

TDK-dolgozat

Banyik Nándor



A Háromdimenziós Utazó Ügynök Probléma Megoldása és Vizualizációja

Készítette

Banyik Nándor

Informatikai Kar

Programtervező Informatikus Bsc. 3. év

Témavezető

Dr. Kovásznai Gergely

Számítástudományi Tanszék

Tanszékvezető, egyetemi docens

EGER, 2022

Tartalomjegyzék

Bevezetés	4
1. Az utazó ügynök probléma	5
1.1. A probléma	5
1.1.1. A probléma felépítése	5
1.1.2. Számítási nehézség	5
1.1.3. Egyéb nehézségek	6
1.2. Meglévő algoritmusok	6
1.2.1. Nearest Neighbour	6
1.2.2. Branch and Bound	8
2. Háromdimenziós utazó ügynök probléma	9
2.1. Alapgondolatok	9
2.1.1. Elméleti alapok	9
2.1.2. Miért lényeges a térbeli vizsgálat?	10
2.2. Implementáció	10
2.2.1. Keretrendszer	10
2.2.2. Generálás	11
2.2.3. Nearest Neighbour implementálása	11
2.2.4. Branch and Bound implementálása	12
3. Segmented Nearest Neighbour	14
3.1. Alapgondolatok	14
3.1.1. Elméleti alapok	14
3.2. Felosztás	14
3.2.1. Generálási intervallum alapú felosztás	14
3.2.2. Generálási intervallumhoz legközelebb álló sarok városok koordinátáin alapú felosztás	15
3.3. Az algoritmus	16
3.3.1. Legutolsónak bejárt él megkeresése	16
3.3.2. Szegmens bejárása	16
3.3.3. Következő szegmens kiválasztása	16

3.3.4.	Befejezés	16
3.3.5.	Javítási lehetőség	16
3.3.6.	Az algoritmus magyarázata	17
3.4.	Négy szegmenses felosztás alkalmazása három dimenzióban	19
3.4.1.	Felosztás	19
3.4.2.	Összehasonlítás a Nearest Neighbourral	20
3.4.3.	Javítási lehetőségek	21
3.5.	Nyolc szegmenses felosztás alkalmazása három dimenzióban	22
3.5.1.	Generálási módosítás	22
3.5.2.	Felosztás	22
3.5.3.	Észrevételek	22
3.5.4.	Implementált javítás	22
3.5.5.	További észrevételek	22
3.5.6.	Megoldás	23
3.5.7.	Konklúzió	24
	Összegzés	25
	Irodalomjegyzék	26

Bevezetés

Kutatásomban az utazó ügynök problémával foglalkoztam, mely egy közismert probléma, aminek a lényege, hogy adott városok között a bejárt út a lehető legolcsóbb legyen. Kezdetben két algoritmust hasonlítottam össze, végül az egyik algoritmust úgy módosítottam, hogy nagy eséllyel szuboptimális utakat járjon be kis futási idővel.

Dolgozatomban írni fogok az utazó ügynök problémáról részletesen, illetve az implementált meglévő algoritmusokról (Branch and Bound, Nearest Neighbour). Az utazó ügynök problémáról háromdimenzióban; miben változtatja a problémát, miért fontos kérdés, és az implementálása. Ezt Unity3D keretrendszerben implementáltam és vizualizáltam. Itt generáltattam véletlenszerű gráfokat, amelyekre alkalmaztam a fentebb említett algoritmusokat, majd összehasonlítottam azok lefutási idejét, távolságát, és a bejárt utat vizualizálom háromdimenzióban.

Továbbá ismertetem a Nearest Neighbour hibáját, és az ennek a hibának a javítására szolgáló módosított algoritmust. Ez egy általam tervezett és implementált algoritmus, mely a Segmented Nearest Neighbour nevet viselni. Ismertetem ennek az algoritmusnak az alapötletét, működését, hibáit, javításait, és a vele végzett kísérleteket.

Kutatásom eredményes volt, a lefutások sorozata után a Segmented Nearest Neighbour számottevően jobb eredményt adott, mint a Nearest Neighbour. A futási idő nőtt, ám csak elhanyagolható mértékben.

Az 1. fejezetben az utazó ügynök problémáról fogok írni általánosságban, majd a meglévő algoritmusokról, amelyeket implementáltam a generált gráfokon. A 2. fejezetben ugyancsak az utazó ügynök problémáról fogok írni, de ebben az esetben három dimenzióban, valamint az ezzel okozott új problémákról. A 3. fejezetben az általam tervezett, implementált algoritmusról fogok írni, amely a Nearest Neighbour módosítása, illetve bővítése. Ezzel az algoritmussal küszöbölöm ki a Nearest Neighbour problémáit. A 3.4. alfejezetben 4 felosztásos módszerrel való próbálkozásaim, kísérleteim eredményeiről írok. A 3.5. alfejezetben 8 felosztásos módszerrel való kísérleteim eredményeit, sikereit taglalom.

1. fejezet

Az utazó ügynök probléma

1.1. A probléma

Adott n város melyeket úgy akarunk bejárni, hogy mindent várost egyszer érintünk, és visszaérünk a kezdő városba, úgy hogy az útiköltség a lehető legkisebb.

1.1.1. A probléma felépítése

A városokat élek kötik össze, és minden két város közötti élnek van költsége, ami a klasszikus esetben mindkét irányban azonos. Tehát $A \rightarrow B$ költsége megegyezik $A \leftarrow B$ költségével. Ahogy a sok város össze lesz kötve egymással, úgy gráfot alkotnak, amelyekből kétfajta lehet: teljesen összefüggő, illetve nem teljesen összefüggő. Ha minden városból tart él minden városba, akkor teljesen összefüggő gráfról, ha pedig egy-egy városból csak egy pár városba tart él, akkor nem teljesen összefüggő gráfokról beszélünk. A probléma megoldása csak az utóbbi esetében kérdőjeleződik meg.

A probléma megoldása csak akkor kérdőjeleződik meg, ha nem teljesen összefüggő gráfról beszélünk. Ez azért van, mert nem feltétlenül lehet ilyenkor bejárni az összes várost úgy, hogy mindet csak egyszer érintenénk. Ahhoz, hogy egy nem teljesen összefüggő gráfot be lehessen járni úgy, hogy minden várost csak egyszer érintünk, kell benne lennie egy Hamilton-körnek. Ellenkező esetben a gráfot nem lehet bejárni az adott szabályokkal, így a problémának nincs megoldása.

1.1. Definíció. A G gráf egy körét (útját) Hamilton-körnek (Hamilton-útnak) nevezzük, ha a gráf minden csúcsán átmegy. [1]

1.1.2. Számítási nehézség

Az utazó ügynök probléma az egyik első bizonyított NP-nehez probléma. Ezt Karp¹ bizonyította 1972-ben. Új algoritmusok lettek kifejlesztve először, vagy legalábbis hasz-

¹ Richard M. Karp, amerikai informatikus.

nálva, hogy megmutassák a hatékonyságukat. Ilyen például a *Branch and Bound*² algoritmus. [2, 225. oldal]

1.1.3. Egyéb nehézségek

Mint korábban említve volt, a gráfnak tartalmaznia kell minimum egy Hamilton-kört. Ha egy gráf nem teljesen összefüggő, akkor nem feltétlenül tartalmaz Hamilton-kört, azonban léteznek elégséges feltételek a Hamilton-kör létezésére, ilyen például a Dirac-tétel.

1.2. Tétel (Dirac-tétel). *Ha az $n \geq 3$ pontú G egyszerű gráfban $\delta = \frac{n}{2}$ – ahol δ minden csúcs foka –, akkor G tartalmaz Hamilton-kört.[3]*

Én a kutatás során teljesen összefüggő gráfokkal dolgoztam.

