyásolta a mérést (de természetesen használat esetén ezt is figyelembe kell venni).

Ennél érdekesebb eredmény született, ha a mérőszámok helyett pusztán a kupacok aktuális mérete szerint döntöttem a soron következő oldalról. Ekkor ugyanis kétirányú Dijkstra esetében 4-5%-os javulást tapasztaltam (alsó kvartilis: 90,11%; felső kvartilis: 90,08%; összesített eredmény: 94,8%), míg a kétirányú A* esetén 7%-os romlást (alsó kvartilis: 97,39%; felső kvartilis: 118,18%; összesített eredmény: 107,3%).

Mivel elméleti szempontból nem tudtam megmagyarázni ezen eredményt, így a kétirányú Dijkstra esetében is az elméleti szempontból is alátámasztható eredeti logikát

hagytam meg.

Külön érdekesség, hogy nem fedeztem fel összefüggést a különböző alternálási szabályok hatékonysága, és a két oldalban eltöltött idő arányában. Arra számítottam, hogy minél inkább ugyanannyi időt „tölts” az algoritmus mind a két oldalon, annál hatékonyabb lesz. De például a Dijkstra esetében a leghatékonyabb megoldás esetén az egyik oldalon 4%-kal töltött több időt, míg a többi szabály használatával csupán 0,5%-kal. Ugyanakkor az A* esetén a legkevésbé hatékony megoldás 7%-kal futott többet az egyik oldalon, szemben a leghatékonyabb megoldás 0,88%-ával. Ugyanakkor az 50-esével váltogató szabály használatával 0,09% volt ugyanezen arány, pedig kis mértékben kevésbé volt hatékony az optimumhoz képest.

3.5. Továbbfejlesztési lehetőségek

Az implementálás valamint a tesztelés során felismertem pár potenciális jövőbeli fejlesztési lehetőséget, ezekről szeretném itt rövid említést tenni.

3.5.1. Az algoritmusok általánosítása, közös felület alá hozása

A szakdolgozatban bemutatott minden a három algoritmus nagy mértékben alapszik az egyirányú Dijkstrán. Ez nem csak működési elvükre igaz, de implementációjukra is. Így ezen 4 algoritmus külön osztályként való szereplése jelentős redundanciát jelent. Ezért érdemes elgondolkozni egy olyan megvalósításon, amely hatékonyságcsökkenés nélkül nyújt egyszerre interfészet minden a 4 algoritmustól, vagy legalább egyet-egyet a két Dijkstrához valamint a két A*-hoz.

Néhány ezen irányba mutató kezdeményezésről már tettem említést, ilyen például, hogy az addTarget() metódus meghívása híján a kétirányú Dijkstra, míg a heurisztika megadásának elmaradása esetén minden a két A* az eredeti, egyirányú Dijkstrát futtatja.

Természetesen a legfőbb szempont a hatékonyság megtartása.

3.5.2. Az irányválasztás funkciót alakítása

Mint azt több helyen említettem, a kétirányú algoritmusok esetén az irányválasztást biztosító függvény a bináris kupac műveletigényein alapszik. Mivel más adatszerkezet is megadható az alap tároló típusának, így szükség lenne ezen metódus megadásának

lehetőségére. Hasonlóan, az adott gráfról való bizonyos információk birtokában előfordulhat, hogy hatékonyabb irányválasztási metodológia is ismert (például ismeretségi hállók adatbázis rendelkezésre állásával, mesterséges intelligencia, stb.), így ilyen esetekben is hasznos lenne ez a lehetőség.

3.5.3. Két külön heurisztika megadása a kétirányú A* esetén

A szakdolgozat során legtöbbször használt heurisztika, az euklideszi távolság nem érzékeny a célcímsúcs választására, de számos olyan heurisztika létezik, amely igen. Így a kétirányú A* esetén gyakran előfordulhat, hogy rendelkezésre áll egy viszonylag hatékony heurisztika az egyik irányú kereséshez, ami a másik irányú keresés esetén nem hatékony, vagy akár nem is használható.

Ilyenkor nagyon hasznos lenne, ha a két oldali kereséshez külön heurisztikapeldány lenne megadható, melyeknek esetlegesen a típusuk is különböző lehet.

3.5.4. A kétirányú A* utófeldolgozásának kiemelése

A 3.4.3.3. részben azt mutattam be, hogy az „utófeldolgozás”-nál használt javítás annak ellenére jelentősen javítja a teljesítményt, hogy így számos olyan csúcs esetén is plusz vizsgálatot jelent, ahol erre nem lenne szükség.

