DIPLOMAMUNKA

Czok Tibor Ádám

Debrecen

2010

Debreceni Egyetem

Informatika Kar

KÉTIRÁNYÚ GRÁFKERESŐ ALGORITMUSOK

Témavezető:
Dr. Nagy Benedek
Egyetemi docens

Készítette:
Czok Tibor Ádám
Programtervező matematikus

Debrecen
2010

Tartalomjegyzék

1. Bevezetés 4
2. Alapismeretek 8
2.1 Állapottér-reprezentáció 8
2.2 Keresési stratégiák 9
2.2.1 Nem informált keresők 9
2.2.1.1 Szélességi keresés (Breadth-first search) 9
2.2.1.2 Mélységi keresés (Depth-first search) 10
2.2.1.3 Mélység korlátozott keresés (Depth-limited search) 11
2.2.1.4 Iteratívan mélyülő keresés (Iterative deepening search) 11
2.2.2 Informált keresők 12
2.2.2.1 Legjobbat-először (Best-first) keresők 12
2.2.2.2 Mohó keresés (Greedy search) 12
2.2.2.3 A* keresés 13
2.2.2.4 Iteratívan mélyülő A* keresés (Iterative deepening A*) 13
3. A kétirányú kereső algoritmusok 14
3.1. A kezdetek, avagy BHPA, Front-to-End 15
3.2 BHFFA (Bi-directional Heuristic Front-to-Front Algorithm) 17
3.2.1 Néhány tétel a BHFFA-val kapcsolatban 20
3.2.2 Legrosszabb eset analízis 23
3.3. BIDA* : egy továbbfejlesztett kerület-kereső algoritmus 25
3.3.1. Kerület-kereső algoritmusok 26
3.3.2 A BIDA* algoritmus 28
3.3.3. A BIDA* algoritmus elemzése 31
4. Kísérleti eredmények 35
5. Konklúzió 39
6. Implementáció 42
7. Összefoglalás 46
8. Irodalomjegyzék 47

1. Bevezetés

A mesterséges intelligenciában jártasak, vagy azok, akik eme tudományterülettel a tanulmányaik során éppen csak megismerkedtek, nagy valószínűséggel hallottak már a gráfkereső algoritmusokról, illetve akár alkalmazták is őket. Ha esetleg a témában laikusnak szeretnénk a teljesség igénye és a hivatalos elnevezések nélkül az egyszerűség kedvéért felületesen felvázolni, mit is takarhat ez a fogalom, talán a következőkkel kezdenénk. Adva van egy probléma, tekintsünk erre, mint egy logikai feladatra, amelyet meg kell oldanunk, illetve a számítógépnek meg kell oldania. A feladat meghatároz egy kezdeti állapotot, helyzetet, melyből kiindulva kell nekünk megoldani az adott feladatot. A feladat meghatározza a célt is, olyan helyzete(ke)t, amit el kell érni a feladat megoldása során, és ha elérjük, akkor a feladat meg van oldva. A megoldás maga pedig a kiinduló helyzetből a célig való eljutás. Ez az eljutás pedig valamilyen cselekvések, műveletek, lépések által történhet, melyeket szintén a feladat határoz meg. Nézzünk esetleg egy konkrét példát erre, legyen az a feladat, hogy egy sakktáblán az A1-es pozícióból el kell jutni a H8-as pozícióba egy huszárral, a sakk szabályai szerint, azaz lólépésben. Ez a feladat megadta a kiinduló helyzetet, az A1-es pozíciót, a célt is, ami a H8-as pozíció, és a lehetséges műveleteket is meghatározta, tehát lólépésben lehet haladni a táblán. Azoknak a lelki szemei előtt, akik már foglalkoztak a gráfkereső algoritmusokkal, valószínűleg úgy futna le a fejben megírt algoritmusok erre a feladatra, hogy a kezdőállapotból kiindulva egy keresés végig futna a gráfon, melyet az egyes lépések után kapott újabb állapotok feszítenek ki, mindaddig, amíg a célállapotba el nem jutnak. Valóban ez a leggyakoribb, illetve legismertebb hozzáállás, de gondoljunk csak bele, mi történne, ha nem csak egy keresést alkalmaznánk, hanem a célállapotból (mivel azt is ugyan olyan jól ismerjük, mint a kezdőállapotot) is indítanánk ugyanakkor egy másik keresést. Az előző konkrét példára visszatérve ez úgy nézne ki, hogy amíg mi keressük a megfelelő mezőket, ahová léphetnénk a huszárral, hogy eljussunk az A1 mezőről a H8 mezőre, addig a H8 mezőről indulva is lépked egy másik személy egy másik huszárral, hogy eljusson az A1 mezőre. Ha esetleg találkozik a két huszár egy mezőn, az azt jelenti, hogy el lehet jutni az A1-ből a H8-ba, mivel ha az A1-ből induló huszár azon az úton halad tovább, amelyiken a másik a találkozási ponthoz eljutott, akkor a célmezőre fog jutni, azaz a feladat megoldható. Sőt, nem fontos, hogy egyszerre találkozzanak, az is elég, ha esetleg az egyik

arra a mezőre lép, amelyiken a másik már járt. Ha belegondolunk, így nagy valószínűséggel hamarabb oldjuk meg ezt a feladatot, mintha csak egyedül próbálnánk végig lépkedni a táblán, úgy, hogy az összes lépést mi gondoljuk ki. A számítógép esetében ez a módszer együtt járhat időmegtakarítással és memória-megtakarítással is. Ezen elgondolás alapján működnek a *kétirányú gráfkereső algoritmusok*. Természetesen akkor is számba lehet venni kétirányú keresőket, amikor nem csak azt vizsgáljuk, hogy létezik-e megoldás, hanem amennyiben létezik, úgy a legjobb, azaz optimális megoldást keressük. A példánkban ezt jelentheti a legkevesebb lépésből álló megoldás. Ha esetleg felmerülne az a kérdés, hogyan is tekinthetünk a számítógép szekvenciális működése miatt a két keresésre úgy, hogy azok egyszerre hajtódnak végre, gondoljunk csak az elosztott rendszerekre. A másik jellemző, hogy az előző konkrét példában a két sakkozó kommunikációja talán nem sokat segíthet a megoldás megtalálásában, a számítógépnek annál többet jelenthet bizonyos esetekben, ha a két keresés között valamiféle adatáramlás, kommunikációs kapocs áll fent, és akár egymást módosíthatják is.

Az ok, hogy a legtöbben mégsem veszik a kétirányú keresők adta lehetőségeket számításba egy alkalmas problémánál, az az lehet, hogy akár tanulmányaik során az oktatóik esetleg csak megemlítették egy mondatban, hogy ilyenfajta kereső algoritmus is létezik, de tovább nem részletezték, valamint, aki a magyar nyelvű szakirodalmat tanulmányozza, az az egyirányú gráfkeresők hosszas és részletes tárgyalása mellett is csak egy pár soros, bevezető jellegű leírást találhat a kétirányú kereső algoritmusokról. Talán ez még annak a jelenségnek tudható be, hogy a kétirányú gráfkereső algoritmusok sokkal később kerültek igazán előtérbe, mint az egyirányú társaik, pedig a két irányzat kutatása kvázi egyszerre kezdődött meg. Ez a következőképpen történhetett meg. Az egyirányú kereső algoritmusok hajnalán felmerült a kétirányú kereső algoritmusok gondolata egy Ira Pohl nevű professzorban, aki több kétirányú kereső algoritmust fejlesztett ki, amelyek kisebb-nagyobb változtatásokban különböznek. Arra a megállapításra jutott, hogy ezek az algoritmusok, habár a nem informált keresők között volt, ami jobban, de a heurisztikus keresők sokkal rosszabbul teljesítettek az egyirányú társaiknál.

Mivel ezt a megállapítást helytállónak és teljes körűen érvényesnek fogadta el mindenki egyöntetűen a témában, valószínűleg ezért hátráltak meg évtizedeken át azok is, akik esetleg kezdetben fantáziát láttak a kétirányú keresők kutatásában. Közel negyed évszázadnak kellett eltelnie ahhoz, hogy ezt a megállapítást újra elővegyék, teljes körűen leellenőrizzék, és hogy

kiderüljön, téves volt. Ezután, az elmúlt bő évtizedben robbanásszerűen megnőtt a kétirányú kereső algoritmusokkal foglalkozó kutatások száma, és az újabb és újabb fejlesztések, és eredmények napjainkban sem hagytak alább. Hogy pontosan hogy is szólt ez a téves megállapítás, és hogy napjainkban hol tart ez a szakterület? A dolgozat későbbi részében meglátjuk.

Mivel a mesterséges intelligencia témaköre már az egyetemi éveim előtt is foglalkoztatott, az ilyen témájú cikkek, filmek, könyvek mindig sok érdekességet nyújtottak számomra, persze akkor még nem rendelkeztem komolyabb ismeretekkel a témakörből, legfeljebb annyi tudással bírtam, mint amilyennel egy átlagos, műszaki tudományok iránt érdeklődő gimnazista. Az egyetemi éveim alatt nyertem betekintést az egyes tantárgyak által a mesterséges intelligencia tudományág egyes területeibe, köztük a gráfkereső algoritmusok osztályába. Ez a témakör igen megtetszett, érdekes volt számomra, hogyan lehet szokványos logikai fejtőröket, vagy akár egy olyan mindennapi problémát, amelyet bizonyos műveletek sorozatos végrehajtásával lehet megoldani, olyan alakra formálni a számítógép számára, hogy azt meg tudja oldani, és sokkal gyorsabban oldja meg, mint ahogy én tudnám vagy bárki más a világon.

A diplomamunkám témájának megválasztásakor arra törekedtem, hogy olyan témakört válasszak, ami érdekes számomra, és amiben látok fantáziát. Ezért esett a választásom természetesen erre a témakörre. Az, hogy ezen belül miért pont a kétirányú gráfkereső algoritmusokra (egyszerűség kedvéért hívhatjuk a továbbiakban kétirányú kereső algoritmusoknak is) esett a választásom, az annak tudható be, hogy az elgondolás, és a benne rejlő esetleges további lehetőségek megragadták a fantáziám. Ugyanakkor, a szembesülés azzal, hogy hosszú ideig mellőzve volt ez a terület a mesterséges intelligencia tudományában, és, hogy manapság jelentős eredményeket tudnak felmutatni, újabb bíztatást adott, hogy ezt válasszam diplomamunkám témájának. Mindemellett az, hogy nem találtam magyar szakirodalmat, amely olyan részletesen említené meg ezt a témakört, amennyire ebben a dolgozatban fogom tenni, először egy teljesen egyedi dolog létrehozásának az érzésével töltött el, később ez már hátránynak tűnt. Ezért kérem az olvasót, hogy ezt a későbbiekben vegye figyelembe.

A dolgozatom célja, hogy egy átfogó képet adjon a kétirányú kereső algoritmusok főbb reprezentánsairól, megtárgyalva működésüket, hatékonyságukat, és azok

összehasonlítását. Mivel mára már rengeteg, akár kisebb-nagyobb mértékben egymáshoz hasonló, illetve teljesen különböző algoritmus látott napvilágot ebben a témakörben, ezért nem is törekszem mindegyiknek bemutatására, hanem helyettük a régebben publikált, a témában már klasszikusnak számító, és a manapság folyó kutatásokhoz alapul szolgáló algoritmusokat említem meg. Az algoritmusok jellemzőinek megértéséhez az elméleti hátterükből kifolyólag szükséges, hogy az olvasó tisztában legyen az állapottér-reprezentáció fogalmával és a népszerűbb egyirányú keresők működésével. Ezért a következő fejezetben végigvesszük ezeket az alapfogalmakat, illetve a későbbiek megértéséhez szükséges algoritmusokat, így jutván el fokozatosan a kétirányú keresőkig, így reményeim szerint egy laikus olvasónak is végig érthető és következetes lesz a dolgozat. A mellékelt alkalmazás is ezen analógián alapul. Egy általános programcsomagot akartam létrehozni, amely az egyes egyirányú illetve a tárgyalt kétirányú kereső algoritmusokat implementálja, egymástól függetlenül használhatóan, de az egyes algoritmusok közötti hasonlóságot, kapcsolatokat tükrözve a program struktúrájában. A programban tehát ugyanúgy fellelhetők a kétirányú keresőkhöz szükséges alapok, mint a dolgozatban, az alkalmazás nem probléma specifikus, valamint további modulokkal bővíthető.