1.2. Meglévő algoritmusok

Több algoritmus is létezik az utazó ügynök probléma megoldására, azonban ezekből kettőt fogok kiemelni, amelyekkel a kutatás során dolgoztam.

1.2.1. Nearest Neighbour

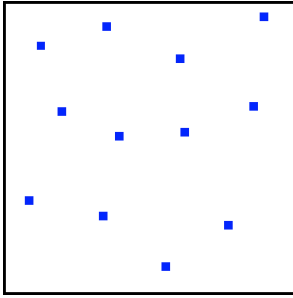
A Nearest Neighbour³ algoritmus egy naívnak nevezett heurisztikus algoritmus. Lényege, hogy egy adott x csúcsból megkeresse a legközelebbi be nem járt csúcsot, mindaddig, ameddig létezik bejáratlan csúcs. Az utolsó bejárt csúcsot összekötjük az első csúccsal. Lépései:

1. Egy csúcs kiválasztása kezdőcsúcsnak
2. Az adatbázisban lévő legközelebbi eddig be nem járt csúcs keresése, majd bejáratlanak megjelölni, és az algoritmus abból a pontból folytatódik
3. 2. lépés ismétlése mindaddig, ameddig az adatbázis nem üres
4. Az utolsó csúcs összekötése a kezdőcsúccsal

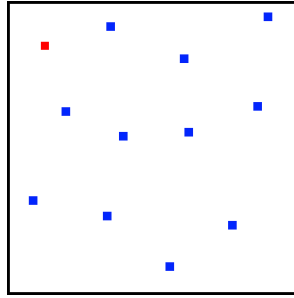
Az algoritmus legnagyobb problémája a naivitása. A minden esetben a legközelebbi csúcs keresése nem egyenlő a lehető legrövidebb úttal. Ez azt jelenti, hogy hiába választja ki a lehetőségek közül a legrövidebb elérést egy csúcsnak, mert lehetnek olyan kitérők, aminek a hatására már csak a jóval távolabbi csúcsok lesznek elérhetőek. Azonban könnyed algoritmusának köszönhetően gyors.

² Magyarul Ág és Korlát algoritmus.

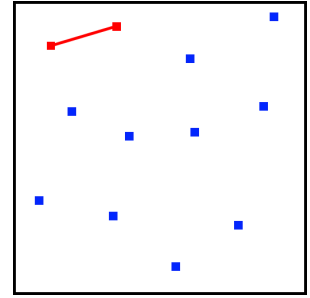
³ Magyarul a Legközelebbi Szomszéd algoritmus.



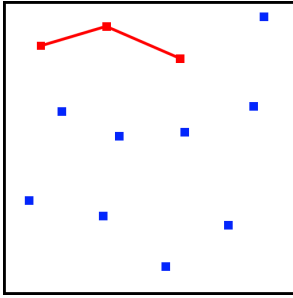
(a)



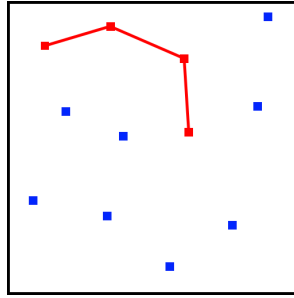
(b)



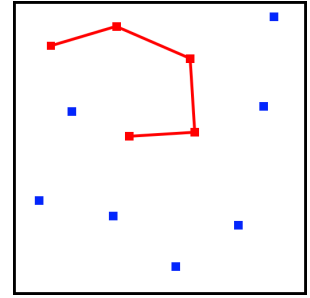
(c)



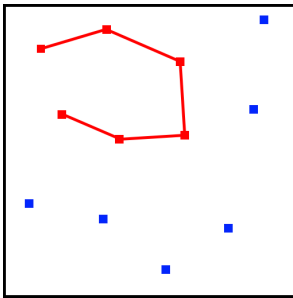
(d)



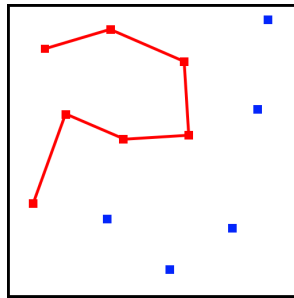
(e)



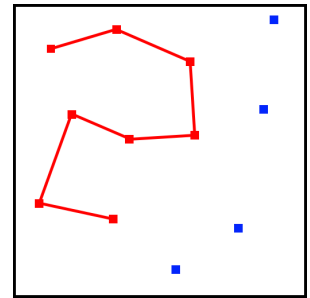
(f)



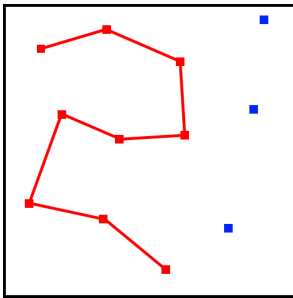
(g)



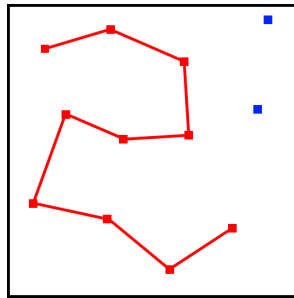
(h)



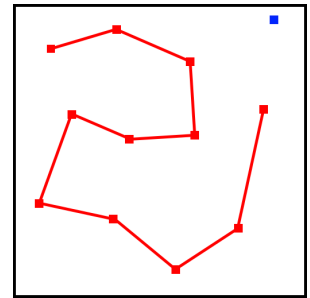
(i)



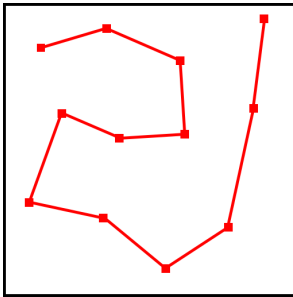
(j)



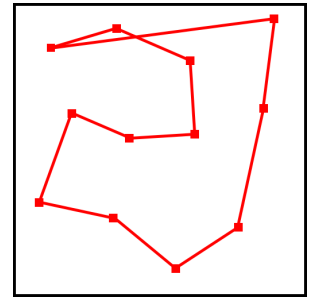
(k)



(l)



(m)



(n)

1.1. ábra. A Nearest Neighbour algoritmus lépései.

Az 1.1. ábrán jól látható, hogy az első és utolsó csúcs közötti távolság szemmel láthatóan nagy. Feltehetően ha az 1.1d. lépésnél a jobb sarokban lévő pontot választotta volna, akkor nem lennének ilyen kitérések, de ez nem garancia rá.

Az algoritmusnak egyik javítása, ha minden bejárás után újraindítjuk, és mindig egy másik kezdőcsúcsot inicializálunk. Azonban minden újraindítás után el kell menteni a már kezdőcsúcsnak jelölt csúcsokat. Ennek következtében az egymásba ágyazott ciklusok száma nő. [4]

1.2.2. Branch and Bound

A Branch and Bound algoritmus alapja a Backtrack algoritmus⁴.

Backtrack ♦ [5] A Backtrack elve, hogy a keresési fát rekurzívan járjuk be a gyökértől kezdve. Minden csomópontban ellenőrzi, hogy az adott csomópont elfogadható-e. Ha nem, akkor az utolsó bejárt csomópontot kitöröljük az adatbázisból, és az azelőtti legutolsó csomóponttól folytatjuk az algoritmus futását, tehát egyet visszalépünk. Ha viszont elfogadja, akkor rekurzívan meghívja önmagát, paraméterében a jelenlegi csomópont összes részfáját adva át.

