Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Általános Informatikai Intézeti Tanszék

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék



Térképmodellek generálása ágens alapú módszerekkel

Diplomamunka

Készítette:
Név: Szacsuri Norbert
Neptunkód: `G0XXTC`
Szak: Mérnökinformatikus MSc
Alkalmazás fejlesztő szakirány

Eredetiségi Nyilatkozat

Alulírott **Szacsuri Norbert**; Neptun-kód: `GOXXTC` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Mérnökinformatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Térképmodellek generálása ágens alapú módszerekkel* című diplomamunkám saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy diplomamunka esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén diplomamunkám visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

Tartalomjegyzék

1. Bevezetés **1**

2. Térképmodellek **3**
2.1. Magasságmező generálása 4
2.1.1. A Diamond-square algoritmus 4
2.2. Úthálózatok generálása 5
2.2.1. Optimalitási feltétel 5
2.2.2. Keresési tér 6
2.2.3. Probléma formalizálása – a Steiner-fa 7
2.2.4. Elterjedt megoldási módszerek 8
2.2.5. Intelligencia alapú megoldási módszerek 12

3. Ágens alapú heurisztikák **13**
3.1. Ágens alapú modellek 13
3.2. Ant Colony Optimization (ACO) 16
3.2.1. A modell áttekintése 16
3.2.2. Az algoritmus 18
3.2.3. Összegzés 19
3.2.4. További változatok 20
3.3. Particle Swarm Optimization (PSO) 20
3.3.1. A modell áttekintése 21
3.3.2. Az algoritmus 22
3.3.3. Összegzés 23
3.3.4. További változatok 24
3.4. Artificial Fish Swarm Optimization (AFSO) 25
3.4.1. A modell áttekintése 25
3.4.2. Az algoritmus 25
3.4.3. Változatok 30
3.4.4. Összegzés 30
3.5. Evolved Bat Algorithm (EBA) 31
3.5.1. A modell áttekintése 31
3.5.2. Az algoritmus 32

3.5.3. Összegzés 33

4. Szimulációs keretrendszerek 34
4.1. Az ACO alapú ötlet 34
4.2. Az AFSO keretrendszere 36
4.3. A saját keretrendszer 40
4.3.1. Számítási komplexitás 41
4.3.2. A keretrendszer lépései 42
4.3.3. Értékelés és összegzés 43
4.4. Az alkalmazás 44
4.4.1. Célok meghatározása 44
4.4.2. Tervezés 45
4.4.3. Implementáció 51
4.5. Optimalizálások 60
4.5.1. JavaScript tömbkezelés 60
4.5.2. Web worker 62
4.5.3. Legrövidebb útszámítás 64
4.5.4. Elsőbbségi sor 65

5. Paraméterek hangolása 69
5.1. A vizsgálatok menete 69
5.2. ACO paramétereinek hangolása 70
5.2.1. A feromonok párolgási együtthatója 70
5.2.2. A feromonok vonzósági együtthatója 71
5.3. PSO paramétereinek hangolása 72
5.3.1. A tanulási együttható 72
5.3.2. A csoport legjobb pozíciójának együtthatója 73
5.3.3. A részecske legjobb pozíciójának együtthatója 74
5.3.4. Az előző mozgási vektor együtthatója 74
5.4. AFSO paramétereinek hangolása 75
5.4.1. A raj méretének paramétere 75
5.4.2. A maximális lépésköz paramétere 76
5.4.3. Az észlelési távolság paramétere 77
5.5. EBA paramétereinek hangolása 78
5.5.1. A PulseRate paraméter 78
5.5.2. A véletlen lépés távolságának paramétere 79
5.6. A módszerek összehasonlítása 80

6. Összegzés 82

7. Summary 84

1. fejezet

Bevezetés

A modern grafikus alkalmazások, azon belül is kifejezetten a számítógépes játékok és szimulációk esetében hatalmas mennyiségű tartalomra van szükség. Az egyik ilyen tartalom, amire rendszerint szükség van, az a teret valamely módon leíró térképmodellek. A térképmodellek elkészítése a hagyományos modellezési folyamatokkal (manuálisan, kézzel) már túlságosan időigényes és költséges, így ezt igyekeznek teljesen, vagy részben automatizáltan generálni, valamilyen algoritmus, vagy program segítségével.

A dolgozat elsősorban a térképmodellek, azon belül is kifejezetten az úthálózatok, ágensekkel történő generálásával foglalkozik. Ennek kapcsán először bemutatásra kerülnek a térképmodellek sajátosságai, mely során ismertetek néhány fontosabb alapfogalmat. Ezután a feladatot két részre bontom: egy magasságmező generálására és egy úthálózat generálására.

A magasságmező generálásához bemutatásra kerül a játékfejlesztők körében elterjedt *diamond-square* algoritmus. Az úthálózat generálási probléma az előzőnél részletesebben is kifejtésre kerül, mely során végül eljutunk a minimális Steiner-fa problémához. Ezt követően áttekintem ennek kapcsán az elterjedt, nem ágens alapú algoritmusokat és azok működését.

A dolgozat további részében már célzottan az ágens alapú módszerekkel foglalkozom. Először is felvezetésre kerülnek az ágens alapú modellek és azok sajátosságai, majd megvizsgálok néhány elterjedtebb módszert, melyekkel optimalizálási problémákat lehet megoldani. Ehhez kiválasztásra kerülnek az ACO (Ant Colony Optimization), a PSO (Particle Swarm Optimization), az AFSO (Artificial Fish Swarm Optimization) és az EBA (Evolved Bat Algorithm) módszerek. Mindegyik kiválasztott módszert részletesen bemutatom, taglalva azok sajátosságait, matematikai modelljét, algoritmusának lépéseit, előnyeit, hátrányait, valamint további változatait.

Az ágens alapú optimalizálási módszerek leírása után, felvezetek 2 különböző keretrendszer ötletet a minimális Steiner-fa probléma megoldásához. Ezek esetén szót ejtek, az algoritmus lépései mellett, az előnyeikről és hátrányaikról, valamint a felmerülő problémákról. Ezután bemutatok egy saját ötletből származó keretrendszert is, melyet össze is hasonlítok az előző kettővel.

A dolgozat során elkészítésre kerül egy böngészőben futtatható alkalmazás is, amivel a különböző ágens alapú módszereket lehet kipróbálni. Ehhez felhasználásra kerülnek az említett ágens alapú optimalizáló algoritmusok és a dolgozatban utolsóként szereplő keretrendszer. A program TypeScript nyelven íródott. A program tervezésének és implementálásának menetét részletesen bemutatom egy-egy szakaszban. Az elkészített forrásfájlok közül viszont csak a fontosabbak kerülnek bemutatásra a dolgozatban (terjedelmi okok miatt).

Külön szót ejtek még az alkalmazás során figyelembe vett optimalizálási lehetőségekről. Ezek során megvizsgálok különböző mátrix kezelési módszereket, legrövidebb útszámítási algoritmusokat és elsőbbségi sor adatszerkezeteket.

A dolgozat végén az elkészült módszerek paramétereit vizsgálom. Ezekhez számos tesztfuttatás alapján meghatározom az optimálisnak vélt paraméterértékeket, valamint össze is hasonlítom azok teljesítményét. Az így elért eredményeket diagramok segítségével szemléltetem.

2. fejezet

Térképmodellek

A mai világban számos témakör kapcsán találkozhatunk térképekkel, legyen akár szó közlekedésről, építészetről, természetjárásról, vagy számítógépes játékokról. Modern megközelítésben térképnek nevezhetünk minden térbeli vonatkozások mértékhez kötött rendezett modelljét [1]. Ezek a térbeli vonatkozások számos dolgot magukba foglalhatnak, attól függően, hogy mit szeretnénk ábrázolni, lehetnek például:

- magasság értékek (magasságmező),
- útvonal adatok (úthálózat),
- vízhálózati adatok (folyók, tavak),
- növényzettel kapcsolatos adatok,
- látványosságok / érdekességek pozíciói (például turistatérképen).

A magasságmező alatt itt a tér egy diszkretizált rácsra történő illesztését értem, ahol minden egyes rácsponthoz meghatározott a felület aktuális magasságértéke. Ez a rácsháló gyakorlatilag egy mátrixot fog eredményezni, amit akár képként is lehet kezelni, ábrázolni.

Az említett térbeli vonatkozásokat akár külön rétegekként is el lehet képzelni, melyekből mindig csak a számunkra fontosakat jelenítjük meg. Viszont egy ilyen modellnél a legtöbb réteget nem lehet teljesen függetlennek tekinteni a többitől, hiszen például a magasságértékek nagyban befolyásolják az adott terület növényzetét, épp úgy, ahogy az útvonalak elhelyezkedését is. Amiatt, hogy ezek nem tekinthetők függetleneknek, korántsem triviális és elég időigényes a valósághű és az egyéb igényeket is kielégítő térképmodellek manuális elkészítése. Ez azokon a területeken, ahol az idő kritikus szempont, mint például a játékiparban, nem fogadható el, ezért amennyire lehet automatizálni próbálják.

A fejezet további részében a térképmodellek számítógéppel történő generálásával, azon belül is főleg az optimális úthálózatok készítésével fogok foglalkozni.

2.1. Magasságmező generálása

Az úthálózatok elhelyezkedése nagyban függ a terület magasságértékeitől, így az úthálózat generálásához először is szükségünk lesz egy magasságmezőre, hogy legyen min elhelyeznünk az útvonalakat. Abban az esetben, ha nem áll rendelkezésünkre magasságmező, akkor azt vehetjük úgy, hogy egy síkon szeretnénk útvonalakat elhelyezni. Ha ez nem felel meg nekünk, akkor akár generálhatunk is egy magasságmezőt. A magasságmezők generálására már számos algoritmust létrehoztak. A dolgozat elsősorban az úthálózatok generálásával foglalkozik, ezért a magasságmező generálására csak érintőlegesen tér ki.

A legtöbb algoritmus valamilyen zaj segítségével (például Gauss, vagy Perlin zajt felhasználva) állítja elő a magasságmezőt [2]. Mivel ezek a zajok önmagukban túlságosan véletlenszerű felületeket produkálnának, így további lépések szükségesek azok finomítására. Egy ilyen módszer például a *Diamond-square* algoritmus, melyet a játékfejlesztők előszeretettel alkalmaznak gyorsasága és valósághű eredménye miatt.

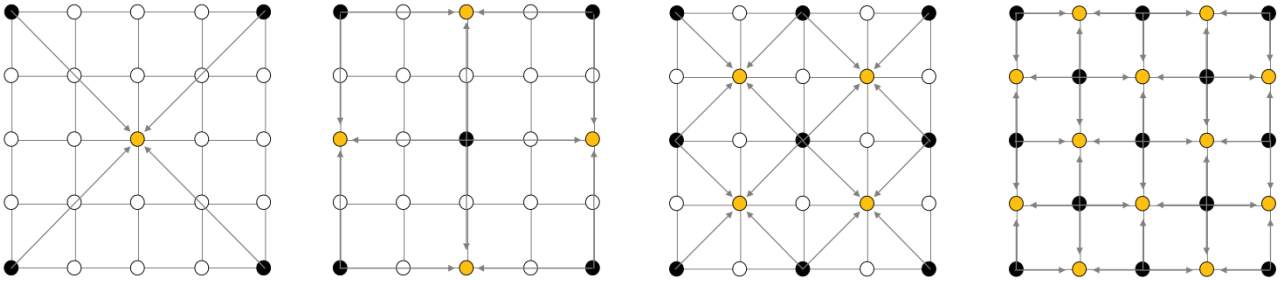
2.1.1. A Diamond-square algoritmus

Az algoritmus 2 egyszerű lépéssel generálja ki számunkra a magasságmezőt, melyhez elsőként meg kell adnunk a térkép 4 sarkának magasságértékét. Ezt követően felváltva végrehajtja a **diamond** és a **square** lépéseket, melyek segítségével végül visszaadja az elkészült térképet [3–9].

A **diamond** lépés során a már meghatározott pontok közül kiválasztásra kerülnek azok, amelyek egy a koordináta tengelyekkel meghatározott négyzet sarkaira illeszkednek. Ha ennek a négyzetnek a középpontjában lévő pont értéke még nincs meghatározva, akkor ez az érték a négyzet sarkainak átlaga lesz, megnövelve egy meghatározott eloszlásból származó véletlen számmal. Erre a véletlen számra azért van szükség, hogy változatos felszínt kapjunk, azonos kezdeti értékek esetén is.

A **square** lépés hasonló, mint az előző, csak itt nem a koordináta tengelyekkel meghatározott négyzetre kell illeszkednie a 4 pontnak, hanem a 45° fokkal elforgatott tengelyekre. Másképp fogalmazva: a vízszintes és függőleges irányok helyett most az átlós irányokat kell vizsgálni.

Ez a két lépés váltakozva végül megadja a térkép összes pontjának az értékét. Ahhoz, hogy a négyzetre illesztés és a közbenső pont kiválasztása mindig elvégezhető legyen, csak $2^n + 1$ méretű 2D-s magasságmezőkkel tud számolni az algoritmus. Amennyiben nekünk ettől eltérő méretekre lenne szükségünk, generálhatunk egy nagyobb méretű magasságmezőt, melyet utólag a megfelelő méretre tudunk vágni. Az algoritmus lépéseit jól szemlélteti a 2.1. ábra, illetve a 2.2. ábrán látható egy példa a végeredményre is.



2.1. ábra. A Diamond-square algoritmus lépései, kezdve a diamond lépéssel [10].

2.2. Úthálózatok generálása

A dolgozat megírása során nagyrészt az úthálózatok generálásával foglalkoztam. Habár egyszerűnek tűnik a feladat, néhány feltétellel nagyon is bonyolulttá lehet tenni. Az alábbi elvárásokat állítottam a generálandó útvonallal szemben.

- Illeszkedjen a generált magasságmezőre!
- Viszonylag kevés számú várost / cél csomópontot tartalmazzon!
- Legyen optimális!

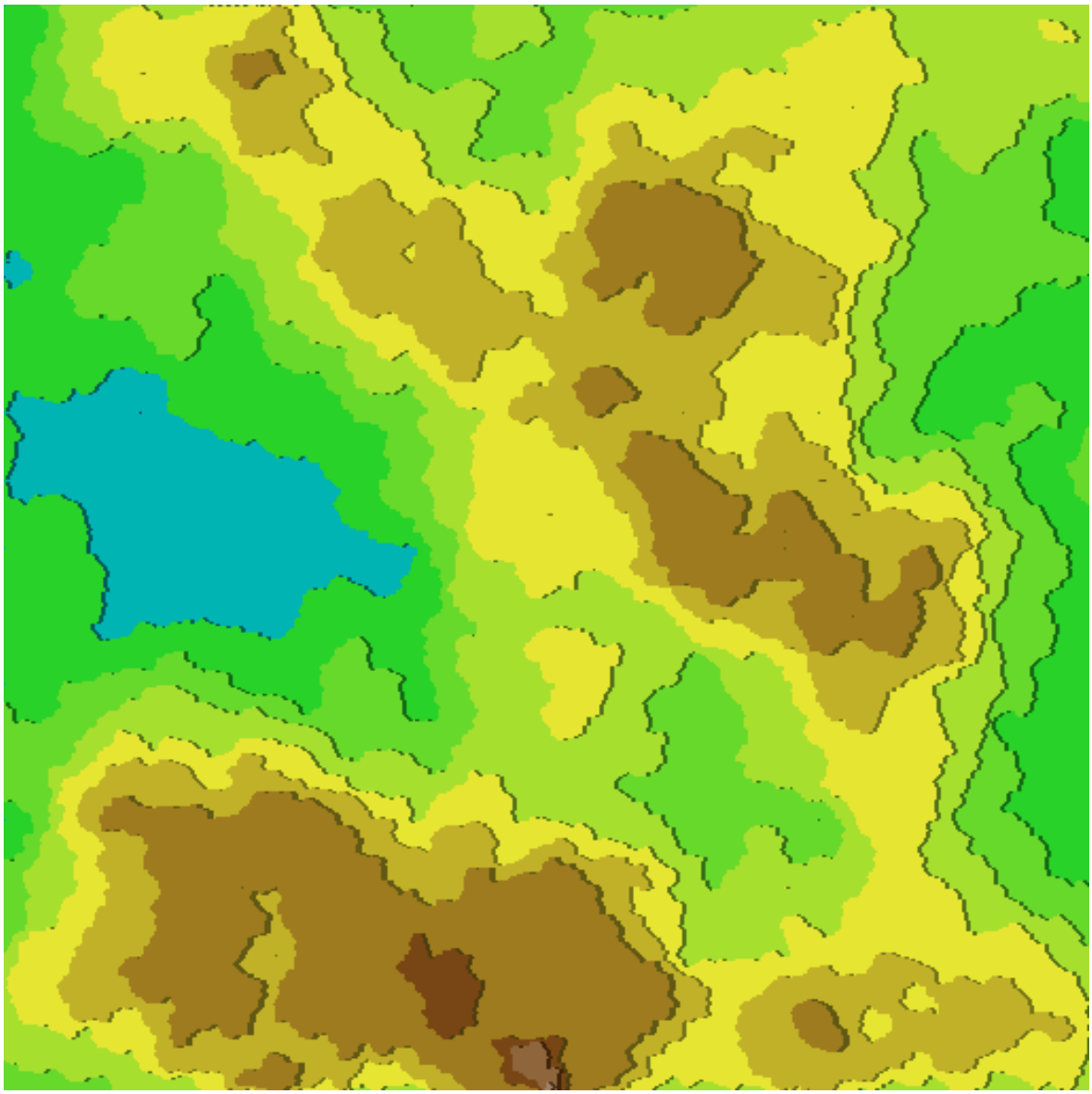
2.2.1. Optimalitási feltétel

Az előzőekben említett feltételek közül az egyedüli, ami problémás lehet az az optimalitás elérése. Ugyanis egy optimális úthálózat meghatározása korántsem triviális feladat. Eleve az sem mindegy, hogy milyen szempontból nézve kell optimálisnak lennie. Lehet például az építési költség, utazási idő, vagy akár egyéb szempont is. Esetemben a minimális építési költséget tekintem az optimalizálási feltételnek, hiszen a valódi utakat is e szerint tervezik és építik. Vegyük például az autópályákat. Ezek gyakran még akkor is kanyarognak amikor a domborzati viszonyok ezt nem igazán indokolják. Ennek oka, hogy habár a domborzat nagyban befolyásolja a költségeket, de azt más szempontok felülmúlhatják. Ilyen lehet például az érintett földek, területek ára, melyeket szükséges lehet megvásárolni, amennyiben magántulajdonban vannak.

Az utazási idő esetére az optimalizálási problémát fel lehetne írni például az alábbi célfüggvény minimalizálásaként is:

$$\text{cost}(X) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{dist}(X_i, X_j),$$

ahol $X = (X_1, X_2, \ldots, X_n)$ a városok pozíciója, a dist pedig két vektor távolságát számolja ki. Ebben az esetben könnyű az optimális megoldást meghatározni, hiszen akkor lesz a legkisebb az utazási idő, ha minden várost minden várossal összekötünk. Ez viszont nem túl valószerű megoldás, a valóságban nem is nagyon találunk rá példát.



2.2. ábra. Példa a Diamond-square algoritmus eredményére *(használt paraméterek: power: 9, corners: [2, 4, 5, 5], offset: 5, range: 16, roughness: 5)*.

Az általam választott feltételhez már nehezebb lenne megadni egy ilyen megfelelő célfüggvényt, így erre most nem térnék ki, a későbbi fejezetekben még úgyis lesz róla szó részletesebben.

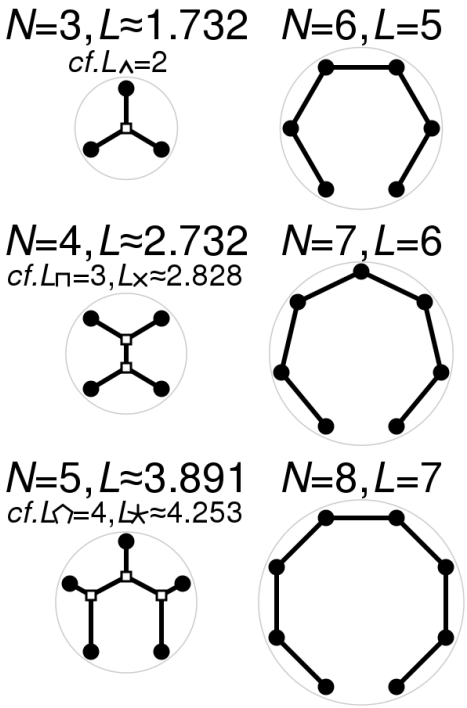
2.2.2. Keresési tér

Az úthálózat generálása során szükséges még figyelembe venni a teret is, amin elhelyezésre kerülnek majd az utak. Ugyanis egy folytonos térben teljesen más lehet az optimális megoldás, mint egy diszkretizáltban. Megfelelően nagy felbontás mellett azonban elhanyagolhatjuk ezt a tényezőt, hiszen a különbség elenyésző lesz. A diszkrét tér olyan szempontból praktikus, hogy ha a térképet képként szeretnénk ábrázolni, akkor így már nem kell diszkretizálni az eredményt, mert eleve a megfelelő formában kapjuk azt meg.

Ezt a diszkrét, 2 dimenziós teret felfoghatjuk egy gráfként is, melyben minden cella egy csomópontnak felel meg, és a szomszédsági viszonyokat az élek reprezentálják [11–14].

2.2.3. Probléma formalizálása – a Steiner-fa

Az előzőekben felvezetett probléma megegyezik egy, a matematika területén már jól ismert problémával, mégpedig a minimális Steiner-fa problémájával [11–14]. Ennek az egyik változata a geometriai Steiner-fa probléma, ami a folytonos térben dolgozik, míg egy másik, ami a gráfelméleti oldalról tekinti a problémát, a diszkrét térben. A minimális geometriai Steiner-fa problémára látható néhány példa a 2.3. ábrán.



2.3. ábra. Minimális geometriai Steiner-fa példák [15]. (N jelöli a pontok számát, míg az L a topológiához tartozó összköltséget.)

A dolgozat során, mivel egyébként is diszkrét értékekkel fogok dolgozni, így a gráfelméleti változatot tanulmányoztam részletesebben. A probléma definiált mind irányított, mind irányítatlan gráfokban, az egyetlen kikötés, hogy az élek súlyainak nem negatívnak kell lenniük. Erre azért van szükség, hogy biztosított legyen az optimális megoldás létezése. Hiszen, ha például lenne egy negatív hosszú kör a gráfban, akkor azt sem lehetne megmondani, hogy mekkora a legrövidebb út két pont között, mivel minél többször mennénk végig ezen a körön, annál kisebb lenne a távolság.

A gráfelméleti megközelítésből a következőképp lehetne leírni a problémát. Legyen $G = (V, E)$ egy súlyozott gráf, és $S \subseteq V$ egy részhalmaza a csúcsoknak. Ekkor a minimális Steiner-fa egy olyan fa G-ben, ami érinti az összes csúcsot S-ben és minimális összköltségű. Azokat a csúcsokat melyek nincsenek benne S-ben és a fokszámuk na-

gyobb mint 2 (vagyis elágazások), Steiner pontoknak nevezzük. A Steiner pontokra látható példa a 2.3. ábra $N = 3$, $N = 4$ és $N = 5$ eseteiben ($N = |S|$).

A probléma felfogható akár másik két ismert probléma általánosításaként is, mégpedig a legrövidebb út és a minimális feszítőfa problémájának. Ha a végpontok száma pontosan 2, vagyis $|S| = 2$, akkor a feladat redukálódott a legrövidebb út megkeresésére, ha viszont a végpontok száma megegyezik a gráf csomópontjainak számával, vagyis $|S| = |V|$, akkor pedig a minimális feszítőfával fog megegyezni a feladat. Ezekre már léteznek optimális megoldást nyújtó algoritmusok, mint például a Dijkstra legrövidebb út algoritmusa, illetve Prim minimális feszítőfa algoritmusa [16–18].

A problémáról megállapították, hogy az általános formájában felírva NP-nehéz, vagyis nem létezik polinomiális idejű algoritmus, ami pontosan megtudná határozni az optimális megoldást. Éppen ezért legtöbbször csak közelítő megoldásokat keresnek, valamilyen heurisztikát felhasználva.

2.2.4. Elterjedt megoldási módszerek

Ahogy a korábbi szakaszban szerepelt, a megoldási módszerek általában valamilyen heurisztikát használnak fel a megoldás közelítésére. Ezek közül most a legismertebbeket fogom részletezni.

Ahhoz, hogy ezeket a módszereket össze tudjuk hasonlítani, definiálnunk szükséges egy mérőszámot, amivel az algoritmus eredményének minőségét tudjuk mérni. Ezt általában az algoritmus teljesítmény arányával (R_A) szokták megadni, melyet az alábbi módon definiálunk:

$$R_A := \sup \left\{ \frac{A(I)}{\mathrm{OPT}(I)} \,\middle|\, \text{minden } I\text{-re} \right\},$$

vagyis az R_A teljesítmény arányt úgy kapjuk meg, hogy az adott A algoritmus által visszaadott eredmény $A(I)$ értékét elosztjuk az optimális megoldás $\mathrm{OPT}(I)$ értékével, és ezt minden lehetséges I bemenetre elvégezzük, majd a legnagyobb értéket kiválasztjuk. Megjegyzendő, hogy a definíció csak minimalizálási probléma esetén érvényes, ha maximalizálási problémáról van szó akkor a legkisebb arányt kell kiválasztanunk. Ez az érték gyakorlatilag azt adja meg, hogy a kapott megoldás a legrosszabb esetben maximum hányszorosa lehet az optimális megoldásának. Például 2 egész érték esetén a megoldásunk értéke kétszerese is lehet az optimálisnak. Természetesen önmagában ez az érték félrevezető lehet, hiszen nem veszi figyelembe a futásidőt, így habár egy algoritmus lehet, hogy jobb teljesítmény aránnyal rendelkezik, mint egy másik, mégsem érdemes használni, mert nagyságrendekkel tovább tart kiszámolnia az eredményt. Éppen ezért a kettőt együtt érdemes kezelni, azok arányát vizsgálva [19].

A következőkben ismertetésre kerülnek az algoritmusok, azonban azok nagyon hasonlóak, csak pár lépésben térnek el egymástól, ezért azok részletezése előtt felsorolom a módszerek általános lépéseit.

- `1. lépés:` *Inicializációs fázis.* Hozzuk létre a kezdeti részleges megoldásunkat, ami egy erdő lesz, amely az önmagukban álló végpontokból fog állni.

- `2. lépés:` *Útvonal választás.* Az eddigi megoldásunk két „megfelelően" kiválasztott pontját kössük össze egy útvonallal. Azt, hogy mely pont számít „megfelelőnek", azt a konkrét módszer fogja meghatározni. A kiválasztott pontoknak nem kell kötelezően megegyezniük a végpontokkal. Ezzel a lépéssel gyakorlatilag az erdő két izolált fáját vonjuk össze egy fává.

- `3. lépés:` *Kilépési feltétel ellenőrzése.* Amíg a megoldásunkban szerepel izolált fa, addig ismételjük a 2. lépést.

- `4. lépés:` *MST számítás.* A megoldás által kijelölt pontokkal és élekkel meghatározott részgráfban számoljunk egy minimális feszítőfát. Az erre nem illeszkedő pontokat és éleket dobjuk el. Ezzel gyakorlatilag kiszűrjük az esetlegesen létrejövő hurkokat.