2. Alapismeretek

2.1 Állapottér-reprezentáció

Adva van egy *p* probléma. Ha a *p* probléma állapottér-reprezentációját el akarjuk készíteni, akkor meg kell határozni, melyek azok a *p* világára jellemző tulajdonságok, amelyek nekünk fontosak a probléma megoldásához. Ezek a jellemzők együttesen adják meg a világ egy állapotát. Ha *n* különböző jellemzője van a világnak, amelyeket $h_1,\ldots, h_n$ értékek jellemeznek, akkor a p világa a $(h_1,\ldots, h_n)$ érték *n*-essel azonosítható állapotban van.
H_i jelentse azt a halmazt, amely azon értékek halmaza, melyeket felvehet az i. jellemző. A *p* állapotai a $H_1 \times \ldots \times H_n$ halmaz elemei. Ezeknek az állapotoknak a halmazát *A*-val jelöljük, és állapottérnek nevezzük. Előfordulhat, hogy egyes jellemzőkhöz tartozó bizonyos értékek egyszerre nem lehetnek jelen, ezért a fenti Descartes-szorzatnak csak egy részhalmaza az *A* halmaz.
Beszélünk kényszerfeltételről is, mely feltétel egy valódi állapotra mindig igazat ad.

$$A = \{ a \mid a \in H_1 \times \ldots \times H_n \, kényszerfeltétel(a) \}$$

Azt az állapotot, melyben a probléma kezdetekor vagyunk kezdőállapotnak nevezzük. Ez is ugyanolyan érték *n*-es, *k*-val jelöljük és az állapotok halmazának eleme.
A célállapotok halmazának elemei azok az állapotok, ahova el akarunk jutni a feladat megoldása során. Ezeket megadhatjuk explicit felsorolással, illetve célfeltétellel határozzuk meg, mely állapotok felelnek meg ennek a követelménynek. Egy kereső célja lehet az egyik vagy akár mindegyik célállapot elérése. Célállapotok halmazát *C*-vel, egy elemét *c*-vel jelöljük.
Ahhoz, hogy az egyik állapotból egy másik állapotba tudjunk jutni, ami feltétlenül fontos a célállapot eléréséhez, szükségünk van természetesen állapotokat megváltoztató függvényekre. Az ilyen állapotból állapotba leképező függvényeket hívjuk operátoroknak. Halmazukat *O*-val jelöljük. Ezeknek a függvényeknek meg kell adnunk egy értelmezési tartományt, mivel általában egy feladat meghatározza milyen állapotokra lehet alkalmazni milyen operátort. Egy

előfeltétel segítségével ez megoldható, mivel ha teljesül az előfeltétel, akkor az adott operátor alkalmazható az adott állapotra.
Ha megadtuk a p= <A,k,C,O> négyest, akkor állapottér-reprezentációval leírtuk a p problémát, ahol

A: a problémához tartozó állapottér

k: a kezdő állapot

C: a célállapotok halmaza

O: az operátorok halmaza, a hozzájuk tartozó előfeltételekkel [1] [2]

2.2 Keresési stratégiák

A továbbiakban néhány alap kereső stratégiával ismerkedünk meg, inkább csak bevezető jelleggel, a felsorolás és a jellemzés nem részletes.

2.2.1 Nem informált keresők

A keresők között megkülönböztetünk informált és nem informált keresőket. Ha a kereső csak azt tudja el dönteni, hogy célállapot-e az aktuális állapot, akkor nem informált keresőről beszélünk. Viszont ha további információkkal bír egy-egy állapot a kereső számára, akkor informált keresőről beszélünk. Most néhány informált kereső algoritmus következik a teljesség igénye nélkül. [1]

2.2.1.1 Szélességi keresés (Breadth-first search)

Ez egy egyszerű keresés, amely először a gyökércsomópontot fejti ki, majd következő lépésben a gyökércsomópontból generált összes csomópontot fejti ki sorra, és majd azok leszármazottjait stb. Ez a keresési stratégia addig nem terjeszt ki egyetlen d+1 szinten lévő csomópontot sem, amíg ki nem terjesztette az összes olyan csomópontot, amelyik a d szinten

van. A szélességi keresés egy nagyon szisztematikus stratégia, mert először az összes 1 egység hosszú utat tekinti, majd a 2 egység hosszúakat stb. Amennyiben létezik megoldás, a szélességi keresés garantáltan megtalálja, illetve több lehetséges megoldás esetén a szélességi keresés mindig a legsekélyebben fekvő megoldást fogja megtalálni. A szélességi keresés teljes, és optimális is, ha az útköltség a csomópont mélységének nem csökkenő függvénye.

Nem mindig ezt a stratégiát választják, és hogy ennek okát lássuk, meg kell vizsgálnunk a keresés végrehajtásához szükséges idő és memória mennyiségét. Nézzünk meg egy olyan állapotteret, melynek elágazási tényezője *b*, azaz minden egyes állapot kifejtése után *b* új állapotot kapunk. Ha az adott probléma megoldása *d* hosszúságú, akkor a szélességi kereső által kiterjesztett csomópontok számának felső korlátja:

$$1 + b + b^2 + b^3 + \ldots + b^d.$$

Habár ennél a kiterjesztett csomópontok száma kevesebb is lehet, mivel a *d.* szint feletti szintek bármelyikén megkaphatjuk a megoldást.
Időigénye $O(b^d)$. [1]

2.2.1.2 Mélységi keresés (Depth-first search)

Ez a fajta keresés mindig azoknak a csomópontoknak egyikét választja ki kiterjesztésre, amelyek a fa legmélyebbi szintjén helyezkednek el. Ha egy olyan csomóponthoz jut el a kereső, amely nem cél csomópont és a kiterjesztése után nem születik új csomópont, akkor a kereső visszalép egy szinttel. A mélységi keresésnek kevesebb a tárigénye, mint például a szélességinek, mivel csak egy utat kell tárolnia, amely a gyökértől a levélcsomópontig vezet. Ha egy mélységi keresésnél minden egyes kiterjesztéskor *b* új csomópont születik, és olyan állapottér mellett, mely *m* maximális mélységű, a keresés tárigénye *b*m* lesz.
A mélységi keresés $O(b^m)$ időigénnyel rendelkezik. A mélységi keresés használatát nagy vagy végtelen mélységű keresési fák esetén jobb, ha mellőzzük.[1]

2.2.1.3 Mélység korlátozott keresés (Depth-limited search)

Ez a keresés egy meghatározott korlát segítségével kerüli el azt a hibát, amit a mélységi keresők könnyen elkövethetnek, hogy túlságosan mélyre mennek le a keresési fában. A korlát szabja meg, mekkora mélység után már nem haladhat lejjebb a kereső. Idő- és tárigénye hasonlít a mélységi keresőjéhez, *l* mélységkorlát mellett $O(b^l)$ az időigénye, és $O(bl)$ a tárigénye.[1]

2.2.1.4 Iteratívan mélyülő keresés (Iterative deepening search)

Az előző keresőnél lényeges volt a jó mélység korlátnak a megválasztása. Ezt a feladatot ez a kereső kiküszöböli azzal, hogy 0-tól kezdve folyamatosan növekvő mélységkorláttal futtat mélységkorlátozott keresést, amíg megoldást nem talál. Ez a felfogás a mélységi és szélességi keresés előnyös tulajdonságaival egyszerre bír. Ez a fajta keresés optimális és teljes, ugyanakkor a mélységi keresésnél tapasztalt kevés tárigény jellemzi.
A csomópontokat hasonló sorrendben terjeszti ki, mint a szélességi keresés, de ez az algoritmus egyes állapotokat többször is kiterjeszt. Ebből a szempontból pazarlónak tűnhet ez a fajta keresés, de a legtöbb problémánál ez a pazarlás elenyésző. A kiterjesztések száma *d* mélységig, és *b* elágazási tényezővel:

$$1 + b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d.$$

Olyan problémáknál ajánlatos használni ezt a fajta keresést, ahol a nagy keresési tér a jellemző, valamint nem ismert a megoldás mélysége.
Időigénye: $O(b^d)$. Tárigénye: $O(bd)$. [1]

2.2.2 Informált keresők

Az informált keresők az előzőekben megismert keresőkkel ellentétben, nem csupán szisztematikusan történő állapot előállítások révén próbálják megtalálni a megoldást, hanem valamilyen probléma-specifikus információ, tudás segítségével a nagy keresési teret le tudják szűkíteni. Ezt a plusz információt hívják heurisztikának, és az informált keresőket más néven heurisztikus keresőknek is hívják. A heurisztika, vagy heurisztikus függvény egy adott állapotból a célba jutás költségét becsli meg. A célállapotban ez általában 0. Az informált keresők egy kiértékelő függvény segítségével határozzák meg, mely csomópontokat érdemes választaniuk, és csak azokra koncentrálniuk. Ez a kiértékelés a keresésnek így egy irányt ad.
Egy heurisztika elfogadható, ha minden állapotban kisebb az értéke, mint az adott állapotból a célba jutás költsége.
Egy heurisztika monoton, ha egy állapot heurisztikája legfeljebb a szülőállapotra alkalmazott operátor költségével kisebb a szülőállapot heurisztikájánál. [1]

2.2.2.1 Legjobbat-először (Best-first) keresők

A név onnan származik, hogy ezek a fajta keresők az újonnan generált csomópontok közül a kiértékelő függvény által legjobbra értékeltet választják ki kiterjesztésre. Bár az nem biztos, hogy amit a kiértékelő függvény a legjobbnak ítélt, az valóban a legjobb csomópont. Általában ezek az algoritmusok egymástól leginkább a kiértékelő függvényekben különböznek.[1]

2.2.2.2 Mohó keresés (Greedy search)

Ez az igen egyszerű legjobbat-először kereső a kiértékelő függvény által a célhoz legközelebbinek ítélt csomópontot választja ki. A kiértékelő függvény a már említett heurisztikát használja ehhez.

Ez a kereső nem optimális és nem is teljes. Mohósága azon alapszik, hogy a cél felé mindig a legnagyobb mértékben akar haladni, nem törődve azzal, hogy ez esetleg hosszú távon nem a legjobb. Általában gyorsan megtalálják a megoldást, de nem mindig az optimálist. Legrosszabb esetben a keresőt $O(b^m)$ tárigény és időigény jellemzi. [1]

2.2.2.3 A* keresés

Bár hasonlít ez a keresés a mohó keresőhöz, a kiértékelő függvénye különbözik tőle. Az A* algoritmus a heurisztika mellett felhasználja az eddig megtett út költségét, ahhoz hogy a megfelelő keresési irányt válassza ki. Így próbálja megtalálni a minimális költségű utat a célcsomópontig.
A kiértékelő függvény tehát az alábbi alakú:

$$f(n) = g(n) + h(n),$$

ahol $f(n)$ az n csomóponton áthaladó legkisebb költségű út becsült költsége. A legkisebb $f(n)$ értékű csomópont kerül kiterjesztésre mindig. Az A* algoritmus teljes, de ahhoz, hogy garantáltan optimális megoldást adjon, egy alulbecslő heurisztika szükséges. A jó heurisztika megválasztása nagyon sokat számíthat, mivel az lecsökkentheti az egyébként igen nagy keresési teret, mely legrosszabb esetben $O(b^m)$. [1]

2.2.2.4 Iteratívan mélyülő A* keresés (Iterative deepening A*)

A nem informált keresőknél látható volt, hogy az iteratívan mélyülő algoritmus nagyon jól bánik a memóriával. Ha az ott alkalmazott szemléletet társítjuk az A* algoritmussal, szintén hasonló előnyre tehetünk szert. Itt is ugyanúgy egy-egy iteráció egy mélységi keresést jelent, csak annyi a különbség, hogy itt az f értékeket használja mélységkorlát helyett. Ezáltal az érhető el, hogy egy bizonyos f költségkorlát alatti csomópontok kerüljenek kiterjesztésre. Ha a korláton belül lévő csomópontok közötti keresés

nem járt megoldás megtalálásával, így a következő alkalmas *f* költségkorláttal újabb iteráció indul.

Az IDA* algoritmus teljes és optimális ugyanazon feltételekkel, mint az A* algoritmus, viszont a memóriaigénye lineáris. [1]

3. A kétirányú kereső algoritmusok

Amikor egy probléma egy állapottér-gráffal van reprezentálva, a megoldást egy ilyen problémára egy út jelenti, amely egy adott *s* startcsomóponttól némely *t* célcsomópontokig tart. Egy ilyen megoldásnak a megtalálása ennek a gráfnak az átvizsgálásával kísérelhető meg. Ha ezt az átvizsgálást, keresést egy heurisztika irányítja, heurisztikus keresésnek hívjuk. A problémamegoldáshoz használatos heurisztikus keresőkkel foglalkozó munkák legnagyobb része az egyirányú szemléletekkel foglalkozik, ahol a keresés az *s*-ből indulva tart valamely *t* irányába.

Amikor meg van adva explicit módon egy *t* célcsomópont, és a keresési műveletek reverzibilisek, a kétirányú keresés alkalmazható, amely mind előrefele *s*-ből *t*-be, mind hátrafele *t*-ből *s*-be halad. Igazán szólva, még az se fontos, hogy az operátoroknak legyen inverzük. Csak az szükséges, hogy minden egyes *n* csomópontra képesek legyünk meghatározni egy olyan halmazt, melynek eleme az összes olyan p_i csomópont, amelyhez létezik olyan operátor, ami p_i -ből *n*-be képez le. A visszafele keresés azt jelenti, hogy a *t* célcsomópontból kiindulva folyamatosan generáljuk a szülőcsomópontokat.

A kétirányú keresés azokban az esetekben is jól működik, amikor az inverz élek költsége valamelyik két csomópont között különböző: a hátrafele kereső a lehetséges szülőcsomópontokat vizsgálja a hátrafele irányban, de ekkor számításba veszi az előrefele haladó keresésnek a költségét. Még hivatalosabban, $k_1(m,n)=k_2(n,m)$ az *m*-ből *n*-be vezető egy, optimális út költsége. Így $k_2(m,n)$ az *n*-ből *m*-be vezető egy, optimális út költsége (k_2-*t csak jelölésbeli kényelmességből használjuk*). Az összes kétirányú algoritmus, amellyel ebben a tanulmányban foglalkozunk, tökéletesen működik ezen feltételek alatt, és nem követeli meg, hogy az operátorok reverzibilisek legyenek, vagy hogy egy út költsége ugyanaz legyen bármely irányban.