Ha a jelenlegi csomópontban az állapot a célállapot, így megvan a megoldás. Minden zsákutcából ki tud lépni, azonban ha a keresőfa gyökéréből kellene visszalépünk, akkor a problémának nincs megoldása. Az algoritmus visszalép, ha már nincs szabad operátor egy adott csomópontban, vagy ha a megtett út nagyobb, mint az esetlegesen előre definiált úthosszkorlát.

Branch and Bound ♦ [5] A Branch and Bound a Backtrackkel ellentétben nem fog az első lehetséges megoldásnál kiugrani a keresési ciklusból és leállni. Ha talált egy megoldást, akkor másolatot készít a jelenlegi adatbázisról, majd visszalép és folytatja a keresést. Így akkor ér véget a keresés, ha az adatbázis kiürült. Azonban az úthosszkorlátos Backtrack dinamikus változatát használjuk, hogy ne kelljen minden egyes lehetséges útvonalat bejárunk. A úthosszkorlát akkor változik, ha egy új megoldást találunk, ilyenkor a korlát az új megoldás hossza lesz. Tehát az úthosszkorlát az új megoldás hossza lesz.

Egyértelműen az első futásnál az úthosszkorlátot inicializálni kell: a lehető legnagyobb értéket adjuk neki kezdeti értéként. A Branch and Boundot szokás körfigyeléssel is kombinálni. Az optimális utat állítja elő, azonban nagy időigénye van, mivel minden lehetséges utat próbál bejárni, ameddig az úthosszkorlát engedi.

⁴Magyarul a Visszalépéses keresés.

2. fejezet

Háromdimenziós utazó ügynök probléma

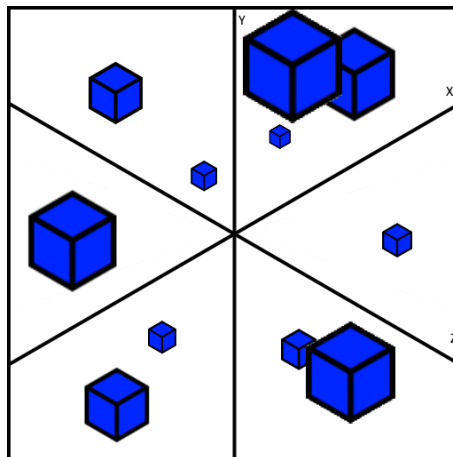
2.1. Alapgondolatok

A problémát háromdimenziós térben bővítjük, így egy extra koordinátát kap minden csomópont. Ez a költségmátrixot is megváltoztatja. A 2.1. ábrán lehet látni, hogy a koordinátarendszeres tájékozódást is megnehezíti a harmadik koordináta hozzáadása.

2.1.1. Elméleti alapok

Mivel háromdimenziós térben mozgunk, az élek költsége nem feltétlenül azonos A -ból B -be és B -ből A -ba, mivel a csúcsok térbeli magasságát is figyelembe kell vennünk; mivel egy dombra költségesebb feljutni, mint lejönni onnan. Ezért az élek irányítottak lesznek, így az $A \rightarrow B$ és a $B \rightarrow A$ költsége eltérhet, de csak akkor, ha a két pont magassága is eltér.

A probléma az, hogy két csomópont euklideszi távolságának mindkét irányban



2.1. ábra. Háromdimenziós utazó ügynök probléma.

ugyanannyi, ezért saját költségfüggvényt vezetünk be. $A \rightarrow B$ költségét a következőképpen számoljuk ki: a két pont közötti euklideszi távolsághoz adjuk hozzá $B.y - A.y$ értéket. Tehát a jelenlegi csomópont és az elérni kívánt csomópont magasságának a különbségét adjuk hozzá a két csomópont közötti alaptávolsághoz.

A klasszikus utazó ügynök probléma esetén minden két csomópont közti távolság egyenlő. Ilyenkor egy költségmátrix szimmetrikus, amire példát a 2.1. táblázatban látnunk. Azonban, mint korábban említve volt, esetünkben két csomópont között az élek költsége eltérő lehet. Ilyenkor a költségmátrix aszimmetrikus, amire a 2.2. ábra mutat egy példát.

0	2	4	5
2	0	3	6
4	3	0	5
5	6	5	0

2.1. táblázat. Szimmetrikus mátrix.

0	1	2	5
2	0	3	3
9	2	0	5
3	6	4	0

2.2. táblázat. Aszimmetrikus mátrix.

2.1.2. Miért lényeges a térbeli vizsgálat?

A klasszikus utazó ügynök probléma légvonalban tekinti a költséget. Térben visszatérhetünk a földre, és nem csak csomópontok közötti távolsággal lehet számolni, hanem a magasságukkal is. Így előfordulhat, hogy két csomópont között kitérünk egy harmadik csomópontra is, mert összességében olcsóbb utat generál.

Térben kalkulálva olcsóbb utakat is találhatunk, mintsem a térképen nézve. Ahhoz, hogy még realisztikusabb modellt alkossunk, két város között több dolgot is számításba lehetne venni, mint például az útminőség vagy a kanyarok száma, íve. Azonban már csak a magasság figyelembevétele is nagyobb fokú realizmust eredményez.

2.2. Implementáció

2.2.1. Keretrendszer

Keretrendszernek a Unity3D-t választottam, ami egy komponens alapú rendszer, főleg játékfejlesztésre. Egy komponens a játékobjektumok funkcionális elemei. Egy-egy

komponens lehet akár egy kód, és ezeket a kódokat magunk meg tudjuk írni C#-ban, amiben lehet hivatkozni a beépített függvényekre, más játékobjektumokra, és a komponenseikre is, ezzel megkönnyítve a programozó dolgát. Alapból egy üres játékobjektumnak mindig van egy komponense, ez pedig a transzform komponens. Ez adja meg, hogy egy-egy objektum hol található, mekkora, és milyen a rotációja.

2.2.2. Generálás

Egy 60×60 -os intervallumban generáltattam le 20 várost. Majd a magasságát Perlin-zaj segítségével adtam meg. A későbbiekben $60 \times 60 \times 60$ -os intervallumon fogom legenerálni a városokat, így a Perlin-zaj elhanyagolható lesz. Ezek a városok egymás klónjai eltérő pozíciókban.

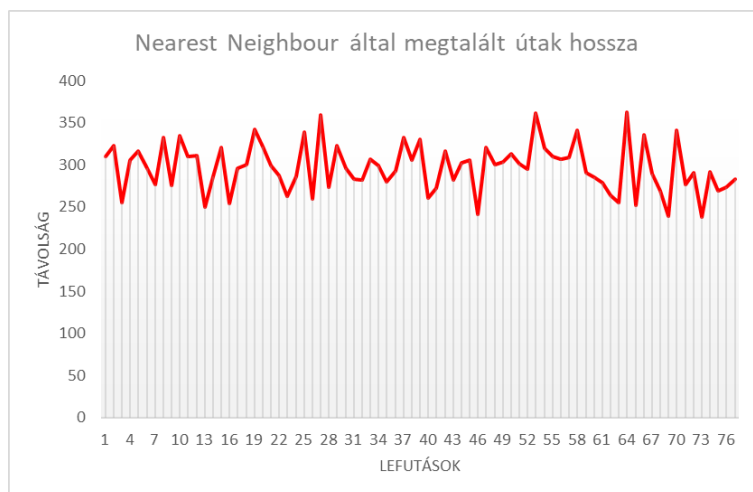
Minden városnak van egy komponens kódja, ami egyfajta kontrollerként viselkedik. Ez a kód tartalmazza az adott várossal összekötött összes város objektum listát, és azokhoz a költség listát. Egy metódus kiszámolja a összes város közötti távolságot a 2.1.1. fejezet alapján.