- `5. lépés:` *Lenyírás.* A megoldásból távolítsuk el azokat a pontokat melyek nem végpontok és a fokszámuk kevesebb mint 2. Egyszerre mindig csak egyet távolítsunk el, egészen addig ismételve, míg már nem tudunk többet eltávolítani. Erre azért van szükség, mert az előző lépés során, amikor hurokmentesítjük a gráfot, „zsákutcákat" hozhatunk létre, melyek fölöslegesek a mi szempontunkból, így el kell távolítani őket.

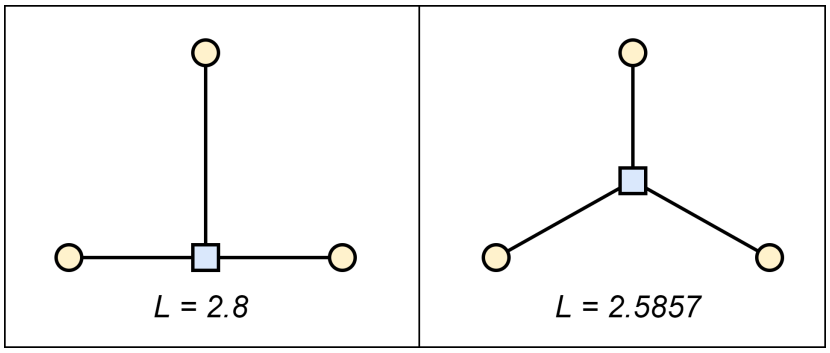
Shortest Path Heuristics (SPH)

Az SPH algoritmust eredetileg Takanashi és Matsuyama fejlesztette ki, de azóta számos másik kutató is tanulmányozta és különböző változatokat hoztak belőle létre [20]. Az algoritmus lépései megegyeznek az előzőekben felsoroltakkal, kivéve a 2. lépést, mely az alábbi formára módosul.

- `2. lépés:` *Útvonal választás.* Előzetesen válasszunk ki egy végpontot, amiből elkezdjük építeni a megoldásunkat. (Ez az iterációk során nem fog változni.) Kössük össze azt a még nem csatlakoztatott végpontot a kiválasztott végponthoz tartozó fába, amelyet a legrövidebb úton érünk el.

Az algoritmus a nevét is erről a lépésről kapta, hiszen mindig a legrövidebb úton elérhető végpontokat csatlakoztatjuk az addigi megoldáshoz [21, 22].

Könnyen látható, hogy az SPH párhuzamba hozható a Prim féle minimális feszítőfa algoritmusával, ugyanis az SPH épp úgy, mint Prim algoritmusa is, fokozatosan, mindig csak egy ággal bővíti a már meglévő fát, melyet az aktuális legkisebb költségű él alapján választ ki. A különbség itt csak annyi, hogy az SPH-nál ezek az élek nem a hagyományos értelemben vett élek, hanem az S pontjait a már meglévő fával összekötő legrövidebb



2.4. ábra. Az SPH eredménye (balra) szemben az optimális megoldással (jobbra). (Az L a topológia összköltségét jelöli.)

utak lesznek. Az is könnyedén belátható, hogy az eredmény korántsem lesz optimális (néhány szélsőséges esetet leszámítva), melyre egy példa is látható a 2.4. ábrán.

Az algoritmus során minden bekötendő végpontból, összesen $|S|-1$ alkalommal kell legrövidebb utat számolnunk. Egy ilyen legrövidebb út kiszámítása legrosszabb esetben $|V|^2$ lépést vesz igénybe, melyből így meghatározható az algoritmus időkomplexitása, szám szerint:

$$T_{\mathrm{SPH}} = \mathcal{O}(|S| \cdot |V|^2).$$

A teljesítmény aránya:

$$R_{\mathrm{SPH}} \leq 2\left(1 - \frac{1}{|S|}\right).$$

Distance Network Heuristics (DNH)

Ezt az algoritmust egymástól függetlenül fejlesztette ki Kou kutatótársaival [23], Plesnik [24], és El-Arbi [25]. Ennél az algoritmusnál a 2. lépés az alábbiak szerint alakul.

- `2. lépés:` *Útvonal választás.* Számoljuk ki minden S-beli pont között a legrövidebb utat, ezeket tároljuk is el. Ezt követően hozzuk létre az S pontjaiból álló teljes gráfot, ahol az élek súlya az előzőekben kiszámolt legrövidebb utak költsége lesz. Az így kapott gráfnak készítsük el a minimális feszítőfáját (például a Prim féle algoritmussal). Végül az így kiválasztott éleknek megfelelő legrövidebb utakból és az S pontjaiból megkapjuk a megoldásunk.

Az elnevezés feltehetően abból ered, hogy az algoritmus folyamán ki kell számolnunk S minden egyes pontja között a távolságot, vagyis a távolsághálót (vagy távolságmátrixot), melyre a szakirodalomban „distance network”-ként is szoktak hivatkozni.

Az algoritmus időkomplexitása hasonlóan számolva, mint az SPH esetén:

$$T_{\mathrm{DNH}} = \mathcal{O}(|S| \cdot |V|^2).$$

A teljesítmény aránya:

$$R_{\text{DNH}} \leq 2\left(1 - \frac{1}{|S^*|}\right) \leq 2\left(1 - \frac{1}{|S|}\right),$$

ahol $S^* = \{v \in S \mid \deg(v, T) = 1\}$ és T a gráf leszűkítése a megoldásban szereplő csúcsokra és élekre. Másképp fogalmazva: S^* elemei azok az S-beli csúcsok, melyek a végeredményben is végpontok [21, 22].

Average Distance Heuristics (ADH)

Az utolsó algoritmusunkat Rayward-Smith és Clare [26, 27] fejlesztették ki. Ez az algoritmus annyiban eltér a többitől, hogy kicsit geometriai szemszögből is közelíti a problémát. Ennél az algoritmusnál a 2. lépés a következő.

- `2. lépés:` *Útvonal választás.* Válasszuk ki a G valamely szempontból (lentebb meghatározott módon) az eddigi megoldásunkban megtalálható részfákhoz képest leginkább központinak tekinthető pontját. Adjuk hozzá a megoldásunkhoz ezt a pontot és a megoldásban található részfákhoz számított legrövidebb utak közül a két legrövidebbet.

Az említett módszer, ami alapján a legközpontibbnak számít egy pont, az alábbi: Minden egyes $v \in G$ ponthoz számoljuk ki a megoldásunkban szereplő részfákhoz vezető legrövidebb utakat. A távolságaik alapján tegyük a részfákat növekvő sorrendbe, jelöljük ezeket $T_1, T_2, \ldots, T_k$-val, ahol k a részfák száma. Ezt követően válasszuk azt a v pontot, ahol a legkisebb lesz az alábbi függvény értéke:

$$\text{MinAvgDist}(v) = \min_{2 \leq n \leq k} \left(\sum_{i=1}^{n} \frac{\text{dist}(v, T_i)}{n - 1} \right).$$

Ez a függvény gyakorlatilag azt csinálja, hogy egy adott v ponthoz megnézi, hogy mennyi a legközelebb lévő 2 részfa (eltolt) átlaga, majd a legközelebb lévő 3, 4 és így tovább egészen k darab részfáig, majd kiválasztja az átlagok közül a legkisebbet.

Az algoritmus geometriai szemlélete is ebben a függvényben nyilvánul meg, hiszen a részfákhoz átlagosan legközelebb lévő pontok kerülnek kiválasztásra, amit tekinthetünk a részfák egyfajta geometriai középpontjának is. A geometria középpont pedig azért tud előnyös lenni, mert a geometriai Steiner problémánál már kiderítették, hogy a Steiner pontok optimális esetben a legközelebb lévő 3 végpontot kötik össze közel 120° fokos szögekkel, melyek szintén egyfajta középpontjai lesznek a pontoknak [21, 22, 28].

Az algoritmus során legrosszabb esetben minden pontból, minden pontba kell legrövidebb utat számolnunk, vagyis az időkomplexitása az alábbi lesz:

$$T_{\text{ADH}} = \mathcal{O}(|V|^3).$$

A teljesítmény aránya az algoritmusnak megegyezik a DNH-nál kapottal, vagyis:

$$R_{\mathrm{ADH}} \leq 2\left(1 - \frac{1}{|S^*|}\right) \leq 2\left(1 - \frac{1}{|S|}\right).$$

2.2.5. Intelligencia alapú megoldási módszerek

Az intelligencia alapú módszerek annyiban térnek el az előzőektől, hogy általában valamilyen kezdeti megoldásból indulnak ki és azt kezdik el folyamatosan javítani, valamilyen heurisztika, vagy szabályrendszer alapján. Előnyük, hogy képesek az optimális megoldáshoz konvergálni, de legrosszabb esetben is egy lokális optimumhoz fognak tartani. Pontosan ezért ezeknek a módszereknek kritikus pontja a megfelelő heurisztika és/vagy szabályrendszer összeállítása.

További előnyük, hogy a számításokat tetszőleges iteráció után megszakítva is kapunk vissza eredményt, habár feltehetően csak kvázi optimálisat. Ez olyan esetben lehet hasznos, ha a felhasználó szeretné eldönteni, melyik a fontosabb számára, a számítási idő, vagy a megoldás minősége.

Általánosságban elmondható, hogy a nem intelligencia alapú módszerekhez képest jobb megoldást tudnak nyújtani kevesebb számítási igény mellett. Természetesen ez mind implementáció függő, hiszen a konvergenciát szándékosan lassabbra is vehetjük, mellyel így nagyobb területet tudunk lefedni, ami kedvezőbb megoldást tud nyújtani, illetve lehet szándékosan gyorsabbra állítani, ami rendszerint lokális optimumot eredményez.

Az általam kipróbált módszerek rendszerint két részből álltak, egy keretrendszerből, mely a probléma megfelelő formára történő alakításáért felelt, illetve egy ágens alapú optimalizálási részből, ami a megoldás finomításáért felelt. Az ágens alapú részekről a 3. fejezetben fogok részletesebben írni, míg a keretrendszerekről a 4. fejezetben lesz szó.

3. fejezet

Ágens alapú heurisztikák

A következőkben részletezésre kerülnek az ágens alapú heurisztikák, azonban mivel ezek gyakorlatilag mind-mind egyben ágens alapú modellek is, így előttük még az ágens alapú modellek leírása következik. A továbbiakban a modellt és a heurisztikát, mint szinonimákat fogom használni [29–31].

3.1. Ágens alapú modellek

Az ágens alapú modellek (Agent Based Model, röviden ABM) a számítógépes modellek egy olyan osztálya, melyek egymással interakcióba lépő ágensekből épülnek fel. Kicsit formálisabban a definíció az alábbi.

3.1. definíció. Ágens alapú modell: Egy számítógépes modell, mely magába foglalja ágensek egy csoportját, melyben az ágensek állapota tipikusan egy véges értékkészletből származik. Az állapotokat egy adott időpillanatra az előre megadott szabályok alapján lehet meghatározni, az előző állapotokból. A szabályok azt írják le, hogy hogyan lépnek interakcióba az ágensek egymással, illetve a környezetükkel. Ezek a szabályok lehetnek akár determinisztikusak (azonos lefutásúak), de akár sztochasztikusak is (véletlentől függőek). Fontos, hogy minden ágens állapota csak az előző állapotától és azon környezetében lévő ágensek állapotától függ, amelyekkel interakcióba is lépett.

Az ágens alapú modellekhez hasonló a multi-ágens rendszerek (Multi Agent Systems, röviden MAS) definíciója. Habár a két fogalom kapcsolódik egymáshoz, de azért nem teljesen azonosak. Az előbbi főleg a természetben megtalálható egyének viselkedési mintáit leutánozva próbál betekintést nyerni a csoport átfogó viselkedésébe, például a madarak egyéni viselkedését vizsgálva próbálja leírni, hogy hogyan alakulnak ki a madárrajok. Az utóbbi főleg adott gyakorlati vagy mérnöki problémára keres megoldást az ágensek segítségével. Mivel kicsi a különbség a két leírási mód között, és lehet átfedés is, ezért a szakirodalmakban gyakran fel szokták cserélni a két fogalmat, vagy azonos jelentéssel rendelkezőként kezelik őket [29, 30, 32, 33].

Egy felbontás szerint nézve, az ágens alapú modellek az alábbi 5 részből állhatnak:

- ágensek,
- döntéshozó szabályok vagy heurisztikák,
- tanulási szabályok,
- interakciók topológiája,
- környezet.

Magukat az ágenseket egy csoportosítás szerint az alábbi két kategóriába lehetne tovább sorolni:

- `passzív ágensek,`
- `aktív ágensek.`

A passzív ágensek nem rendelkeznek céllal, így általában nem csinálnak semmit, például csak egy akadály szerepét töltik be. Ellenben az aktív ágensek már rendelkeznek céllal és annak elérésére törekednek a viselkedésükkel.

Egy másik szempont alapján a következő két csoportra lehetne tovább osztani az ágenseket:

- `reflex ágensek,`
- `intelligens ágensek.`

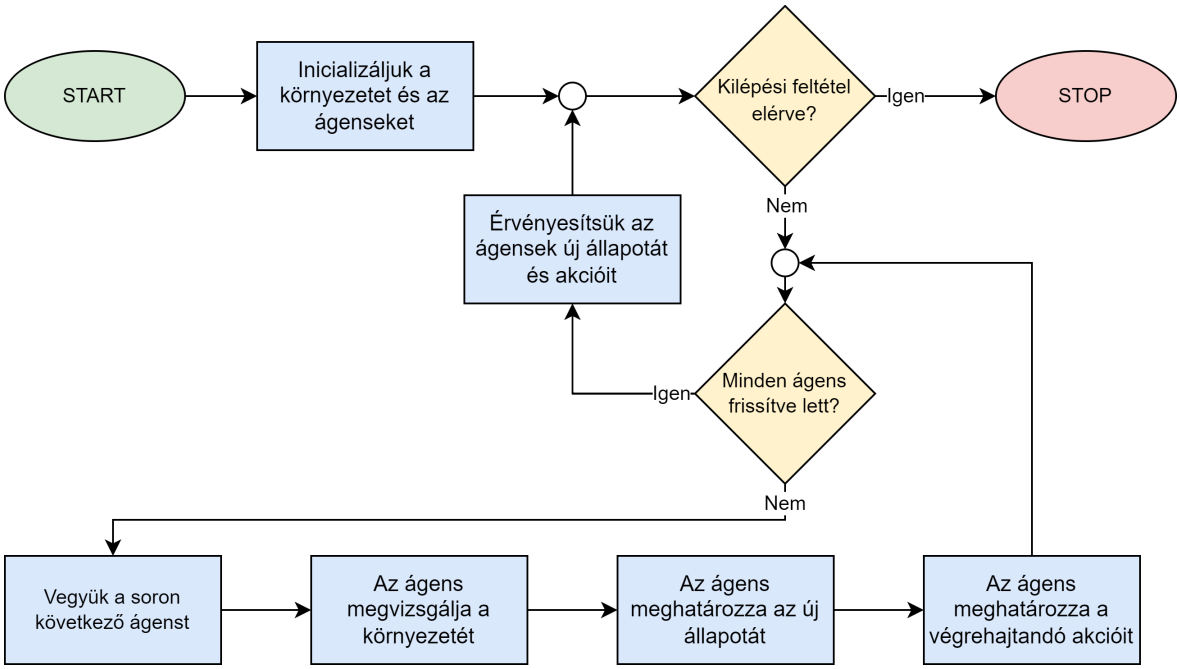
A reflex ágenseket nevezhetnénk „buta” ágenseknek is, ugyanis ők a megadott szabályok alapján csak reagálnak a környezet változásaira. Ezzel szemben az intelligens ágenseknél megtalálható egy tanulási faktor, mellyel módosítani tudják a viselkedésüket, például tudnak új szabályokat létrehozni, vagy törölni.

Végül a környezet reprezentációja alapján az alábbiak szerint csoportosíthatjuk tovább őket:

- diszkrét térrel rendelkezők,
- folytonos térrel rendelkezők.

A modell működése megtekinthető a 3.1. ábrán.

Az ágens alapú modellek nagy előnye, hogy számos problémára jól lehet alkalmazni őket. Rengeteg területen vették már hasznát, például a társas interakciók tanulmányozásában, a járványok terjedésének szimulálásában, a termelésnél az ütemezés és a hatékonyság elemzésében, a gyógyszerek az emberi sejtekre mért hatásának vizsgálatában. Érdemes megemlíteni, hogy a mai világ tudományos problémái általában nem egy területre korlátozódnak csak, hanem több tudományágra is szükség van a megoldásukhoz. Például a rák ellenszerének megtalálásában valószínűleg szerepet fognak játszani



3.1. ábra. Az ágens alapú modellek folyamatábrája.

biológusok, genetikusok, matematikusok, kémikusok, de még akár más tudományágakkal foglalkozó kutatók is. Az ágens alapú modellek az ilyen kutatásokban nagy előnyt jelentenek, hiszen a mögöttes, bonyolult matematikai modellt elfedik és egy könnyen kezelhető felületet (rendszerint grafikus felületet) biztosítanak a felhasználók számára. Másképpen fogalmazva: anélkül tudják használni akár a biológusok, a kémikusok vagy más tudósok is a modellt, hogy meg kellene érteniük a matematikai hátterét.

Még a matematikai modellek körében is van néhány egyértelmű előnye az ágens alapú modelleknek. Az egyik ilyen, hogy hatékonyak az olyan rendszerek modellezésében, ahol a sok entitásnak (ágensnek) ugyan azokat a szabályokat kell követnie. Bizonyos rendszereket sokkal egyszerűbb ezzel modellezni, mint mondjuk a hagyományos vagy a parciális differenciálegyenletek segítségével. Különösen igaz ez, ha szükséges lehet a modell részeit a későbbiek során változtatni, hiszen ezt az ABM esetén néhány kódsor megváltoztatásával meg lehet valósítani, míg a differenciál egyenletek használata esetén ez sokkal bonyolultabb, akár kivitelezhetetlen is lehet [34, 35].

Kutatóknak akadt azonban problémája is az ágens alapú modellekkel kapcsolatban, kifejezetten azok leírásával, melyek a következőek [34].

- Nincs szabványosítva a modell leírása. Vagyis nincs szabványos sorrendje és formája a modell részeinek. Emiatt egyrészt az olvasó számára nehéz lehet a különböző ágens alapú modelleket összehasonlítani, másrészt egy adott modellt is nehezebb értelmezni, mert nem feltétlenül egyértelmű, hogy hol kell lennie a keresendő résznek.

- Sok esetben pontatlanul vagy hiányosan vannak megfogalmazva a modell részei. Például a bonyolultabb szabályok nem egyértelmű módon vannak leírva, esélyt

adva így a félreértésre. Erre egy jó példa lehet az a modell, amelyben az ágensek nyulakat reprezentálnak és minden iteráció során egy véletlenszerű irányba mozognak. Könnyedén belátható mi a probléma ezzel a leírással, hiszen nem derül ki egyértelműen például, hogy az irányok diszkrétek-e vagy folytonosak, ha diszkrétek akkor konkrétan melyek, illetve, hogy a mozgás mértéke változó-e vagy konstans.

A következő szakaszokban az általam részletesen tanulmányozott ágens alapú modellekről lesz szó. Megjegyzendő, hogy természetesen nem csak ez a 4 modell létezik, hanem még sok másik is. A választás azért ezekre esett, mert vagy már közismertek voltak, vagy mostanában alkalmazták az előző fejezetben említett problémára. A modellek sorrendjénél igyekeztem időrendben haladni azok kifejlesztésének megfelelően.

3.2. Ant Colony Optimization (ACO)

A következőkben a hangyakolónia optimalizációról (Ant Colony Optimization, röviden ACO) lesz szó. A hangyakolónia optimalizálási eljárást először 1992-ben fejtette ki Marco Dorigo a PhD értekezésében [41]. Alapvetően egy valószínűségen alapuló módszer, mellyel olyan gráfokban értelmezett problémák oldhatóak meg viszonylag jó hatékonysággal, melyek valamely módon redukálhatóak a legrövidebb útvonal keresésre [36–40].

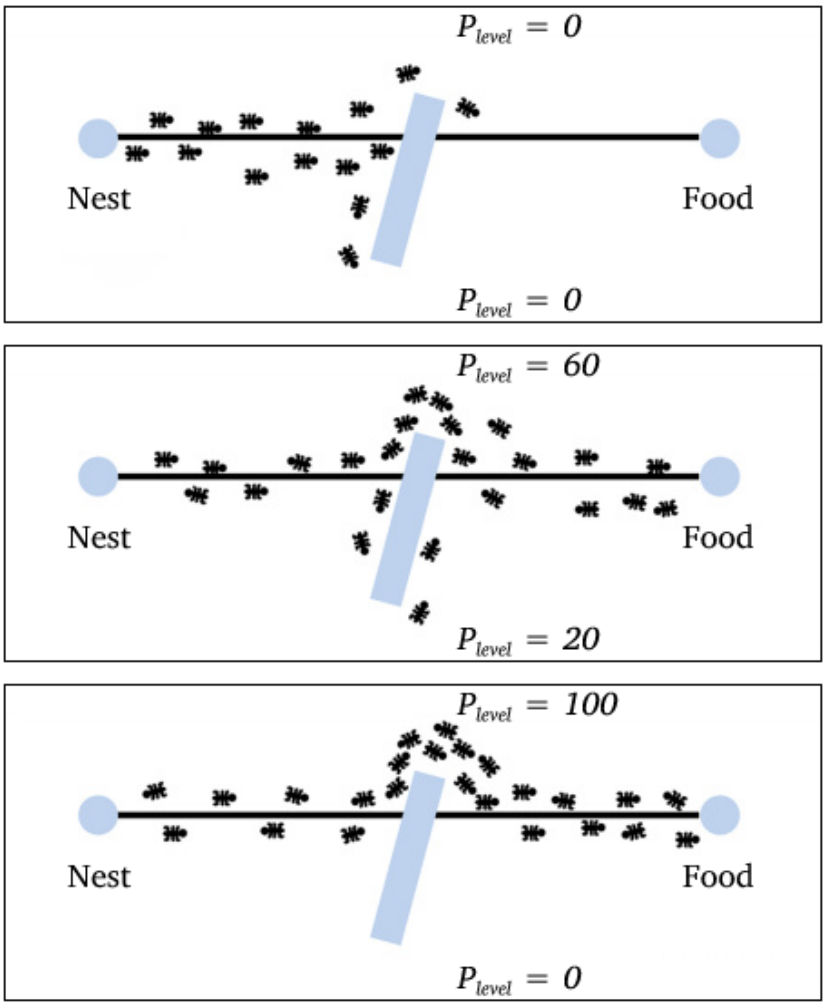
3.2.1. A modell áttekintése

Ahogy a nevéből sejthető is, a módszer a természetben fellelhető hangyák viselkedésén alapszik. A természetben a hangyák alapvetően véletlenszerűen vándorolnak, mindaddig amíg nem találnak élelemforrást. Miután találtak egy élelemforrást, visszafordulnak a kolóniához és az útjuk során feromonokat tesznek le. A többi hangya, amennyiben érzékeli ezeket a feromonokat, nagy valószínűséggel már nem fog tovább véletlenszerűen bolyongani, hanem elkezdi követni ezt az útvonalat egészen az élelemforrásig. Ha ténylegesen találnak élelmet az útvonal végén, ők is visszaindulnak a kolónia felé és feromonokat húznak az útjuk során, mellyel megerősítik az eredetileg talált útvonalat.

Idővel azonban a feromonok elkezdenek elpárologni, ezzel csökkentve a vonzerejét az útvonalnak. Minél több ideig tart egy hangyának eljutnia a céljához, annál több ideje van a feromonoknak elpárolognia. Egy rövidebb útvonalon ugyan annyi idő alatt többször is végig tudnak menni a hangyák, ezért az útvonal feromon szintje is magasabb lesz, hiszen kevesebb ideje van elpárologni a feromonoknak, illetve többször is tudnak új feromont elhelyezni a hangyák útjaik során. Ha nem lenne a feromonok párolgása, akkor a legelső megtalált útvonal gyakorlatilag az egyetlen megtalált útvonal is lenne, mivel onnantól az összes hangya nagy valószínűséggel csak ezt az egy útvonalat követné.

Összegezve, amikor egy hangya talál egy jó útvonalat a kolóniától egy élelemforráshoz, a többi hangya elkezdi követni ezt az útvonalat, és a feromonok pozitív visszacsato-

lását okozza, ami végül oda vezet, hogy egyre több hangya fogja ezt az útvonalat követni. A hangyakolónia optimalizáció ötlete az lenne, hogy leutánozva a hangyák ilyesfajta viselkedését, útvonalkeresési problémákat tudjon megoldani gráfokban. A hangyakolóniák útvonal optimalizációjának működését szemlélteti a 3.2. ábra.



3.2. ábra. A hangyakolóniák útvonal optimalizációjának működése. (A P_{level} az adott irányhoz tartozó feromon szintet jelöli.)

A modellben szereplő ágensek a mesterséges hangyák lesznek, melyek viselkedését a valódi hangyákról imitálták.

A hangyák az egymással történő kommunikációra az úgy nevezett feromonokat használják. Ez egy indirekt kommunikációs forma, hiszen nem közvetlenül jeleznek egymásnak a hangyák, hanem a környezetüket módosítják oly módon, hogy az a többiek számára jelentéssel bírjon. A valódi hangyák ezeket az információkat az élelem megtalálásához használják fel, míg esetünkben a legrövidebb útvonal megkereséséhez használjuk fel.

A módszer során a mesterséges hangyáink úgy határozzák meg az optimális megoldást, hogy az összes lehetséges megoldást tartalmazó paraméter térben mozognak. A mozgásuk során felfedezik a környezetüket, vagyis meghatározzák az aktuális meg-

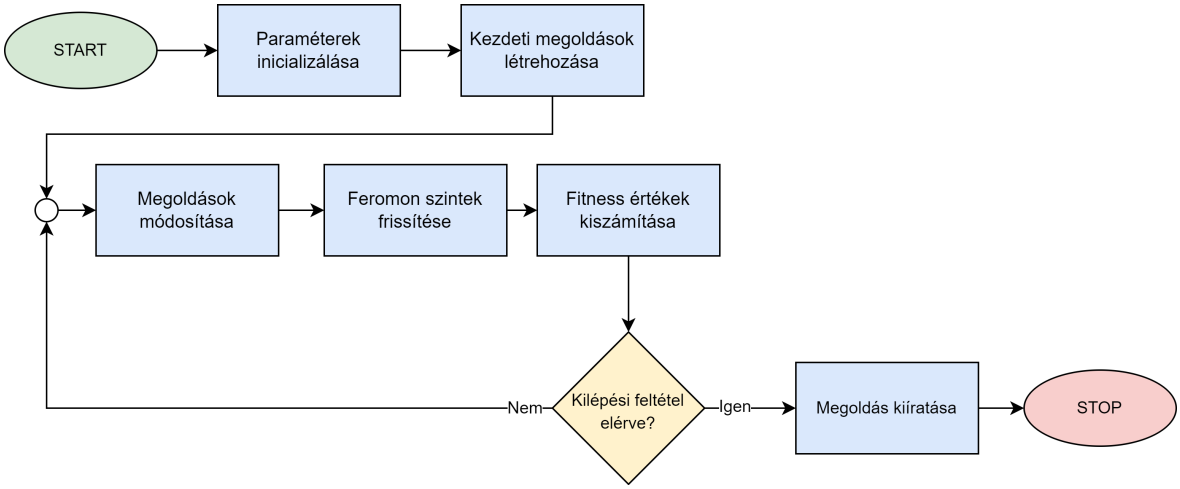
oldásuk minőségét, majd a megoldás minőségével arányosan, a felhasznált paraméter értékekhez feromonokat helyeznek el. Ezzel a többi hangyát arra ösztönzik, hogy szintén ezeket az értékeket válasszák a saját megoldásaikhoz.