Példaként tekintsünk egy nem informált keresést alkalmazó esetet. Ha mindkét irányban *b* elágazási tényező a jellemző, akkor egy *d* mélységű megoldást a keresés $O(2b^{d/2}) = O(b^{d/2})$ lépésben találja meg, mivel a keresések csak félútig haladnak. Ha esetleg *b*=10 és *d*=6, akkor 1 111 111 csomópontot generál a szélességi keresés, ezzel szemben a kétirányú keresés 3 mélységnél célt ér, mindkét irányban, úgy, hogy 2222 csomópontot generált. Ez alapján ez az elgondolás nagyon kecsegtetőnek tűnik, de bizonyos kérdéseket, és problémákat is felvet ez a szemléletmód, melyek a későbbiekben kiderülnek. [1] [4]

3.1. A kezdetek, avagy BHPA, Front-to-End

Mivel a BHPA képviseli a dolgozatban a *front-to-end* becsléssel rendelkező algoritmusokat, ezért különösebben nem térek ki a jellemzésükre. A fő különbség ezen algoritmusok és a *front-to-front* becslést alkalmazók között a következő sorokból kiderül. A BHPA rossz teljesítményét nem célom most részletezni, a fontos tulajdonságokat a következőkben megemlítem, majd utána folytatjuk a tárgyalást az érdekesebb, *front-to-front* becslést alkalmazó BHFFA illetve BIDA* algoritmusokkal, melyek közül az előbbi tradicionális, az utóbbi nem tradicionális kétirányú keresőnek számít.

A kétirányú heurisztikus keresők megítélése helytelen volt mióta először publikálták több mint negyed évszázaddal ezelőtt. Egészen sokáig, ez a keresési stratégia nem érte el az elvárt eredményeket, és ennek az okával kapcsolatban nagy félreértés volt. Habár még mindig elterjedt az a hiedelem, hogy a kétirányú heurisztikus keresők attól a problémától szenvednek, hogy a kereső határok, vagy más néven kereső frontok elhaladnak egymás mellett, később látni fogjuk, hogy ez a feltevés téves. A tradicionális és az újabb szemléleteket vesszük górcső alá, annak érdekében, hogy egy átfogó képet kapjunk a témáról. Az újabb megközelítésekkel foglalkozó kísérletekben a tapasztalati eredmények azt mutatják, hogy a kétirányú heurisztikus keresés nagyon hatékonyan végrehajtható, még limitált memóriával is. Ezek az eredmények azt sugallják, hogy a kétirányú heurisztikus keresés bizonyos bonyolult problémák megoldásában jobbnak tűnik a hasonló egyirányú keresésnél. Ez bizonyítékul szolgál egy olyan keresési stratégia használhatóságáról, melyet sokáig elhanyagoltak.

Összességében be fogjuk látni, hogy a kétirányú heurisztikus keresés egy életképes dolog, és joggal vehető bizonyos problémáknál elsődlegesen számításba.

A kétirányú keresésről bebizonyosodott, hogy sokkal hatékonyabb, mint az egyirányú változata, amikor heurisztikus tudás nem áll rendelkezésre. A kétirányú *heurisztikus* keresőnél eredetileg ezzel ellentétes eredményre jutott Pohl 1971-ben. Mivel ez a fajta keresés nem úgy működött, ahogy várták, konszenzus volt azon feltevés tekintetében, hogy a kétirányú heurisztikus keresők attól a problémától szenvednek, hogy a kereső frontok elhaladnak egymás mellett, anélkül, hogy kereszteznék egymást. Ezt a szituációt Pohl képletesen egymást elkerülő rakétákhoz hasonlította. Kimutatták, hogy ebben az esetben a kétirányú keresés kétszer annyi csomópontot terjeszthet ki, mint amennyit egy egyirányú keresés terjesztene ki.

Amíg a Pohl által 1971-ben tervezett BHPA eredeti algoritmusa valóban ilyen rossz teljesítményt mutathatott fel, addig maga a rakéta-hasonlat téves és félrevezető volt. Láthatjuk majd, hogy a kétirányú heurisztikus kereső valójában nem szenved attól a problémától, hogy a kereső frontok elhaladnak egymás mellett. A BHPA teljesítménye sokkal rosszabb, mint ami eredetileg várt volt két nagyban különböző ok miatt:

1. A BHPA kereső frontjai tipikusan *túlmennek egymáson*.

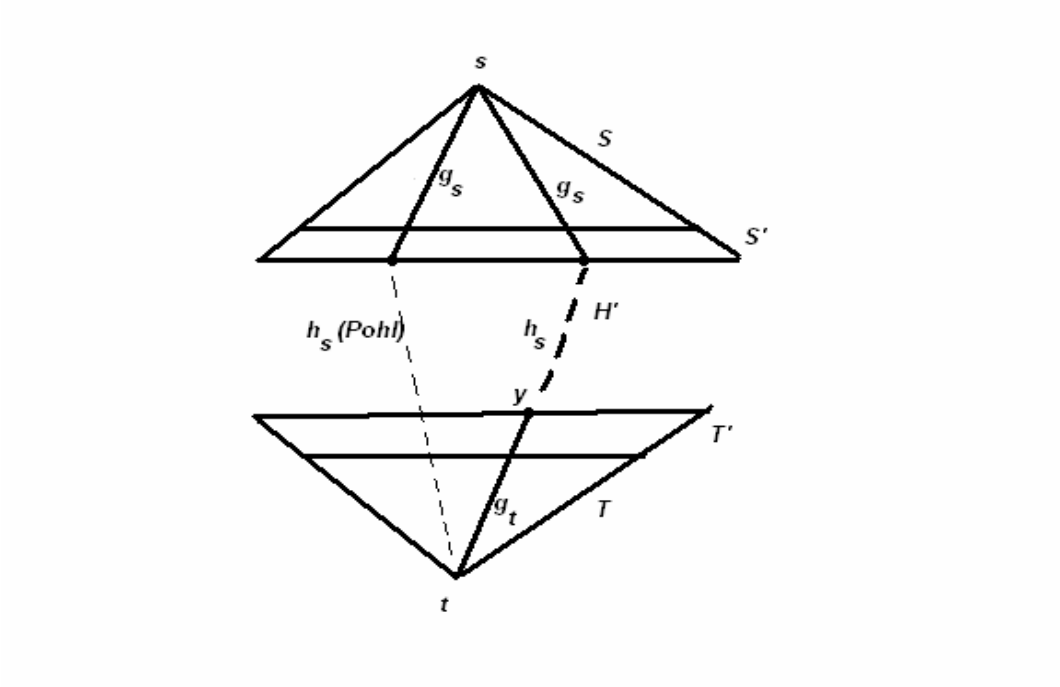
2. A fő erőfeszítést akkor fejti ki, *miután* már a kereső frontok találkoztak: hogy jobb megoldást találjon, egészen az optimálisig, mint amilyet talált a kereső frontok első találkozásánál; és végül, hogy bebizonyítsa, hogy valóban nincs annál jobb megoldás.

Az első ok a BHPA sajátossága, melyet később, más algoritmusokban már sikerült kiküszöbölni. A második, és egyúttal ez a fő hátránya azon mai kétirányú heurisztikus keresőknek is, amelyek *front-to-end* becslést használnak, azaz egy keresési fronton lévő csomópontból a célig vezető út minimális költségét becsülik. Ne feledjük, hogy ugyanezt a fajta kiértékelést alkalmazzák az egyirányú keresők is. Bizonyos újabb fejlesztések, ezzel

ellentétben *front-to-front* becslést alkalmaznak, mely azt jelenti, hogy az aktuális csomóponttól becsüli annak az útnak a minimális költségét, amely a szemközti keresési front egyik csomópontjához vezet. Ezekkel a fogalmakkal jó tisztában lenni, mivel a következőkben ezek alapján csoportosíthatjuk a tárgyalt keresőket. [4]

3.2 BHFFA (Bi-directional Heuristic Front-to-Front Algorithm)

Mielőtt belemennénk az algoritmus precíz jellemzésébe, képzeljük el az 3.2.a ábrán látható szituációt, ahol *S* és *T* a zárt csomópontok halmaza, valamint *S'* és *T'* a nyílt csomópontok halmaza, az egyik és a másik keresési iránynak megfelelően. Éppen úgy döntöttünk, hogy kiterjesztünk egy csomópontot az *S'*-ből. Pohl BHPA algoritmusában az az *S*-beli csomópont lett kiválasztva kiterjesztésre, amelyik a legkisebb értékkel bírt a g_s+h_s függvények összegeként, ahol g_s volt az aktuális minimális költség a kezdőcsúcstól, és h_s volt a csomóponttól a terminális csomópontig lévő távolság becslő függvénye. Az itt használt h_s eltérő, mivel itt ez a $H'+g$ összeg minimumát jelöli minden egyes csomópontra nézve a szemközti *T* fronton, ahol a *g* hasonló az előzőleg ismertetett *g*-hez , csak itt a terminális csomópontra érvényes, valamint *H'* tetszőleges csomópont-párok tagjai között a legrövidebb távolságot becslő függvény. Azonnal világossá válik ennek az algoritmusnak a hátránya a BHPA-hoz képest, mivel az ebben az algoritmusban használt *h* függvénynek a kiszámítása sokkal komplikáltabb, mint a *t* csomóponttól való távolság kiszámítása a BHPA-ban. Másrészt van haszna is ennek az tradicionális, *front-to-front* becslést alkalmazó algoritmusnak, de erről még később lesz szó. [3]



3.2.a ábra

A szaggatott vonallal jelölt távolságot becsüljük egy heurisztikus függvénnyel.

Azért, hogy leírjuk a BHFFA működését, néhány definíciót meg kell adnunk, ahol minél szorosabban követni fogjuk a klasszikus terminológiát.

- *s* a probléma kezdőállapotát tartalmazó csomópont (ezentúl startcsomópont);
- *t* a probléma célállapotát tartalmazó csomópont (ezentúl célcsomópont, vagy terminális csomópont);
- *S* azoknak a csomópontoknak a gyűjteménye, melyeket *s*-ből értünk el és ismert f_s értékkel rendelkeznek;
- *T* azoknak a csomópontoknak a gyűjteménye, melyeket *t*-ből értünk el és ismert f_t értékkel rendelkeznek;
- *S'* azoknak a csomópontoknak a gyűjteménye, melyek nem elemei *S*-nek, de közvetlen leszármazottjai az *S*-beli csomópontoknak;
- *T'* azoknak a csomópontoknak a gyűjteménye, melyek nem elemei *T*-nek, de közvetlen leszármazottjai a *T*-beli csomópontoknak;
- *H(x,y)* a minimum távolság *x* csomópont és *y* csomópont között;
- *H'(x,y)* egy becsült távolság *x* csomópont és *y* csomópont között, ahol *H'(x,y)= H'(y,x)*;
- $g_s(y)$ a minimum távolság *s* és *y* között, ahol $y \in S \cup S'$ és az út$\in S \cup S'$;

– $g_t(y)$ a minimum távolság t és y között, ahol $y \in T \cup T'$ és az út$\in T \cup T'$;

– $h_s(n) = \overset{\min}{y \in T'} (H'(n,y) + g_t(y))$;

– $h_t(n) = \overset{\min}{y \in S'} (H'(n,y) + g_s(y))$;

– $f_s(x) = g_s(x) + h_s(x)$;

– $f_t(x) = g_t(x) + h_t(x)$;

– $\Gamma(x)$ azon csomópontok véges halmaza, amelyek megkaphatók az x-re alkalmazható operátorokkal;

– $\Gamma^1(x)$ azon csomópontok véges halmaza, amelyek megkaphatók az x-re alkalmazható inverz operátorokkal;

– $l(n,x)$ az élhosszúság n és x között.

Most pedig definiáljuk a BHFFA-t:

1. $s \rightarrow S'$, $t \rightarrow T'$, és $f_s(s) := f_t(t) := H'(s,t)$
2. IF $S' \cup T' = 0$ THEN stop megoldás nélkül ,
   ELSE eldöntjük, hogy előre megyünk $\rightarrow$ goto 3 , vagy hátra $\rightarrow$ goto 10
3. Kiválasztjuk $n \in S'$, ha $f_s(n) = \overset{\min}{y \in S'} (f_s(y))$,
   eltávolítjuk n-t S'-ből és S-be rakjuk, legyen $Leszármazottak(n) := \Gamma(n)$
4. IF $n \in T'$ THEN stop megoldással
5. IF $Leszármazottak(n) = 0$ THEN goto 2
6. Legyen $x \in Leszármazottak(n)$ és távolítsuk el $Leszármazottak(n)$-ból.
7. IF $x \in S'$ THEN
   IF $g_s(n)+l(n,x) < g_s(x)$ THEN
   $g_s(x) := g_s(n)+l(n,x)$
   ENDIF;
   IF $g_s(x)+h_s(x) < f_s(x)$ THEN
   $f_s(x) := g_s(x)+h_s(x)$
   ENDIF;
   goto 5
   ENDIF

8. IF $x \in S$ THEN
   IF $g_s(n)+l(n,x)<g_s(x)$ THEN
      $g_s(x) := g_s(n)+l(n,x)$
   ENDIF;
   IF $g_s(x)+h_s(x)<f_s(x)$ THEN
      $f_s(x) := g_s(x)+h_s(x)$;
      távolítsuk el x-t S-ből és rakjuk S'-be;
   ENDIF;
   goto 5
   ENDIF
9. Berakjuk S'-be x-t a hozzátartozó $f_s(x)$ értékkel; goto 5
10. Ugyanúgy végrehajtjuk a 3. lépéstől a 9. lépésig, kicserélve az (s,S,S',Γ) paramétereket (t,T,T',Γ^{-1}) –re.

Az nincs megszabva, hogy mi alapján történjen az a döntés a 2. lépésben, amely meghatározza, hogy előre vagy hátrafele folytassuk a keresést. Pohl kutatásai alapján, a legígéretesebb eljárás, hogy megszámoljuk a csomópontokat S'-ben és T'-ben, és azt a frontot választjuk, amelyikben a legkevesebb van (de legalább egy). [3]

3.2.1 Néhány tétel a BHFFA-val kapcsolatban

Látni fogunk néhány tételt és bizonyítást a BHFFA-val kapcsolatosan, melyek hasonlóak az egyirányú A* algoritmusnál szokásos tételekhez és bizonyításokhoz.