Miután a városok le lettek generálva, összekötöm mindegyiket a többivel. Egymásba ágyazott ciklusokkal végigmegyek az összes városon és megnézem, melyik város van már összekötve a jelenlegivel. Ha össze vannak kötve, akkor nem csinálok semmit, ha nincs, akkor a két város kontrollerében berakom őket az összekötött városok listájába.

2.2.3. Nearest Neighbour implementálása

Első algoritmusnak a Nearest Neighbour algoritmust implementáltam, amit különböző, véletlenszerű, 20 várost tartalmazó készleteken alkalmaztam. Több, mint 75 lefutás eredménye a 2.2. ábrán látható, ahol 250 és 375 egység közötti össztávolságot talált az algoritmus, melyek egyike sem optimális megoldás.

Az algoritmus itt nem restartos, azaz nincs benne újraindítás, valamint nem mátrixban keresi meg a legközelebbi városokat, hanem a város kontrollében lévő listákkal dolgozik.



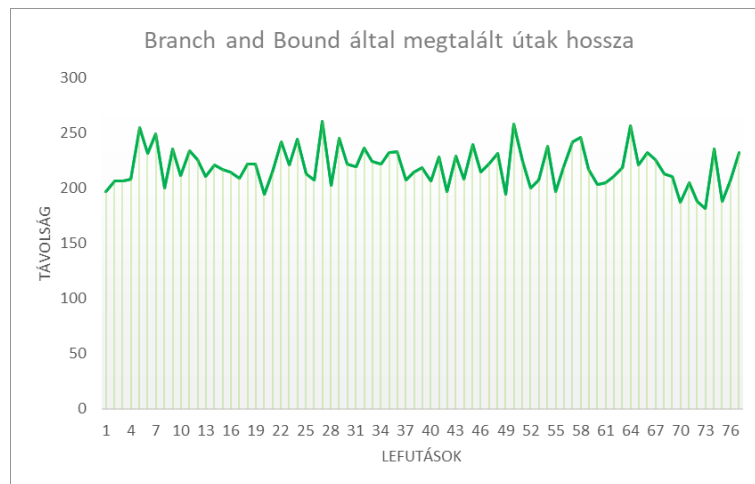
2.2. ábra. A Nearest Neighbour lefutások eredményei.

2.2.4. Branch and Bound implementálása

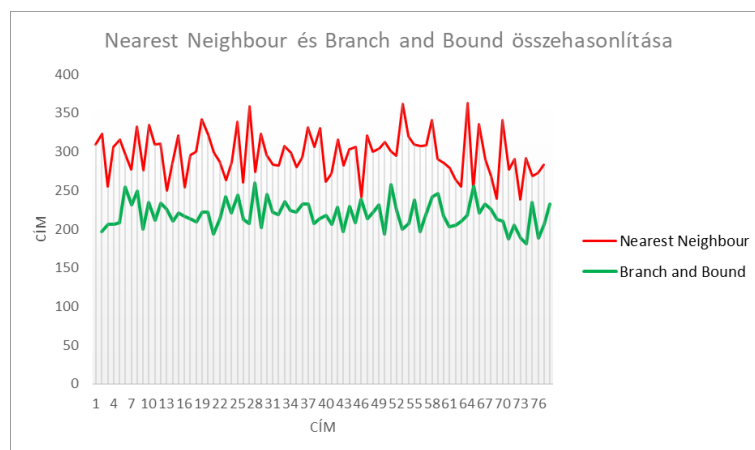
Második algoritmusnak a Branch and Boundot választottam implementálásra, melyet ugyanazokon a készleteken alkalmaztam, mint a Nearest Neighbourt. Több, mint 75 lefutás eredménye a 2.3. ábrán látható, ahol 200 és 275 egység közötti össztávolságot talált az algoritmus.

Az algoritmus hosszú lefutási idővel, de jó eredményeket adott, azonban voltak olyan hosszú lefutások, amikor kénytelen voltam egy 60 másodperces időkorlátot kiszabni, amit többször is túlteljesített volna. Ilyenkor az algoritmus ilyen tekintetben sikertelen volt. A 2.4. ábrán látható egy összehasonlítás a Nearest Neighbourral.

Ellentétben a Nearest Neighbourral, itt mátrixokat használva futtattam az algoritmust, mivel átalakítani a kontrollerek által tartalmazott listabeli elemek összehasonlításához nagyobb munka lett volna, mintsem ezeket a listákat mátrixokba exportálni. Ezért írtam erre egy metódust, amivel az összes város összes listáján végigmenve egy mátrixot generálok, melynek az elemei a városok közötti távolságok.



2.3. ábra. A Branch and Bound lefutások eredményei.



2.4. ábra. A Branch and Bound és Nearest Neighbour lefutások eredményeinek az összehasonlítása.

3. fejezet

Segmented Nearest Neighbour

3.1. Alapgondolatok

A Segmented Nearest Neighbour¹ algoritmus egy általam tervezett és implementált javítása a Nearest Neighbour algoritmusnak. Azért választottam ezt az algoritmust javításra, mert könnyed és gyors. A módosítások, még ha nem is drasztikusan, de nagyban csökkentik a megtalált megoldások útiköltségét, és az időigény csak minimálisan nő.

3.1.1. Elméleti alapok

A meglévő városokat koordináta alapján felosztjuk kisebb bejárható gráfokra, azaz szegmensekre, amelyek között ki- és belépési pontok vannak. Ez azt jelenti, hogy egy-egy szegmensben minden elemet végig kell járnunk ahhoz, hogy a következő szegmensbe léphessünk.

Ezzel a megközelítéssel kiküszöbölhetjük a Nearest Neighbour korábban említett problémáját, amikor az algoritmus a naivitásának köszönhetően eltért egy kedvezőbb lehetőségtől, mivel más csúcspontot talált a legközelebbinek.

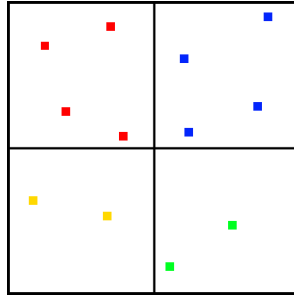
A felosztásnak többféle megközelítése is lehet. Lehet generálási intervallum alapú felosztás is, valamint generálási intervallumhoz legközelebb álló sarok városok koordinátáin alapú felosztás. A jelenlegi gráfot feloszthatjuk többféleképpen; elfelelhetjük két szegmensbe, vagy harmadolhatjuk, akár negyedelhetjük is.

3.2. Felosztás

3.2.1. Generálási intervallum alapú felosztás

Az 1.1. ábrán látható városokat bejárási korlátozottságban 4 szegmensre osztjuk, aminek az eredményét a 3.1. ábra mutatja. Az elv egyszerű: az ismert generálási interval-

¹ Magyarul Szegmentált/Felosztott Legközelebbi Szomszéd algoritmus.



3.1. ábra. Generálási intervallum alapú felosztás.

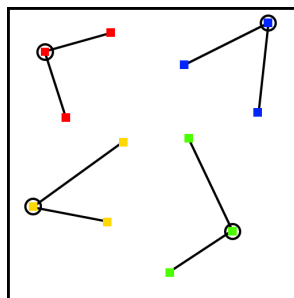
lumok értékét felhasználva megszabjuk, hogy melyik város melyik szegmensbe kerül az elhelyezkedésétől függően. Tehát ha a generálási intervallum 60×60 , akkor az első szegmensbe kerülő városok azok lesznek, amelyeknek az x és y koordinátája kisebb, mint 30. A második szegmens városai pedig azok lesznek, amelyekenek az x koordinátájuk nagyobb, mint 30, azonban az y koordinátájuk kisebb, mint 30, és így tovább.