Konvergencia

A konvergenciát az biztosítja, hogy a hangyák a nagyobb feromon szinttel rendelkező irányba nagyobb valószínűséggel mennek, így egyfajta önerősítő módon, idővel a legjobb útvonalrészeken egyre nagyobb lesz a feromon szint, amitől egyre többet mennek azokra a részekre, és így tovább. Fontos megjegyezni, hogy mindig van esély a kisebb feromon szinttel rendelkező irányokba történő haladásra is, mellyel így a módszernek van esélye kijönni a lokális minimumokból. A konvergenciához kritikus rész a feromonok párolgása is, mivel enélkül az algoritmus a legelső megoldáshoz konvergálna, függetlenül annak minőségétől.

3.2.2. Az algoritmus

Az algoritmus lépéseit jól szemlélteti a 3.3. ábrán látható folyamatábra. A felsorolt lépések közül a „megoldások módosítása” és a „feromon szintek frissítése” az érdekesebbek számunkra.



3.3. ábra. A hangyakolónia optimalizáció lépései.

A **megoldások módosítása** során a mesterséges hangyáknak el kell dönteniük, hogy mekkora valószínűséggel fogják a paramétereik értékét megváltoztatni a lehetséges értékek közül valamelyik másikra. Egy ilyen döntés során két dolgot vehetnek figyelembe: az új értékre történő váltás η *vonzóságát*, melyet előzetesen meghatározunk egy heurisztikával (ilyen lehetne például az él költsége), illetve az új értékre történő váltás feromon szintjét, ami azt jelzi, hogy a múltban mennyire bizonyult jónak ez az irány. Általánosságban felírva a valószínűsége, hogy a k-adik hangya az x állapotból áttér az

y állapotba (ahol az állapot a paramétereket foglalja magába) a következő:

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha) \cdot (\eta_{xy}^\beta)}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^\alpha) \cdot (\eta_{xz}^\beta)},$$

ahol a τ_{xy} az x állapotból y állapotba történő változás feromon szintje, az η_{xy} a változás előzetesen meghatározott vonzósága, az $\alpha \geq 0$ a feromonok, a döntésre mért befolyását kontrolláló paraméter, a $\beta \geq 1$ a változás vonzóságának befolyását kontrolláló paraméter, a $z \in \text{allowed}_x$ pedig az összes állapot, melyre x-ből át lehet térni.

A **feromon szintek frissítése** során két dolog történik. Egyrészt a hangyák elhelyezik a megoldásuk minősége alapján kiszámolt feromon értékeket a használt paraméter értékekhez, másrészt globálisan csökkentjük az összes eddig elhelyezett feromon értéket. Képlet formájában felírva:

$$\tau_{xy} = (1 - \rho) \cdot \tau_{xy} + \sum_k^m \Delta\tau_{xy}^k,$$

ahol az előző jelölések mellett a ρ a feromon párolgási együttható, az m a hangyák száma, a $\Delta\tau_{xy}^k$ pedig a k-adig hangya által, ebben az iterációban elhelyezett feromonok értéke. A $\Delta\tau_{xy}^k$ értékét tipikusan az alábbiak szerint szokták meghatározni:

$$\Delta\tau_{xy}^k = \begin{cases} Q/L_k, & \text{ha a } k\text{-adik hangya használja a megoldásában az } xy \text{ élt,} \\ 0, & \text{egyébként,} \end{cases}$$

ahol az L_k az aktuális megoldás hossza, a Q pedig egy konstans paraméter, amivel az elhelyezett feromonok mennyiségét lehet befolyásolni.

3.2.3. Összegzés

A módszert számos problémára alkalmazták már sikerrel, például az utazó ügynök problémájára, képek éleinek a detektálásra, számos útvonaltervezési problémára és ezek további változataira. Előnye a többi hasonló problémát megoldó algoritmussal (mint például a genetikus algoritmussal) szemben, hogy a dinamikusan, futásidőben változó gráfok esetén is jól tud működni, képes alkalmazkodni azok változásaihoz anélkül, hogy hozzá kellene nyúlni a programhoz. Ezen tulajdonsága különösen a számítógépes hálózatok útválasztásánál lehet hasznos, hiszen ezek a hálózatok folyamatosan változnak, melyek során például új csomópontok jöhetnek létre, vagy összeköttetések szűnhetnek meg. További előnye még, hogy nem szükséges egzaktan megtalálnia a hangyáknak az optimális megoldást, elég, ha minden hangya csak egy részét fedezi fel, majd ezekből a feromon értékek miatt úgyis összetevődik az optimális megoldás.

Alapvetően a módszer jó eredményeket tud produkálni, de fontos megkötése, hogy csak gráfokon értelmezett problémák estén használható, valamint ha a problémát át lehet alakítani egy legrövidebb út megkeresésére, ezért általános problémákra nem igazán alkalmazható.

3.2.4. További változatok

Az algoritmusnak számos változata létezik, melyek rendszerint javítanak valamilyen formában a működésén.

- `Ant System (AS)`: A legelső változat, a fentebb már definiált módon működik, kifejezetten az utazó ügynök problémához hozták létre.
- `Ant Colony System (ACS)`: Az AS egy továbbfejlesztett változata. Itt csak a legjobb megoldáshoz tartozó hangya tehet le feromonokat. A hangyák állapot változtatási döntése át lett alakítva, hogy jobban preferálja a rövid és nagy feromon szinttel rendelkező állapotokat. Továbbá minden állapot változtatás során a hangyák frissítik a változáshoz tartozó feromonok értékét, ezzel lassítva azok párolgását.
- `Elitist Ant System`: Ebben a változatban a globális (eddigi legjobb) megoldáshoz tartozó paraméter értékekhez minden iteráció során teszünk le feromonokat (akkor is, ha nem járt erre egyetlen hangya sem). Ezzel az eddig megtalált legjobb megoldás felé tudjuk terelni a hangyákat.
- `Max-Min Ant System (MMAS)`: Ez az algoritmus a feromonok szintjét egy minimális $T_{\min}$ és egy maximális $T_{\max}$ érték közé szorítja. Feromonok elhelyezését csak az iteráció legjobb és a globális legjobb megoldások hangyáinak engedélyezik. Ez könnyedén a megoldások stagnálásához vezethetne, azért, hogy ezt elkerüljék, kezdetben minden feromon értéket $T_{\max}$-al inicializálnak. Ennek hatására a hangyák a keresési tér sokkal nagyobb részét fogják bejárni. Amikor a feromon értékek nagyon kicsik kezdenek lenni, újrainicializálják őket $T_{\max}$-ra.
- `Rank-based Ant System (ASrank)`: Ebben a változatban az iterációk végén a megoldások a minőségeik alapján sorba lesznek rendezve, és csak az n legjobb megoldáshoz tartozó hangyának engedélyezik, hogy feromont helyezzenek el. Az elhelyezett feromonok mennyisége továbbra is a megoldás minőségével arányosan alakul.
- `Recursive Ant Colony Optimization`: Az AS-nek egy rekurzív változata. A keresési teret több kisebb részre bontja, melyekben külön-külön hajtja végre a hangyakolónia optimalizációt.

3.3. Particle Swarm Optimization (PSO)

A következőkben a részecske raj optimalizációról (Particle Swarm Optimization, röviden PSO) lesz szó. A PSO-t eredetileg Kennedy és Eberhart fejlesztette ki 1995-ben [43]. Egy viszonylag jó hatékonysággal működő optimalizáló módszer, amit gyakorlatilag

minden olyan problémára lehet használni, melyet át lehet alakítani minimalizálási vagy maximalizálási feladatra [42–47].

3.3.1. A modell áttekintése

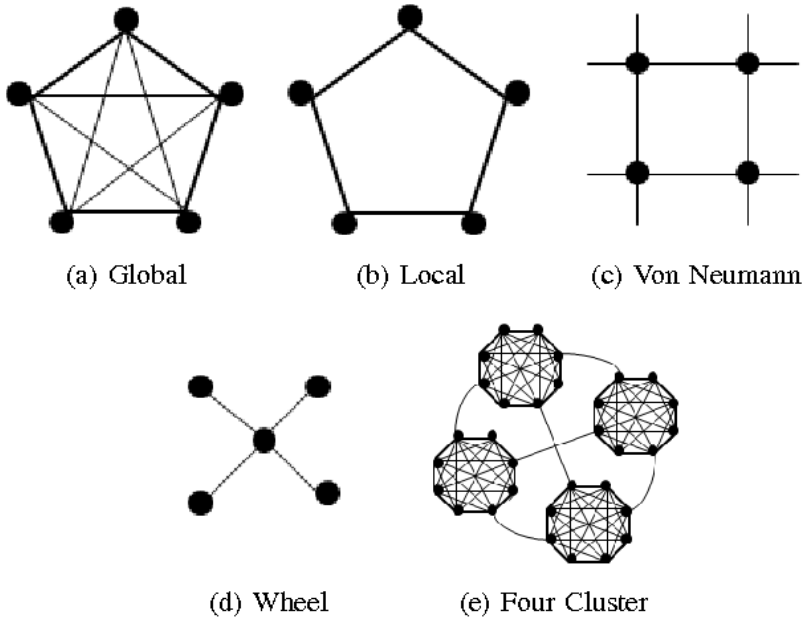
A PSO ötlete is a természetből jött. A madárrajokat modellező BOIDS algoritmus egyszerűsítéséből alakult ki a módszer. A lényege, hogy az ágenseink szerepét a részecskék fogják felvenni. Minden egyes részecske rendelkezik egy pozícióval és sebességgel, valamint néhány egyszerű szabály alapján mozogni is tudnak a keresési térben. A részecskék mozgását két dolog befolyásolja: az eddig ismert legjobb pozíciójuk, és a csoport által ismert legjobb pozíciója, mind a kettő vonzó hatást gyakorolva. Ezeket az értékeket rendszerint frissíti a számítás, amint jobb pozíciót sikerül találnia valamelyik részecskének.

Topológia

A módszer topológiája azt határozza meg, hogy mely részecskék kommunikálhatnak egymással, vagyis melyek tartoznak egy csoportba. A legegyszerűbb változatban mindegyik részecske kommunikálhat mindegyikkel, mely a gyakorlatban az egyetlen globális (g) legjobb értékben nyilvánul majd meg. Más topológiák is léteznek, melyek lehetnek geometriai alapúak vagy szociális alapúak is. A geometriai alapú topológiára példa a legközelebbi n részecskék topológiája, vagy a legfeljebb d távolságra lévő részecskék topológiája. A geometriai topológiák hátránya, hogy általában több számítást igényelnek, mert szerepet játszik bennük valamiféle távolságszámítás, éppen ezért inkább a szociális alapú topológiákat szokták preferálni. A szociális alapú topológiák nem függnek semmilyen távolságtól, helyette nekünk kell meghatároznunk, hogy pontosan melyik részecske melyikkel kommunikálhat. Általában a szociális topológiák idővel geometriaiak is lesznek, hiszen az adott csoport rendszerint elkezd egy adott ponthoz közeledni, névlegesen az optimumhoz. Szociális topológiára egy példa a gyűrű topológia, mely során minden részecske csak az előtte és az utána lévőt ismeri. Különböző topológiák láthatók még a 3.4. ábrán.

Paraméterek hangolása

A módszer paramétereinek nagy hatása van annak teljesítményére. Úgy kell őket meghatározni, hogy megfelelő egyensúlyt teremtsünk a keresési tér felfedezése (exploration) és a konvergencia (exploitation) között. A rosszul megválasztott paraméterek egyrészt a lokális optimumhoz történő túl korai konvergenciát okozhatják, másrészt pont ellenkezőleg, a túl lassú konvergenciát eredményezhetik. Éppen ezért fontos a paraméterek jó megválasztása, mely számos további kutatás tárgya volt már eddig is. Legtöbbször tapasztalati alapokon szokták meghatározni őket.



3.4. ábra. Néhány példa a PSO topológiákra [48].

Konvergencia

A módszer konvergenciáját már számos tanulmány vizsgálta, de teljességgel bizonyítani annak helyességét még nem sikerült egyiknek sem. Ennek fő oka, hogy bonyolultsága miatt, az egyszerűbb esetektől eltekintve csak tapasztalati alapokon tudják vizsgálni a módszert. Például magasabb dimenzióknál már ez a helyzet. Ezzel pedig az a probléma, hogy csupán néhány eset vizsgálatából nem lehet teljesen biztos következtetést levonni az összes esetre vonatkozóan.

3.3.2. Az algoritmus

Az algoritmus lépései megfigyelhetőek a 3.5. ábrán található folyamatábrán. A továbbiakban az ábrán szereplő lépések részletezéséről lesz szó a legegyszerűbb változat esetében, melynél a globális topológiát használjuk.

A részecskék kezdeti pozícióját a keresési téren értelmezett, egyenletes eloszlású véletlen vektorral kell meghatározni, formálisan:

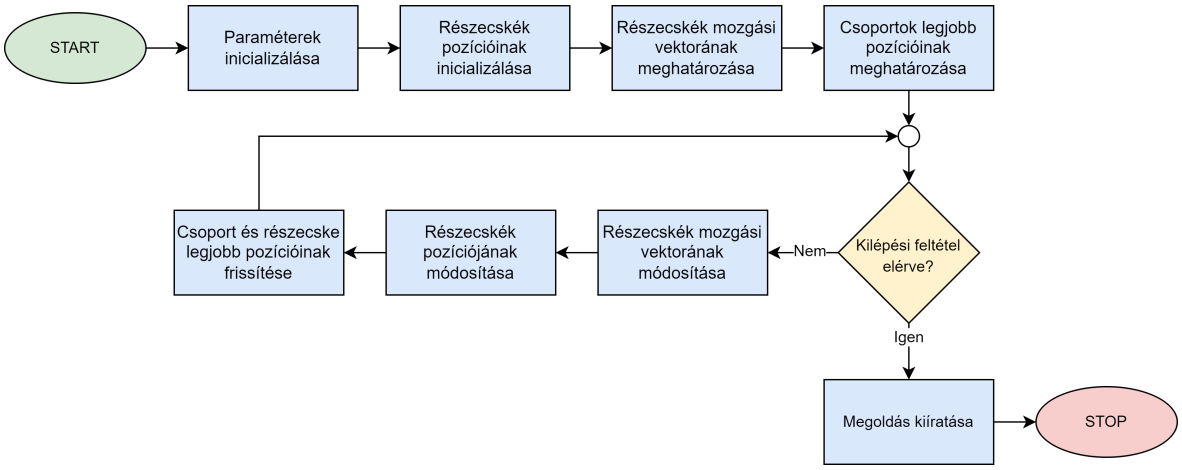
$$x_i \sim U(b_{\text{lower}}, b_{\text{upper}}),$$

ahol x_i az i-edik részecske pozíciója, a b_{lower} és b_{upper} értékek pedig a keresési tér alsó és felső határai.

A részecskék kezdeti mozgási vektorát szintén egy egyenletes eloszlású véletlen vektorral kell meghatározni, mégpedig:

$$v_i \sim U(-|b_{\text{upper}} - b_{\text{lower}}|, |b_{\text{upper}} - b_{\text{lower}}|),$$

ahol a v_i az i-edik részecske mozgási vektorát jelöli.



3.5. ábra. A részecskeraj optimalizáció lépései.

A részecskék mozgási vektorának módosítása az alábbi formula szerint történik:

$$v_{i,d} = w \cdot v_{i,d} + \Phi_p r_p(p_{i,d} - x_{i,d}) + \Phi_g r_g(g_{i,d} - x_{i,d}),$$

ahol az i alsóindexek az i-edik részecskét jelölik, a d alsóindexek a d-edik dimenzióját a keresési térnek, a v a részecske mozgási vektorát, az r_p és az r_g a $U(0,1)$ egyenletes eloszlásból generált véletlen számok, a p a részecske eddig ismert legjobb pozíciója, a g a részecske csoportjának eddig ismert legjobb pozíciója, az x a részecske aktuális pozíciója, a w egy paraméter amivel az előző mozgási vektor új mozgási vektorra mért hatását lehet befolyásolni, a Φ_p és a Φ_g pedig szintén paraméterek, melyekkel a lokális legjobb pozíciónak és a csoport legjobb pozíciójának hatását lehet befolyásolni.

A részecskék pozíciójának módosítása a mozgási vektorral történő eltolással fog megegyezni, vagyis az i-edik részecske új pozícióját az

$$x_i + L \cdot v_i$$

formában lehet meghatározni, ahol az L a tanulás mértékét meghatározó paraméter.

Az ábra többi lépése véleményem szerint már nem igényel több kifejtést, mert azok magától értetődőek, esetleg a kilépési feltételről ejtenék még pár szót. A kilépési feltétel megválasztásánál relatíve szabad kezet kapunk: megadhatjuk például az iterációk számát, egy értékhatár elérését, a megoldás minőségének javulásának bizonyos szint alá esését.

3.3.3. Összegzés

A módszernek rengeteg alkalmazása létezik, hiszen számos területen fel lehet használni. Általános optimalizálási célokra is gyakran használják, hiszen nagy előnye a többi módszerhez képest, hogy nem igényli a célfüggvény folytonosságát, differenciálhatóságát. Hasonlóan a többi heurisztikán alapuló optimalizációs módszerhez, ez sem garantálja a globális optimum megtalálását.

Úgy gondolom, hogy általánosságban optimalizálási feladatokra tökéletesen jól használható, így szerintem elsődlegesen ezzel érdemes megpróbálni megoldani az adott problémát, és csak ha valamiért nem lenne megfelelő ez a módszer, akkor érdemes szétnézi más módszerek, vagy esetleg a PSO különböző változatai között is.

3.3.4. További változatok

A PSO-nak rengeteg változata létezik, így nem is próbálkoznék most mindegyik felsorolásával. Már csupán a paraméterek beállításával, a részecskék inicializációjának módosításával is hatalmas változásokat lehet elérni. Például a részecskék sebességeit inicializálhatjuk véletlenszerűen, vagy akár 0-val is, illetve a lokális p_i és globális g legjobb értékeket frissíthetjük az iterációk végén, vagy azonnal amint már van jobb értékünk. Tovább árnyalja a képet, hogy számos módszer gyakorlatilag visszavezethető a PSO-ra és inkább annak egy változataként lenne érdemes kezelni, mintsem különállóan, de ez valamilyen okok miatt mégsem így történt. Néhány nevesebb változatot mindenesetre felsorolnék, melyek az alábbiak.

- `Adaptive PSO (APSO)`: Ahogy a neve is sugallja, ennél a változatnál jelen van egyfajta alkalmazkodó képesség. Ez abban nyilvánul meg, hogy nem nekünk kell meghatározni mekkora mértékben vegye figyelembe az algoritmus a globális és lokális legjobb pozíciókat, hanem azt automatikusan eldönti alkalmazkodva az aktuális állapothoz. Ennek előnye, hogy a kezdeti iterációkban lehetőség van inkább a felfedezésre (exploration) fókuszálni, ezzel meghatározva minél több lehetséges optimum helyet, majd ezután a konvergenciára (exploitation) fókuszálni, vagyis pontosítani az optimum közelítését. Ezeknek hála az APSO jobb optimumokat tud találni gyorsabb konvergencia mellett, mint a normál PSO, és kevésbé hajlamos a lokális optimumokba történő „beszorulásra”.

- `Binary PSO (BPSO)`: Alapvetően a PSO valós számokkal tud csak dolgozni, ezért gyakori megoldás, hogy diszkrét problémák esetén átkonvertálják a keresési teret egy folytonos térre, majd abban végzik el a PSO számításait, végül visszakonvertálják a megoldást [46, 47]. Ezzel az a probléma, hogy a konvertálás ronthatja a megoldás minőségét, illetve sok esetben maga a konvertálás sem annyira triviális. A bináris esetben, ahol minden dimenzió csak a 0-t vagy az 1-et veheti fel, módosítani szükséges az algoritmust. A lényegi különbség, hogy a BPSO esetében a mozgási vektorokat inkább valószínűségekként értelmezzük. Konkrétan egy ilyen vektor minden egyes komponense azt fogja meghatározni, hogy mekkora valószínűséggel fogjuk 1-nek választani az adott komponenst a pozíció módosítása esetén.

3.4. Artificial Fish Swarm Optimization (AFSO)

A mesterséges halraj optimalizáció (Artificial Fish Swarm Optimization, röviden AFSO), melyet szokás még AFSA-nak (Artificial Fish Swarm Algorithm) is nevezni, egy relatíve új és modern algoritmus az optimalizációk körében [49–51]. Az AFSO-t eredetileg 2002-ben fejlesztette ki Li a kutatótársaival egy tanulmányában [52].

A többihez hasonlóan ennek a módszernek az ötlete is a természetből fakadt, konkrétabban a halak viselkedéséből. Az eredményeinek köszönhetően rövid idő alatt megtalálta helyét a többi optimalizációs módszer között, így nem is csoda, hogy számos alkalmazása létezik, illetve, hogy a mai napig kutatások tárgya.

3.4.1. A modell áttekintése

Ahogy az előbb említettem, a módszer halak viselkedéseit tanulmányozva és azokat leutánozva jött létre. A lényege, hogy az ágenseink szerepét a mesterséges halak fogják betölteni. Minden egyes hal rendelkezik egy aktuális állapottal vagy pozícióval és egy környezettel. A hal környezetét az érzékelő, vagy látótávolsága határozza meg. Ebben a környezetben lévő többi hal befolyással lehet az adott hal viselkedésére. Ezeket az információkat a különböző viselkedési mintáknál használjuk fel a halak állapotainak frissítésére.

A környezet alatt gyakorlatilag a keresési teret értjük, melyben az optimalizálandó paraméterek értékei határozzák meg az ágensek konkrét pozícióját.

A viselkedések általános szerkezete a következő műveletekkel írható le.

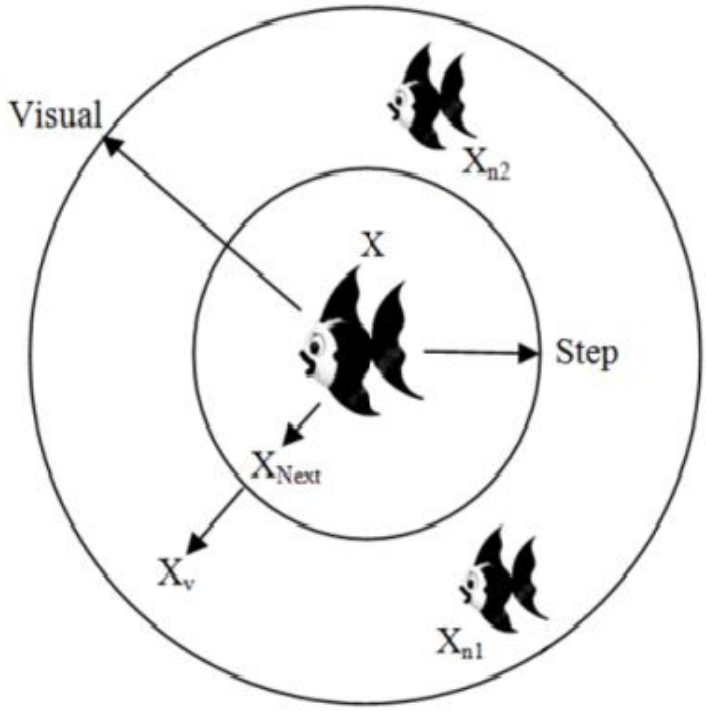
- Generáljunk ki egy új pozíciót a látótávolságon belül.
- Döntsük el, hogy ez az új pozíció megfelel-e az elvárásainknak.
- Frissítsük az aktuális állapotot, úgy, hogy közelebb kerüljünk a generált állapothoz.

A modellben használt konkrét viselkedési formákról részletesebben a következő szakaszban lesz szó. Az általános viselkedésnél használt értékeket jól szemlélteti a 3.6. ábra, melyen a $Visual$ a látótávolságot, a $Step$ a maximális lépésközt, az X az aktuális állapotot, az X_{n1} és X_{n2} a környező halak állapotát, X_v a vizsgált új állapotot és az X_{Next} pedig a tényleges új állapotot jelöli.

A modell során végig figyelemmel kell kísérni a legjobb hal állapotát, ugyanis a kilépési feltétel elérése esetén ez lesz a visszaadott eredmény.

3.4.2. Az algoritmus

A módszer egy absztraktabb leírását láthatjuk a 3.7. ábrán. Bal oldalt látható az



3.6. ábra. Az AFSO viselkedéseiben használt paraméterek [53].

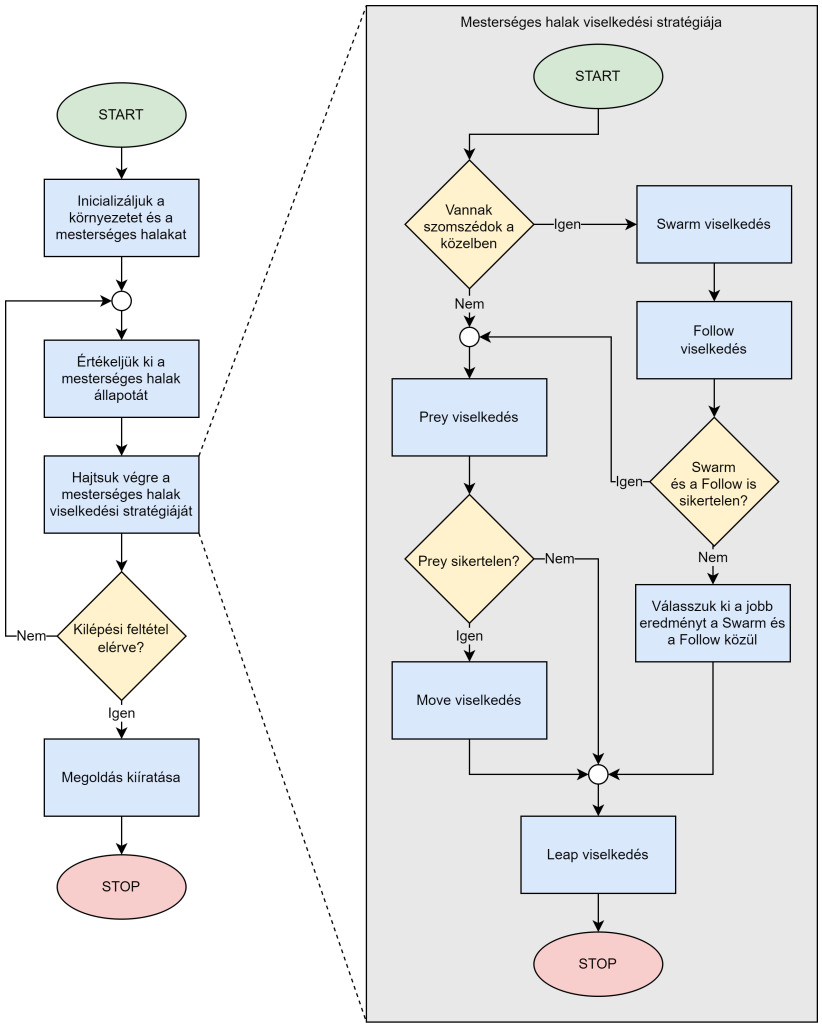
általánosabb, keretrendszerszerű része a módszernek, míg jobb oldalt, a tipikus megvalósításban szereplő viselkedések. Fontos kiemelni, hogy a viselkedési stratégia nem kell, hogy egzakt módon ez legyen, az szabadon változtatható, például új viselkedések hozzáadásával, vagy eltávolításával, de természetesen ennek csak akkor van értelme, ha ezzel jobb teljesítményt vagy eredményeket érünk el.