Tétel 3.2.1.1. *Ha $H'(x,y) \leq H(x,y)$ és minden él-címke nem kisebb mint valamilyen pozitív δ, akkor a BHFFA megáll az s és t közötti legrövidebb úttal (feltéve ha van ilyen).*

Bizonyítás.
Ahogy az egyirányú esetben, először egy lemmát bizonyítunk be.

Lemma 3.2.1.2. *Ha $H'(x,y) \leq H(x,y)$, akkor a BHFFA minden iterációjához és minden s-ből t-be vezető P optimális úthoz léteznek $n \in S'$, $m \in T'$ csomópontok a P-n $f_s(n) \leq H(s,t)$ és $f_t(m) \leq H(s,t)$ értékekkel.*

Bizonyítás.

Legyen n az első csomópont a P-n az s-től számítva, úgy hogy $n \in S'$. Legyen m az első csomópont a P-n a t-től számítva, úgy hogy $m \in T'$ (ezek léteznek, mert különben a BHFFA már megállt volna).

$f_s(n) = g_s(n) + h_s(n)$

$= g_s(n) + H'(n,y) + g_t(y)$ ($y \in T'$ -re)

$\leq g_s(n) + H'(n,m) + g_t(m)$ h_s definíciója miatt

$\leq g_s(n) + H(n,m) + g_t(m)$

$= H(s,t)$ mivel egy optimális úton vagyunk.

$f_s(m) \leq H(s,t)$ is hasonló módon bizonyított.

Most tegyük fel, hogy Tétel 3.2.1.1. nem igaz. Ekkor három eset létezik:

1. BHFFA nem áll meg;
2. BHFFA megáll megoldás nélkül;
3. BHFFA megáll a legrövidebb út nélkül.

1. eset: Legyen P egy optimális út s-ből t-be.

A Lemma 3.2.1.2. értelmében mindig létezik egy nyílt n csomópont $S' \cup T'$-ben a P-n, olyan $f_s(n)$ illetve $f_t(n)$ értékkel, ami kisebb, mint $H(s,t)$. Ennek következtében, a kiterjesztett csomópontoknak $H(s,t)$-nél kisebb vagy vele egyenlő f-értékkel kell rendelkezniük. Tehát a g-értékeik kisebbek vagy egyenlők $H(s,t)$-vel. Ily módon BHFFA legfeljebb $H(s,t)/6$ lépésre az s-től illetve t-től terjeszt ki csomópontokat, és ez egy véges szám. Legyen M_s és M_t megfelelően minden olyan csomópontnak a halmaza, amelyeket valaha az s illetve t csomópontokból állítottunk elő. Ahogy minden csomópontnak csak véges számú leszármazottja lehet, és ahogy a lépések maximális száma az s-től vagy t-től is véges számú, mind az M_s és mind az M_t is csak véges számú csomópontot tartalmazhat, és így

$M = M_s \cup M_t$ is véges méretű. Legyen az M-beli csomópontok száma v. Legyen p a (természetesen véges) száma a különböző utaknak s-ből m-be ha $m \in M_s$, és t-ből m-be ha $m \in M_t$, és legyen p' a maximum az összes p közül. Ekkor p' a maximális száma annak, hogy egy csomópontot hány különböző alkalommal lehet újra kiterjeszteni. A BHFFA $p*v$ iterációja után az M összes csomópontja véglegesen zárt lesz. Tehát $S' \cup T' = 0$ és a BHFFA megáll, amely ellentmondást szül.

2. eset: Most bizonyítottuk, hogy a BHFFA végső soron megáll, és ez csak két okból lehetséges: megtalált egy megoldást, illetve $S' \cup T'$ üres. Ha az utóbbi áll fenn, akkor az utolsó kiterjesztett csomópontnak egyáltalán nincsenek leszármazottjai (máskülönben elhelyeztük volna őket az S'-ben vagy a T'-ben). De ez azt jelenti, hogy nem létezik s-ből t-be vezető út, ellentmondva a feltevésnek. Tehát a BHFFA megáll, mivel megtalált egy megoldást.

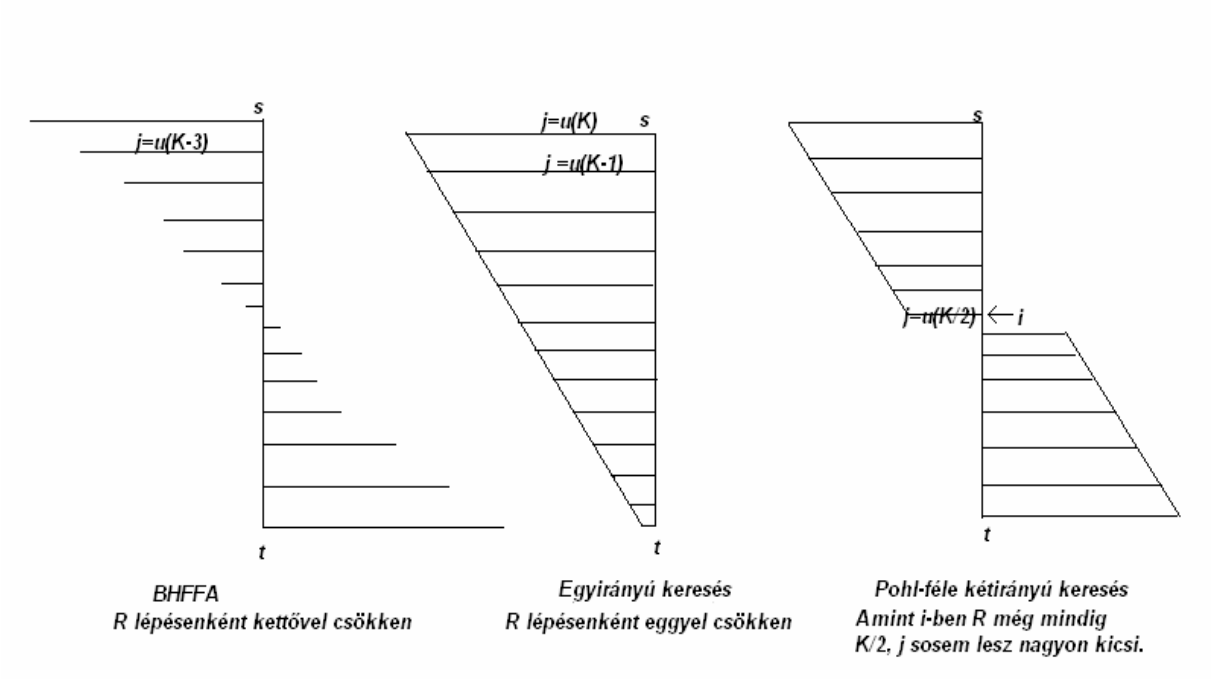
3. eset: Mielőtt éppen befejeznénk az m csomóponttal, lennie kell (Lemma 3.2.1.2. miatt) egy n csomópontnak az S'-ben $f(n) < H(s,t) < f(m)$ értékkel, így az n lesz kiválasztva kiterjesztésre m helyett.

A következő tétel az optimalitás tétele, amely azt fejezi ki, hogyha két heurisztika, H és H^*, olyan viszonyban áll minden n csomópont esetén, hogy $H^*(n,t) < H(n,t)$, és a heurisztikák következetesek, akkor minden egyes csomópont melyet a H kiterjeszt, H^* is kiterjeszt. Ez a tétel nem tartható a BHFFA-ra. Az ok az, hogy ellentétben az egyirányú és Pohl kétirányú algoritmusával szemben, egy nyílt csomópont f-értéke nem statikus a BHFFA-ban, és az, hogyan változik, függ az ellentétes frontokon lévő csomópontok közötti heurisztikus és a valódi távolság pontos formájától, és mivel részletes információkat ezekről nem feltételezhetünk, így nehéz megjósolni bármilyen heurisztika konkrét viselkedését a BHFF-ban. Még nem sikerült találni egyetemes bizonyítékot arra, hogy ha egy heurisztika jobb (a fent említett jelentésben) mint egy másik, akkor az mindig kevesebb iteráció alatt fog végezni. [3]

3.2.2 Legrosszabb eset analízis

A fentebb említett algoritmusok elsőrendű összehasonlítása úgy történhet, hogy megvizsgáljuk a működésüket a legrosszabb esetben. Mintákat készítettek az egyirányú, Pohl féle kétirányú és a BHFFA algoritmusokhoz, feltételezve, hogy a használt heurisztika relatív határok közt egy maximális hibát fog adni. Legyen a keresési tér csomópontok egy megszámlálható halmaza, két olyan csomóponttal, a start és a cél csomóponttal, melyek m (m>1) éllel rendelkeznek, míg minden más csomópontból m+1 él ered, valamint nincsenek körök. Legyen az összes élhosszúság egyenlő 1-el, és legyen egy K hosszúságú út a start és a cél csomópont között. Az egyirányú nézőpontből ez a tér egy fa, m elágazási értékkel, mivel az algoritmus nem fog túl nézni a cél csomóponton. Ebben a térben az alábbi eredmények kaphatóak:

1. Legyen R a valódi távolság a megoldásként kapott út valamely csomópontjától, a cél csomópontig egyirányú esetben, a cél vagy a start csomópontig a Pohl–féle kétirányú esetben, illetve az ellentétes frontig a BHFFA esetében. Ha a megoldás útvonalán lévő minden egyes csomópontot kiterjesztünk a megoldás útvonalától távol valamilyen j mélységig, és ha ez a j egy monoton csökkenő függvénye R-nek, akkor mindegy, hogy néz ki pontosan a heurisztika, $\mathcal{N}_{\text{BHFFA}} < \mathcal{N}_{\text{egyirányú}} < \mathcal{N}_{\text{Pohl-kétirányú}}$, ahol $\mathcal{N}_{\text{A}}$ jelöli az A-algoritmus által kiterjesztett csomópontok számát. Ez könnyen megérthető, mivel magától értetődően látszik ha megnézzük a 3.2.2.a ábrát, ahol a vonalak hosszúsága reprezentálja a mélységét a megoldás útvonalától távol kiterjesztett csomópontoknak. Az összes vonal hosszúságának összege balról jobbra haladva növekszik.



3.2.2.a ábra

2. Tegyük fel, hogy két csomópont a megoldás útvonalán fekszik és ekkor érvényes, hogy $H'(n,m) = H(n,m)*(1+\delta)$ valamely δ>0-ra, illetve ellenkező esetben (legalább az egyik az útvonalon kívül fekszik) $H'(n,m) = H(n,m)/(1+\delta)$. Ekkor δ egy relatív hibakorlát és ez a legrosszabb lehetséges eset. Most a BHFFA, az egyirányú, és a Pohl-féle kétirányú algoritmusnak a megoldásként visszaadott úton kívül eső csomópontjait terjesztjük ki a j=6*R+δ*(δ+1)/(δ+2) mélységig. Így a monoton feltételek érvényesek itt a fentebb bemutatott eredményekkel. Továbbá konkrét képleteket írhatunk fel a különböző algoritmusok által kiterjesztett csomópontok számát illetően ebben a bizonyos legrosszabb esetben:

$$\mathcal{N}_{\text{BHFFA}} = 2 * m^{\delta - c_\delta} * (m^{K\delta} - 1) / (m^{2\delta} - 1);$$

$$\mathcal{N}_{\text{egyirányú}} = m^{\delta - c_\delta} * (m^{K\delta} - 1) / (m^{\delta} - 1);$$

$$\mathcal{N}_{\text{Pohl-kétirányú}} = 2m * m^{(K/2)* \delta - c_\delta} * (m^{(K/2)\delta} - 1) / (m^{\delta} - 1);$$

ahol m az elágazási-tényező, K a megoldás útvonalának hossza, és $c_\delta = \delta*(\delta+1)/(\delta+2)$. [3]

3.3. BIDA* : egy továbbfejlesztett kerület-kereső algoritmus

Egy újabb kétirányú heurisztikus kereső algoritmust vizsgálunk meg, amely a nem tradicionális kétirányú keresők közé tartozik. Az elméleti és a gyakorlati eredmények azt mutatják, hogy a *kerület-kereső algoritmusok* jobban teljesítenek a többi kétirányú kereső algoritmusnál, ezért érdemes komolyabban szemügyre venni az újabb kutatásokat.

Sajnos, minden publikált kétirányú algoritmusnak van néhány hátránya, és az eddigiekben tárgyaltak egyike sem teljesít a legtöbb esetben jobban, mint a jól ismert egyirányú A* és IDA* algoritmusok.

Dillenburg és Nelson bevezette a kétirányú algoritmusok egy új osztályát, melyet kerület-kereső algoritmusok osztályának neveztek el, és az algoritmusokban a hátrafele és az előrefele való keresés szekvenciálisan történik, nem pedig szimultán. Először is, ezek az algoritmusok egy ún. kerületet generálnak, ami a célcsomópontot körülvevő csomópontokból álló halmaz. Azután, egy előrefele haladó keresés hajtódik végre; ez a második keresés akkor fejeződik be, ha a kerületet elérte.

Kerület-kereső algoritmusok általában kevesebb csomópontot generálnak, mint az egyirányú testvéreik, de sokkal több heurisztikus számítást igényelnek. Ennélfogva, ezek az algoritmusok csak akkor hasznosak, ha a heurisztikus függvény számítási költsége alacsony egy új csúcs generálásának költségéhez képest.