Az így kapott szegmenseknek megadhatunk egy bejárési sorrendet, amely alapján halad végig a városokon az algoritmus.

3.2.2. Generálási intervallumhoz legközelebb álló sarok városok koordinátáin alapú felosztás

Az 1.1. ábrán látható városokat bejárési korlátozottságban 4 szegmensre osztjuk, aminek az eredményét a 3.2. ábra mutatja. Az elv egyszerű: megkeressük azt a 4 várost, amelyek a generálási intervallumok sarkaihoz állnak a legközelebb, majd megjelöljük azokat, mint sarokelemek. Az összes maradék várost megnézzük, hogy melyik sarokelemhez állnak a legközelebb és azokhoz rendeljük.

Az előző felosztáshoz képest nagyobb időigényű, azonban jobb eloszlású, egyenletesebb felosztást eredményez.



3.2. ábra. Generálási intervallumhoz legközelebb álló sarok városok koordinátáin alapú felosztás.

3.3. Az algoritmus

Elsőként az algoritmus lépéseit ismertetem, majd a 3.3.6. fejezetben magyarázom a 3.3. ábra alapján.

3.3.1. Legutolsónak bejárt él megkeresése

Első lépésnek megkérdőjelezhető, de lényeges ahhoz, hogy elkerüljük az 1.1n. ábrán látható hosszú bejárt élt az első és utolsó város között. Csak egyszer futtatjuk le ezt a lépést, és azt is csak az első szegmensben, még mielőtt elindítottuk volna a keresést. A legutolsó városnak érdemes a kezdővárostól a lehető legtávolabbi várost megkeresni az első szegmensben, és azt megjelölni, mint utolsó város.

3.3.2. Szegmens bejárása

Miután megvolt az utolsónak bejárható város megkeresése, elindíthatjuk a szegmens bejárását. Az algoritmus ugyanaz, mint a Nearest Neighbour, azzal a megkötéssel, hogy csak az adott szegmens elemeiben kereshet legközelebbi várost. Miután a szegmenst bejárta, akkor ugrik a következő szegmensre.

3.3.3. Következő szegmens kiválasztása

Miután bejárt egy adott szegmenst az algoritmust, akkor ugrik a következőre, de melyikre? Akár megtarthatjuk az alapsorrendet, és sorban végigmegy a szegmenseken. De akár úgy is el lehet dönteni, hogy melyik szegmenshez vagyunk közelebb. Ez megbonyolítja a koncepciót, mert ha $\delta \geq 4$ – tehát a szegmensek száma 4 vagy annál nagyobb –, akkor átléphet egy szembelevő szegmensbe, amivel azt érjük el, amit el akartunk kerülni: olyan két csomópontot járunk be, melyek között a távolság nagy. Ha eldöntöttük, mi alapján választjuk meg a következő szegmenst, akkor bejárjuk.

3.3.4. Befejezés

Ha az utolsó szegmenst is bejártuk, és nincs létező bejárható város, akkor az utolsónak bejárt várost összekötjük az utolsónak kijelölt várossal, amit összekötünk a kezdővárossal.

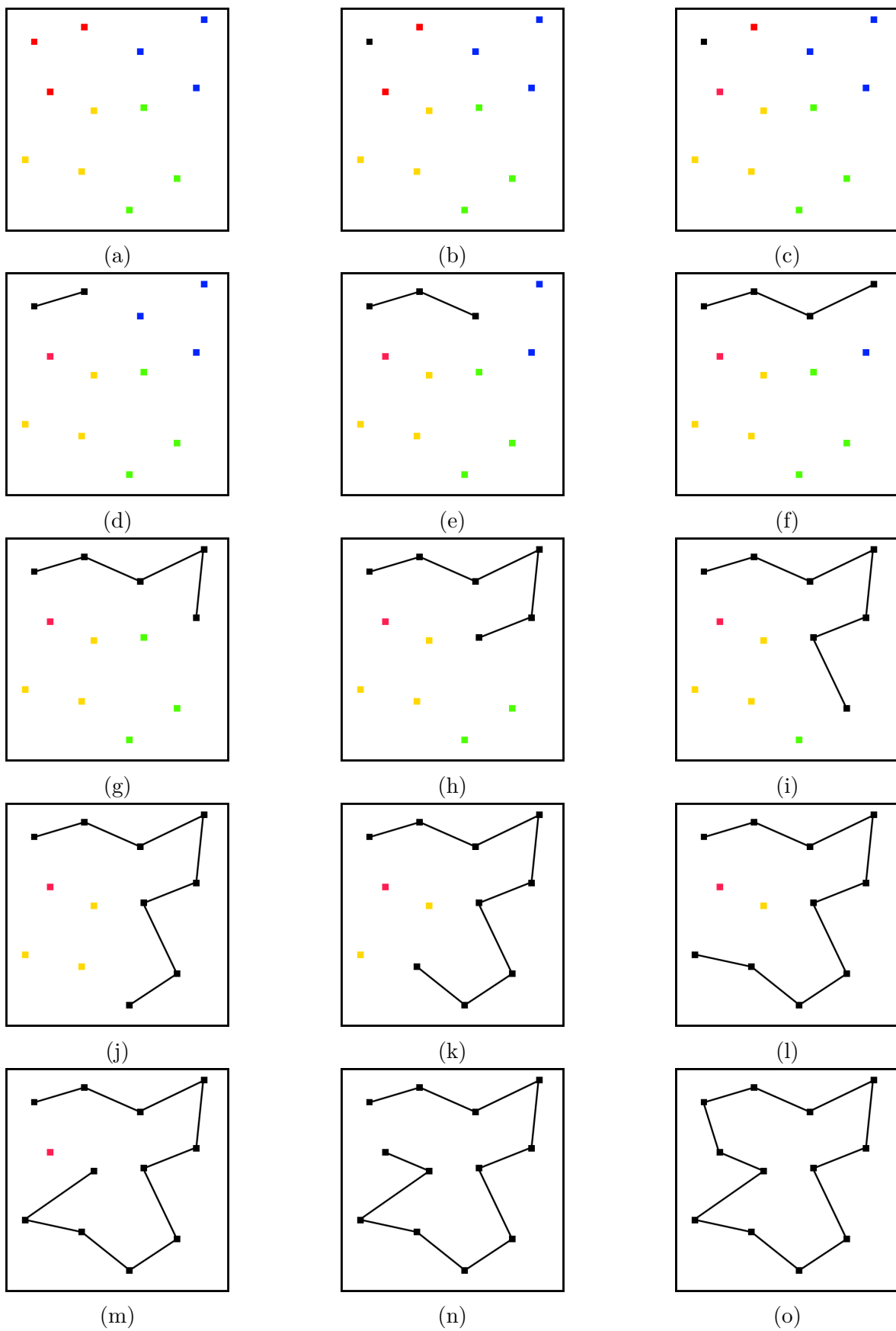
3.3.5. Javítási lehetőség

Ha a kezdő szegmens elemeinek száma több, mint három, és a következő szegmens kiválasztása sorrendi alapú, akkor megkeressük, hogy melyik az a kettő város, amely

eltérő szegmensben van, és a legközelebb állnak egymáshoz. Tehát megnézzük, hogy az A szegmens városai közül melyik van legközelebb B szegmens egyik városához.