A következőkben az ábrán szereplő viselkedésekről lesz szó részletesebben. Ehhez szükséges néhány jelölést előre definiálni. Legyen $\mathrm{f}(x_1, x_2, \ldots, x_n)$ a célfüggvényünk melyet minimalizálni szeretnénk, melyben az x_i értékek a meghatározandó paraméterek. Jelölje $X = (x_1, x_2, \ldots, x_n)$ a mesterséges hal állapotát, melyben az x_i értékek a célfüggvény paraméterei. Valamint jelölje $Y = \mathrm{f}(X)$ az X állapothoz tartozó célfüggvény értéket.

A Prey viselkedés

Ezt a viselkedést a halak élelem keresési szokása ihlette. A halak észlelni tudják az élelem koncentrációját a víz adott irányaiban a látásukkal és szaglásukkal, majd ezen információk alapján eldöntik, hogy melyik irányba haladjanak tovább, hogy élelmet szerezzenek.

A viselkedés konkrétabb leírása, a következő. Legyen X_i az adott mesterséges hal aktuális állapota. Generáljunk véletlenszerűen egy X_j új állapotot a hal $Visual$ (észlelési)



3.7. ábra. Az AFSO tipikus lépései.

távolságán belül. Értékeljük ki a generált állapot, vagyis határozzuk meg a célfüggvény értékét: $Y_j = \mathrm{f}(X_j)$, és ha ez az érték jobb, mint az aktuális állapot, haladjunk ebbe az irányba.

Az X_j állapot generálásához az alábbi formulát használjuk fel:

$$X_j = X_i + \mathrm{random}(n) \cdot Visual,$$

ahol a random(n) egy n dimenziós $U(0,1)$ eloszlású véletlen vektort előállító függvény, a $Visual$ pedig a módszer egy paramétere, mely a halak észlelési távolságát adja meg.

Ezt követően ha $Y_j < Y_i$ teljesül, akkor az alábbi formulával lehet meghatározni a következő állapotot:

$$X_i^{(t+1)} = X_i^{(t)} + \frac{X_j - X_i^{(t)}}{\left\| X_j - X_i^{(t)} \right\|} + \mathrm{random}(n) \cdot Step,$$

ahol az $X^{(t)}$ a t-edik iterációhoz tartozó állapotot jelöli, a $Step \leq Visual$ pedig szintén a módszer egy paramétere, mely a halak mozgásának maximális mértékét határozza meg.

Ha nem teljesül az előbbi feltétel, akkor generáljunk egy új X_j-t és próbáljuk újra. Ha nem sikerül $TryPreyLimit$ próbálkozás alatt megfelelő X_j-t generálni akkor ezt a viselkedést sikertelennek minősítjük és folytatjuk a következővel, ami tipikusan a $Move$ viselkedés szokott lenni.

A $TryPreyLimit$ az algoritmus egy paramétere. Nagy értéke esetén az algoritmus több időt fog tölteni a halak lokális környezetének felfedezésével, vagyis az adott lokális optimumok meghatározásával, ellenben kisebb érték esetén, a halak szabadabban tudnak mozogni (függően a többi viselkedéstől), így jobban felderítik a keresési teret és ezáltal több lokális optimumot találnak meg. Ha nagynak választjuk a $Visual$ paramétert, akkor ez a viselkedés eltolódik a globális keresés felé, míg ha kisebbnek, akkor a lokális keresés felé.

A Swarm viselkedés

Ezt a viselkedést a halak csoportosulási szokása ihlette. Ezt a gyakorlatban nem csak élelemszerzési célokból teszik a halak, hanem hogy elkerüljék a különböző ragadozókat, veszélyeket.

A viselkedés konkrétabb leírása, a következő. Legyen X_i az adott mesterséges hal aktuális állapota. Határozzuk meg X_c-t, ami legyen a középpontja a környező halak pozíciójának (állapotának). A környező halak halmazát jelöljük AF_{near}-rel, melyet az alábbiak szerint kapunk meg:

$$AF_{\text{near}} = \{X_j \in AF \mid \text{distance}(X_i, X_j) < Visual\},$$

vagyis a maximum $Visual$ távolságra lévő halak. A továbbiakban jelöljük a környező halak számát k_{near}-rel, a teljes populációt pedig k-val. Ekkor az X_c-t az alábbiak szerint kapjuk meg:

$$X_c = \frac{\sum_{X_j \in AF_{\text{near}}} X_j}{k_{\text{near}}}.$$

Ezt követően, ha $Y_c < Y_i$ teljesül, vagyis jobb a célfüggvény értéke a középpontban és $\frac{k_{\text{near}}}{k} < \delta$, vagyis nincs túlzsúfolva a terület, akkor menjünk közelebb az X_c ponthoz, az alábbi képlettel:

$$X_i^{(t+1)} = X_i^{(t)} + \frac{X_c - X_i^{(t)}}{\left\| X_c - X_i^{(t)} \right\|} + \text{random}(n) \cdot Step.$$

Ellenkező esetben ezt a viselkedést sikertelennek minősítjük. Függetlenül a sikerességtől, a viselkedés befejezése után tipikusan a $Follow$ viselkedés szokott következni.

A feltételben szereplő δ az algoritmus egy paramétere, ami a rajok méretét szabályozza. Nagy értéke esetén nagyobbak lesznek a halrajok, és így kisebb területre összpontosulnak a halak, míg kis értéke esetén kisebbek lesznek a rajok, melytől a halak összességében nagyobb területet tudnak bejárni.

A Follow viselkedés

Ezt a viselkedést a halak követési szokása ihlette. Gyakorlatban, ha egy hal élelemforrást talál, a környezetében lévő többi hal gyorsan követni kezdi őt, amivel ők is élelemhez juthatnak.

A viselkedés konkrétabb leírása, a következő: Legyen X_i az adott mesterséges hal aktuális állapota. Határozzuk meg az X_{best} értéket, amely legyen a legjobb a környező halak állapotai közül. Ezt a korábbi jelölések mellett, az alábbiak szerint kaphatjuk meg:

$$X_{\text{best}} = \inf(AF_{\text{near}}).$$

Ezt követően, ha $Y_{\text{best}} < Y_i$ teljesül, vagyis jobb a célfüggvény értéke itt, és $\frac{k_{\text{near}}}{k} < \delta$, vagyis nincs túlzsúfolva a terület, akkor menjünk közelebb az X_{best} ponthoz, az alábbi képlettel:

$$X_i^{(t+1)} = X_i^{(t)} + \frac{X_{\text{best}} - X_i^{(t)}}{\left\| X_{\text{best}} - X_i^{(t)} \right\|} + \text{random}(n) \cdot Step.$$

Ellenkező esetben ezt a viselkedést sikertelennek minősítjük. Általában a $Follow$ és a $Swarm$ viselkedést együtt szokták vizsgálni. A kettő eredménye közül a jobbat szokták megtartani, ha pedig egyiknek sincs eredménye, akkor folytatják a következő viselkedéssel, ami tipikusan a már említett $Prey$ viselkedés szokott lenni.

A Move viselkedés

Ezt a viselkedést a halak véletlenszerű mozgása ihlette. Ez a legegyszerűbb viselkedési forma is egyben, és az algoritmus során egyfajta tartalék viselkedésként szolgál, ami akkor hajtódik végre, ha minden más sikertelen volt.

A viselkedés konkrétabb leírása, a következő. Legyen X_i az adott mesterséges hal aktuális állapota. Generáljunk véletlenszerűen egy X_r új állapotot a hal $Visual$ (észlelési) távolságán belül és legyen ez az új állapot:

$$X_i^{(t+1)} = X_r = X_i^{(t)} + \text{random}(n) \cdot Visual.$$

Ez a viselkedés mindig sikeresnek van minősítve, de opcionálisan szoktak még egy viselkedést betenni ezután, mégpedig a $Leap$ viselkedést.

A Leap viselkedés

Ezt a viselkedést nem kifejezetten a halak inspirálták, sokkal inkább az optimalizálás javítása érdekében született. Lényege, hogy amikor a konvergencia elkezd lelassulni, akkor elképzelhető, hogy egy lokális minimumpontba sikerült beszorulnunk. Ekkor, ugrásszerűen véletlenszerűen megváltoztatva az állapotokat kikerülhetünk ebből a gödörből.

A viselkedés konkrétabb leírása, a következő. Legyen X_i az adott mesterséges hal aktuális állapota és jelölje $X_{\text{best}}^{(t)}$ a t-edik iteráció legjobb állapotát. Ekkor, ha teljesül az alábbi feltétel:

$$X_{\text{best}}^{(t-l)} - X_{\text{best}}^{(t)} < \epsilon,$$

ahol l a vizsgálandó iterációk száma ($l > 0$), akkor generáljunk egy véletlen számot és ha az nagyobb, mint az előre meghatározott α paraméter, akkor módosítsuk az állapotot az alábbiak szerint:

$$X_i^{(t+1)} = X_i^{(t)} + \text{random}(n) \cdot \beta \cdot Visual.$$

A képletekben szereplő ϵ, α, β az algoritmus paraméterei. Az ϵ a megoldás javulásának minimális mértékét határozza meg, az α a módosítandó halak a teljes populációhoz mért arányát határozza meg ($0 \leq \alpha < 1$), a β pedig a módosítás nagyságának mértékét szabályozza.

Az AFSO módszernél, amennyiben szerepel, akkor tipikusan ez szokott lenni az utolsó viselkedés.

3.4.3. Változatok

Az elmúlt pár évben számos tudós próbált javítani az algoritmuson, így jó pár változat jött létre. Ezek egy része új viselkedési mintákkal bővítette, más része pedig hibrid módon, más módszerekkel ötvözte az eredeti AFSO-t. A következőkben felsorolok néhányat közülük.

- `Improved AFSO (AFSO1)`: Ebben a változatban a globális legjobb hal állapota is figyelembe van véve a halak viselkedése során, így gyorsabb konvergenciát értek el, minimális plusz számítási igény hozzáadásával.
- `Cultured Artificial Fish-swarm Algorithm (CAFAC)`: Ennél a változatnál megjelenik a genetikus algoritmusnál már ismert keresztező operátor. Ezt az operátort arra használja, hogy ötvözze a halak állapotát és így a gyerekek megörököljék a szülők tulajdonságait, vagyis az állapotaikat.
- `Improved Artificial Fish-Swarm Optimization (IAFSO2)`: Ebben a változatban jelenik meg a már részletezett *Leap* viselkedési forma.
- `AFSA-PSO`: Ez a változat gyakorlatilag az AFSO és a PSO kombinációja, ami használja a halaknál tapasztalható viselkedési formákat, kibővítve a részecske raj optimalizációnál látott információkkal.

3.4.4. Összegzés

Az AFSO, mint említettem, relatíve új optimalizálási módszer, így lehetnek még gyermekbetegségei. Az egyik ilyen lenne a szakirodalom pontatlansága, ugyanis számos

tanulmányban pontatlanul, vagy legalábbis nem egyértelműen vannak leírva a módszer lépései. Ennek ellenére számos helyen tudták már alkalmazni, mint például általános optimalizációra, képfeldolgozásra, adatbányászatra, mesterséges neurális hálók javítására, útválasztási problémákra, jelfeldolgozásra. Mindenképp egy megfontolandó választást képez a többi módszer között.

Előnyei közt szerepel a robosztussága, a rugalmassága a változásokra, a globális keresési képessége, az érzéketlensége a kezdeti értékekre, valamint a jó hibatűrése. Rendelkezik a genetikus algoritmus (röviden GA) néhány vonzó tulajdonságával is, mint például a gradienstől való függetlenséggel vagy a komplex nemlineáris sokdimenziós problémákra való alkalmazhatósággal. Nem rendelkezik viszont keresztező és mutáló operátorokkal, amitől könnyebb lehet implementálni, szemben a GA-val.

Hátrányai között szerepel, hogy nehéz lehet megtalálni az egyensúlyt a paraméterekkel a globális és a lokális keresés között, nem használja az előző lépésekben megkapott eredményeket, így egy állapotot akár többször is kiértékelhet, hajlamos a lokális optimumokba beszorulni, valamint a relatíve lassú a konvergenciája.

3.5. Evolved Bat Algorithm (EBA)

Az utolsó általam tanulmányozott ágens alapú optimalizációs módszer az EBA (Evolved Bat Algorithm), ami a denevérek viselkedésén alapszik [54–56]. A módszert Xin-She Yang fejlesztette ki, melyet egy 2010-es tanulmányában ki is fejtett [55]. Ezt a módszert általános optimalizálási feladatokra lehet használni, ami így a mi esetünkben is használható a Steiner probléma megoldására.

3.5.1. A modell áttekintése

A többi módszerhez hasonlóan ez is egy rajintelligencián alapuló megoldás. Az ötlet szokásosan a természetből jött ismét, konkrétan a denevérek mozgásán és tájékozódásán alapszik. Az ágensek szerepét ezesetben a mesterséges denevérek fogják betölteni. Az eddigiekhez hasonlóan minden ilyen ágens rendelkezni fog egy pozícióval, ami a minimalizálandó függvény paramétereinek felel meg. Az ágensek mozgását csak a véletlen és a globális legjobb pozíció fogja befolyásolni.

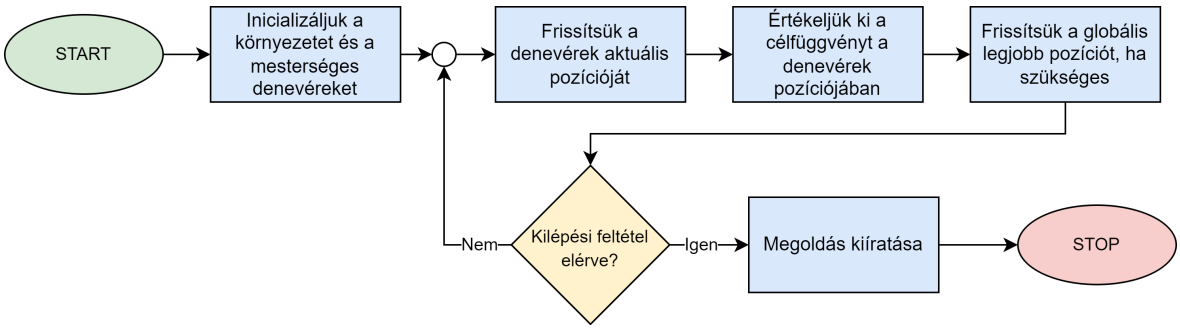
A módszer alapköve az ultrahang alapú tájékozódás koncepciója. Minden egyes virtuális denevér véletlenszerűen mozog a térben, mely során az ultrahangok alapján tájékozódnak. Amint találnak egy zsákmányt (esetünkben egy jó pozíciót), megváltoztatják az ultrahang paramétereit, hogy pontosabban megtudják határozni a zsákmány pozícióját. Ezzel gyakorlatilag folyamatosan finomhangolják a megfelelő egyensúlyt a lokális és a globális keresés között.

3.5.2. Az algoritmus

Az algoritmus lépései láthatóak a 3.8. ábrán, illetve a következő felsorolásban.

- `1.lépés`: Inicializáljuk a mesterséges denevéreinket véletlenszerűen elszórva a keresési térben.
- `2.lépés`: Frissítsük a denevérek pozícióját a következőekben említésre kerülő formulákkal.
- `3.lépés`: Értékeljük ki a célfüggvényt és ha szükséges módosítsuk az globális eddigi legjobb értéket.
- `4.lépés`: Kilépési feltétel ellenőrzése, teljesülése esetén adjuk vissza eredményül az eddigi legjobb értéket, ellenkező esetben vissza a 2. lépéshez.

A második lépés két további részre bontható, egy véletlenszerű mozgásra és egy csoportosulásos mozgásra.



3.8. ábra. Az EBA lépései.

A véletlenszerű mozgás az alábbi formula alapján történik:

$$X_i^{(t+1)} = X_i^{(t)} + D,$$

vagyis egy egyszerű vektorral történő eltolásról van szó, ahol az $X_i^{(t)}$ az i-edik ágens aktuális pozíciója a t-edik iterációban, a D pedig az elmozdulási vektor melyet az alábbi módon kaphatunk meg:

$$D = \frac{v \cdot \Delta T}{2} = 0.17 \cdot \Delta T.$$

Ez a képlet a fizikából már jól ismert út-idő-sebesség képletből származtatható, ahol a D a távolságot, a T az időt, a v pedig a sebességet jelöli. Ezesetben a sebességnek a hang terjedési sebességét vették értékül km/h-ban mérve, ami 0.34 lenne, de mivel az ultrahangnak a távolságot oda vissza is meg kell tennie, hogy a denevérek tudjanak tájékozódni, így ez még el lett osztva 2-vel, így 0.17-et kaptunk. Esetünkben a ΔT egy véletlen generált vektor lesz a $[-1, 1]$ tartományon, ami az elmozdulás irányát és mértékét fogja megadni.

A csoportosulásos mozgás nincs minden esetben végrehajtva, ehhez először generálnunk kell egy véletlen számot 0 és 1 között egyenletes eloszlással, és ha ez nagyobb, mint a paraméterként megadott $PulseRate$, csak akkor módosítjuk a pozíciót az alábbi formula szerint:

$$X_i^{(t+1)} = X_i^{(t)} + \beta \cdot \left(X_{\text{best}} - X_i^{(t)} \right),$$

ahol a β egy véletlen szám 0 és 1 között és X_{best} pedig az eddig globálisan megismert legjobb pozíció.

Az algoritmus egyetlen paramétere a $PulseRate$, ami a csoportosulásos mozgás végrehajtásának valószínűségét adja meg. Minél nagyobb ez az érték, annál hamarabb fog konvergálni a módszer, viszont annál nagyobb is az esély arra, hogy egy lokális minimumot találunk csak.

3.5.3. Összegzés

Véleményem szerint ez a módszer nagyon hasonlít a PSO-ra, éppen ezért én nem különálló módszerként tartanám számon, hanem csak mint a PSO egy változatát. Ez könnyedén belátható, hiszen ez az EBA felfogható úgy, mint a globális topológiával rendelkező PSO egy sajátos paraméterezése, ahol nem vesszük figyelembe a részecskék korábbi legjobb pozícióját.

Egy tekintetben hasonlít az AFSO-hoz is, mégpedig, hogy ez is egy relatíve új optimalizálási módszer, így még vannak gyermekbetegségei, melyek közül itt is a pontatlan megfogalmazásokat és leírásokat említeném a kapcsolódó cikkekben.

Pei-Wei Tsai és kutatótársai tanulmánya [56] alapján a módszer jó eredményeket tud felmutatni a numerikus optimalizálási feladatok kapcsán, és az általuk vizsgált függvények esetén jobb eredményeket is tudott produkálni, mint a PSO.

Habár az említett tanulmány kecsegtető eredményeket ígér, én azért fenntartásokkal kezelném az ott prezentált adatokat, viszont mivel viszonylag kevés lépésből áll az algoritmus, így könnyű implementálni, ezért egy próbát mindenképp megér a hatékonyságának vizsgálata.

4. fejezet

Szimulációs keretrendszerek

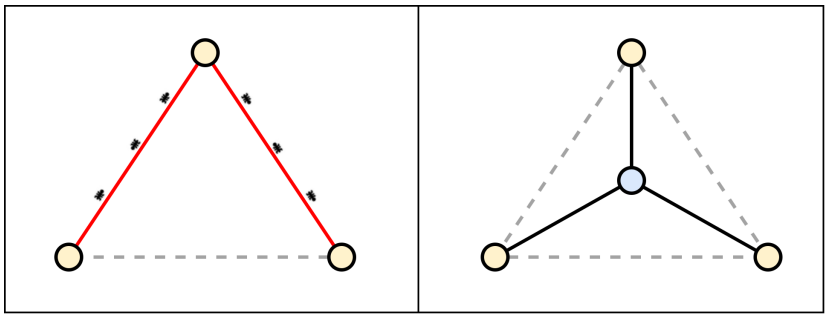
A most következő fejezetben először a Steiner probléma megoldásához készült ötletekről és keretrendszerekről fogok ejteni pár szót, ezt követően pedig az ezek alapján készített alkalmazásról.

4.1. Az ACO alapú ötlet

Ahogy már említettem, az ACO-val alapvetően legrövidebb utakat tudunk meghatározni, mivel valamilyen szinten a Steiner probléma is kapcsolódik ehhez, így adta magát az ötlet, hogy ezzel próbáljam megoldani. A hangyakolónia algoritmusával egyszerűen meg lehet találni a legrövidebb utat egy 2 dimenziós térben, vagy egy annak megfelelő diszkretizált gráfban 2 pont között. Viszont esetünkben több pont alapján kellene megtalálni a legrövidebb összköltségű úthálózatot, ezért szükséges módosítani az algoritmust.

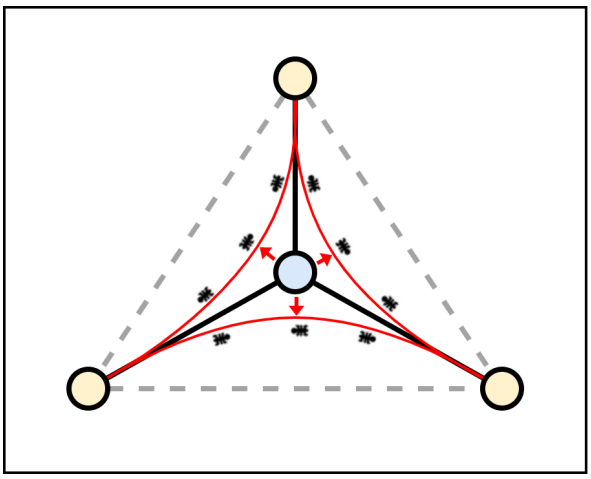
Az algoritmus módosításához először megvizsgáltam, hogy különböző esetekben hogyan viselkedik az. Ekkor arra jutottam, hogy mivel a hangyák letérhetnek a feromonokkal kijelölt útvonalakról, de ezt legtöbbször csak egy rövid ideig teszik, így fokozatosan, apránként elkezdik majd javítani az útvonalat. Ezt követően kipróbáltam hogyan viselkednek a hangyák, ha egy meghatározott pontból indulnak ki, és a célpontjukat pedig véletlenszerűen sorsolom ki a lehetőségek közül. Ekkor azt tapasztaltam, hogy habár minimalizálják az útvonalukat a hangyák, de nem az úthálózat egészére nézve, hanem csak mindig az adott két pont között. Példa erre az egyenlő szárú háromszög 3 pontjának esete, ami a 4.1. ábrán látható, melynél a háromszög két oldalával lettek összekötve a pontok, pedig az optimális a csillag hálózat lett volna.

Ekkor elgondolkodtam, hogyan is lehetne elérni, hogy a pontokat csillag alakban kössék össze a hangyák. Jött az ötlet, hogy ha nem rögzítem a kezdőpontot, hanem azt is véletlenszerűen választom ki és minden hangya egy közös feromon mátrixot használna, akkor elképzelhető, hogy egyenlő arányban minden ponthoz próbálnák optimalizálni az útvonalat. Ugyanis egy adott hangya mindig csak azt tudja hova kell eljutnia, viszont azt nem tudja, milyen útvonalon tudja ezt a leggyorsabban megtenni, ezért ha egy



4.1. ábra. A kezdetleges ACO alapú keretrendszer eredménye (bal oldalt), szemben az optimális eredménnyel (jobb oldalt).

rossz (hosszú) útvonalon fog elindulni (a feromonok miatt), fokozatosan elkezdi azt az adott útvonalat úgy átalakítani, hogy gyorsabban lehessen eljutni az általa elérendő célba (lásd 4.2. ábra). Más hangyák pedig másik célponthoz képest kezdik optimalizálni az adott útvonalakat. Mivel egyenletesen vannak elosztva a kezdő és a végpontok a hangyák között, így minden ponthoz azonos prioritással lesz optimalizálva az úthálózat.

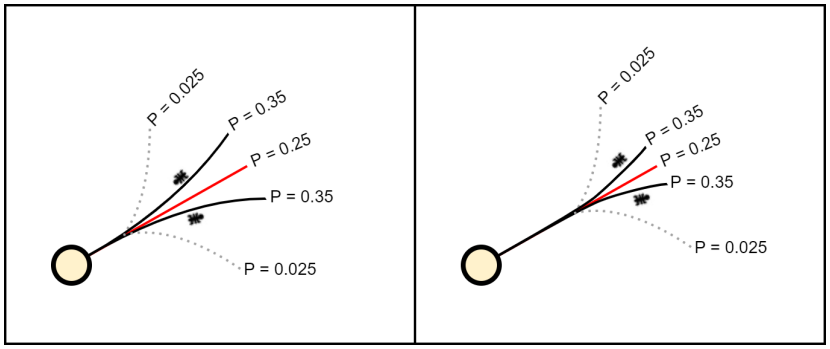


4.2. ábra. Az ACO alapú keretrendszer útoptimalizáló hatása.

Ebben az esetben a megoldásunkat a feromonok jeleznék. Alapvetően szükséges lenne, hogy a legjobb úthálózathoz tartozó pontokon mi is frissítsük a feromonokat, mivel kis számú hangya esetén könnyedén elpárologna nak és nem lehetne meghatározni a végén a megoldást. A feromonok különösebben nem jeleznék mely pontokat kötik össze, így ha egy hangya több irányba is érzékelne feromont, véletlenszerűen kellene döntenie közöttük.

A módszert tovább javíthatja, ha egy viszonylag jó kezdeti megoldásból indulunk ki, amely például lehet a végpontokat legrövidebb úton összekötő hálózat. Ezzel gyakorlatilag csak a konvergenciát gyorsítjuk valamelyest. További optimalizálási lehetőség, ha a feromonokat nem úgy kezeljük, hogy szigorúan csak egy irányra vonatkoznak, hanem az adott iránytól kis mértékben eltérő irányokra is. Ez olyan esetben lesz kifejezetten hasznos, ha például egymáshoz közel található 2 útvonal, melyek ugyebár az úthálózat

szempontjából nagyon költségesek, ilyenkor a két közeli irány mellett a kettő irány átlagát is viszonylag nagy eséllyel választhatják a hangyák, amivel azt érik el, hogy a két útvonal fokozatosan egybeolvad. Ezt szemlélteti a 4.3. ábra.



4.3. ábra. Az ACO alapú keretrendszernél az utak egybeolvadása és a választások valószínűségei.

A megvalósítás során viszont számos problémába ütköztem az ötlet kapcsán. A legelső, hogy a hangyák nehezen találtak célba a túl sok lehetőség miatt, mely sokszor elég kiszámíthatatlan is volt, ami így a számításigény kárára ment. Másik probléma, hogy nehezen találtak vissza a hangyák a feromonokkal kijelölt utakra, mely megint rossz hatással volt az algoritmus időigényére. Illetve még egy probléma, ami a hagyományos ACO esetén is megjelenhet, hogy sokszor léptek olyan helyre a hangyák, melyen már jártak, sőt néha még körbe-körbe is haladtak.

A problémák javítására volt néhány ötletem, az egyik, hogy az útvonalaktól csak bizonyos távolságig engedtem volna a hangyákat elmenni, ezzel biztosítva, hogy hamar visszataláljanak, de ezt csak költséges távolság számítással lehetett volna implementálni, ami még lassabbá tette volna az algoritmust. A másik, ami a körbejárás problémáját orvosolta volna, hogy a hangyák tárolták volna az eddigi útvonalukat, és csak olyan mezőre léphetnének, ahol még nem voltak (egyfajta tabulistaként működve). Azonban ez további problémákat szült volna, mint például a zsákutcába kerülés esélyét, valamint a nagy gráfok és hosszú útvonalak esetén a jelentősen megnövekedett memóriaigényt.

Végezetül azért nem fordítottam több energiát az algoritmus vizsgálatára, mert nagy erőforrásigény mellett lassú konvergenciát tapasztaltam, és nem tűnt úgy, hogy a problémákon érdemben lehetett volna javítani.

4.2. Az AFSO keretrendszere

Egy másik keretrendszer ötlet, melyet a [49] cikkből vettem át, már jobb eredményeket ígért.