A továbbiakban a BIDA* (Bi-directional Iterative Deepening A*) kétirányú kereső algoritmust fogom bemutatni, mely Dillenburg és Nelson munkásságától függetlenül született meg. A kísérleti tesztek, a 15-ös Puzzle problémát használva, azt mutatták, hogy a BIDA* kevesebb csomópontot generál, és kevesebb heurisztikus becslést hajt végre, mint az IDA*. Ennek következtében, a BIDA* még akkor is hasznos lehet, ha a heurisztikus függvény kiértékelésének költsége túlsúlyban van egy új csúcs generálásának költségéhez viszonyítva

A továbbiakban a következő szerkezet szerint történik a tárgyalás: a 3.3.1 pontban megvizsgáljuk a kerület-kereső algoritmusokat, a 3.3.2 pontban a BIDA* algoritmus leírása következik, és végül a 3.3.3. pontban ezt elemezzük. [5]

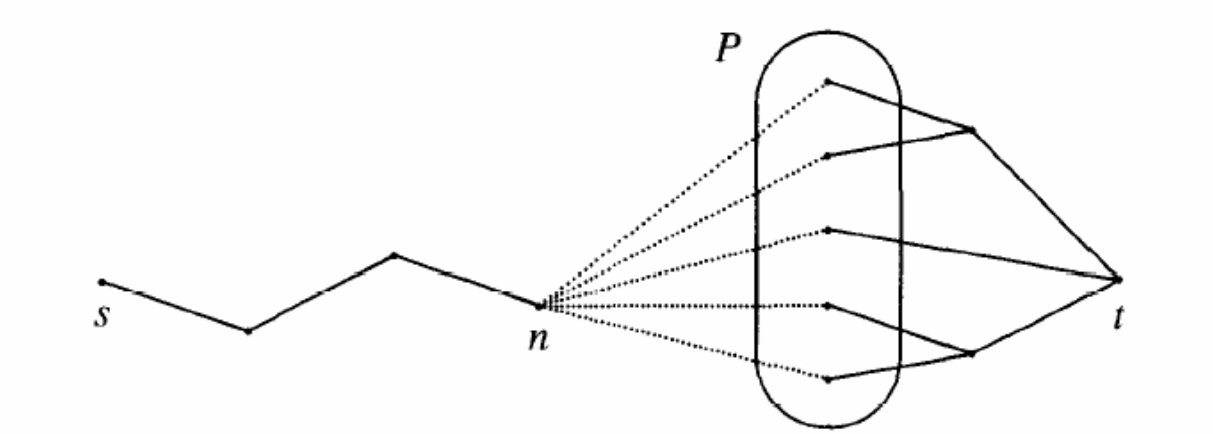
3.3.1. Kerület-kereső algoritmusok

Ebben a szegmensben a kerület-kereső algoritmusok koncepcióját vizsgáljuk meg, és dokumentáljuk a kerület-kereső algoritmusok által használt heurisztikus függvények néhány tulajdonságait.

Bár már a dolgozat egy korábbi pontján már tárgyalva volt, de idézzük fel a teljesség kedvéért az alábbiakat. Kétirányú kereső algoritmusokat használhatunk azokra a problémákra, amikor meg van adva egy kezdőcsomópont *s*, egy célcsomópont *t*, és invertálható operátorok egy véges halmaza. Bár szigorúan véve, nem feltétel, hogy az operátoroknak legyen inverzük. Csak az szükséges, hogy minden egyes *q* csomópontra képesek legyünk egy listát létrehozni az összes olyan p_i csomópontból, amelyhez létezik olyan operátor, amely p_i -ből *q*-ba képez le. Ha létezik egy *n*-ből *n'*-be leképező operátor, azt mondjuk, hogy *n őse n'*-nek, és *n' leszármazottja n*-nek, valamint *c(n,n')* jelöli az operátor költségét. Ha α egy út (ami operátorok egy sorozata) *n*-ből *m*-be, akkor ennek a költségét $C_\alpha(n,m)$ jelöli. Adott két csomópont, *n* és *m*, *H*(n,m)*-el jelöljük az *n*-től *m*-ig tartó út minimális költségét. Feltesszük, hogy rendelkezésre áll egy elfogadható *H* heurisztikus függvény úgy, hogy $H(n,m) \leq H^*(n,m)$ minden egyes *n*,*m* csomópont-párra.

A kerület-kereső algoritmus a következőképpen működik (lásd 3.3.1.a ábra). Először, a célcsomópontból egy hátrafele keresés létrehozza a *P* kerületet. Azután, a keresés a startcsomópontból halad a $\hat{h}_p(n) = \min_{m \in P} [H(n,m)+H^*(m,t)]$ heurisztikus függvényt használva. Megjegyzem, bármilyen típusú heurisztikus kereső algoritmust használhatunk az előrefele kereséshez.

Még hivatalosabban, definiálunk egy A_d halmazt az összes olyan *m* csomópontra, mely $H^*(m,t) \leq d$, ahol $d \geq 0$. Más szóval, $m \in A_d$, ha létezik út *m*-től a célcsomópontig, és az út költsége kisebb vagy egyenlő *d*-vel. Egy P_d halmazt is definiálunk, amely az összes olyan $m \in A_d$ csomópont halmaza, amelynek létezik legalább egy olyan őse, amelyik nincs az A_d-ben. A P_d halmaz tekinthető az A_d frontjának, és a *d* mélység kerületének nevezzük. Megjegyzem, $d = 0$-ra azt kapjuk, hogy $A_0 \equiv P_0 \equiv \{t\}$.



3.3.1.a ábra. A heurisztikus függvény kiszámítása a kerület-kereső algoritmusoknál. A folyamatos vonalakkal jelölt utak költsége ismert, a szaggatott vonalakkal jelölt utak költsége a *H* függvényt használva becsülve vannak.

Definiáljuk $d \geq 0$ -ra a következő heurisztikus h_d függvényt:

ha $n \in A_d$, akkor $h_d(n) = H^*(n,t)$;

ha $n \notin A_d$, akkor $h_d(n) = \min_{m \in P_d} [H(n,m)+H^*(m,t)]$. (1)

A kerület-kereső algoritmusok a h_d heurisztikus függvényt használják, és akkor fejeződnek be, amikor az előrefele keresés eléri a kerületet.

A következőkben feltesszük, hogy a *H* függvény monoton, vagyis, ha *n'* egy leszármazottja *n*-nek, akkor

$$H(n,m) \leq c(n,n') + H(n',m). \quad (2)$$

Feltesszük, hogy ez a tulajdonság érvényes H második argumentumára is; azaz, ha *m'* egy leszármazottja *m*-nek, akkor

$$H(n,m') \leq H(n,m) + c(m,m'). \quad (3)$$

A h_d függvény tulajdonságait a következő lemmák foglalják össze.

Lemma 3.3.1.1. *A h_d függvény egy elfogadható heurisztika a $d \geq 0$ –ra, azaz, $h_d(n) \leq H^*(n,t)$.*

Lemma 3.3.1.2. *A h_d függvény egy monoton heurisztika a $d \geq 0$ –ra, azaz, ha n' egy leszármazottja n-nek, akkor $h_d(n) \leq c(n,n') + h_d(n')$.*

Lemma 3.3.1.3. *Ha $d' \geq d$ és a (3) egyenlőtlenség fennáll, akkor $h_{d'}(n) \geq h_d(n)$.*

Megjegyzem, ha $d' \geq d$, a Lemma 3.3.1.1. és Lemma 3.3.1.3. miatt $h_d(n) \leq h_{d'}(n) \leq H^*(n,t)$; vagyis, $h_{d'}(n)$ mindig egy jobb becslése az ismeretlen $H^*(n,t)$ értéknek. Azonfelül, $d = 0$ esetén $h_0(n) = H(n,t)$, azaz, h_0 az egyirányú kereső algoritmusok által rendszerint használt heurisztikus függvény. Más szóval, a kerület-kereső algoritmusok által használt heurisztika „jobban informált", mint $H(n,t)$.

A kerület-kereső algoritmusoknak van egy fő hátrányuk, az, hogy a h_d függvény kiszámítása nagyon költséges lehet. Ha az (1) képletet használjuk, a h_d egy kiértékelése $|P_d|$ számú kiértékelését igényli a H függvénynek. A következő fejezetben a h_d érték hatékony kiszámításának egy új módszerét ismerhetjük meg. Ez a módszer alkalmazható bármelyik mélységi(depth-first) stratégián alapuló algoritmusnál. Most csak az IDA* algoritmust fogjuk figyelembe venni, az egyszerűsége és a hatékonysága miatt. [5]

3.3.2 A BIDA* algoritmus

Ebben a szegmensben a BIDA* kétirányú algoritmussal foglalkozunk, amely abban áll, hogy IDA* algoritmust hajt végre h_d–t használva heurisztikaként. A BIDA* egy kerület-kereső algoritmus, és ha a H heurisztika elfogadható és monoton, akkor a BIDA* mindig megtalálja a legkisebb költségű utat, ha az egyáltalán létezik.

Mindvégig ebben a fejezetben, az s és a T fogja jelölni a startcsomópontot és az aktuális küszöbértéket(*Threshold*). Jelöljön n egy csomópontot az úton, amelyet jelenleg derít fel a BIDA*$_d$ algoritmus. Ha $n \notin A_d$, a következőt definiáljuk:

$$P_d(n,T) = \{ m \in P_d \mid g(n) + H(n,m) + H^*(m,t) \leq T \}. \quad (4)$$

Legyen $f_d(n) = g(n) + h_d(n)$. Az $n \notin A_d$ –ra fennáll, hogy $f_d(n) > T$ akkor és csak akkor, ha a $P_d(n,T)$ halmaz üres. A következő lemma kimutatja a $P_d(n,T)$ halmaz egy másik fontos tulajdonságát.

Lemma 3.3.2.1. *Jelölje $\langle s = n_1, n_2, n_3, \ldots, n_k \rangle$ a BIDA*$_d$ algoritmus által jelenleg felderített utat. Továbbá*

$$P_d(n_i,T) \subseteq P_d(n_{i-1},T), \qquad i=1,2,\ldots k.$$

Bizonyítás. $m \in P_d$ esetén fenáll, hogy

$$\begin{aligned} & g(n_{i-1}) + H(n_{i-1},m) + H^*(m,t) \\ & \quad = g(n_i) - c(n_i,n_{i-1}) + H(n_{i-1},m) + H^*(m,t), \\ & \quad \leq g(n_i) + H(n_i,m) + H^*(m,t). \end{aligned}$$

Így, $m \in P_d(n_i,T)$ magában foglalja, hogy $m \in P_d(n_{i-1},T)$, és ebből következik a lemma.

A BIDA*$_d$ algoritmus egy utat addig derít fel, amíg az utolsó csomópont f_d-értéke meg nem haladja az aktuális küszöbértéket, vagyis, amíg meg nem talál egy n csomópontot amelyhez tartozó $P_d(n,T)$ üres. Lemma 3.3.2.1. szerint, ha n' egy leszármazottja n-nek, akkor azért, hogy megvizsgáljuk $P_d(n',T)$ üres-e, csak a $P_d(n,T)$-ben lévő csomópontokat kell figyelembe vennünk. Ennek következtében, az alábbiak szerint implementálhatjuk a BIDA*$_d$ algoritmust.

Algoritmus 1. Eljárás BIDA*$_d$

1. Határozzuk meg az A_d halmazt, és a H*(m,t) értékeket minden $m \in A_d$ –ra.
2. Ha $s \in A_d$,akkor a megoldás kiíratása, és kilépés.
3. Számítsuk ki a kezdeti küszöbértéket: $T = f_d(s)$.
4. Határozzuk meg a $P_d(s,T)$ halmazt, és hajtsuk végre Keres(s, $P_d(n,T)$).
5. Számítsuk ki az új T küszöbértéket és folytassuk a 4. lépéssel.

Algoritmus 2. Eljárás Keres(n,S)

1. n minden egyes n' leszármazottjára do
2. legyen g(n') = g(n) + c(n,n');
3. IF n'$\in P_d$ THEN
   számítsuk ki a C(n') = g(n') + H*(n',t)
   ENDIF;
   IF C(n') $\leq T$ THEN
   a megoldás kiíratása és kilépés,
   ELSE a következő leszármazottat vesszük figyelembe
   ENDIF;
4. Határozzuk meg $S' = \{ m \in S \mid g(n') + H(n',m) + H^*(m,t) \leq T \}$;
5. IF S' nem üres THEN Keres(n',S') végrehajtása; od;

Megjegyzések.

1. Az fontos, hogy lássuk, amikor a Keres(n,S) eljárást hajtjuk végre, akkor S = $P_d(n,T)$. Figyeljük meg, hogy S' meghatározása és annak vizsgálata, hogy az esetleg üres-e, ekvivalens azzal a vizsgálattal, hogy $f_d(n') > T$ fennáll-e. Habár, a keresés egy bizonyos út mentén halad, az S halmaz egyre kisebb és kisebb lesz; és mivel a kereső algoritmusok a legtöbb idejüket a keresőfa leveleinek közelében töltik, az S' halmaz meghatározása sokkal gyorsabb, mint kiértékelni a $h_d(n')$ függvényt az (1) egyenletet használva.
2. Minden $m \in P_d$ csomóponthoz kiszámít az algoritmus egy minimális költségű utat *m*-ből *t*-be. Ha ez az út tartalmaz egy másik $m'' \in P_d$ csomópontot, akkor

   $$H(n,m'') + H^*(m'',t) \leq H(n,m) + H^*(m,t)$$

   érvényes minden *n* csomópontra. Ezért az *m* csomópont nem befolyásolhatja az f_d függvény kiszámítását. Az összes olyan csomópont, amelyre ez a tulajdonság igaz, nyugodtan törölhető a $P_d(s,T)$ halmazból.
3. A BIDA*$_d$ eljárás 5. lépésében az új küszöbérték kiszámítása az IDA* algoritmus általános szabálya alapján történik, vagyis a régi *T* küszöbértéket meghaladó

összes f_d értéknek vesszük a minimumát. Nyilvánvaló, hogy ténylegesen a minimum kiszámítása a Keres eljárás 4. lépése közben történik.