3.3.6. Az algoritmus magyarázata

A 3.3a. ábra a gráfot mutatja közvetlenül a szegmensekre való felosztás után. A 3.3b. ábrán kijelöljük a kezdőcsúcsot (feketével), amely az egyik szegmens sarokeleme. A 3.3c. ábrán megjelöljük az utolsó bejárható csomópontot (pinkkel). Ez azért szükséges, hogy az utolsónak bejárt csomópont és az utolsónak bejárható csomópont közötti távolság minimális legyen. Tehát az első szegmensben ezt a csomópontot nem járhatjuk be, csak az utolsó szegmensben az utolsónak bejárt csomóponttal lesz összeköttetésben.



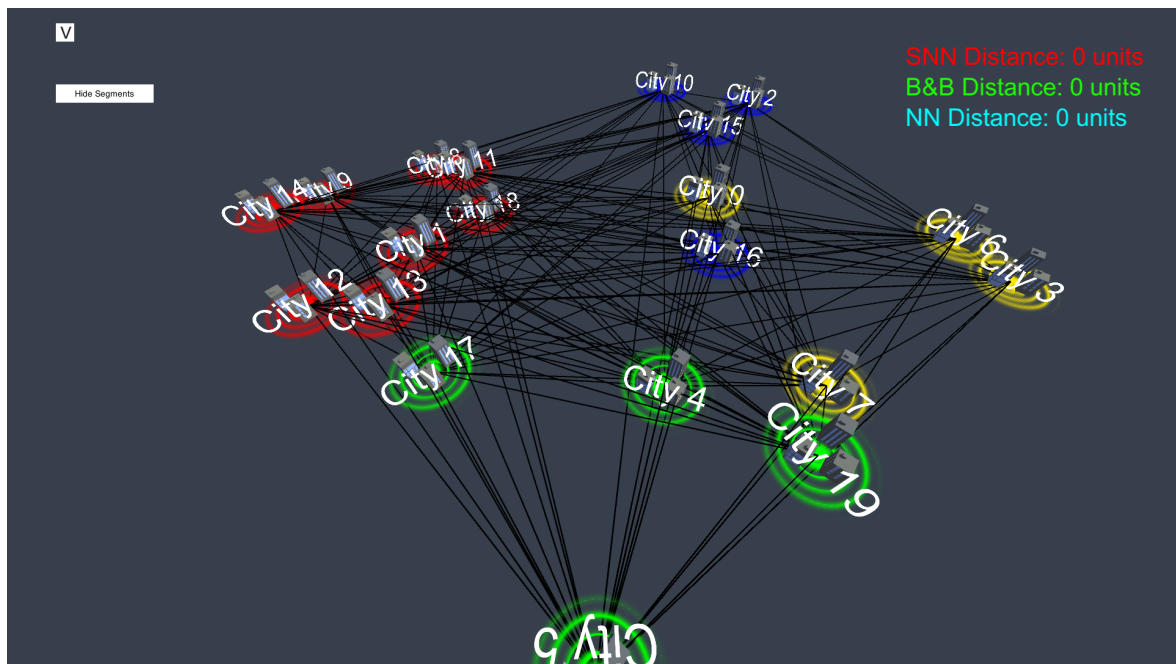
3.3. ábra. A Segmented Nearest Neighbour algoritmus lépései.

A 3.3d. ábrán az első szegmensben bejárjuk az összes bejárható csomópontot, de mivel itt már csak egy darab bejárható csomópont létezik, a következő lépésben átugrunk a következő szegmensre, ahogy a 3.3e. ábrán lehet látni. Ezután addig ismételjük ezt a lépéssorozatot, míg az utolsó szegmens utolsó bejárható csomópontját be nem járjuk, amit a 3.3m). ábra mutat. Ezután az utolsónak bejárható csomópontot is bejárjuk (3.3n. ábra), majd pedig visszatérünk a kezdővárosba (3.3o. ábra).

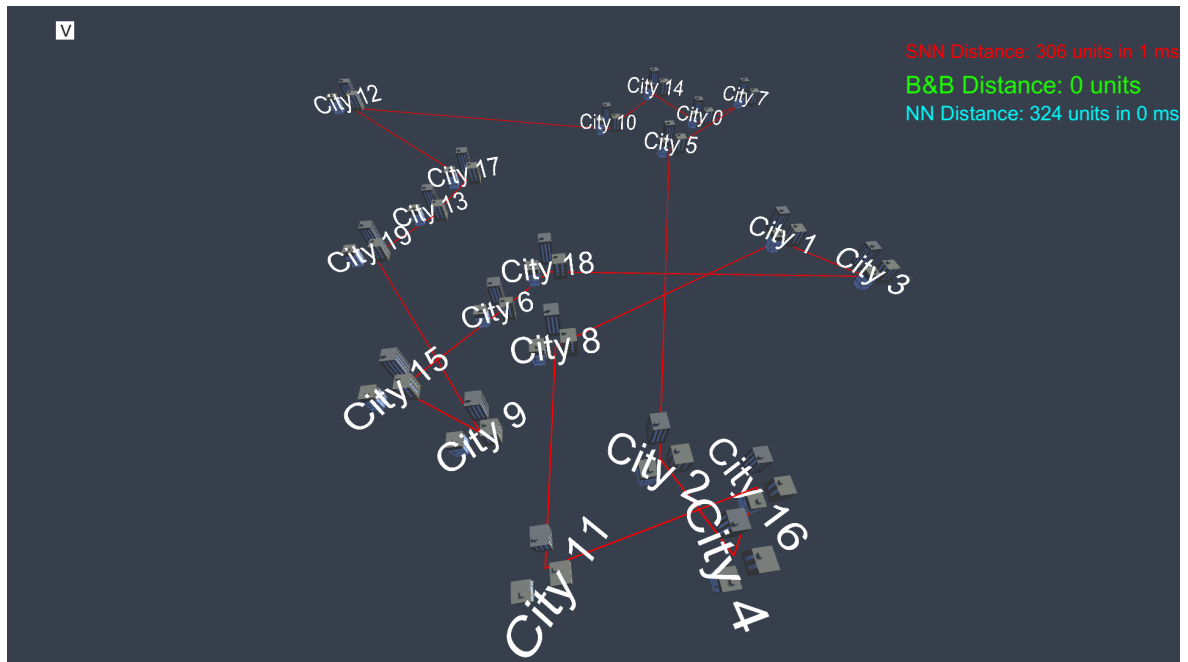
3.4. Négy szegmenses felosztás alkalmazása három dimenzióban