Az előzővel ellentétben, ez már ténylegesen egy keretrendszer a probléma megoldásához. Ez gyakorlatilag átalakítja számunkra a problémát egy könnyebben kezelhető

optimalizálási feladatra. Ezt úgy oldja meg, hogy a gráf csomópontjait kódolja 0-1 értékekre, melyeknél az 1 azt jelöli, hogy a csomópontot kiválasztottuk a megoldásunkhoz, míg a 0 azt, hogy nem. Ha megvan minden ponthoz, hogy kiválasztottuk-e vagy sem, akkor a kiválasztott pontok által meghatározott részgráfban kiszámolja a minimális feszítőfát, és ennek a hossza lesz az adott állapot értéke, vagy költsége. Amennyiben éppen a megfelelő pontokat választjuk ki (azokat, amik a minimális Steiner-fában lennének), akkor ez a minimális feszítőfa meg fog egyezni a minimális Steiner-fával. Ezt a 0-1 kódolást akár úgy is el lehet képzelni, mint egy nagyon nagy ($|V|$) dimenziós egységkockát, amiben minimumot kell keresnünk [49–52].

Az ágensek állapota ebben az esetben a teljes gráf lesz 0-1-ként kódolva, illetve az ehhez kiszámolt költség. A 0-1 kódolást nyelvtől függően meg lehet valósítani akár bitekkel is, így minimalizálva a memóriaigényét az algoritmusnak. Habár ez mindenképp egy fontos szempont a keretrendszer tekintetében, ugyanis nagy gráfoknál, például egy 1 millió pontból álló gráfnál, ami az 1024×1024 pixeles képnek feleltethető meg, csak a 0-1-ként kódolt gráf tárolására kell 1 millió bit, vagy esetemben a JavaScript nyelvnél, ahol nem lehet közvetlenül biteket tárolni, 1 millió bájt = 1 megabájt memóriára lesz szükség ágensenként. Könnyen belátható, hogy a kép méretének növelésével arányosan gyorsan növekszik a memóriaigény is. Például egy 2048×2048 pixel felbontású kép már 4 millió pont = 4 megabájt memória. Ezt ellensúlyozhatjuk, kisebb populáció használatával, viszont azzal azt érhetjük el, hogy romlik a konvergencia, illetve, hogy lokális minimumba ragad az algoritmus. A térkép / gráf méretének növelésével arányosan a populációt is illene növelni, hogy megfelelően elkerülhetőek legyenek a lokális minimumok, ami így még inkább a memóriaigény és a teljesítmény kárára fog menni.

A keretrendszer további problémája, hogy a kiválasztott pontok nem feltétlenül adnak ki egy összefüggő gráfot, és egy nem összefüggő gráfban nem lehet feszítőfát megadni, hiszen nem lehet minden pontot elérni minden pontjából. Tehát az ilyen eseteket megfelelően büntetni kellene, ezt pedig úgy érheti el az algoritmus, hogy virtuálisan minden pontot minden ponttal összeköt egy megfelelően nagy költségű éllel. Ezáltal a feszítőfa számolható lesz, és a virtuális él költségétől függően megfelelően büntetve is lesz. A tanulmányban leírtakkal ellentétben, szerintem nem szükséges minden pontot összekötni minden ponttal, hiszen, ha csak a végpontokat kötjük össze, akkor is el lehet jutni minden érintendő pontból minden érintendő pontba, és számunkra csak ez a fontos a költség szempontjából, mivel a virtuális él költsége mindenhol azonos, így lényegtelen, hogy egy félig befejezett út végéből választjuk ki a virtuális élt, vagy a kezdőpontból (lásd 4.4. ábra).

Ennek a virtuális él költségének az eredeti gráfban kiszámolt minimális feszítőfa összköltségét adták. Ezt azzal indokolták, hogy ha a teljes gráfban megoldható akkora költséggel bármelyik két pont összekötése, akkor a részgráfban (amennyiben létezik megoldás) annak biztosan kisebb költségűnek kell lennie. Így, ha egy megoldás tartalmaz virtuális élt, az mindenképp rosszabb lesz egy bármely más nem virtuális élt

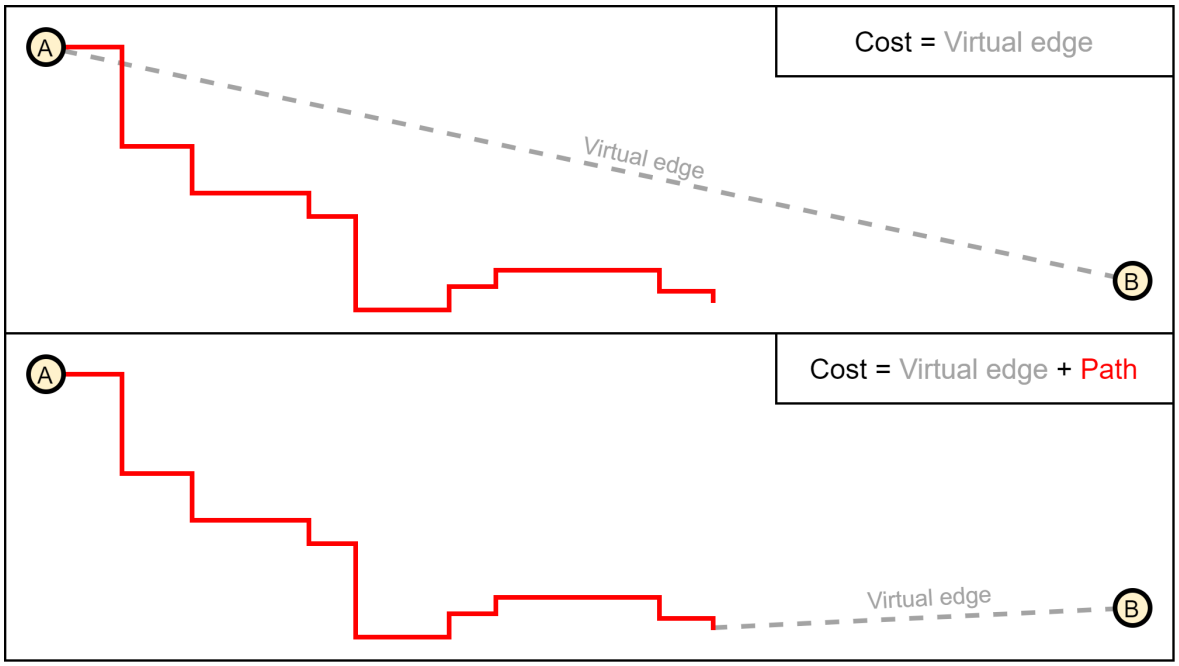
tartalmazó megoldástól (akár csak azt az esetet nézve is, hogy minden egyes pontot beleveszünk). Megjegyzendő, hogy ez a költség lehetne más érték is, de az algoritmus helyessége miatt mindenképp nagyobbnak vagy egyenlőnek kell lennie a teljes gráf MST-jével, de kisebbnek mint végtelen, hiszen a végtelen esetében nem lenne megkülönböztethető az a megoldás, ahol 2 virtuális él van, szemben azzal, ahol csak 1. A konvergencia szempontjából az a jobb, ha ez a szám minél kisebb, ezért esett a választás az MST értékére.

Pontok kiválasztásánál az inicializációs lépésben véletlenszerűen dönti el az algoritmus, hogy melyik pontokat választja ki, ez pedig sok esetben nem egymás melletti pontokat jelent, ami számunkra nem kedvező, hiszen ha nincsenek egymás mellett a pontok, akkor nem lehetnek összefüggőek sem. Ugyan így igaz a későbbi transzformációs lépésekben, amikor az ágensek mozognak. Nem figyel az algoritmus arra, hogy összefüggően válasszon ki új pontokat, mely a konvergenciát, és a megoldhatóságot is rontja, ahogy azt majd később részletezem is miért.

Tapasztalat alapján az 1024 × 1024-es térképnek megfelelő gráfnál, a négy sarkot kiválasztva az előző probléma miatt percek alatt sem tud, nem, hogy optimális, de akármilyen megoldást sem generálni (nem köti össze a kívánt pontokat utakkal) a keretrendszer.

További probléma a keretrendszerben a virtuális élekkel kapcsolatos, ugyanis amíg nincs egy kiinduló megoldásunk, addig nem ösztönzi megfelelően a megoldás keresését. Értem ez alatt, hogy ha van egy már majdnem kész utunk A-ból B-be, akkor ennek a költsége, az útvonal költsége + 1 virtuális él költsége lesz, viszont ha semmilyen útvonalunk nincs, akkor csak a virtuális él költsége számít, mely így kisebb költségű lesz, mint a számunkra kedvezőbb, majdnem kész útvonal esete (lásd 4.4. ábra). Emiatt az optimalizálás eltolódott a minél rövidebb útvonalak létrehozására, a minél rövidebb, de összefüggő útvonalakéról. Megoldható lenne, hogy kiküszöböljük ezt a problémát azzal, hogy nem konstans lenne minden virtuális él költsége, hanem a távolság függvényében változna, de ez sok plusz számítást igényelne minden egyes állapot költségének meghatározásakor, ami nagyban lassítaná az algoritmust (legrövidebb utakat keresni minden nem összefüggő végpontból a már meglévő úthálózathoz, sok végpont esetén sok számítást igényelne).

A tapasztalat alapján nagyobb méretű (közel 1 millió csúcspontú) gráfoknál már nem tudott használható eredményt adni a keretrendszer, mert mindig volt benne virtuális él. Ezt valamelyest kiküszöbölve, utolsó lépésként minden el nem ért végpontból a legrövideb úttal becsatlakoztattam a már meglévő úthálózatba a pontot. Ez korántsem jelentett optimális eredményt, de legalább kaptunk vissza eredményt. A tanulmányban ezt annyival megoldották, hogy mivel a véletlenre bízták a keresést, az algoritmus egyszer úgyis fog találni olyan megoldást, amiben nincs virtuális él, és ha egy ilyen már van, attól kezdve megfelelően van honorálva az optimális érték irányába történő elmozdulás, tehát idővel konvergálni fog.



4.4. ábra. Az AFSO keretrendszernél a virtuális éllel számított útköltség 2 lehetséges esete.

A tanulmányban az AFSO ágens alapú modellt alkalmazták, de bármilyen általános optimalizáló módszert lehetne alkalmazni, mivel keretrendszerről van szó. Ebben az esetben az AFSO-nak csak a *Prey*, *Move* és *Follow* viselkedéseit implementálták. Ezekhez szükséges volt azonban további módosításokra is, így például a mozgást a bitek átbillentése jelentette, a távolságszámítást a Hamming távolsággal oldották meg.

Saját véleményem, hogy az AFSO nem épp a legszerencsésebb választás volt ezesetben. Ugyanis a *Prey* viselkedés eléggé hasonlít a *Move* viselkedésre, viszont sokkal több számításigénye van, hiszen amíg a *Follow* és a *Move* mozgás során 1 pozíciónak kell kiszámolni a célfüggvény értékét, addig a *Prey* esetében akár $TryPreyLimit$ számszor is. Ez pedig azért problémás, mert a gráf kódolása és az egyébb számítások miatt nagyon költséges kiértékelni a célfüggvényt. Szerintem jobb lett volna, ha a *Prey* viselkedést kicsit átalakították volna: csak 1 új pozíciót generáljon ki, hasonlóan, mint a véletlen bolyongáskor, csak itt nem véletlenül választaná ki az új pozíciót, hanem úgy, hogy olyan távol kerüljön a környezetében lévő aktuális legjobb másik ágenstől, hogy ne tudja már azt követni, vagyis legalább $Visual$ távolságra. Ezzel ugyanazt érték volna el, mint eredetileg, vagyis, hogy a lokális minimumba csak egy része kerülne bele az ágenseknek, a többi automatikusan elmenne onnan.

Egy másik lehetőség lehetne, hogy az AFSO helyett a PSO-t implementáljuk, pontosabban annak a bináris változatát, ahol ugyebár a momentumot, nem mint sebességet, hanem mint a bitek „ragadósságát” értelmezzük, vagyis mennyire nehéz megváltoztatni az adott bitet, azaz mekkora rá az esélyünk.

Az előzőhöz hasonlóan, a hangyakolóniás implementáció is érdekesnek hangzik. Eh-

hez konkrétan még nem találtam tanulmányt, viszont mindenképp megérne szerintem egy kutatást. Egy egyszerű implementáció annyiból állna, hogy minden ágens az általa kiválasztott pontokra (aktuális pozíció) az aktuális költség alapján tenne bizonyos mennyiségű feromont. Amikor új pozíciókat generálnának, akkor a feromonok is közrejátszanának egy pont kiválasztásának a valószínűségében. Ennek, amennyiben valamilyen problémára nem derülne fény, az lenne az előnye, hogy nem kellene egyetlen ágensnek sem megtalálni a konkrét optimális megoldást, elég lenne csak mindegyiknek egy optimális szakaszt találnia, melyekből végül összeállna a végső optimális megoldás.

Végül ezt a megoldási ötletet is elvetettem, mivel túl lassú volt a konvergencia és nem voltak elfogadhatóak a visszaadott eredmények. A konvergencián tudtam volna még javítani, de a rossz eredményeken (virtuális éleket tartalmazókon), már nem tudtam. Az említett cikkben természetesen voltak teszteredmények is, de ezek csak kisebb, maximum pár ezer csúcspontú gráfokhoz készültek. Természetesen adódik az a feltételezés, hogy nagy gráfok esetén ezt a keretrendszert nagyon sok ideig kellene futtatni. Természetesen nem a cikket szeretném kritizálni, hiszen elképzelhető, hogy valamit én nem értelmeztem úgy, ahogy a szerző gondolta, de a legjobb tudásom szerint próbálva, a már leírt következtetésekre jutottam.

4.3. A saját keretrendszer

Ennek a keretrendszernek az ötlete onnan jött, hogy rájöttem, hogy attól függetlenül, hogy a gráfokban egy diszkretizált változatát próbálom megoldani a Steiner problémának, attól még gyakorlatilag ez a geometriai Steiner problémával lesz ekvivalens. Ez azért fontos, mert a geometriai Steiner problémánál bizonyított, hogy a közbenső elágazások, vagyis a Steiner pontok száma maximum $|S| - 2$ lehet. Ez könnyedén belátható, hiszen $|S| = 2$ esetben nincs közbenső pontunk, mert a két pont a legrövidebb úttal lesz összekötve. A többi esetet pedig lehet úgy értelmezni, hogy az eggyel kevesebb ponttal rendelkező probléma kibővítése, oly módon, hogy az új pontot valahol be kell csatlakoztatni az úthálózatba, úgy, hogy az optimális legyen. Ezt a helyet az adott ponthoz tartozó Steiner pont fogja betölteni.

Éppen ezért kijelenthető, hogy elegendő számunkra csak ezeknek a Steiner pontoknak a meghatározása is, melyből már egyértelműen meghatározható a megoldás. Az is könnyedén belátható, hogy a maximum $|S| - 2$ pont meghatározását értelmezhetjük úgy is, hogy pontosan $|S| - 2$ pontot kell meghatároznunk, melyek eshetnek akár a hozzájuk tartozó végpontokra is, vagy más Steiner pontokra, ami az eredeti értelmezésben azt jelentené, hogy nincs is Steiner pontunk.

Minden egyes Steiner pont pozíciójára maximum $|V|$ lehetőségünk van, vagyis viszonylag kicsi $|S|$ esetén sokkal kevesebb lépésből meg lehet határozni az optimális megoldást, mint például az előző keretrendszer esetén. Másképp fogalmazva sokkal jobban skálázódik a gráf méretének növekedésével ez a megoldás az előzőhöz képest.

4.3.1. Számítási komplexitás

A számítási komplexitást a következő példán keresztül vezetem le. Legyen a vizsgálandó térképünk egy 1024 × 1024 pixelből álló kép. Ekkor a gráfunk csúcsainak száma:

$$|V| = 1024 \cdot 1024 = 1048576$$

lesz. Legyen továbbá a végpontjaink száma

$$|S| = 10.$$

Ekkor az AFSO keretrendszernél, mivel minden ponthoz el kell döntenünk, hogy kiválasztjuk-e vagy sem, az összes lehetséges megoldás száma a következőek szerint alakul:

$$MaxStep = 2^{|V|} = 2^{1048576},$$

ami egy hatalmas szám. Ellenben ennél a keretrendszernél csak $|S| - 2$ pontot kell meghatároznunk, melyek mindegyike maximum $|V|$ féle képpen állhat elő, vagyis a lehetséges megoldások maximális száma:

$$MaxStep = |V|^{(|S|-2)} = 1048576^8,$$

ami már egy jóval kisebb szám.

Megjegyzendő, hogy ha a gráfunk kicsi, és az S sok pontot tartalmaz belőle, akkor az AFSO keretrendszer esetében lesz kevesebb a lehetőségek száma. Az adott gráfmérethez mindig meghatározható (a következő módon) a végpontok maximális száma, aminél még az általam felvetett keretrendszernél lesz kevesebb lehetőség:

$$k_{\max} = \frac{|V| \cdot \log(2)}{\log(|V|)},$$

ahol a $k_{\max} = |S| - 2$ a végpontok számát jelöli. A különböző gráfméretekhez kiszámolt $k_{\max}$ értékei láthatóak a 4.1. táblázatban.

Gráfméret	$k_{\max}$ értéke
$\|V\| = 16 \cdot 16$	32
$\|V\| = 1024 \cdot 1024$	52428
$\|V\| = 2048 \cdot 2048$	190650

4.1. táblázat. A saját keretrendszer $k_{\max}$ értékei különböző gráfméretek esetében.

Az is könnyedén meghatározható, hogy relatíve mennyivel kevesebb a lehetőségek száma a két keretrendszer esetében, mégpedig:

$$SpeedUp = \frac{2^{|V|}}{|V|^{(|S|-2)}}.$$

Gráfméret	Végpontok száma	Relatív gyorsítás
$\|V\| = 16 \cdot 16$	$\|S\| = 10$	$6.2771017353 \cdot 10^{57}$
$\|V\| = 1024 \cdot 1024$	$\|S\| = 10$	$4.6124752469 \cdot 10^{315604}$
$\|V\| = 2048 \cdot 2048$	$\|S\| = 10$	$2.1560269349 \cdot 10^{1262558}$

4.2. táblázat. A saját keretrendszer relatív gyorsítása az AFSO keretrendszerhez képest különböző gráfméretek esetében.

A különböző gráfméretekhez számított relatív gyorsítás a 4.2. táblázatban látható.

Ennek a hatalmas különbségnek az oka, hogy fix $|S|$ esetén az általam létrehozott keretrendszer lehetőségeinek száma a $|V|$ egy polinomiális függvénye lesz, míg az AFSO esetén ez egy exponenciális függvény, az pedig már bizonyított, hogy egy polinomiális függvény egy bizonyos határértéktől kezdve mindenképp kisebb lesz, mint egy exponenciális függvény. Megjegyzendő, hogy a kiszámolt értékek csak a maximális lehetőségek számára, vagyis a maximális lépésszámra vonatkoznak, a konkrét időkomplexitás függ még az adott lépés időkomplexitásától is. Mivel ennek a meghatározása már bonyolultabb lett volna, ezért ezzel már nem is foglalkoztam, de valószínűsítettem, hogy akkora különbség nem lenne a két lépés időkomplexitása között, mint amit az előbb tapasztaltunk a lépésszám esetén.

4.3.2. A keretrendszer lépései

A keretrendszer működése során az alábbi lépéseket hajtja végre.

- `1. lépés`: Minden S-beli végponthoz számítsuk ki a tőle vett legrövidebb utakat a gráf összes többi pontjába. Ez egy egyszerű legrövidebb útkereséssel megoldható (például Dijkstra algoritmussal), melyet csak akkor állítunk le, ha már érintettük a gráf összes pontját. Ekkor minden végponthoz tudni fogjuk, hogy egy véletlenszerűen kiválasztott pontból mekkora legkisebb költséggel lehet eljutni oda.

- `2/a. lépés`: Ha $|S| = 3$, vagyis csak egy Steiner pontot kell meghatároznunk, akkor elég a 3 végpont legrövidebb útjainak költségeit tartalmazó mátrixokat összeadni és a legkisebb értéket kiválasztani. Ennek a költségnek megfelelő pont lesz a minimális Steiner-fához tartozó Steiner pontunk. Ezzel az algoritmus véget is ér.

- `2/b. lépés`: Ha $|S| > 3$, vagyis több Steiner pontot kell meghatároznunk, akkor minden ágensre hajtsuk végre a következőket.

  - `2.1. lépés`: Valamilyen algoritmus alapján határozzuk meg az ágens pozícióját, ami a Steiner pontok pozícióit fogja reprezentálni.

  - `2.2. lépés`: Az ágens Steiner pontjai közül a legelsőt leszámítva, számítsuk ki az 1. lépésben megismert módon, a legrövidebb utakat a gráf összes többi pontjához. Ekkor már azt is tudni fogjuk, hogy a Steiner pontok között mekkorák lesznek a legrövidebb utak.
  - `2.3. lépés`: Ezt követően írjuk fel a végpontokból és az ágens Steiner pontjaiból álló teljes gráfot, a már megismert legrövidebb utak hosszának segítségével. (Ez a gráf pontosan $|S| \cdot |S| - 2$ nagyságú lesz.)
  - `2.4 lépés`: Ebben a gráfban számoljunk egy minimális feszítőfát, például Prim algoritmusával. Ezzel megkapjuk, hogy pontosan mely pontokat is kell összekötnünk.
  - `2.5 lépés`: A teljes gráf feszítőfájában szereplő éleket alakítsuk vissza a hozzájuk tartozó legrövidebb utakra (az eredeti gráfban).
  - `2.6 lépés`: A célfüggvényünk értékét a kiválasztott legrövidebb utak összköltsége adja meg.

- `3.lépés` Kilépési feltétel ellenőrzése. Teljesülése esetén adjuk vissza a legjobb ágens által meghatározott úthálózatot, ellenkező esetben ismételjük a 2/b. lépéstől.

4.3.3. Értékelés és összegzés

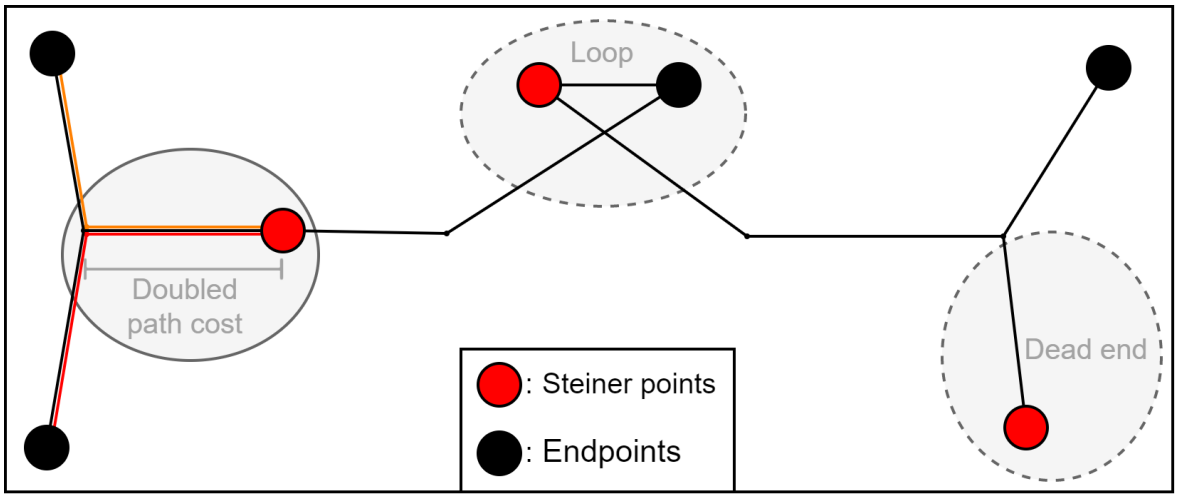
Ami hátrányt találtam az algoritmusban, hogy ugyan ahhoz az úthálózathoz több különböző megoldást is tud generálni különböző célfüggvény értékekkel. Ez azért lehetséges, mert hiába tettük fel, hogy a Steiner pontoknak az elágazásoknak kell lennie, de a minimális feszítőfa számításakor elképzelhető, hogy pont úgy kerülnek kiválasztásra az élek, hogy mégse lesznek azok. Ez pedig azért történik így, mert így lesz a legjobb az úthálózat összköltsége.

Emiatt megeshet az is, hogy egy Steiner pont valójában egy útvonal közepére esik, vagy egy zsákutca végpontja lesz. Azoknál az eseteknél, ahol a Steiner pont egy zsákutcát eredményez, egy utólagos lépéssel ki lehet szűrni ezeket a zsákutcákat, de a költségnél ettől függetlenül ugyanúgy meg fog jelenni az értéke.

Ugyanezen okok miatt megeshet, hogy bizonyos útszakaszokat többször is figyelembe vesz az algoritmus az összköltség kiszámításakor. Ebben az esetben a tényleges elágazás és a Steiner pont közötti szakaszt számolhatja többszörözötten az algoritmus, mivel ez a szakasz több útvonalhoz is tartozhat.

Ezek az anomáliák azért nem feltétlenül jelentenek tényleges problémát, mert ezekre az esetekre egyszerűen mondhatjuk azt, hogy nincsenek elég jól megválasztva a Steiner pontjaink. Ugyanis a feladatunk a Steiner pontok pozícióinak meghatározása lenne, de attól függően milyen messze kerülnek a tényleges elágazás helyétől ezek a pontok, annál

nagyobb lesz a célfüggvényünk értéke is, vagyis hosszútávon, a konvergencia miatt, ezek az anomáliák el fognak tűnni, nem fogják befolyásolni az algoritmus helyességét.



4.5. ábra. A saját keretrendszer problémái: duplikált útvonal költség, hurok, zsákutca. (A fekete pontok a végpontokat, a piros pontok a kiválasztott Steiner pontokat jelölik.)

Az előző keretrendszerhez hasonlóan ennek is előnye, hogy szabadon választhatjuk meg milyen optimalizálási módszerrel szeretnénk végrehajtani a minimalizálást. Egy további előnye a másik algoritmushoz képest, hogy mindig érvényes megoldást ad vissza, így akármikor leállítható az algoritmus, ha nem szeretnénk tovább várni, legrosszabb esetben csak egy, az optimálistól távolabbi megoldás lesz az eredményünk.

Véleményem szerint, ha a geometriai Steiner problémát szeretnénk megoldani a gráfok segítségével, és elfogadjuk, hogy nem lehet akármekkora az érintendő pontjaink halmaza, akkor ezt a keretrendszert mindenképp érdemes kipróbálni, ellenkező esetben pedig használhatjuk az AFSO által inspirált keretrendszert.

4.4. Az alkalmazás

A következőkben az elkészített alkalmazást fogom bemutatni. Első sorban az elérendő célokat határozom meg, majd megtervezem azok elérési módját, ezután bemutatom a konkrét megvalósítást.

4.4.1. Célok meghatározása

A dolgozathoz készített alkalmazás célja, a korábban már taglalt elméleti módszerek implementálása olyan módon, hogy azok könnyedén kipróbálhatóak legyenek a felhasználó számára. Emiatt az alábbi elérendő célokat tűztem ki az alkalmazással szemben:

- platformfüggetlenség,
- közös felhasználói felület a módszerekhez,

- magasságmezők generálása,
- magasságmezők térképszerű színezése,
- végpontok elhelyezésének lehetősége,
- végpontokat összekötő optimális úthálózat meghatározása,
  - különböző módszerek implementálása,
  - paraméterek állításának lehetősége,
- eredmények mentésének lehetősége,
- amennyiben nem biztosítható a pár percen belüli futásidő, legyenek megszakíthatóak a számítások, folyamatos visszajelzéssel az aktuális állapotról.