4. Az aktuális út minden n csomópontjához el kell tárolnia a $P_d(n,T)$ halmazt a BIDA* algoritmusnak. Habár a Lemma 3.3.2.1. alapján következik, hogy lehetséges úgy is gondoskodni ezekről a halmazokról, hogy csupán egy mutatót használunk az út minden egyes csomópontjához. [5]

3.3.3. A BIDA* algoritmus elemzése

Ebben a részben a BIDA* algoritmust fogjuk elemezni, és összehasonlítjuk az IDA* algoritmussal, ami az egyirányú testvére. Az előző pontban láthattuk, hogy a BIDA*$_d$ abban különbözik az IDA*-tól, hogy a h_d függvényt használja heurisztikának. A Lemma 3.3.1.3. által tudjuk, hogy a h_d „jobban informált", mint az IDA* által használt h heurisztika. Ugyanakkor a h_d kiszámítása sokkal költségesebb, mint a h heurisztikáé. Mindkét tényezőt figyelembe kell venni, amikor a két algoritmust össze akarjuk hasonlítani.

A BIDA*$_d$ algoritmus nem számolja ki nyíltan a h_d függvényt. A Keres eljárás 4. lépésében a BIDA*$_d$ csak azt vizsgálja, hogy $g(n') + h_d(n') \leq T$ fennáll-e, ahol T jelöli az aktuális küszöbértéket. Ha n' szülője n, akkor ez a vizsgálat annyi számú kiértékelését követeli meg a H függvénynek, amekkora méretű a (4) egyenletben definiált $P_d(n,T)$ halmaz. Sajnos, nagyon nehéz felmérni ennek a halmaznak a méretét, mivel nem tudjuk felmérni magának a h_d –nek az átlagos kiszámítási költségét. Világosan fogalmazva, a h_d kiszámítása mindig drágább a h kiszámításánál, amely a H függvény egy kiértékelését igényli csak, de a két költség közötti különbség nagyban függ a problémakörtől.

Elemezzük a h_d heurisztika használatának előnyeit a csomópont-kiterjesztések szerint. Hangsúlyozandó, hogy az itt tárgyalt eredmények nem csak a BIDA*-ra érvényesek, hanem bármely iteratívan mélyülő kerület-kereső algoritmusra is. A következő egyszerű lemma az elemzésünk egyik fő kelléke.

Lemma 3.3.3.1. *Tegyük fel, hogy végrehajtunk egy egyszeri mélységi keresést a BIDA*$_d$ és a BIDA*$_{d'}$ algoritmussal, ugyanazt a küszöbértéket és ugyanazt a leszármazott elrendezést használva. Ha d' > d, akkor BIDA*$_d$ által kiterjesztett csomópontoknak egy részhalmazát terjeszti ki a BIDA*$_{d'}$.*

Mivel IDA* ≡ BIDA*$_0$, az előző lemmából következik, hogy a BIDA*$_d$ soha nem terjeszt ki több csomópontot, mint az IDA*, amikor a két algoritmus ugyanazt a küszöbértéket használja. Habár ez a tény nem vonja maga után, hogy a BIDA*$_d$ általánosságban kevesebb csomópontot terjeszt ki, mivel lehetséges, hogy ugyanannál a probléma-példánynál különböző iterációkat különböző küszöbértékekkel hajt végre a két algoritmus. Azért, hogy összehasonlítsunk különböző iteratívan mélyülő algoritmusokat, meg kell állapítanunk, mely küszöbértékeket használják az egyes algoritmusok. A következőkben, *s* és *t* fogja jelölni a startcsomópontot illetve a célcsomópontot, és *L* jelöli egy optimális út költségét.

Lemma 3.3.3.2. *Legyen $\hbar$ egy elfogadható és monoton heurisztika. Ha az IDA* algoritmus a $\hbar$-t használja, végrehajt egy mélységi keresést a T küszöbértékkel, akkor és csak akkor, ha T ≤ L, valamint létezik egy n csomópont és egy α út s-től n-ig, úgy, hogy $C_\alpha(s,n) + \hbar(n) = T$.*

Ebből a lemmából következik, hogy a BIDA*$_d$ több csomópontot terjeszthet ki, mint az IDA*, ha például, a h_d függvény sok eltérő értéket vesz fel. Ugyanakkor a továbbiakban bebizonyítjuk, hogyha a *H* függvény bizonyos feltételeket kielégít, akkor a BIDA*$_d$ mindig kevesebb csomópontot terjeszt ki, mint az IDA*.

Ha *n'* egy leszármazottja *n*-nek, deifiniáljuk

$$\delta(n,n') = H(n',t) - H(n,t) + c(n,n');$$

mivel *H* monoton, ezért $\delta(n,n') \geq 0$. Figyeljük meg, hogy a $\delta(n,n')$ mennyire méri fel a *H* pontosságának növekedését *n*-től *n'*-ig haladva: ha $\delta(n,n') = 0$, akkor a $H(n,t)$ becslés van olyan jó, mint a $H(n',t)$. Bármely $\alpha \equiv \langle n_1, n_2, n_3, \ldots, n_k \rangle$ útra igaz indukció által

$$C_\alpha(n_1, n_k) + H(n_k, t) = H(n_1, t) + \sum_{i=1}^{k-1} \delta(n_i, n_{i+1}). \qquad (5)$$

Adva van három csomópont *n*, *m*, *m'*, úgy, hogy *m'* leszármazottja *m*-nek, ekkor definiáljuk a következőt:

$\Delta_n(m,m') = H(n,m) - H(n,m') + c(m,m')$.

A Δ függvény segít megállapítani a kapcsolatot a $h_d(n)$ és a $h(n) \equiv H(n,t)$ között.

Lemma 3.3.3.3.

Legyen $m \in P_d$ *úgy hogy* $h_d(n) = H(n,m) + H^*(m,t)$, $n \notin A_d$, *és legyen* $\langle n_1, n_2, n_3, \ldots, n_k \rangle$ *a legkisebb költségű út m-től t-ig. Ekkor*

$$h_d(n) = H(n,t) + \sum_{i=1}^{k-1} \Delta_n(m_i,m_{i+1}).$$

A következőkben azt mondjuk, hogy a H függvény *egyszerű*, ha létezik olyan $\lambda > 0$, hogy

1. ha n' egy leszármazottja n-nek, akkor $\delta(n,n') \in \{0, \lambda\}$,
2. ha m' egy leszármazottja m-nek, akkor $\forall n\ \Delta_n(m,m') \in \{0, \lambda, 2\lambda, 3\lambda, \ldots\}$.

Például a Manhattan-távolság heurisztika a 15-ös Puzzle problémához egyszerű, mivel $\delta(n,n') \in \{0, 2\}$, és $\Delta_n(m,m') \in \{0, 2\}$.

Tétel 3.3.3.4. *Ha H egyszerű, akkor a BIDA*$_d$ algoritmus soha nem terjeszt ki több csomópontot, mint az IDA*.*

Bizonyítás. A Lemma 3.3.3.1. alapján tudjuk, hogy elegendő azt bizonyítani, hogy a BIDA*$_d$ által használt küszöbértékek részhalmazát képezik az IDA* által használtakénak. Tegyük fel, hogy BIDA*$_d$ végrehajt egy iterációt egy $T \leq L$ küszöbértékkel. A Lemma 3.3.3.2. alapján tudjuk, hogy létezik egy n csomópont és egy α út s-től n-ig úgy, hogy $C_\alpha(s,n) + h_d(n) = T$. Továbbá, a Lemma 3.3.3.3. és az (5) képlet szerint tudjuk, hogy létezik két egész szám N_1, N_2 úgy, hogy

$$T = C_\alpha(s,n) + H(n,t) + N_1\lambda = H(s,t) + N_2\lambda. \qquad (6)$$

Jelölje β a legkisebb költségű megoldást (ami egy út), $\beta \equiv \langle n_1 = s, n_2, n_3, \ldots, n_k = t \rangle$. Definiáljuk az $i = 1,\ldots,k$ esetén

$$T_i = C_\beta (s,n_i) + h(n_i).$$

Ekkor ott tartunk, hogy $T_1 = H(s,t)$, $T_k = L$, és $T_{i+1} = T_i + \delta(n_i, n_{i+1})$. Továbbá, mivel a H heurisztika egyszerű, igaz az, hogy

$$T_{i+1} = T_i \qquad \text{vagy} \quad T_{i+1} = T_i + \lambda. \tag{7}$$

Most nézzük a $T_1, T_2, \ldots, T_k$ sorozatot. Azt látjuk, hogy $T_1 \leq T \leq T_k$, mivel a (7) és a (6) egyenletből következik, hogy létezik olyan i, hogy $T_i = T$. Ezzel bebizonyítottuk a tételt.

Ha a H függvény nem egyszerű, akkor az IDA* és a BIDA*$_d$ algoritmusok aszimptotikus viselkedését hasonlíthatjuk össze. Az ismert, hogy az IDA* algoritmus gyengén teljesít, ha minden egyes iterációban csak egy kevés számú új csomópontot terjeszt ki. Ennek a hátránynak a legyőzésére találtak ki egy technikát, aminek a neve Szabályozott Újrakiterjesztés (Controlled Reexpansion). Ez a módszer garantálja, hogy B^i új csomópont kerül kiterjesztésre az i. iterációban, ahol B egy, felhasználó által definiált paraméter. A módosított algoritmus által generált összes csomópont száma $O(|\mathcal{N}|)$, ahol

$$\mathcal{N} = \{ n \mid g(n) + h(n) \leq L \}.$$

Ekkor $|\mathcal{N}| \approx b^L$, ahol b jelöli az effektív elágazási tényezőt a h heurisztikához. Ha a Szabályozott Újrakiterjesztés technika és a BIDA*$_d$ kombinálva van, akkor az így kapott algoritmus $O(|\mathcal{N}_d|)$ csomópontot generál, ahol

$$\mathcal{N}_d = \{ n \mid g(n) + h_d(n) \leq L \}.$$

Mivel $h_d \geq h$, azt kapjuk, hogy $\mathcal{N}_d \subseteq \mathcal{N}$.

Az előző elemzésben nem vettük számításba, hogy a BIDA* két különálló keresés végrehajtásával jut hozzá a megoldáshoz. Meglepőnek tűnhet ez, mivel általánosan elismert tény, hogy a fő előnyük a kétirányú kereső algoritmusoknak az, hogy két kis fát derítenek fel egy nagy helyett. Valóban gondolhatnánk azt, hogy a BIDA*$_d$ algoritmusnak egy d mélységig történő hátrafele keresést, és egy L - d mélységig történő előrefele keresést kellene elvégeznie, ahelyett, hogy egy egyszeri keresést hajt végre L mélységig. De meg kell jegyeznünk, hogy a két keresési front csak akkor találkozik, amikor a küszöbérték megegyezik L-lel. Továbbá, mind a BIDA*$_d$ és mind az IDA* is L mélységig keres, a fő különbség a két algoritmus között, hogy a BIDA*$_d$ kevesebb csomópontot generál a h_d heurisztika nagyobb *metszési* erejéből kifolyólag. [5]

4. Kísérleti eredmények

A kétirányú algoritmusok teljesítményének tényleges megfigyelésére és összehasonlítására a mellékelt programot használtam. Ennek segítségével valódi, gyakorlati eredményeket kaphattam, és így lehetőség nyílt ezeknek az eredményeknek a publikált eredményekhez való hasonlítására.
A kísérletek a 8 Puzzle nevű problémát használták fel. Ez tulajdonképpen abból áll, hogy egy 3*3-as táblán (tehát összesen 9 mező) van egy üres mező, a többin pedig 8 számozott lapocska 1-től 8-ig. Az üres mezőre lehet mindig egy szomszédos lapocskát áttolni. Kezdetben a táblán össze-vissza lehetnek a lapocskák. A cél az, hogy végül úgy nézzen ki a tábla, hogy a bal felső sarokban legyen az üres mező, a többi mezőn pedig sorfolytonosan a többi lapocska a rajtuk lévő szám szerint emelkedő számsorrendben. A felhasznált heurisztika a nem megfelelő helyen lévő lapocskák számának összeadására alapult.
A program Java nyelven implementált. A programot futtató számítógép egy AMD Athlon XP 2400+ processzorral, 1536 MB RAM-mal ellátott rendszer.

Még a kezdetek kezdetén Pohl levonta azt a következtetést, hogy a nem informált keresők tekintetében a kétirányú keresők nagyon jól teljesítenek. Ugyanez látszódik a nem

informált keresőkkel végzett kísérleteimben is. Amikor két szélességi keresőt indítottam el egymással szemben, általában ötöd-tized annyi csomópontot terjesztett ki és állított elő.

Amikor két A* algoritmus indult el egymással szemben, vegyesek voltak a tapasztalatok, volt amikor a szélességihez hasonló eredményt produkálták, volt amikor elég rosszul teljesített a kétirányú keresés az egyirányúhoz képest.

Körülbelül 20 különböző kezdőállapottal elindítva vizsgáltam meg az A*, a BHFFA és a BIDA* keresők teljesítményét. A következő sorokban foglalnám össze az említésre méltó esteket, illetve az átlagos teljesítmény arányukat.