3.4.1. Felosztás

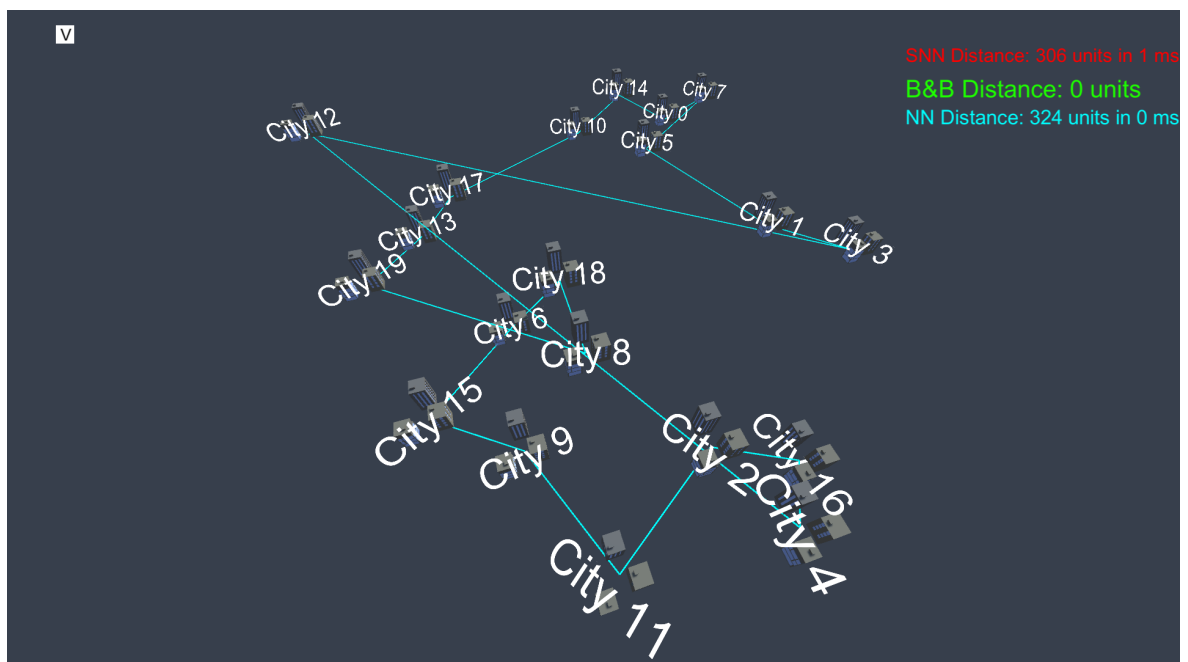
A felosztásnál már a magasságot is figyelembe vettem, ennek következtében az általam készített vizualizációs szoftver által mutatott felülnézetben (3.4. ábra) látszólagosan egy szegmens elemei közé belekerülhet egy másik szegmens eleme is; mindez azonban csupán a két dimenzióra való vetítés miatt látszik így.



3.4. ábra. Szegmensek látszólag összeolvadása.



3.5. ábra. A Segmented Nearest Neighbour bejárása.

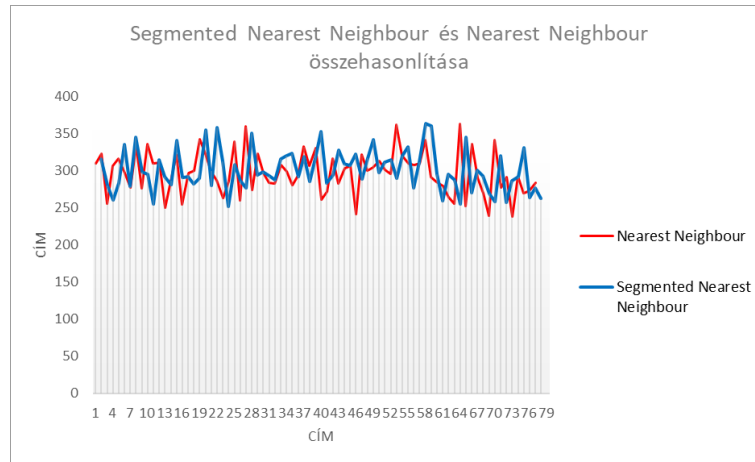


3.6. ábra. A Nearest Neighbour bejárása.

3.4.2. Összehasonlítás a Nearest Neighbourral

Ugyanazokon a készleteken lefuttattam az algoritmust, a kezdő város mindkét algoritmusnál ugyanaz volt. Ahogy a 3.7. ábrán látható, vegyes, de biztató eredményeket kaptam. Voltak olyan esetek, melyeknél a Segmented Nearest Neighbour sokkal rövidebb utat talált, mint a Nearest Neighbour, azonban voltak olyan megoldások, amelyek

hosszabbak voltak. Az sem volt idegen, hogy ugyanazt az utat járta be a két algoritmus, illetve az sem, hogy más utat, de ugyanolyan távolságot jártak be.



3.7. ábra. A Segmented Nearest Neighbour és Nearest Neighbour lefutások eredményeinek az összehasonlítása.

Összességében, ahogy a 3.7. ábrán is látható, sokszor talált olcsóbb megoldást a Segmented Nearest Neighbour, mint a Nearest Neighbour algoritmus. Ezzel arra a következtetésre juthatunk, hogy az algoritmusnak van jövője.

A jobb lefutásra mutatnak példát a 3.5. és a 3.6. ábrák, ahol ugyanazon a problémán lett lefuttatva mindkét algoritmus. A 3.5. ábrán lehet látni, hogy a Segmented Nearest Neighbour rendezetten járja be a városokat, míg a 3.6. ábrán a Nearest Neighbour közismert hibáját lehet látni: a hosszú utolsónak bejárható élek használatát. Mindkét algoritmus átlag lefutási ideje 0-5 ms közé tehető.

3.4.3. Javítási lehetőségek

A tapasztalatokat levonva a Segmented Nearest Neighbour algoritmust úgy lehet javítani, ha a felosztást is háromdimenzióban végzem el. Azonban ehhez növelni fogom a városok mennyiségét és, mint korábban említettem, a generálásnál $60 \times 60 \times 60$ -os intervallumot fogok bevezetni. A felosztás a három dimenzióban úgy fog kinézni, hogy 4 szegmens helyett 8 szegmens lesz. Ezt úgy kell elképzelnünk, mintha egy kockát 8 apróbb kockára osztanánk fel.

3.5. Nyolc szegmenses felosztás alkalmazása három dimenzióban

3.5.1. Generálási módosítás

Mint korábban említettem, elvetem a feleslegesnek bizonyuló Perlin-zaj alapú magasság generálását, és áttérek a $60 \times 60 \times 60$ -as generálási intervallumra; valamint 20 város problémák helyett 30 város problémákat fogok generálni.

3.5.2. Felosztás

A felosztás elve nem változik, azonban 4 szegmens helyett már 8 kell. Tehát 8 sarokelem lesz, melyek a generálási intervallum alapján lesznek meghatározva.

3.5.3. Észrevételek

Miután létrehoztam a nyolc szegmensre való felosztást, egy alap sorrenden lefuttattam az algoritmust. Az eredmények azonosak voltak, mint a négy szegmenses felosztásnál, azonos aránnyal. Nyilván a több város és a nagyobb generálási intervallum miatt megnőtt a bejárt út hossza, azonban azt vettem figyelembe, mikor produkált rövidebb megoldást a Segmented Nearest Neighbour.

Mivel az eredmények nem javultak annyira, amennyire elvártam, ezért egy javítási lehetőséget implementáltam az algoritmusban.

3.5.4. Implementált javítás

A 3.3.5. pontban említett javítást implementáltam: az alap sorrend alapján a jelenlegi, éppen érintett szegmensben meghatározom a kilépési csomópontot aszerint, hogy melyik csomópont áll a legközelebb a következő szegmenshez.

Ez a javítás a vártnál jobb eredményeket hozott, azonban átlagban még mindig feleannyiszor volt kedvezőbb az algoritmus által szolgáltatott megoldás.

3.5.5. További észrevételek

Sokadik lefutásnál észrevettem, hogy a megkötések miatt rosszabbul teljesít az algoritmus bizonyos helyzetekben, vagy sokkal jobban teljesít a megkötések miatt. A Segmented Nearest Neighbour előnyben volt, amikor a bejárési sorrendhez igazodott a generálás, illetve akkor, amikor a Nearest Neighbour-nak költségesebb volt egy-egy élbejárása.

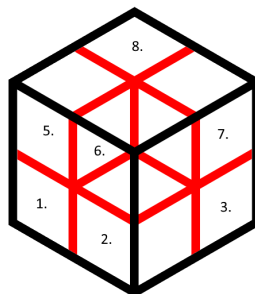
Különböző bejárési sorrendekkel próbálkoztam, aminek nyomán találtam olyan eseteket, melyek jobb eredményt adtak, illetve fordítva is: voltak olyan sorrendek is, melyeknél jóval csökkent a kedvező lefutások száma.