A célok elérésének tényleges mivoltáról a következő részben lesz szó.

4.4.2. Tervezés

Ahhoz, hogy a korábban megnevezett célokat biztosan elérjük, szükséges az alkalmazást előzetesen megtervezni. Ebben a részben erről lesz szó.

A közös felület és a platformfüggetlenség biztosítását azzal oldom meg, hogy gyakorlatilag egy webalkalmazást fogok készíteni, ami a böngészőben fog futni. Ehhez kiválasztásra került az Angular keretrendszer, aminek a komponens alapú felépítése miatt könnyen lehet a felhasználói felületeket elkészíteni, és a Service rétege, illetve a függőség kezelése miatt könnyen el lehet különíteni a felület logikáját a térképgenerálás logikájától. Ezt az elkülönítést még tovább is fokoztam azzal, hogy külön létrehoztam egy „UI” és egy „Library” modult.

Mivel a weboldalakon többnyire JavaScript-et lehet csak futtatni, így programozási nyelvnek azt kellett választanom. Azonban mivel volt lehetőségem, így éltem is azzal, hogy a JavaScript forráskód közvetlen megírása helyet TypeScript programozási nyelvet használjak. A TypeScriptet a JavaScript egy típusokkal kiegészített változataként tekinthetjük. A benne megírt kód JavaScript nyelvűre fordul egy fordítási folyamat során, vagyis a böngésző nem is fog tudni a TypeScript változatról.

A JavaScript a teljesítmény szempontjából azonban nem a legjobb választás, mivel csak egy szálon tud futni, illetve dinamikus nyelv jellege lévén számításigényes feladatokra nem feltétlenül ideális. Esetemben nem volt elsődleges szempont a teljesítmény, és ebben a nyelvben van a legtöbb tapasztalatom, ezért ennél a nyelvnél maradtam. Egy dolgot kellett megoldanom: hogy ha sokáig tart egy számítás, akkor az ne blokkolja a fő szálat, mert attól a felület megfagyna, de ez is megvalósíthatónak bizonyult a WebWorker-ök segítségével, amiről majd az optimalizációk során több szót is ejtek még.

A Library modul

Ez a modul fogja a tényleges szimulációkat végezni, melyet a UI modul fog majd felhasználni. Feladatköre a magasságmezők és úthálózatok generálása egy kiválasztott módszerrel és az inputként megkapott paraméterekkel.

A modult a következő részekre bontottam fel:

- MST számító,
- elsőbbségi sor,
- legrövidebb út számító,
- ágensek (több változat),
- szimulációk (több változat).

A részek kapcsolatait a 4.6., a 4.7., a 4.8., a 4.9., a 4.10. és a 4.11. ábrákon lehet megfigyelni. Látható, hogy több absztrakciós szintet is létrehoztam, hogy minél jobban elkerülhessem a kódduplikációt. A következőkben az osztálydiagramon szereplő osztályokról fogok részletesebben beszélni.

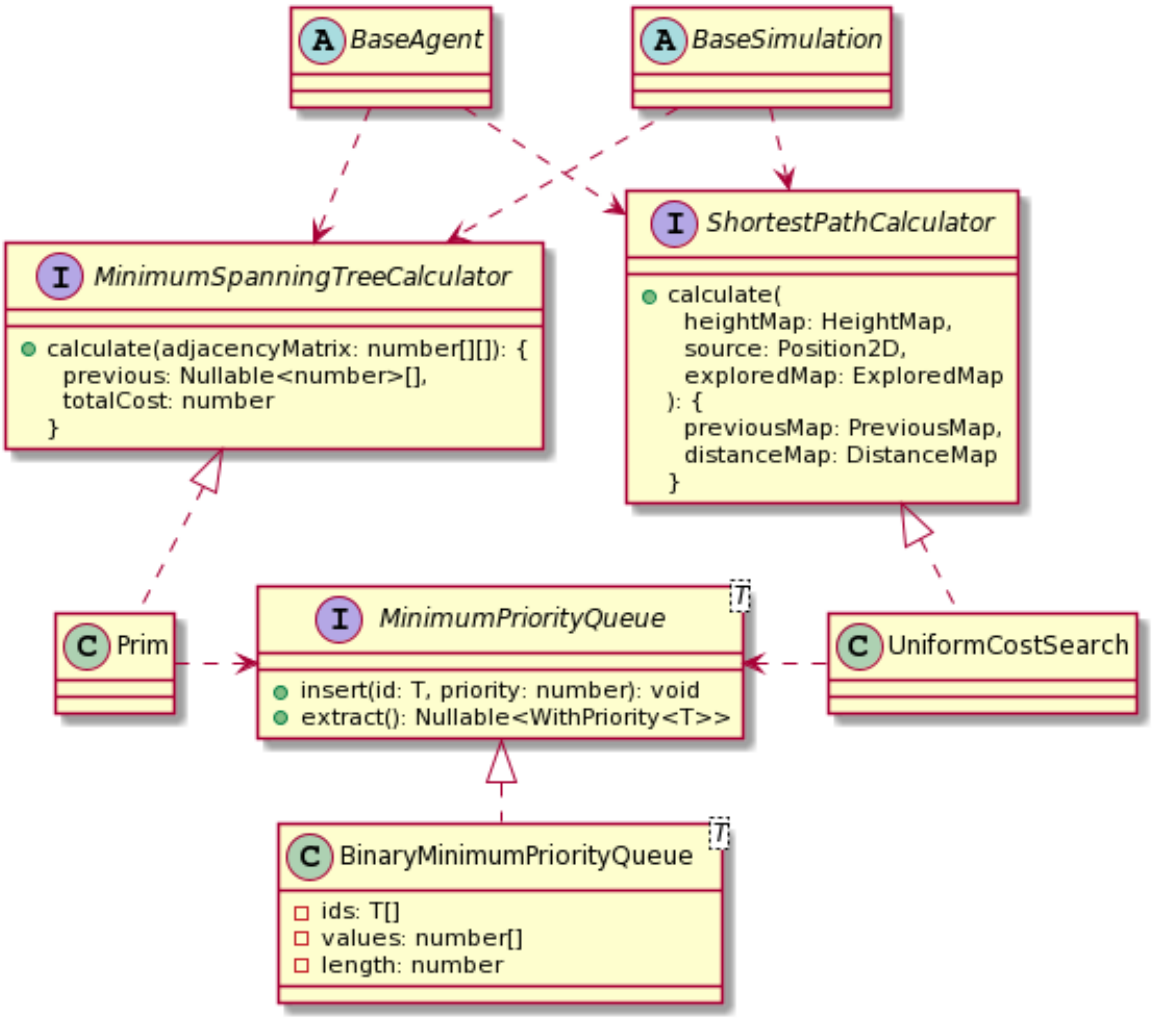
A `BinaryMinimumPriorityQueue` osztály feladata az elsőbbségi sor adatstruktúra implementálása lesz. Két metódusa lesz, az *insert* és az *extract*, melyek az elemek beszúrásáért és kivételéért felelnek majd. Ezt az osztályt a legrövidebb út (`UniformCostSearch`) és a minimális feszítőfa (`Prim`) számításánál fogom felhasználni.

A `Prim` osztály feladata a minimális feszítőfa kiszámítása a megadott gráfra. Egyetlen metódusa a *calculate(. . .)*, ami az előbb említett minimális feszítőfát számolja ki a paraméterben megkapott gráfra. Ehhez fel fogja használni a `MinimumPriorityQueue` implementációt.

A `UniformCostSearch` osztály feladata a legrövidebb utak számítása lesz, a megadott gráf megadott kezdőpontjából. Ennek is csak egy metódusa lesz, a *calculate(. . .)*, ami az említett legrövidebb utakat fogja kiszámolni a paraméterként megkapott gráfra és kezdőpontra. Ez a metódus utolsó paraméterében opcionálisan megkapja a figyelmen kívül hagyandó pontokat. A számításokhoz fel fogja használni a `MinimumPriorityQueue` implementációt.

A `BaseSimulation` egy absztrakt osztály, ami a szimulációk közös metódusait fogja implementálni oly módon, hogy ha szükséges, azt tudják bővíteni a leszármazottak. Ehhez felhasználja a `ShortestPathCalculator` és a `MinimumSpanningTreeCalculator` implementációit. Az osztály metódusai a *run(. . .)*, melyet a leszármazottnak kell majd implementálni, az *updateBestAgentState(. . .)*, ami frissíti az eddig megtalált legjobb állapotot, illetve a *getEdgesForBestAgentState(. . .)*, ami a végeredményt konvertálja át megjeleníthető útvonal szegmensekre.

A `BaseAgent` egy absztrakt osztály, ami az ágensek közös metódusait fogja implementálni oly módon, hogy ha szükséges, azt tudják bővíteni a leszármazottak. Ehhez



4.6. ábra. A Library modul osztálydiagramja (főbb részek).

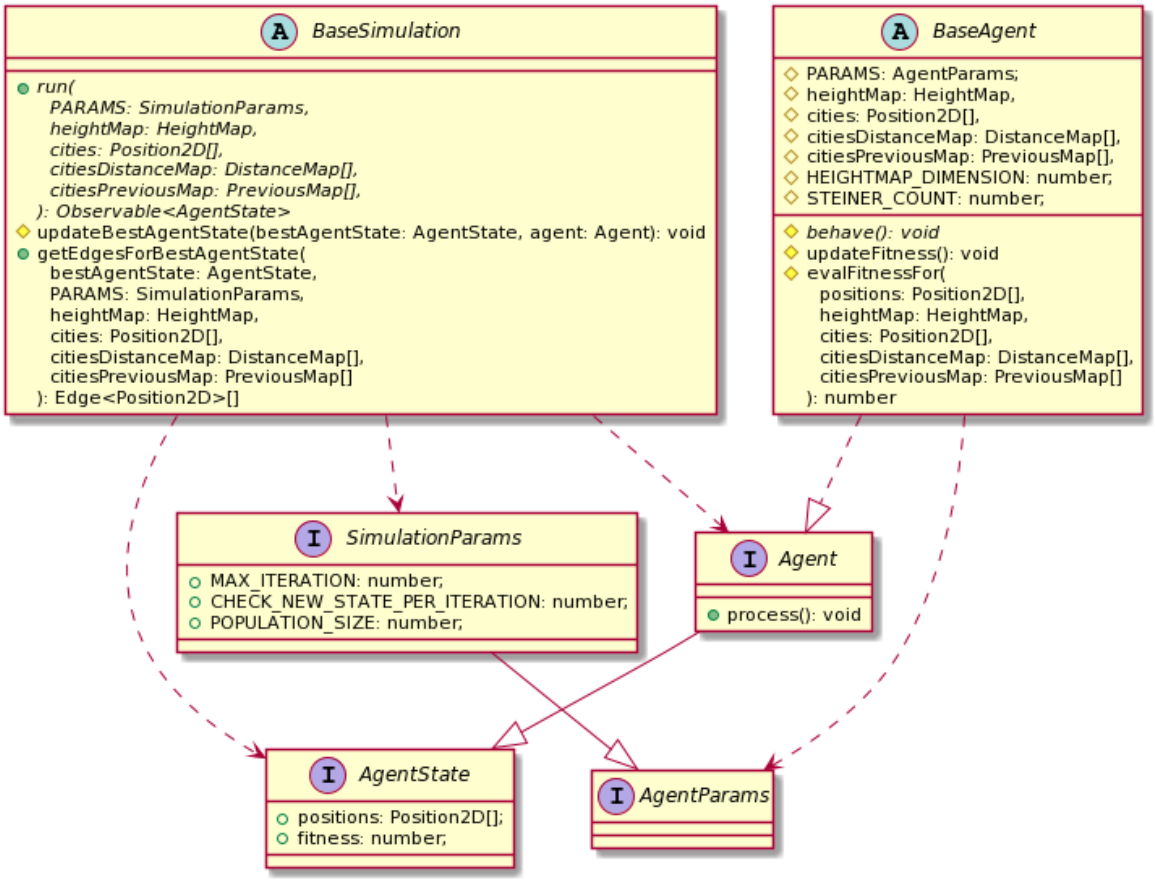
felhasználja a `ShortestPathCalculator` és a `MinimumSpanningTreeCalculator` implementációit. Az osztály metódusai a *behave(. . .)*, amit a leszármazottaknak kell majd implementálni, és ez fogja meghatározni az ágens konkrét viselkedését, az *updateFitness(. . .)*, ami frissíti az ágens költségét, amit az utolsó metódus, az *evalFittnesFor(. . .)* segítségével fog tudni meghatározni az aktuális pozíció alapján.

A `*Agent` osztályok a megfelelő módszerhez tartozó ágens implementációk, melyek mindegyike kibővíti a `BaseAgent` absztrakt osztályt.

A `*Simulation` osztályok a megfelelő módszerhez tartozó szimuláció implementációk, melyek mindegyike kibővíti a `BaseSimulation` absztrakt osztályt.

A `*AgentParams` ami a megfelelő módszer paraméterei közül azokat tartalmazza, melyekre az ágenseknek szüksége van.

A `*SimulationParams` ami a megfelelő módszer paramétereit tartalmazza. Ebbe már beletartoznak az előbb említett `*AgentParams` osztály tagjai is.



4.7. ábra. A Library modul osztálydiagramja (absztrakt osztályok).

A UI modul

Ennek a modulnak a feladata a megjelenítés lesz, valamint a számításokhoz szükséges adatok tárolása és átadása a Library modul felé.

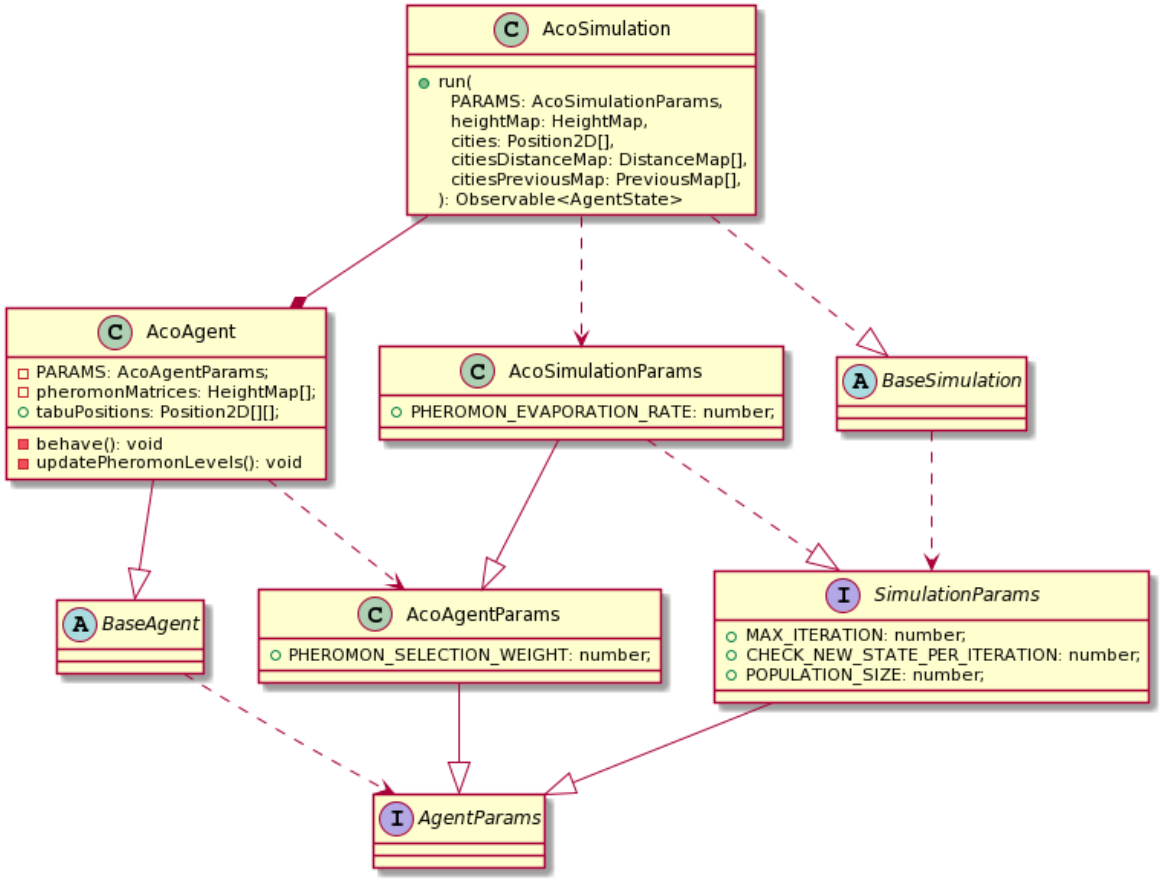
A modult a következő főbb részekre bontottam fel:

- UI komponensek,
- magasságmező generáló szervíz,
- úthálózat generáló szervíz,
- Store a különböző adatok tárolására.

A részek kifejtését és kapcsolatait a 4.12. ábrán lehet megfigyelni. A következőkben a komponensdiagramon szereplő komponensekről fogok részletesebben beszélni.

Az ábrán látható, hogy 3 csomagba szerveztem a komponenseket, melyekről szintén ejtenék pár szót.

A `UI Components` csomag a tényleges megjelenítésért felelős komponenseket tartalmazza. Ezek a megjelenítéshez szükséges adatokat rendszerint a `Store` csomagban található komponensektől kapják. A felületen elvégzendő interakciók során, például



4.8. ábra. A Library modul osztálydiagramja (ACO változat).

egy gombra kattintáskor, a `Store`-ba akciókat küldenek be, mely hatására az majd megváltoztatja az alkalmazás állapotát.

A `Sidebar` komponens feladata az oldalsó menüsáv megjelenítése és kezelése lesz. Ehhez fel fogja használni a `Terrain Generator Forms`, a `Road Generator Forms` és az `Export` komponenseket.

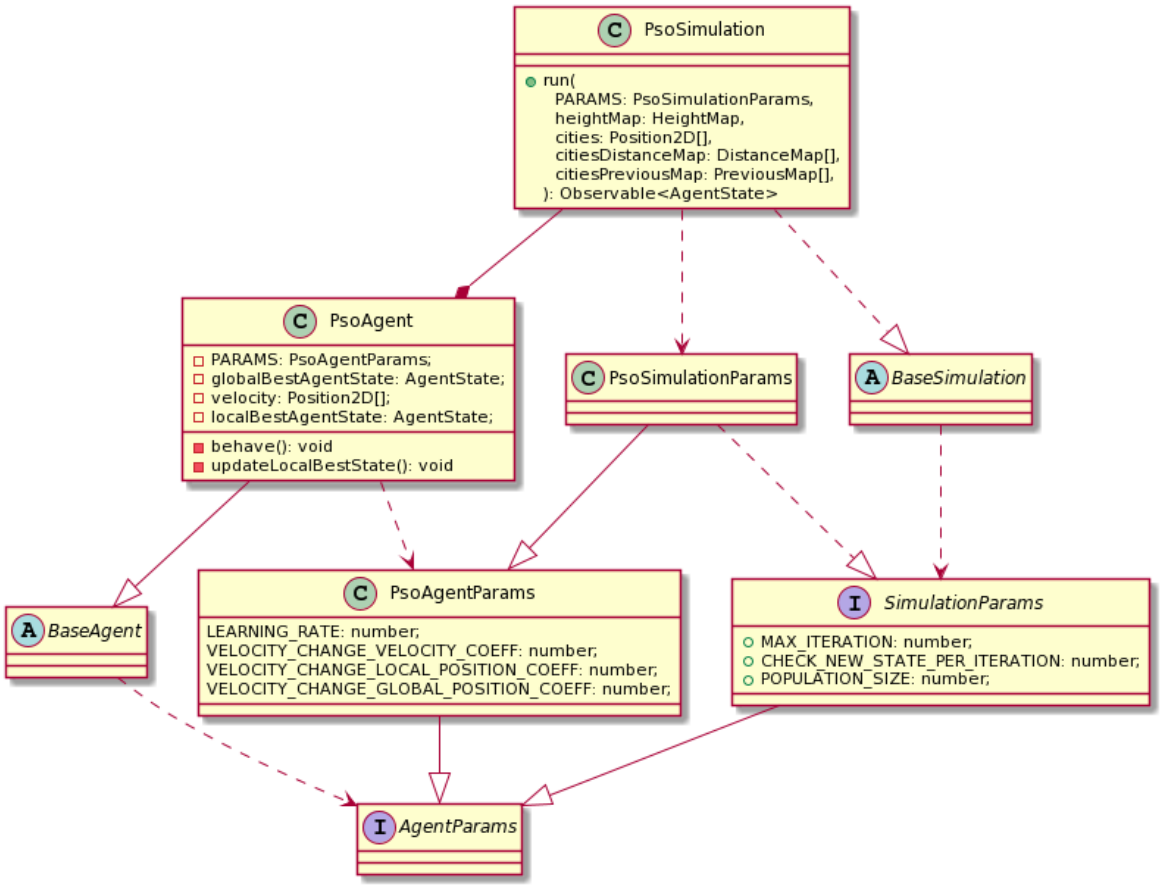
Az `Image Preview` komponens a generált képek megjelenítésért fog felelni.

A `Terrain Generator Forms` komponens a magasságmező generálásának űrlapját fogja kezelni és megjeleníteni. Lehetőséget fog adni egy algoritmus kiválasztására, valamint a hozzá tartozó paraméterek beállítására.

A `Road Generator Forms` komponens az úthálózat generálásának űrlapját fogja kezelni és megjeleníteni. Lehetőséget fog adni egy algoritmus kiválasztására, valamint a hozzá tartozó paraméterek beállítására.

Az `Export` komponens a generált képek letöltéséhez szükséges linkeket fogja kezelni és megjeleníteni.

A `Store` csomagban találhatóak az alkalmazás állapotát tartalmazó komponensek. Ezek visszaadják az állapotot a többi komponens számára, valamint figyelve az akciókra, ha szükséges meg is változtatják azt, és ilyenkor értesítik a többi komponenst a változásról. Ezek a komponensek egymással is kommunikálnak, például az egymástól



4.9. ábra. A Library modul osztálydiagramja (PSO változat).

függő adatok esetén.

A `Terrain` komponens a magasságmező generálás algoritmusát, a hozzá tartozó paraméterértékeket, valamint ezeknek az eredményét fogja tárolni.

A `Cities` komponens az elhelyezett városok pozícióit fogja tárolni.

A `Roads` komponens az úthálózat generálás algoritmusát, a hozzá tartozó paraméterértékeket, valamint ezeknek az eredményét fogja tárolni.

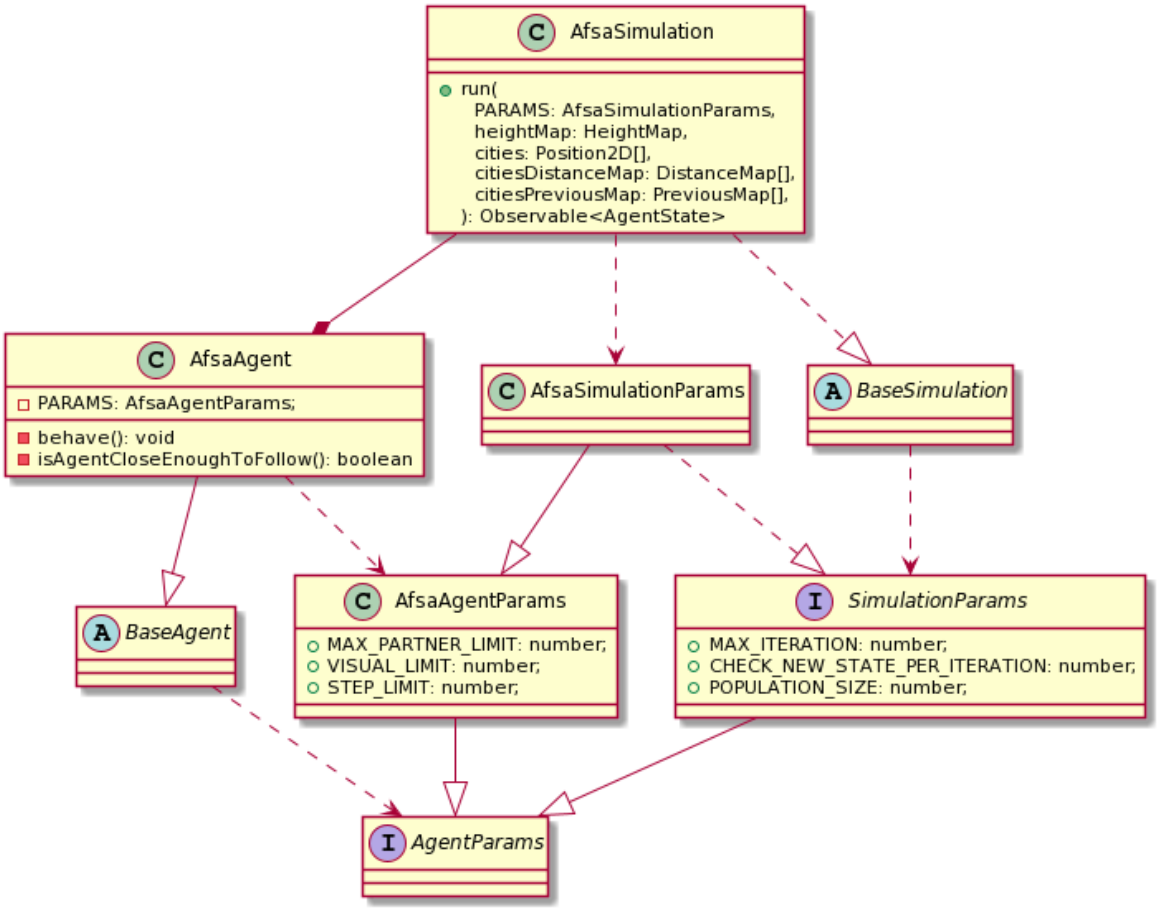
Az `Image Preview` komponens a megjelenítendő képet fogja kezelni, hogy a számos generált kép közül pontosan melyiknek is kell megjelennie.

A `Services` csomag komponensei fogják végrehajtani a konkrét számításokat, melyhez a Library modult fogják felhasználni. Ezeket a `Store` komponensei fogják meghívni, a megfelelő felhasználói interakcióra reagálva.

A `Terrain Generator` a magasságmezők generálásáért lesz felelős, melyet a Library modul felhasználásával valósít meg.

A `Road Generator` az úthálózat generálásáért lesz felelős, melyet a Library modul felhasználásával valósít meg. Az eredményt folyamatosan küldi vissza a `Store` megfelelő komponensébe, ami miatt így lehetőség van a felületen is nyomon követni a folyamatot.