Azokban az esetekben, amikor az optimális megoldás költsége 15- 20 lépés között volt, az A* algoritmus kb. 1160 kiterjesztést 300 ms alatt, a BHFFA kb. 240 kiterjesztést 40640 ms alatt, a BIDA* 1132 kiterjesztést 46 ms alatt igényelt. Ez utóbbi 11-es mélységet használó kerület-keresővel. Látható, hogy az A*-nál kb. ötöd annyit terjesztett ki a BHFFA, de az ehhez felhasznált idő rengeteg, kb. 35-szöröse az A* algoritmusénak. A BIDA* kiterjesztéseinek száma 97%-a az A* algoritmus kiterjesztésszámának, viszont hatod annyi idő alatt érte le ezt a teljesítményt. Ha az arányokat megnézzük, láthatjuk, hogy a BHFFA amit behozott a kiterjesztések számával, elvesztette időben, mivel ugye a költséges heurisztika-számítások nagyon sok erőfeszítést igényelnek. A BIDA* nem terjesztett ki nagyságrendekkel kevesebbet az A* algoritmusnál, sőt, a többi mélységű kerület-keresésnél sokkal rosszabb eredményt produkált; de az időt is figyelembe véve, összességében a BIDA* teljesített ebben az esetben a legjobban.

Amikor az optimális megoldás költsége az előbbieknél kevesebb, vegyük pl. a 9-et, akkor nem tapasztalható nagy eltérés az egyes algoritmusok teljesítménye között, valószínűleg azért, mivel nincs rá elég „lehetőség” hogy kijöjjenek az előnyök illetve hátrányok. A BHFFA most is kevesebb csomópontot terjesztett ki, mit az A*, de most csak 1/3-al kevesebbet. Ugyanakkor a lefutási ideje most is több volt, de nem olyan sokkal, „csak” négyszerese az A* idejének. A BIDA* kiterjesztések számában hasonló az A* algoritmushoz ismét, a legjobb esetekben. Ezek az esetek kis költségű út, és viszonylag könnyű kiinduló állapot miatt az 1-es és a 2-es mélységű kerület-kereséseknél fordult elő. Az 1 mélységű esetben a kiterjesztések száma teljesen megegyezik az A* algoritmuséval, de a felhasznált

ideje annyira kevés volt, hogy pontos adat nincs róla, mivel 1 ms-nél is kevesebb volt, míg ehhez az A*-nak 16 ms-re volt szüksége. A 2-es mélységű BIDA*-nál még jobb a helyzet, hasonlóan kevés idővel, még kevesebb csomópontot terjesztett ki, ami 76%-a az A* algoritmusénak. A nagyobb mélységszámmal lefutott BIDA* keresők, bár ugyanolyan kevés idő alatt, egyre több csomópontot terjesztettek ki, ez annak tudható be, hogy a kerület-kereső algoritmus már túl sok csomópontot tárolt el a probléma nehézségéhez képest.

Azokban az esetekben, amikor az optimális megoldás költsége meghaladta a 20-as értéket, jobban kijöttek az egyes különbségek, igen sok helyen nagyságrendekkel eltérő értékek jelentek meg. A BHFFA kereső ekkor kb. tized annyi csomópontot terjesztett ki, mint amennyit az A*, de a felhasznált ideje akár 200-1000-szer több volt, mint amennyit az A* igényelt. Itt is látható, amit már előbb említettem, hogy mennyire le tudja rontani a futási időt a nagy számításigényes kiértékelő függvények. A BIDA* keresés esetében, a kis mélységű kerület-kereséssel a csomópontok tekintetében igen rosszul állt, sokkal többet terjesztet ki, mint az előző kettő algoritmus, de az idő tekintetében a kettő között helyezkedik el. Ha növeljük a kerület-kereső mélységét, akkor egyre jobb eredményeket kapunk, az egyes mélységek között kb. ötödére csökkennek az eredmények. A legjobb eredményeket körülbelül a 8-as mélységnél kaphatjuk. Ez ugye azért lehetséges, mert viszonylag bonyolult a kiinduló állapot, sokat kell kutatni a gráfban, ezért kell nagyobb mélységben feltárni a kerületet, de a 8 Puzzle probléma sajátosságai miatt ennél nagyobb mélységnél már esetleg hátrány lehet a további felderítés, illetve felesleges is lehet.

Tehát láthatjuk az egyes eredményekből, hogy mindegyik keresőhöz található olyan eset, amikor az tűnik hatékonyabbnak. Ha az egyes esetek eredményeit átlagoljuk, a leggyakoribb arányokat nézzük az egyes keresőknél, akkor talán azt mondható, hogy az egyszerűbb feladatoknál jól láthatóan kijön a BIDA* fölénye, hiszen mind időben, mind kiterjesztésben jobb a többinél, és nem kell mérlegelni így az egyes előnyök és hátrányok között, mint a BHFFA-nál, ahol egy bizonyos szűk keresztmetszetet határoz meg a kiterjesztések/idő arány.
Az, hogy az egyszerűbb feladatoknál jobban szemügyre vehető a BIDA* hatékonysága, az talán még azt hordozza magával, hogy az IDA* algoritmus általában egyszerűbb problémákkal könnyebben boldogul.

Láthatjuk, hogy ezek az eredmények igazán változatosak. Nagyban függnek a problémától, a megoldás bonyolultságától, a heurisztikától és egyéb plusz információktól, valamint ne felejtsük el, az implementációtól is.

Mivel kevés tesztelési időm volt, és valószínűleg nem sikerült minden algoritmust a megfelelően optimalizálnom, ezért lehetséges, hogy ezek az eredmények nem pontosan tükrözik a tényleges teljesítményadatait az egyes algoritmusoknak. De mivel, a program leginkább saját elgondolásaim megvalósításához, és a tanulmányozott algoritmusok kipróbálásához készült, így nem is az volt a célom, hogy hivatalos eredményeket közöljek. Mint minden programban, ebben is lehetnek hibák, ezért előfordulhatnak nem megfelelő futási eredmények.

5. Konklúzió

Láthatjuk, hogy a *front-to-end* becsléssel dolgozó algoritmusoknál jobb eredményeket produkálnak a *front-to-front* becslést alkalmazók, valamint a nem tradicionális keresők általában jobban teljesítenek, mint a tradicionálisak. A ma folyó kutatások ezért inkább a BIDA* elgondolására épülő algoritmusokkal kapcsolatban folynak, főleg abból a szempontból kifolyólag, hogy a keresési teret lineárissá alakítsák, valamint, minél jobb heurisztikus becslést használjanak a futás során. Ugyanakkor vegyük figyelembe, hogy az egyes tényezők javulása általában más tényezők romlásával jár. Ilyen az, amikor a kevesebb csomópont ára a nagy futási idő, a nagy számításigény. Meglátásom szerint minden egyes kifejlesztett algoritmusnak a meghatározója, a többihez viszonyított jósága azon múlik, hogy ez a bizonyos arány hogyan alakul akár a legrosszabb esetekben.

Az egyes kutatásokból, publikációkból látható, hogy a BIDA* nagyon sok helyen szerepet játszik manapság, mint olyan algoritmus, melynek eredményei összehasonlítás alapjául szolgál más algoritmusok teszteredményeihez. Ugyanez igaz az A* algoritmusra, mivel nincs olyan algoritmus, amely minden egyes esetben, bármilyen problémára nézve abszolút jobb teljesítményt mutatna nála, ezért, mint az egyirányú keresők legnépszerűbb képviselője ma is a legtöbb kutatás etalonjának számít.

Az, hogy a BIDA* illetve a kerület-kereső algoritmusok többnyire jobban teljesítenek, mint a korábban használt, más szemlélet alapján felépített algoritmusok, nem jelenti azt, hogy ez a biztos út. Manapság is sok olyan újabb és újabb kutatás folyik, melyek azt igazolják, hogy akár a tradicionális *front-to-front* kereső algoritmusok továbbfejlesztése egyes problémakörökben, egyes heurisztikák mellett sokkal jobban teljesítenek a nem tradicionális keresőknél is. Mióta bebizonyosodott, hogy a Pohl kijelentése óta tanúsított csekély figyelem a kétirányú keresők iránt alaptalan volt, napjainkra igen megnőtt a kétirányú kereső algoritmusok fejlesztése iránti „kedv”. Attól függetlenül, hogy a nem tradicionális keresők, köztük akár a BIDA*, igen sok területen jobban teljesítenek konkurenseinél, egyre több, általában esetleg rosszabb teljesítményt mutató algoritmusokhoz nyúlnak vissza továbbfejlesztés céljából. Ez annak tudható be, hogy bizonyos probléma-osztályoknál, bizonyos megkötések mellett eltérő sorrend alakul ki az algoritmusok között hatékonyság tekintetében. Tehát fontos azt figyelembe venni, hogy az algoritmusok hatékonysága nagyban

függ a probléma-osztálytól, a bonyolultságtól, az alkalmazott heurisztikától is akár, és egyéb olyan plusz információtól, melyet a keresés hatékonyságát növelheti.

Összességében, tehát, Pohl által tévesen levont következtetését megcáfolva széleskörűen elfogadottá vált az a nézet, hogy a kétirányú keresőknek igen is van létjogosultsága, és vannak olyan problémakörök és specifikációk, amikor jobban teljesítenek bármelyik egyirányú kereső algoritmusnál.

A dolgozatban tárgyalt kereső algoritmusok vizsgálatakor, tekintve az alap elgondolásokat, illetve az egyes keresők előnyeit illetve hátrányait, további ötleteim támadtak, milyen fejlesztésekkel lehetne javítani még jobban a kétirányú kereső algoritmusokon, illetve milyen kísérleteket lehetne végrehajtani további eredmények szerzésére.

Legfőképpen az osztott rendszerek adta lehetőségeket emelném ki, ami már éppen csak meg lett említve a dolgozat elején is. Egy több processzoros rendszernél, melynél az egyes processzoroknak szánt feladatot meg tudjuk szabni, lehetőség lenne valóban szimultán kétirányú kereséseket megvalósítani. Az, hogy megszabható, a munkának mely részét melyik mag végezze el, lehetőséget ad arra, hogy olyan kétirányú keresők heurisztika értékeit számoltassuk ki egy maggal, amelyek függetlenek a másik irányból jövő kereső heurisztikájának a kiszámításától. Tehát azoknál a keresőknél alkalmazható elsősorban, ahol nem kell a heurisztika kiszámításához esetleg figyelembe venni a másik kereső által folyamatosan létrehozott csomópontokat, illetve azok heurisztikáját, költségeit. Habár, esetleg ebben az esetben is alkalmazható ez a módszer, ha az egyes kalkulációkat valamiféleképpen függetleníteni tudjuk egymástól, és szét tudjuk őket választani, hogy megfelelő legyen egy több processzoros rendszer előnyeinek kihasználásához. Egy másik elgondolás, mely szintén jól tudná hasznosítani mellékesen az elosztott rendszerek előnyeit, az az lenne, hogy kvázi kettőnél több irányú keresést hajtanánk végre, pontosabban egyszerre akár kettőnél több keresés is futna a gráfot bejárva. Ezt úgy lehetne megvalósítani, hogy valamilyen sejtés alapján a kezdő és a cél csomópont közötti úton megpróbálunk konstruálni egy vagy több állapotot. Ezután ezen a kezdeti állapoton próbálunk folyamatosan javítani, hogy elérjük a célállapotot, tehát onnan is elindul egy keresés a gráfban. Hasonlóan működnek az iteratívan javuló kereső algoritmusok, amelyek ilyen sejtés alapján indulnak el keresni a gráfban.

Tulajdonképpen úgy is fel lehet fogni ezt az elgondolást, hogy a gráfot felosztjuk több részre, és kisebb részeken végzünk, esetleg jobban becslő heurisztikával keresést. Abból kifolyólag, mivel több, független részgráfot kapunk, és függetlenek az egyes keresési iterációk, ehhez a módszerhez is nagyon hasznosnak bizonyulhat egy elosztott rendszer. Az, hogy igazából milyen eredményeket szolgáltathat egy ilyen algoritmus, optimális-e és teljes-e, illetve egyáltalán milyen probléma-osztályokra alkalmazható, az még nyitott kérdés.

Magával a BIDA* algoritmussal kapcsolatban is akadtak ötletek a dolgozat írása folyamán, melyekkel talán javulást lehetne elérni. Ezek leginkább arra épülnek, hogy az IDA* javításait ültetném át a kétirányú változatba. Azt tudjuk, hogy az IDA* lineáris térben keres, ezért nagyon jól bánik a memóriával. Ehhez hozzátartozik, hogy nincs „memóriája”, mivel csak egyetlen f határértéket tárol el, ezért újra és újra végigjárhatja ugyanazt a részgráfot. Ha esetleg a BIDA* is tárolna az egyes csomópontokban információt a bejárt, de törlésre került részgráf legjobb útjáról, csak akkor generálná le újra azt a részgráfot, ha az érték alapján az tűnik a legjobb választásnak az összes tárolt út közül. Egy másik ötlet, amivel javulást lehetne elérni, olyan esetekben lehet jó, amikor a heurisztika minden egyes csomópontnál más és más, mivel ilyenkor felmerülhet az a probléma, hogy egy új küszöbértékkel indított iterációban csak egy csomóponttal több közül lehet választania az algoritmusnak, mint az előző küszöbértékkel. Erre lehet megoldás egy, más algoritmusoknál már használt módszer, az, hogy az egyes f értékeket egy alkalmas, rögzített ε értékkel növeljük. Ekkor 1/ε-al arányos iterációt fog eredményezni a keresés. Habár így a keresési költség csökkenhet, de a megoldás minősége romolhat, mivel az optimálisnál rosszabb lehet, de legfeljebb csak ε-val.