3.5.6. Megoldás

Mivel több jó sorrend is létezik és mindegyik egy-egy bizonyos helyzetben alkalmazható a generálás végett, ezért összegyűjtöttem 10 sorrend mintát, amikkel egymás után lefuttattam az algoritmust, majd a megoldásaik közül kiválasztottam a legrövidebbet. Mindez lassítja az algoritmust, de nem drasztikusan, mivel egyszeri lefutása 0-5 ms közé tehető, így az átlagos lefutási idő nem lesz nagyobb, mint 50 ms.

A bejárési sorrendek a következők (melyhez a 3.8. ábrán lehet látni a szegmensek számozását):

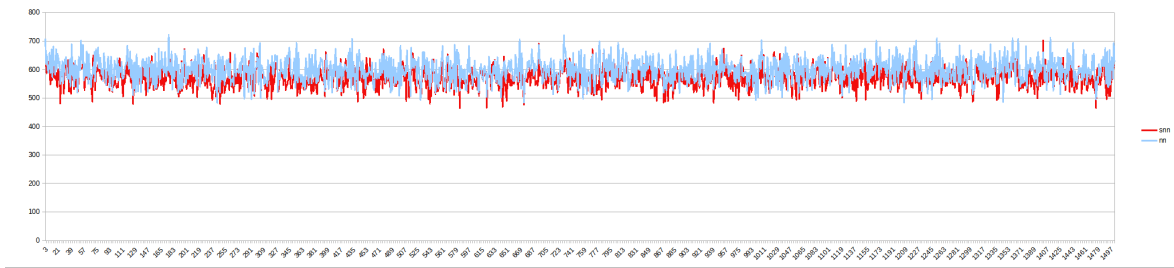
1. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 1$
2. $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 5 \rightarrow 1$
3. $1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1$
4. $1 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 6 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$
5. $1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 5 \rightarrow 1$
6. $1 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 1$
7. $1 \rightarrow 5 \rightarrow 6 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 4 \rightarrow 1$
8. $1 \rightarrow 5 \rightarrow 8 \rightarrow 4 \rightarrow 3 \rightarrow 7 \rightarrow 6 \rightarrow 2 \rightarrow 1$
9. $1 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 8 \rightarrow 4 \rightarrow 1$
10. $1 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 6 \rightarrow 2 \rightarrow 1$



3.8. ábra. A nyolc szegmenses felosztás vizualizációja. (Nem látható a 4. szegmens.)

Ezzel drasztikusan jobbak lettek a lefutási eredmények. Az algoritmussal elvégeztem 1500 lefutást, a kapott eredményeket összehasonlítottam a Nearest Neighbourral

eredményeivel, ahogy azt a 3.9. ábrán lehet látni. Az 1500 lefutásból 1166-szor adott jobb eredmény a Segmented Nearest Neighbour.



3.9. ábra. A Segmented Nearest Neighbour és Nearest Neighbour lefutások eredményeinek az összehasonlítása.

3.5.7. Konklúzió

Mivel az algoritmus alapja a Nearest Neighbour algoritmus, ezért nem lehet szabadjárá engedni úgy, hogy önmaga kiválassza a következő szegmenst. Ez azért van, mert olyan szegmensbejárési sorrendeket generálhat, amelyek keresztezik önmagukat, illetve távoli csomópontokat járhatnak be, ha nem lenne más lehetőségük. Ezért ezt is korlátoznunk kell, mégpedig előre definiált szegmensbejárési sorrendekkel.

A sorrendi megkötésekkel és a próbálgatásukkal bizonyított, hogy lényeges a bejárési sorrend a szegmensek között, és több ilyen minta létezik.

Ez több kérdést felvet: Hány ilyen minta létezik? Mi alapján kell eldönteni, hogy melyik mintát érdemes használni? Muszáj mintákat keresni, vagy egyfajta gráfvizsgálat után létre lehet hozni minden egyes problémára egy egyéni kedvező bejárást?

Az összes minta használata nem kifizetődő, hiszen ha az első szegmenst nem számoljuk, akkor is $7!$ szegmensbejárési sorrend minta létezik. Azonban ha nem vizsgálunk meg minden szegmensbejárési mintát, akkor fennáll a veszélye, hogy hosszabb utat találunk, mint a Nearest Neighbour.

Ezért minden tekintetben – az újraindítással szemben – a legkifizetődőbb megoldás lenne egy gráfvizsgálat után létrehozni egy szegmensbejárési sorrendet. Azonban maga az alkalmazandó gráfvizsgálat is lehet nagyon költséges.

Összegzés

Eleinte a kutatás két algoritmus összevetése volt, ám sokkal nagyobb szabású projekt alakult ki ebből a véleményem szerint. Céлом a Nearest Neighbour algoritmus módosítása volt úgy, hogy csekély időigényért cserébe javítsak a bejárt út költségén. Sikereim közé tartozik, hogy a több hónapnyi kutatás, próbálkozás, tervezés és kísérletezés után a saját algoritmusom, a Segmented Nearest Neighbour sikeresnek bizonyult. Kísérleti úton megmutattam, hogy a Nearest Neighbour algoritmus módosított verziója megbízható és van a jövője. Természetesen nem sikerült 100%-os sikert elérnem. Az algoritmus jelenleg előre definiált szegmensbejárési sorrendekre támaszkodik, és a sorrendek meghatározásának a szabálya jelenleg nem ismert.

Számomra ez egy hatalmas eredmény, amivel még rengeteget tudok foglalkozni annak céljából, hogy a Segmented Nearest Neighbour még kedvezőbb megoldásokat generáljon. Jövőbeli terveim közé tartozik, hogy a felosztási algoritmuson javítsak. Ezt úgy tervezem, hogy a meglévő felosztási algoritmust kibővítem hisztogramos átrendezéssel: megvizsgálom a városok koordinátái alapján, hogy nagy számban hol tömörülnek össze városok, és eszerint egy-egy szegmenshez társítom őket. Ez akkor segíthetne leginkább, amikor két szegmens határvonalán nagy számban helyezkednek el városok, és ezért ezt a sokaságot nem célszerű két különböző szegmensbe osztani. További jövőbeli feladat lehet olyan, viszonylag olcsón vizsgálható gráftulajdonságok meghatározása, melyek segíthetnek előnyös szegmensbejárési útvonalakat meghatározni adaptív módon.

Irodalomjegyzék

- [1] SZŐNYI TAMÁS: *Hamilton-utak, Hamilton-körök*.
- [2] M.O. BALL ET AL: *Eds., Handbooks in OR & MS, vol. 7*, 1995, Elsevier Science B.V.
- [3] MELISSA DELEON: *A Study of Sufficient Conditions for Hamiltonian Cycles*, Department of Mathematics and Computer Science, Seton Hall University, South Orange, New Jersey, 2000.
- [4] GREGORY GUTIN, ANDERS YEO, ALEXEY ZVEROVICH: *Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*, Discrete Applied Mathematics, vol. 117, issues 1–3, 2002, pp. 81-86.
[https://doi.org/10.1016/S0166-218X\(01\)00195-0](https://doi.org/10.1016/S0166-218X(01)00195-0)
- [5] DR. KOVÁSZNAI GERGELY, DR. KUSPER GÁBOR: *A Mesterséges Intelligencia Kérdései a Középiskolai Oktatásban*, Eszterházy Károly Főiskola, Matematikai és Informatikai Intézet, 2009.