A felület alapvetően két részből áll majd, egy előnézeti szekcióból és egy menüsávból. A menüsáv fogja tartalmazni a már említett komponensek segítségével, a különbö-



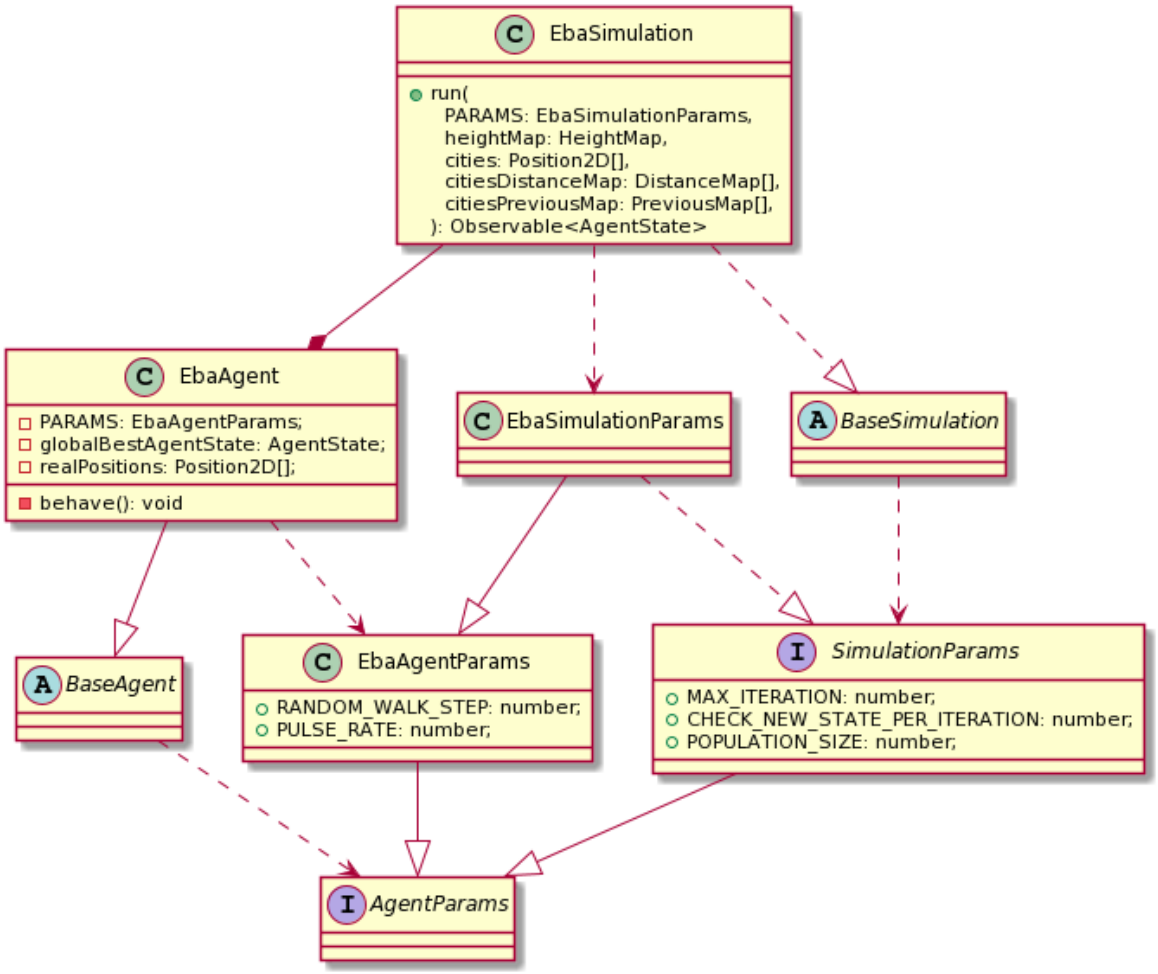
4.10. ábra. A Library modul osztálydiagramja (AFSO változat).

ző algoritmusokkal történő magasságmező és úthálózat generáláshoz szükséges beviteli mezőket és gombokat, valamint az eredmények mentéséhez szükséges linkeket. Az előnézetben a legutolsó eredmény képét lehet majd látni. A felülethez készített wireframe diagramok megtekinthetőek a 4.13. és a 4.14. ábrákon.

4.4.3. Implementáció

Az alkalmazás implementációja előtt először is felsorolnám milyen eszközöket és technológiákat használtam fel annak elkészítése közben:

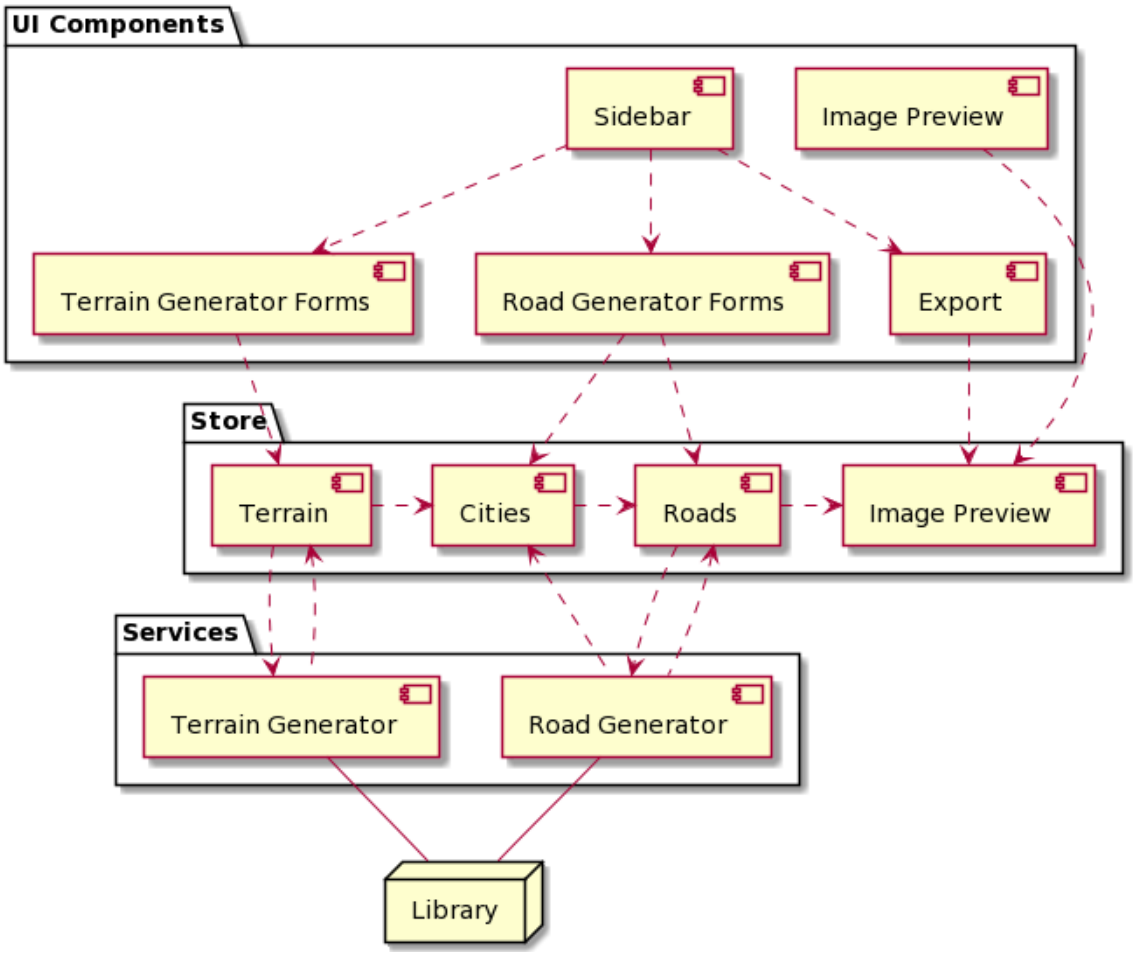
- `TypeScript`: a választott programozási nyelv,
- `Node.js`: egy szoftvercsomag amivel böngészőn kívül is lehet futtatni JavaScript alkalmazásokat,
- `NPM`: a Node.js csomagkezelője,
- `VS Code`: a Microsoft által fejlesztett integrált fejlesztői környezet (IDE). Nagyon elterjedt, főleg a JavaScript programozás esetén, ugyanis kicsi erőforrásigényű, jól



4.11. ábra. A Library modul osztálydiagramja (EBA változat).

testreszabható, számos kiegészítő létezik hozzá, de már önmagában is rendelkezik mindennel, ami a fejlesztéshez szükséges,

- `Git`: verziókezelő rendszer,
- `Angular`: egy frontend keretrendszer, amivel a felhasználói felületeket lehet könnyedén elkészíteni,
- `NgRx`: egy állapotkezelő (state management) függvénykönyvtár az Angularhoz,
- `RxJS`: a reaktív programozást lehetővé tevő függvénykönyvtár, ami nagy hangsúlyt fektet az Observer tervezési minta használatára,
- `colormap`: egy NPM függvénykönyvtár, amivel különböző színskálákat lehet létrehozni térképek színezéséhez,
- `NgxSlider`: egy az Angularhoz készített UI komponens, amivel úgynevezett slider (csúszka) beviteli mezőket lehet hozzáadni az alkalmazáshoz,



4.12. ábra. A UI modul komponensdiagramja.

- `ds-heightmap`: egy NPM függvénykönyvtár, ami a Diamond-square algoritmussal tud magasságmezőket generálni.

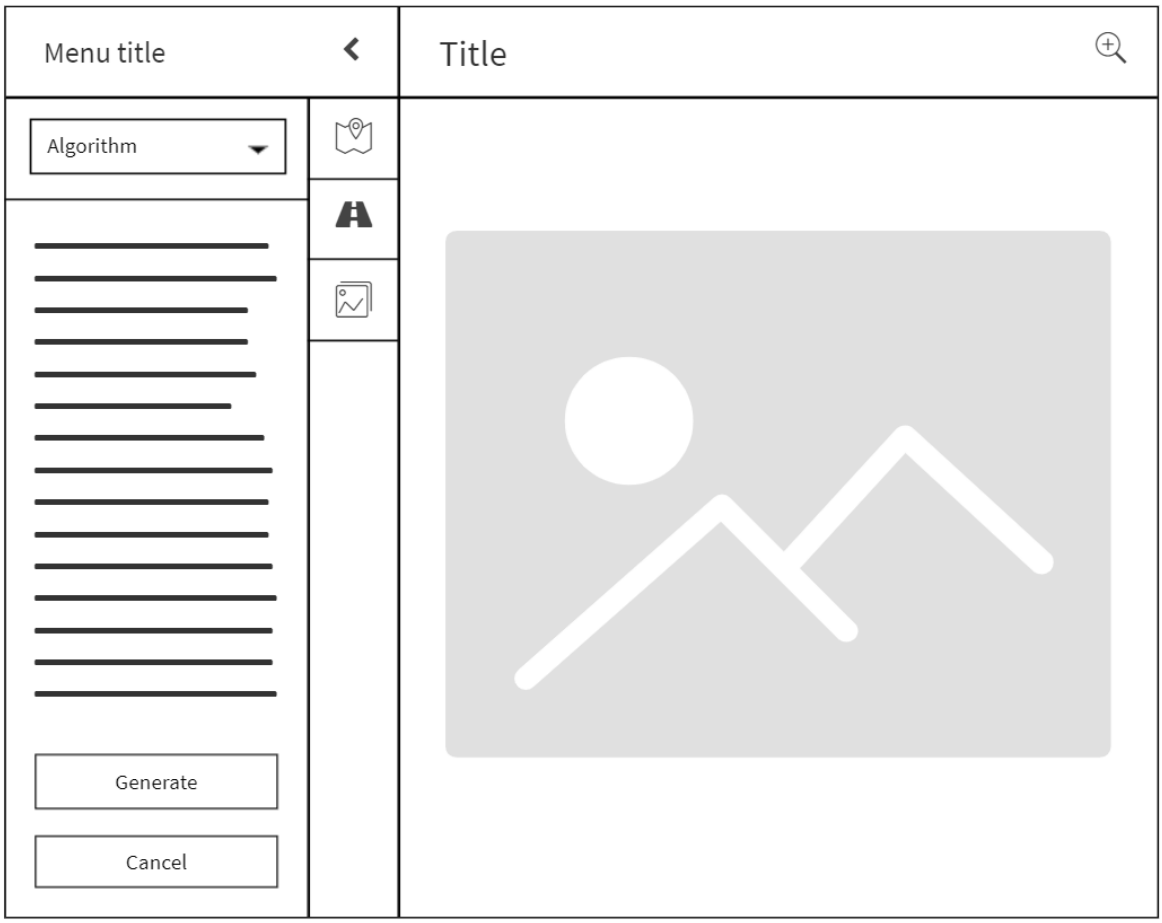
A projekt felépítése nagyjából megegyezik az előző fejezetben említettekkel. A tényleges szerkezet a 4.15. ábrán tekinthető meg. Ami lényeges változás, hogy a UI modulban megjelentek a Worker-ök, melyek esetünkben a Library modul kódját fogják majd futtatni, így a Service csomagban található komponensek már nem közvetlenül fogják használni a Library modult, hanem csak a Worker-ökön keresztül.

Az összes elkészült forrásfájlt nem részletezném azok mennyisége miatt, inkább csak a fontosabbakat. Szót ejtek majd az elkészült UI komponensekről, az állapotkezelésről, valamint bemutatom a felületeket is.

Az elkészült UI komponensek az alábbiak.

- `AppSidebarComponent`: Ez az alkalmazás bal oldalsó menüsávjának komponense. Funkciója a menüsáv ki-be csukhatóságának biztosítása. A felületet az `AppSidebarHeaderComponent` és az `AppSidebarBodyComponent` komponensek segítségével valósítja meg.

- `AppPreviewComponent`: Az alkalmazás előnézeti képének komponense. Funkciója



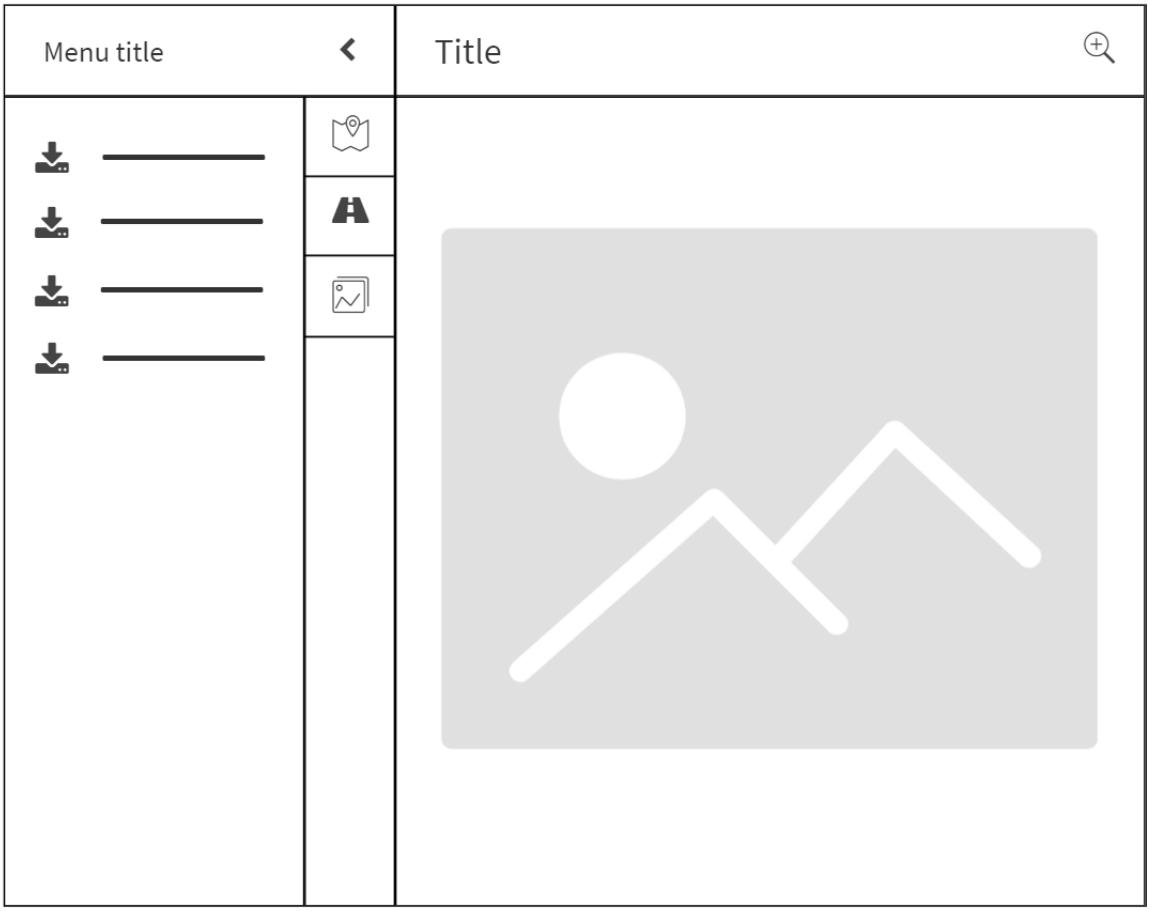
4.13. ábra. A főoldal wireframe diagramja.

a generált képek, valamint a hozzájuk tartozó címek megjelenítése. Lehetőséget ad még a megjelenített kép méretét váltani az eredeti mérete és a képernyő mérete között.

- `AppSidebarHeaderComponent`: Az oldalsó menüsáv fejléc komponense. Funkciója az `AppSidebarHeaderTitleResultComponent`, az `AppSidebarHeaderTitleRoadComponent` és az `AppSidebarHeaderTitleTerrainComponent` komponensek segítségével a megfelelő cím megjelenítése, valamint a menüsáv ki-be csukhatóságának biztosítása.

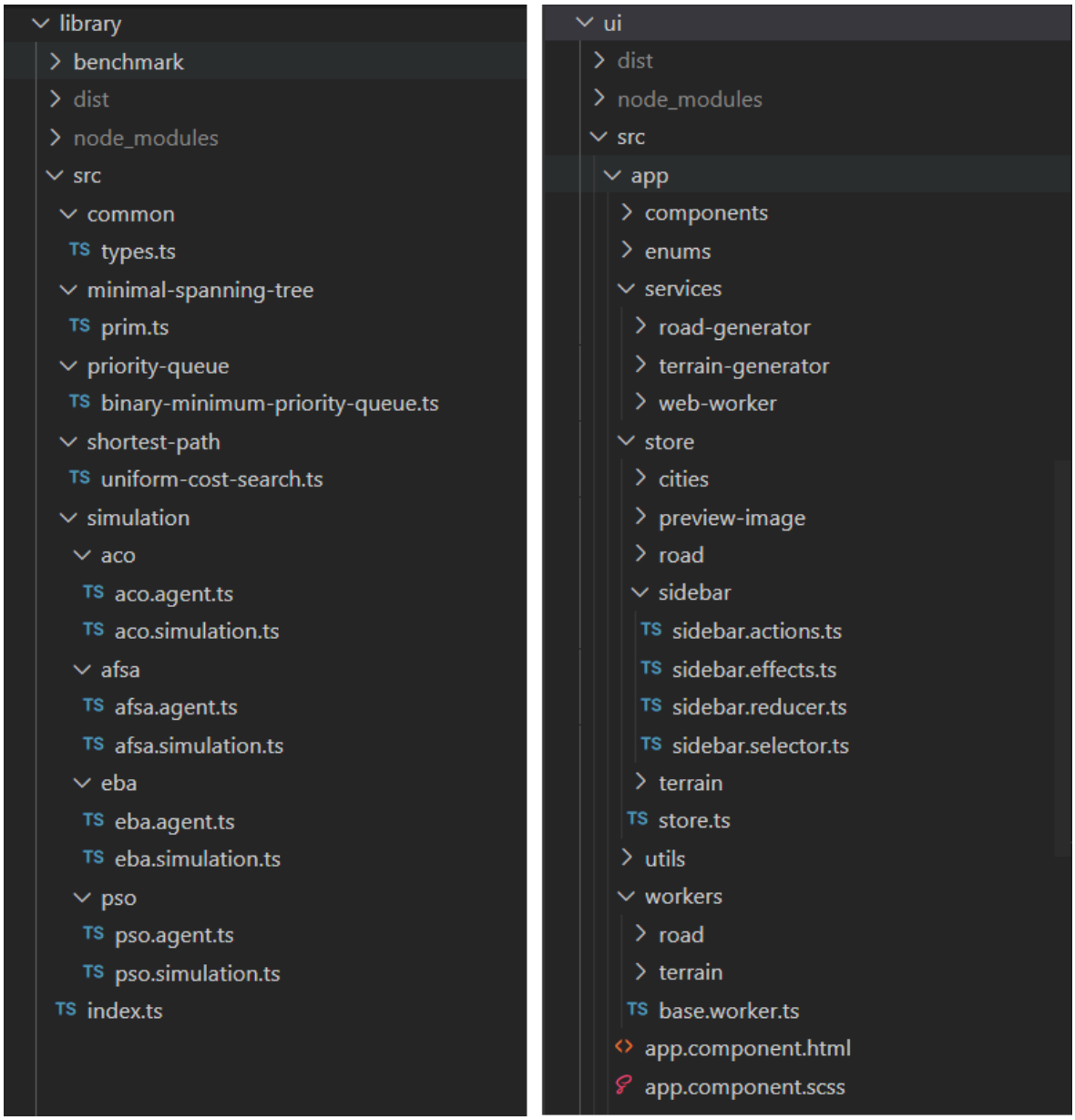
- `AppSidebarBodyComponent`: Az oldalsó menüsáv törzs komponense. Funkciója a menüpontok és a megfelelő tartalom megjelenítése, melyeket az `AppSidebarBodyMenuComponent` és az `AppSidebarBodyContentComponent` komponensek segítségével valósít meg. A tartalmat csak akkor jeleníti meg, ha a menüsáv nyitott állapotban található.

- `AppSidebarHeaderTitleTerrainComponent`: A menüsáv fejlécének implementációja, amely akkor fog megjelenni, ha a magasságmező generálása menüpont az aktív.



4.14. ábra. Az exportálás menüpont wireframe diagramja.

- `AppSidebarHeaderTitleRoadComponent`: A menüsáv fejlécének implementációja, amely akkor fog megjelenni, ha az úthálózat generálása menüpont az aktív.
- `AppSidebarHeaderTitleResultComponent`: A menüsáv fejlécének implementációja, amely akkor fog megjelenni, ha az eredmények exportálása menüpont az aktív.
- `AppSidebarBodyContentComponent`: Az oldalsó menüsáv törzsének tartalmi komponense. Funkciói: a kiválasztásnak megfelelő menüponthoz tartozó tartalom megjelenítése, melyet az `AppSidebarBodyContentTerrainComponent`, az `AppSidebarBodyContentRoadComponent`, és az `AppSidebarBodyContentResultComponent` komponensek segítségével valósít meg.
- `AppSidebarBodyMenuComponent`: Az oldalsó menüsáv menüpontjait tartalmazó komponens. Funkciói: a menüpontok megjelenítése, az aktív menüpont kiemelése, valamint a felhasználói interakciók kezelése.
- `AppSidebarBodyContentTerrainComponent`: A magasságmező generálásának tartalmi komponense. Funkciói: a magasságmezőt generáló algoritmus kiválasztásá-



4.15. ábra. A projekt struktúrája: balra a Library modul, jobbra a UI modul.

nak biztosítása, valamint ez alapján a megfelelő űrlap komponens megjelenítése, mely esetünkben csak az `AppSidebarBodyContentTerrainDiamondSquareComponent` komponens lehet.

- `AppSidebarBodyContentRoadComponent`: Az úthálózat generálásának tartalmi komponense. Funkciói: az úthálózatot generáló algoritmus kiválasztásának biztosítása, az elhelyezett városok számának és az aktuális úthálózat költségének, valamint a megfelelő űrlap komponensének megjelenítése, mely esetünkben az `AppSidebarBodyContentRoadAcoComponent`, az `AppSidebarBodyContentRoadAfsaComponent`, az `AppSidebarBodyContentRoadEbaComponent` vagy az `AppSidebarBodyContentRoadPsoComponent` komponensek lehetnek.

- `AppSidebarBodyContentResultComponent`: Az eredmények exportálásának tartalmi komponense. Funkciói: a generált képekhez a linkek elkészítése és megjelenítése, szükség esetén eltávolítása.

- `AppSidebarBodyContentTerrainDiamondSquareComponent`: A Diamond-square algoritmussal történő magasságmező generálás űrlapja. Funkciói: az algoritmushoz szükséges paraméterek űrlap formában történő bekérése, valamint a generálás indításának és megszakításának kezelése.

- `AppSidebarBodyContentRoadAcoComponent`: Az ACO módszerrel történő úthálózat generálás űrlapja. Funkciói: az algoritmushoz szükséges paraméterek űrlap formában történő bekérése, valamint a generálás indításának és megszakításának kezelése.

- `AppSidebarBodyContentRoadPsoComponent`: A PSO módszerrel történő úthálózat generálás űrlapja. Funkciói: az algoritmushoz szükséges paraméterek űrlap formában történő bekérése, valamint a generálás indításának és megszakításának kezelése.

- `AppSidebarBodyContentRoadAfsaComponent`: Az AFSA módszerrel történő úthálózat generálás űrlapja. Funkciói: az algoritmushoz szükséges paraméterek űrlap formában történő bekérése, valamint a generálás indításának és megszakításának kezelése.

- `AppSidebarBodyContentRoadEbaComponent`: Az EBA módszerrel történő úthálózat generálás űrlapja. Funkciói: az algoritmushoz szükséges paraméterek űrlap formában történő bekérése, valamint a generálás indításának és megszakításának kezelése.

- `AppSelectComponent`: Egy általános komponens, ami az egyedi legördülő listák megvalósításáért felel.

- `AppInputComponent`: Egy általános komponens, ami az egyedi beviteli mezők megvalósításáért felel.

- `AppFormFieldComponent`: Egy általános komponens, ami az űrlapok egy mezőjének megfelelő megjelenítéséért felel.

- `AppDividerComponent`: Egy általános komponens, ami az egyedi vízszintes határoló vonalak megjelenítéséért felel.

- `AppButtonComponent`: Egy általános komponens, ami az egyedi gombok megjelenítéséért felel. Ha szükséges, megjelenít egy töltést jelző spinner-t, amíg a gomb nem kattintható.

- `AppBoxButtonComponent`: Egy általános komponens, ami az egyedi gombok megjelenítéséért felel. Funkcióiban megegyezik az `AppButtonComponent` komponenssel, viszont ennek már más a kinézete.