6. Implementáció

A dolgozathoz mellékletként tartozik egy program, mely a dolgozatban tárgyalt algoritmusokat implementálja, és azokról kísérleti eredményeket szolgáltat. Az implementáció Java nyelven készült. A program, mely inkább tekinthető egy programcsomagnak, széleskörű lehetőségeket biztosít az egyes algoritmusok implementációinak, külön-külön illetve egymással kombinálva történő felhasználására, illetve további fejlesztések alapja lehet. A program készítésének kezdetén egy átlátható, logikusan felépített, jól strukturált és nem túlspecifikált alapot szerettem volna létrehozni. Szándékomban volt még az alap egyirányú algoritmusokat is implementálni hasonló módon, hogy a későbbi kétirányú algoritmusok ezekre épüljenek, illetve látható legyen a forráskódban is ezeknek az algoritmusoknak a viszonya. Időközben rátaláltam a [1] forrás szabadon felhasználható forráskód mellékletére, mely megfelelt az előbb említett elvárásaimnak, ezért részben fel lehetett használni a programcsomag elkészítéséhez. Némi módosítással, már jó alapot képezett a kétirányú algoritmusoknak az implementálásához. A felhasznált forráskódok az alábbi címen voltak elérhetők [6]. A program forráskódját tartalmazó fájlok nevei a files.txt-ben vannak felsorolva, rövid megjegyzésekkel, hogy mely forráskódok lettek esetleg módosítva, vagy teljes egészében saját magam által készítettek.

Nem célom a dolgozatban elhelyezni egy rendes dokumentációt a mellékelt programról, mivel a fő hangsúlyt nem erre kívánom a dolgozatban helyezni, másfelől, a programról készített részletesebb dokumentációt az érdeklődők megtalálhatják a forráskódokkal együtt a mellékleten. Mivel a program készítésének egyik fő iránymutatója az volt, hogy általánosan használható, és logikusan felépített legyen, és ez az osztályhierarchiából maximálisan látszik is, ezért inkább azt részletezném pár mondat erejéig a teljesség igénye nélkül, útmutatással, mely segítséget adhat a program jobb olvashatóságához is.

A *Problem* osztály

A *Problem* osztály segítségével határozhatunk meg egy konkrét problémát, feladatot a programunk számára. A *Problem* osztály az állapottér-reprezentáció szemlélete szerint várja

el, hogy specifikáljunk egy problémát, melyet meg akarunk oldani. A *Problem* osztály egy példányát jellemzi egy *Object* típusú kezdőállapot, egy célfeltételt meghatározó *GoalTest* interfészt implementáló osztály példánya, egy, olyan *SuccessorFunction* interfészt implementáló osztály példánya, amely az állapottér-reprezentációban operátorokként ismert műveletek alkalmazását implementálja. Továbbá opcionálisan megadható olyan interfészeket implementáló osztályok, amelyek az operátorok alkalmazásának költségét határozzák meg, esetleg az informált keresőknél a heurisztika kiszámításának módját adja meg. Ezeknek az opcionális információknak a megadása természetesen az adott algoritmusoktól függnek, mely információkra van szükségük a működéshez, illetve, van, amit éppen maga az adott algoritmus eleve determinál, ezért nem kell külön megadni. Tehát minden egyes keresőhöz meg kell tudnunk adni egy problémát reprezentáló *Problem* típusú objektumot, egyes kétirányú keresőknél kettőt is, mivel némely algoritmusok két *Problem* példányból építik fel logikailag azt a problémát, mely elegendő információval bír a kétirányú kereséshez.

A Search interfész

A *Search* interfészt implementáló osztályok maguk az egyes kereső algoritmusokat megvalósító osztályok. A *Search* interfész egyfajta struktúrát ad nekik, meghatározza a keresők közös tulajdonságát, és általános érvényű kommunikációs csatornát is épít így ki, melyet a keresésekről kapott információk összegyűjtésére is jól lehet alkalmazni. Ezt az interfészt implementáló osztályok között olyan öröklődési hierarchia van, amely tükrözi az egyes algoritmusok viszonyát, az elméleti osztályozását a tárgyalt keresőknek. Például a *Search* interfészt implementálja a *PrioritySearch* abstract osztály, mely olyan kereső algoritmusok absztrakciója lehet, melyek az adott csomópontok között valamilyen sorrendet tartanak fent folyamatosan. Ennek az osztálynak a leszármazottja például a *BHFFAOneDirSearch*; vagy a *BestFirstSearch* osztály, mely a legjobbat először stratégiát alkalmazó kereső algoritmusokat reprezentálja általánosságban. Ennek az osztálynak a leszármazottja az *AStarSearch* osztály, mely az A* algoritmust valósítja meg. Tehát az osztály-hierarchia jól mutatja az algoritmusok hierarchiáját, és ez az analógia a program más részeiben is megtalálható.

Egyes egyszerűbb, illetve az előbbi hierarchiából sajátosságaik miatt kimaradt kereső algoritmusok a *NodeExpander* leszármazottjai, mivel ez az osztály tartalmazza a keresők azon motorját, mely a csomópontok előállításáért felelős. Azok az osztályok melyek nem ennek a leszármazottjai, azok a *NodeExpander* osztály valamely leszármazottjainak segítségével valósítják meg ezt a funkciót, egy komplett kereséssel, kibővítve a kereső-specifikus műveletekkel és információkkal. Ilyen osztály a *QueueSearch* és annak leszármazottja, a *GraphSearch* osztály. Azt, hogy ez a két, különálló osztályhierarchia együttműködjön a keresés során, egy *SearchAgent* osztály gondoskodik. Ebben az osztályban lévő metódusok segítségével történik meg a tényleges keresés, az egyes argumentumként megadott osztály-példányok szolgáltatásai végrehajtódnak. A *SearchAgent* osztály gondoskodik a megoldás rendezett formában való visszaadásáról, valamint a keresési metrikák összegyűjtéséről. A metrikák tárolására a *Metrics* osztály ad lehetőséget, mely tetszőleges különböző műveletek számszerűsített jellemzőinek, adatainak eltárolását és előhívását teszi lehetővé.

A kétirányú kereső algoritmusok egyes példányaihoz külön kellett egy osztályt létrehozni, amely a *SearchAgent* osztályéhoz hasonló funkciókat biztosít, ez lett a *SearchAgentForBiSearch* osztály. Annyi a különbség, hogy úgy értelmezi a kétirányú keresést, mint két egyirányú keresés, azaz mindegyik rendelkezik egy *Problem* példánnyal, és a két keresés részeredményei folyamatosan egyeztetve vannak. Ez azt jelenti, hogy például az egyes kiterjesztések között mindig megvizsgálja, hogy a két keresés találkozott-e. Ezenfelül, ezen osztály metódusai arról is gondoskodnak, hogyha megtörténik ez a bizonyos találkozás, akkor ellenőrizze a megoldás minőségét, és ha megfelelő, akkor kiíratásra alkalmas formára hozza.

A BHFFA algoritmushoz, és a hasonlóan működő keresőkhöz lett létrehozva a *FrontToFrontEvaulationFunction*, mely lehetőséget ad a megfelelő heurisztikával párosítva olyan becslésre, ahol egy állapotot dinamikusan lehet egy futás közben meghatározandó állapot vagy állapotokhoz viszonyítani. Természetesen, ahogy a *Search* interfész, az *EvaulationFunction* és a *HeuristicFunction* interfész is meghatározza, hogy nézhet ki egy kiértékelő és egy heurisztikát kalkuláló metódusokkal rendelkező osztály.

A BHFFA algoritmus implementálása a *SearchAgentForBiSearch* osztály funkcióinak a felhasználásával történt. A keresési szisztémát, ahogy más összetettebb keresőknél, itt is az

egyik *Search* interfészt implementáló osztály valósítja meg, amely a *BHFFAOneDirSearch*. Mivel csak az egyik irányba történő haladást valósítja meg, igazából két példányt kell belőle használni, és a keresés folyamán egymást tudják befolyásolni. Így valósítva meg a BHFFA által kívánt folyamatos dinamikusságot.

Implementáltam egy egyirányú kereső algoritmust, az IDA* algoritmust is, melyet nem tartalmazott az alapként használt programcsomag. Ezt a keresési stratégiát az *IDAStarSearch* osztály valósítja meg, mely a *NodeExpander* osztály leszármazottja, valamint implementálja a *Search* interfészt is. A végrehajtást és az eredmények illetve metrikák feltárását a *SearchAgent* osztály metódusai valósítják meg. Az IDA* algoritmus inicializálása hasonlóan történik, mint például az A* algoritmusé, ugyanazt a kiértékelési függvényt megvalósító osztály példányát adtam meg neki a programban. Persze ez lehetne akár egy másik *EvaulationFunction* interfészt implementáló osztály példánya is, mely megfelel a specifikációknak.

A BIDA* kereső algoritmust egy osztály valósítja meg, a *PerimeterSearch* osztály, mely metódusai felelnek a kerület létrehozásáért, majd az IDA*-hoz hasonló algoritmus a kerületet képező csomópontok kollekcióját felhasználva fut le, a dolgozatban tárgyalt algoritmus alapján. Ennek a keresőnek a hívása úgy történik, mint egy egyirányú keresőé a szekvenciális jellege miatt.

Ha futtatni akarjuk a programot, úgy, hogy saját magunk által készített feladathoz kívánjuk felhasználni az implementált kereső algoritmusokat, az egyszerűen megtehető. A *Problem* osztály által várt objektumok típusai megadják, milyen osztályokat milyen adattagokkal és metódusokkal kell létrehozni. Ilyen az állapotot leíró osztály esetleg az operátorok alkalmazásának előfeltételeit leíró metódusokkal; az operátorok alkalmazását megvalósító osztály; a célfeltételt vizsgáló metódust tartalmazó osztály; és esetleg az operátor alkalmazásának költségeit illetve a heurisztikát kiszámító osztályok. Látható, hogy ezek azok a minimális információk, amelyek feladat specifikusak, és mindenképpen meg kell adni a programnak. Valamint természetesen gondoskodni kell a *main* metódusról, és a megkapott keresési eredmények kiíratásáról is a kívánt formában.
[6]

7. Összefoglalás

Az előzetes célom az volt, hogy a dolgozatban egy átfogó képet adjak a kétirányú keresők világáról, és azon belül azokról a kétirányú kereső algoritmusokról, amelyek a téma meghatározó szemléleteinek képviselői. Ezek a kereső algoritmusok egy-egy oldalról bástyaként zárják közre a kétirányú keresők életterét, valamint a sokáig mellőzött kétirányú keresési stratégiákat alkalmazó algoritmusok kifejlesztéséhez további útmutatást is adnak.
A diplomamunka papír alapú részét próbáltam úgy megszerkeszteni, hogy egy laikus olvasó számára is következetes, egymásra épülő információkból egy, a témához szükséges tudásbázist alakítson ki egészen végig. Továbbá szerettem volna úgy összeállítani, hogy az olvasó a dolgozat végére felismerje a kétirányú algoritmusokban rejlő lehetőségeket, valamint a levont konklúziók segítségével el tudja helyezni a kétirányú kereső algoritmusokat a problémamegoldás világában. Úgy gondolom, hogy a fent említett célokat sikerült elérnem, habár a tapasztalati eredményeket tárgyaló rész, és az egyes pontok hagynak némi kívánni valót maguk után. Ez annak is betudható, hogy a téma megismeréséhez felhasznált irodalom angol nyelven készült, valamint némelyik nehezen hozzáférhető, nem éppen aktuális, és rossz minőségű volt.

Az alkalmazást nagyjából sikerült úgy megvalósítanom, ahogy az elején terveztem. Szerettem volna egy olyan programot létrehozni, amely általános célú, egyszerűen továbbfejleszthető, illetve bármilyen feladathoz felhasználhatóak a keresők, és könnyen cserélhetőek az egyes részek. Sok olyan fejlesztést, és a használhatóságot könnyítő funkciót terveztem még a programhoz, amit sajnos idő hiányában már nem tudtam befejezni, de az egyes keresők teljes körű használhatóságát ezek nem befolyásolják. A program fejlesztésénél is törekedtem a dolgozat papír alapú részében megmutatkozó logikai felépítés tükröződésére. Az alkalmazás készítése közben magam is fejlődtem a Java nyelven való programozás területén, és további tapasztalatokat szereztem komplexebb programok implementálásában.

Ezúton szeretném megköszönni Dr. Nagy Benedek témavezetőmnek a türelmét és a munkámat segítő tanácsait.

8. Irodalomjegyzék

Magyar nyelvű forrás:

[1] Stuart J. Russel, Peter Norvig Mesterséges Intelligencia Modern Megközelítésben. Panem Könyvkiadó Kft. Budapest, 2000

[2] http://www.inf.unideb.hu/~varteres/mi/part1/index.htm (2010-06-01)

Angol nyelvű forrás:

[3] AN IMPROVED BI-DIRECTIONAL HEURISTIC SEARCH ALGORITHM, Dennis de Champeaux & Lenie Sint. Instituut voor Bedrijfseconomie en Accountancy Universiteit van Amsterdam/Netherlands MirrorCy; 1974 December; Revised 1975 June

[4] Hermann Kaindl-Gerhard Kainz. Bidirectional Heuristic Search Reconsidered Journal of Artificial Intelligence Research 7 (1997) 283-317 Submitted 8/97; published 12/97

[5] Manzini, G. (1995). BIDA*: an improved perimeter search algorithm. Artificial Intelligence, 75(2),347-360

[6] http://code.google.com/p/aima-java/ (pontosítva: 2010-06-01)