Amennyiben az algoritmus nem csak így szemléletesen, de ténylegesen is szét lenne választva elő- és utófázisra, úgy ezen felesleges pluszköltségek kiiktathatók lennének némi kódredundanciáért cserébe.

3.5.5. Kétirányú algoritmusok többszálúvá tétele

Mind a kétirányú Dijkstra, mind pedig a kétirányú A* esetében az algoritmus túlnyomó részében a két oldal egymástól teljesen függetlenül dolgozik. Emiatt nagy hatékonyágbeli növekedést lehetne elérni, ha a két oldal külön szalon futna. Ennek megvalósításban a „szokásos” szálkezeléssel kapcsolatos problémákon felül az is nehézséget jelenthet, hogy minden algoritmus esetén minden a két oldal használ bizonyos adatokat a másik oldalról. Ezen adathasználat azonban a két oldal összeérése előtt csak egy-egy csúcs állapotára korlátozódik, és ezen szakasz teszi ki a futás nagyobbik részét.

A megvalósíthatóságot az biztosítja, hogy minden algoritmus helyes eredményt

ad a két oldal találkozásának helyétől függetlenül, így nem gond, ha a többszálúsítás következtében eltolódik ez a „találkozási pont”. Legfeljebb annyit eredményezhet, hogy a szükségesnél valamivel több csúcsot vizsgálnak meg, és az optimálishoz képest egy kicsit nagyobb lesz a futásidejük, ám ez így is sokkal alacsonyabb lesz, mint ha csak egy szalon futna mind a két oldal (ez természetesen csak több processzor vagy processzormag esetén igaz).

A hozzávetőlegesen feleződő futásidő mellett járulékos előnye lenne ezen megoldásnak az is, hogy így a két oldal közti váltás 3.5.2-ben leírt problémája irrelevánssá válna, hiszen nem is kellene váltani a két oldal között. Ez persze a váltás költségének megszűnését is jelentené.

4. Irodalomjegyzék

- [1] Wim Pijls, Henk Post: Bidirectional A*: Comparing balanced and symmetric heuristic methods
Econometric Institute Report EI 2006-41
RePub: Erasmus University Institutional Repository,
<http://repub.eur.nl/res/pub/8035>, 2011
- [2] LEMON Homepage, <http://lemon.cs.elte.hu>, 2011
- [3] EGRES Homepage, <http://www.cs.elte.hu/egres/>, 2011
- [4] Edsger W. Dijkstra, http://en.wikipedia.org/wiki/Edsger_Dijkstra, 2011
- [5] Dijkstra's algorithm, http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, 2011
- [6] A* search algorithm, http://en.wikipedia.org/wiki/A*_search_algorithm, 2011
- [7] Installing LEMON,
<http://lemon.cs.elte.hu/trac/lemon/wiki/InstallAutotool#InstalltheLatestDevelopmentVersion>, 2011
- [8] LEMON Tutorial: Graph Structures,
<http://lemon.cs.elte.hu/pub/tutorial/a00014.html>, 2011
- [9] LEMON: Graph Class Reference,
<http://lemon.cs.elte.hu/pub/doc/1.1.1/a00127.html>, 2011
- [10] LEMON: Digraph Class Reference,
<http://lemon.cs.elte.hu/pub/doc/1.1.1/a00083.html>, 2011
- [11] LEMON Tutorial: Maps,
<http://lemon.cs.elte.hu/pub/tutorial/a00019.html>, 2011
- [12] LEMON Tutorial: Graph Adaptors,
<http://lemon.cs.elte.hu/pub/tutorial/a00008.html>, 2011
- [13] LEMON Tutorial: Algorithms,
<http://lemon.cs.elte.hu/pub/tutorial/a00009.html>, 2011
- [14] 9th DIMACS Implementation Challenge - Shortest Paths,
<http://www.dis.uniroma1.it/~challenge9/download.shtml>, 2011
- [15] LEMON Graph Format (LGF),

- <http://lemon.cs.elte.hu/pub/doc/1.0.6/a00002.html>, 2011
- [16] CPU cache, http://en.wikipedia.org/wiki/CPU_cache, 2011
- [17] LEMON Contribution Repository,
<http://lime.cs.elte.hu/~kpeter/hg/hgwebdir.cgi/lemon-astar/>, 2011
- [18] Andrew V. Goldberg:
Point-to-Point Shortest Path Algorithms with Preprocessing,
<http://www.avglab.com/andrew/pub/sofsem07.pdf>, 2011
- [19] Daniel Delling, Peter Sanders, Dominik Schultes, Dorothea Wagner:
Engineering Route Planning Algorithms
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.164.8916&rep=rep1&type=pdf>, 2011