Az alkalmazás állapotkezelését az NgRx csomag segítségével valósítottam meg, ami gyakorlatilag egy Redux implementáció Angular környezethez. Ennek a megoldásnak az az előnye, hogy központi helyen van tárolva az alkalmazás állapota, a Store-ban, így egyszerűbb szinkronizálni az állapotokat a különböző komponensek között. A Redux architektúra megköveteli az adatok egyirányú mozgását, illetve az Immutable (megváltoztathatatlan) adatszerkezetek használatát. Ennek az lesz az előnye, hogy jól követhető lesz az adatok mozgása és változása, valamint elkerülhetőek lesznek a referencia egyezőségből fakadó hibák, hiszen mindig új objektumokat kell létrehoznunk a módosítás helyett. A UI komponensek feliratkoznak a Store-ra, ahonnan megkapják az aktuális állapotot, valamint az Observer tervezési minta használatának köszönhetően mindig értesülnek majd a változásokról. A Store állapotát Action-ökkel (akciókkal) tudjuk megváltoztatni. Ezek gyakorlatilag események, amire a Store figyel és hatásukra megváltoztatja valamely módon a tárolt állapotot. Az NgRx tartalmaz további funkciókat is, melyek közül én még az Effect-et (effekteket) emelném ki. Ez a funkció arra szolgál, hogy egy akció hatására egy másik akciót tudjunk létrehozni. Egy egyszerű példa erre a `Sidebar` effektjei, melyek alább láthatóak:

```
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { Store } from '@ngrx/store';
import { tap } from 'rxjs/operators';
import { AppState } from '../store';
import { selectMenuRoad, selectMenuTerrain, open, selectMenuResult }
    from './sidebar.actions';

@Injectable()
export class SidebarEffects {
  public sidebarEffect1 = createEffect(() => this.actions.pipe(
    ofType(selectMenuTerrain, selectMenuRoad, selectMenuResult),
    tap(() => this.store.dispatch(open()))
  ), {dispatch: false});

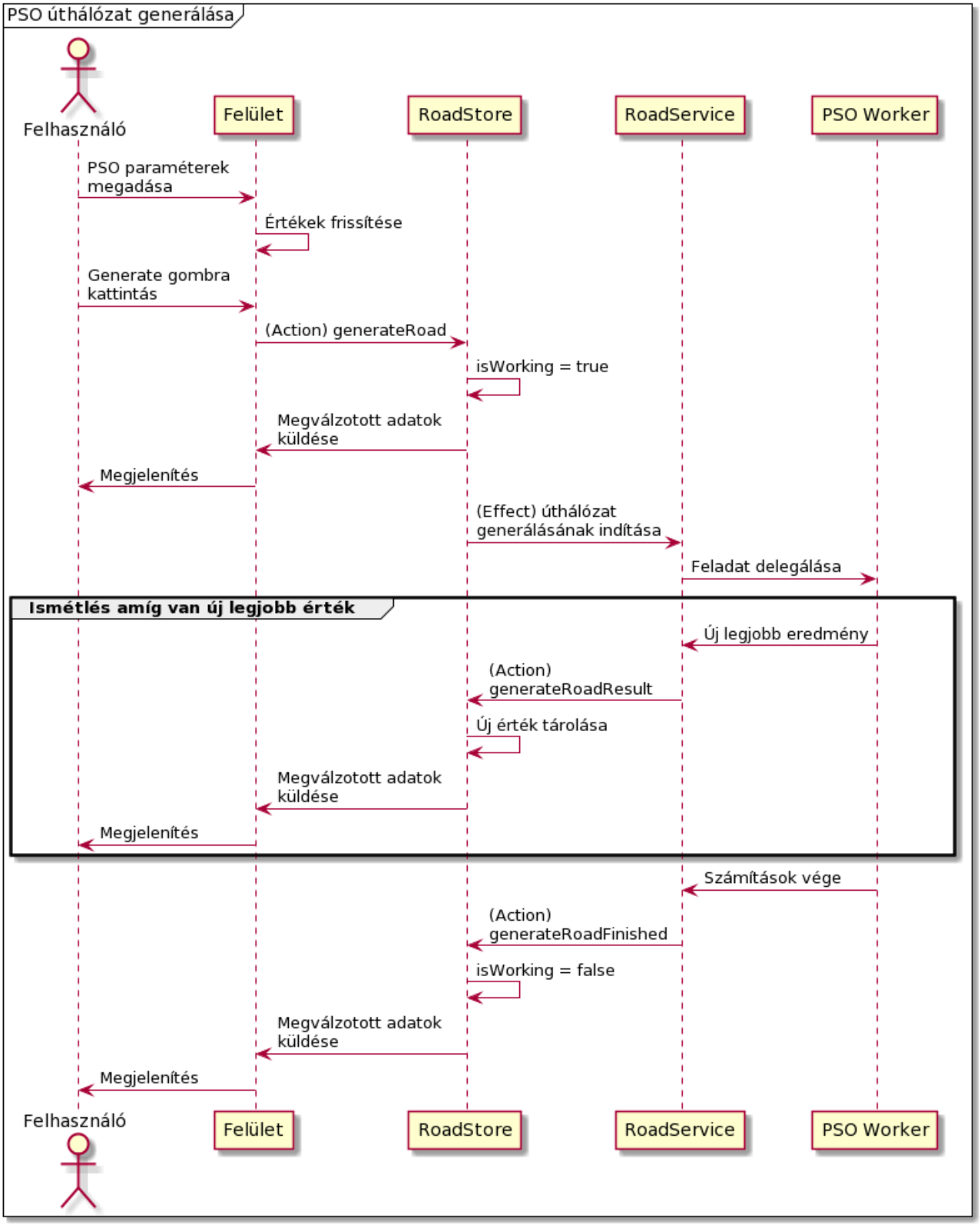
  constructor(
  private actions: Actions,
  private store: Store<AppState>,
  ) {}
}
```

Ennek az effektnek az a szerepe, hogy ha kiválasztjuk az egyik menüpontot (selectMenuTerrain, selectMenuRoad, selectMenuResult), akkor a kiválasztás mellett még ki is nyissa a menüsávot, hogy látható legyen.

Az állapotkezelés működésére látható egy példa a 4.16. szekvenciadiagramon.

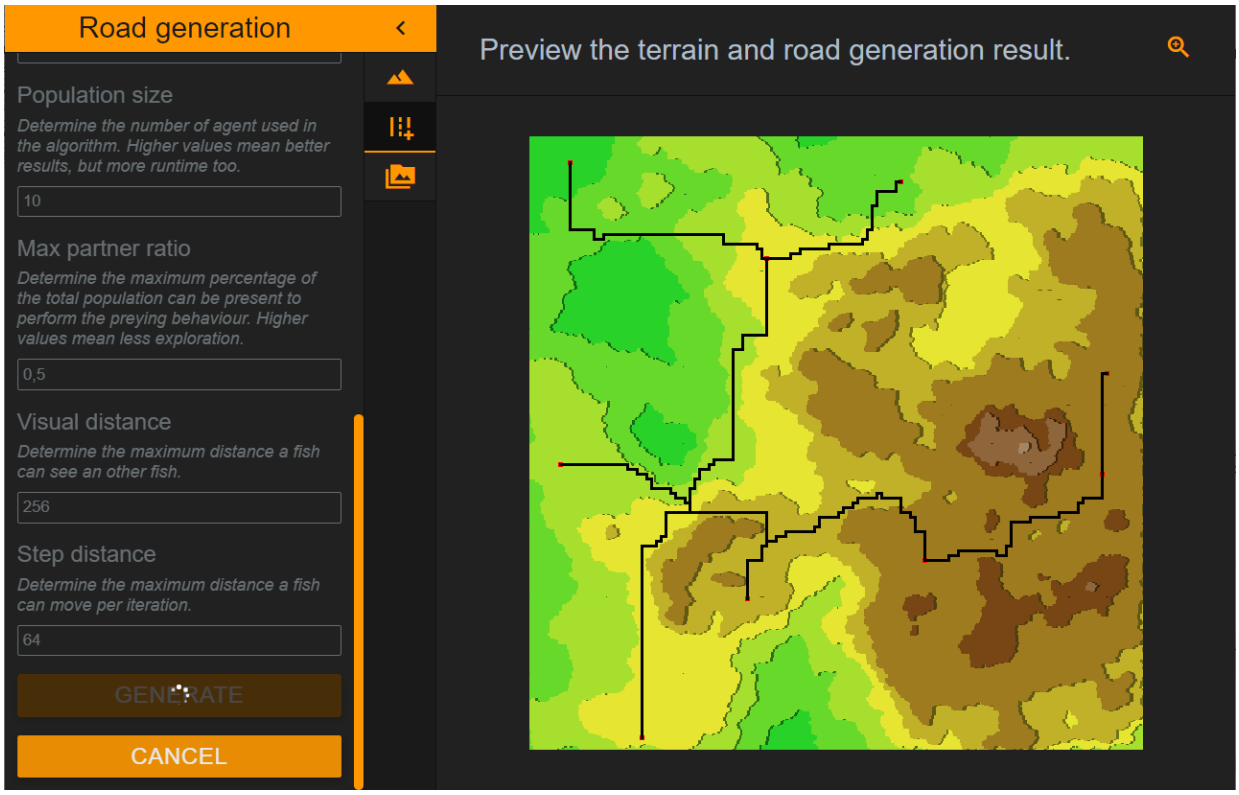
Az elkészült felületek a 4.17. és a 4.18. ábrákon láthatóak.

Az alkalmazás használatához csak az NPM-re és internetkapcsolatra van szükségünk. A projekt UI moduljában lefuttatva az „npm i” parancsot tudjuk feltelepíteni a szükséges függőségeket. Ez a parancs a Library modulhoz is feltelepíti a függőségeket,



4.16. ábra. A PSO úthálózat generálásának szekvencia diagramja.

valamint el is készíti annak a futtatható változatát, hogy tudja azt használni a UI modul. Ezt követően „npm start” paranccsal tudjuk lokális fejlesztői szerveren futtatni az alkalmazást, valamint az „npm run build” paranccsal pedig elkészíthetjük az alkalmazás



4.17. ábra. Az alkalmazás fő felülete.

lefordított változatát, vagyis a statikusan kiszolgálandó fájlokat.

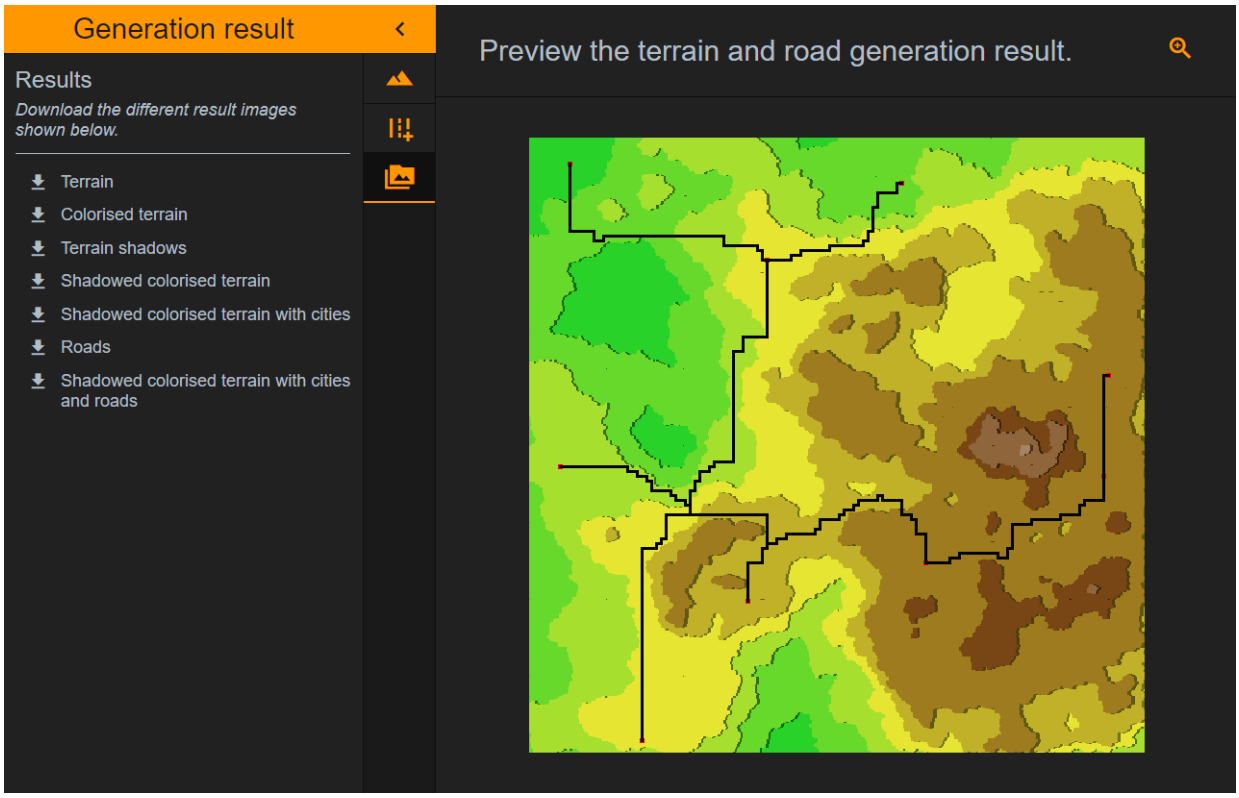
4.5. Optimalizálások

Az alkalmazás készítése során számos szempontból próbáltam optimalizálni az algoritmusokat, adatszerkezeteket. Ezekről lesz itt most részletesebben szó.

4.5.1. JavaScript tömbkezelés

Mivel a JavaScript esetében a tömbök nem egyeznek meg a statikus típusos nyelveknél (például C, Java) megszokott tömbökkel, így érdemesnek tartottam ezeket is megvizsgálni [57]. A különbség alapvetően az, hogy a JavaScript lista szerűen kezeli a tömböket, vagyis dinamikusan változtatható a méretük, illetve nincs megszabva, hogy milyen elemeket tartalmazhatnak. Ez a többszempontból is dinamikusnak tekinthető megvalósítás a teljesítmény rovására tud menni. Éppen ezért több, az alábbiakban felsorolt változatot is kipróbáltam a mátrixok hatékony tárolására:

- 2 dimenziós tömb használata (2D Array),
- 1 dimenziós tömb használata sor-, vagy oszlopfolytonos tárolással (1D Array),
- 2 db 2 dimenziós tömb sor- és oszlopfolytonos tárolással (2 × 2D Array),



4.18. ábra. Az eredmények letöltésére szolgáló felület.

- 2 db 1 dimenziós tömb sor- és oszlopfolytonos tárolással (2 × 1D Array),
- az 1 dimenziós megoldások TypedArray használatával (1D TypedArray, 2 × 1D TypedArray).

Az első az optimalizáció mentes változat lenne, ahol tömböket tárolok tömbökben.

A második változatra azért gondoltam, mert ha kevesebb tömböt kell kezelni (elhagyjuk a tömbök a tömbökben részt), akkor talán gyorsabb lesz az elemek elérése, cserébe viszont manuálisan kell számolni az indexeket.

A harmadik és negyedik változat a mátrix szorzásnál már tapasztalt egyidejű sor- és oszlopfolytonos tárolásból adódó gyorsítást igyekezett kihasználni. Ugyanis, ha tároljuk mindkét formájában a mátrixot, akkor mind az oszlop indexek léptetése, mind a sorindexek léptetése gyors tud lenni, hiszen a processzor rendszerint előre betölti a tömb / mátrix folytonosan következő elemeit, melyet így gyorsabban tudunk elérni.

Az utolsó változatnál a JavaScript TypedArray megoldását igyekeztem kihasználni. Ennek lényege, hogy lehetőség van közvetlenül is használni a memóriát, mint ahogy például a C nyelv esetében. Ezzel így tényleges típusos tömböt lehet létrehozni, ami csak az adott típusnak megfelelő elemeket tud majd tárolni és csak a meghatározott méretig. Várhatóan ennek kellene a leggyorsabbnak lennie. A mért eredményeket a 4.3. táblázatban lehet látni.

Mint látható, a leggyorsabb meglepő módon a legelső, az optimalizáció mentes változat volt, de csak minimális a különbség az első 3 helyezett között. Ennek okát kezdet-

Módszer megnevezése	Átlagos ops/s	Százalékos eredmény
2D Array	779.47	100.00%
1D Array	775.87	99.54%
2 × 2D Array	610.22	78.29%
2 × 1D Array	553.34	70.99%
1D TypedArray	771.98	99.04%
2 × 1D TypedArray	584.77	75.02%

4.3. táblázat. Különböző JS tömbkezelési módszerek összehasonlítása (ops/s az operáció/másodperc rövidítése).

ben én sem értettem, de utánaolvasva kiderült, hogy a JavaScriptet futtató V8 motor magától is számos optimalizációt elvégez, mielőtt még futtatná (értelmezné) a kódot. Így amikor csak teheti, mint az esetünkben is, a dinamikus tömbök helyett valójában a TypedArray-eknek megfelelő típusos tömböket használja, emiatt sokkal gyorsabb lesz a kód. Fontos, hogy ezt nem tudja minden esetben végrehajtani, például ha nem kiszámítható módon használjuk a tömböket, mint amikor véletlenszerűen generált pozícióit próbáljuk olvasni a tömbnek.

Mivel érdemi különbséget nem sikerült mérni a különböző változatok között és bízva abban, hogy a V8 megfelelően fog tudni optimalizálni, így az alkalmazáshoz inkább maradtam a legegyszerűbb változatnál, az elsőnél.

4.5.2. Web worker

Mivel az alkalmazás elsődlegesen a webes világhoz készülne, ez azt jelentené, hogy a kód böngészőben fog futni. Alapvetően a JavaScript alkalmazások függetlenül attól, hogy hol futtatjuk őket, csak 1 szálasok, vagyis nincs bennük lehetőség párhuzamosításra. Ez a böngészőben való futtatás esetén különösen problémás lehet, ugyanis nem szeretnénk blokkolni a fő szálat a számításokkal, mert az a megfagyott oldal érzetét keltheti. Erre kínálnak megoldást a web worker-ök. Ezek használata a mi szempontunkból nem számít tényleges optimalizációnak, viszont a felhasználó szempontjából mindenképp fontos, az előbb említett ok miatt.

A web worker-ök segítségével gyakorlatilag háttérben futó szálat tudunk létrehozni. Ennek viszont számos megkötése van. Az egyik ilyen megkötés, hogy nem férhetünk hozzá semmilyen módon a DOM-hoz, illetve a globális kontextushoz. Ez alól egyedüli kivétel a Canvas, ugyanis arra tudunk rajzolni web worker-ön belülről is [58].

A fő szál és a web worker szála között a kommunikáció üzenetek formájában folyik. Ezeket az üzeneteket a *postMessage* és az *onmessage* metódusokkal lehet elküldeni és

fogadni. Megjegyzendő, hogy az üzenetek tartalmáról másolat készül, és úgy kapja meg a másik fél, így nagy mennyiségű adat továbbítása időigényes lehet.

További fontos tulajdonsága, hogy jelentős létrehozási költsége van a web worker-öknek, így kis, illetve ad-hoc számításokra nem feltétlenül érdemes létrehozni őket.

Az alkalmazásban a web worker-ök könnyebb létrehozására az alábbi metódust készítettem el:

```
import { Observable } from "rxjs";

// WebWorker types, for typescript compilation
declare var WorkerGlobalScope: any;
declare function postMessage(...args: any)[]: any;

export function createWorker<T extends unknown[], K>(execute:(...args: T)
    => Observable<K>) {
  if (typeof WorkerGlobalScope !== 'undefined' && self instanceof
      WorkerGlobalScope) {
    self.addEventListener('message', function(e: MessageEvent<T>) {
      execute(...e.data).subscribe({
        next: (data) => postMessage({event: 'NEXT', data: data}),
        error: (data) => postMessage({event: 'ERROR', data: data}),
        complete: () => postMessage({event: 'COMPLETE'}),
      });
    });
  }

  return {execute};
}
```

Ez gyakorlatilag a paraméterként megadott metódust fogja futtatni. Ehhez először megvárja, hogy kapjon egy üzenetet a fő száltól, ami a metódushoz szükséges bemenő paramétereket fogja tartalmazni. Ezt követően lefuttatja a metódust, majd feliratkozik arra és az összes visszakapott eredményt, vagy hibát továbbítja a főszál felé.

A web worker-ök futtatására az alábbi kódrészletben láthatunk példát:

```
public generateTerrainDiamondSquare(...args: TerrainAlgorithmDiamondSquareArgs):
    Observable<TerrainAlgorithmResult> {
  if(this.webWorkerService.hasWebWorkerSupport) {
    //Create worker if none present.
    if(this.diamondSquareWorker == null) {
      this.diamondSquareWorker = new Worker(new URL(
        'src/app/workers/terrain/diamond-square.worker',
        import.meta.url
      ))
    }

    return this.webWorkerService.executeWorkerAndKeep(this.
        diamondSquareWorker, ...args);
  } else {
    return DiamondSquareWorker.execute(...args);
  }
}
```

Itt látható, hogy először megvizsgálom támogatottak-e a web worker funkciók a böngészőben, és ha igen, akkor megvizsgálom hoztam-e már létre worker-t az adott feladatra, ha nem, akkor létrehozok egy újat, melyhez a futtatandó script elérési útvonalát kell megadni. Ezt követően átküldöm a worker-nek a szükséges paramétereket, ami pedig

majd egyszer visszaküldi az eredményt. Tartalék megoldásként, ha nem lennének támogatottak a web worker funkciók, akkor közvetlenül a fő szálon futtattom le ugyan azt a kódot, ami a workerben futna, így biztosítva a kompatibilitást.

A web worker-ök és az Angular kapcsán egy komoly problémába futottam bele. Mivel az Angular-nál TypeScript-et használok, így a web worker-öknél is azt szerettem volna. Erre lehetőséget is ad az Angular, de mint kiderült, nem teljeskörűt. A probléma abból fakadt, hogy a web worker-öknek más kontextusban kell futnia, így a globális változók típusainak is másnak kell lennie, és ha a különböző globális típusdeklarációkat megpróbáljuk vegyíteni, azt nem tudja kezelni a fordító. Erre számomra azért lett volna szükség, mert nem szerettem volna duplikálni a kódot a worker és az alkalmazás között, vagyis szerettem volna beimportálni a kódot a worker fájljából az alkalmazásba. Ekkor kaptam is a hibaüzeneteket, hogy az egyik helyen definiálva vannak a DOM kezeléséhez szükséges függvények, a másik helyen pedig nincsenek és ezt nem tudja értelmezni a fordító. Végül úgy kerültem meg a problémát, hogy a worker fájlban nem tettem bele a web worker kontextus összes típusdefinícióját, csak azokat, amiket ténylegesen is használtam, valamint úgy írtam meg a kódot, hogy el tudja magáról dönteni, hogy egy worker szálon fut-e vagy a fő szálon, és ez alapján másképp viselkedjen.

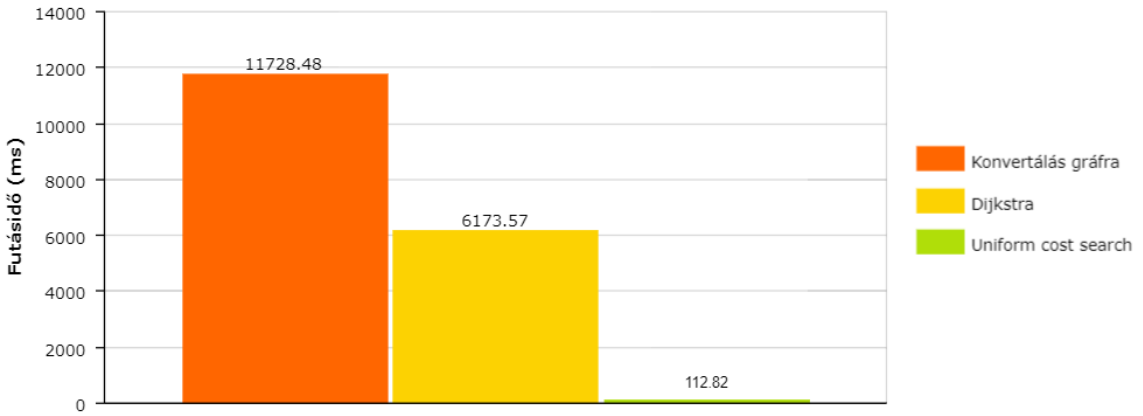
4.5.3. Legrövidebb útszámítás

A különböző Steiner-fa megoldó módszerek közös pontja, hogy gyakran kell legrövidebb utat / utakat számolniuk, így ezt a lépést mindenképp érdemes a lehető legjobban optimalizálni. Ezt alapvetően két részre lehetne bontani: az algoritmus és az adatszerkezet optimalizációjára. Az adatszerkezettel a következő részben külön is fogok foglalkozni, így most inkább az algoritmusra fogok csak koncentrálni [16, 17, 59].

A legrövidebb útvonal kiszámításához kezdetben a Dijkstra algoritmusát használtam fel. Mivel ez egy jól ismert algoritmus, így nem részletezném a lépéseit, inkább csak egyet emelnék ki közülük, mégpedig azt, amikor egy elsőbbségi sorba (priority queue) belerakjuk a gráf összes pontját. Mivel az elsőbbségi sorban prioritás szerint kell tárolni az elemeket, ez egyrészt sok időt vehet igénybe, másrészt sok memóriába is kerülhet. Éppen ezért, ezt ha lehet, el kellene kerülni.

Erre nyújt megoldást a Uniform Cost Search (UCS) algoritmus, ami a Dijkstra algoritmusának egy kis mértékben módosított változata. A különbség a kettő között, hogy az UCS-nél csak a kezdő pontot rakjuk bele az elsőbbségi sorba, és utána pedig folyamatosan rakjuk bele az újonnan felfedezett pontokat. Ha egy csúcs esetleg már benne lenne a sorban, akkor pedig módosítjuk annak a prioritását az új értékre. Ezzel azt érjük el, hogy egyszerre viszonylag csak kevés csúcs van az elsőbbségi sorban, ami gyorsítani tudja a műveleteket. Ezt egyrészt a kevesebb memóriahasználat miatt teheti, másrészt pedig az elsőbbségi sor adatszerkezetének műveleteiből kifolyólag, ugyanis azok ideje általában a sorban tárolt elemek számától nagyban függ.

Még egy érdekesség, amit megvizsgáltam, hogy az UCS algoritmusának minimális



4.19. ábra. A Dijkstra és a Uniform Cost Search futásidejének összehasonlítása.

átalakításával könnyedén elérhető, hogy ne kelljen módosítani a már beszúrt elemek prioritását, hanem elég legyen csak még egyszer beszúrni őket a sorba az alacsonyabb prioritással. Ezzel néhány esetben javítani lehet a futásidőn, különösen akkor, ha a prioritás módosításnak nagy a költsége az adatszerkezet tekintetében. Valamint azért lehet még ez előnyös, mert az adatszerkezettől függően, nehéz lehet optimálisan implementálni minden szükséges metódust, de ha csak egy párat kell, akkor ezzel az egyszerűsítéssel könnyebb az optimális megvalósítást elérni. A konkrét módosítás csak annyiból áll, hogy amikor kivesszük a legkisebb elemet a sorból, meg kell vizsgálni, hogy egy korábbi lépésben feldolgoztuk-e már azt, ha igen, akkor ezt eldobjuk és vesszük a következő elemet.

Továbbá még egy apró módosítást végeztem az algoritmuson, mégpedig, hogy nem a gráfon számítom a legrövidebb utat, hanem közvetlenül a mátrix formában tárolt térképen, ezzel megspórolva egy oda-vissza alakítást a mátrix és a gráf között. A két algoritmus, illetve a gráfra történő konvertálás idejének összehasonlítása látható a 4.19. ábrán.

4.5.4. Elsőbbségi sor

A legrövidebb utak kereséséhez, mint említettem, szükségünk lesz egy adatstruktúrára, az elsőbbségi sorra (priority queue). Az eredeti algoritmusnál, az adatszerkezetnek rendelkeznie kell a beszúrás (insert), a prioritás módosítása (decrease-key) és a minimum kivétele (extract-min) műveletekkel. Ahogy már ezt is említettem, a módosított változatban ezek közül a prioritás módosítása elhagyható [60, 61].

Habár az jól dokumentált, hogy milyen számítási komplexitással rendelkeznek az egyes adatszerkezetek, de ezek sok esetben csak nagy elemszámoknál jönnek elő. Például egy $\mathcal{O}(1)$ beszúrás igényű adatszerkezet lehet lassú is kevés elem szám esetén, ugyanis a konstans 1 időigény csak az jelenti, hogy nem függ a sorban lévő elemek számától, de ettől még lehet, hogy van olyan plusz konstans idejű lépése, ami egy

nem $\mathcal{O}(1)$ változatban nincs. Általában pont azért vannak ezek a konstans idejű plusz lépések, hogy ne függjön az adott művelet az elemek számától. Egy ilyen lépés lehetne például az elem hash-ének kiszámítása. Az elterjedt elsőbbségi sor adatszerkezetek időkomplexitása a 4.4. táblázatban látható.

	find-min	extract-min	insert	decrease-key	meld
Binary	$\Theta(1)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\Theta(\log n)$
Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$
Fibonacci	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Pairing	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$o(\log n)$	$\Theta(1)$
Brodal	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Strict Fibonacci	$\Theta(1)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(1)$	?

4.4. táblázat. Az elterjedt elsőbbségi sor adatszerkezetek időkomplexitása [60, 61]. (A „find-min” a minimum meghatározását, a „extract-min” a minimum kivételét, az „insert” a beszúrást, a „decrease-key” az elem prioritásának csökkentését és a „meld” az összefésülést jelenti.)

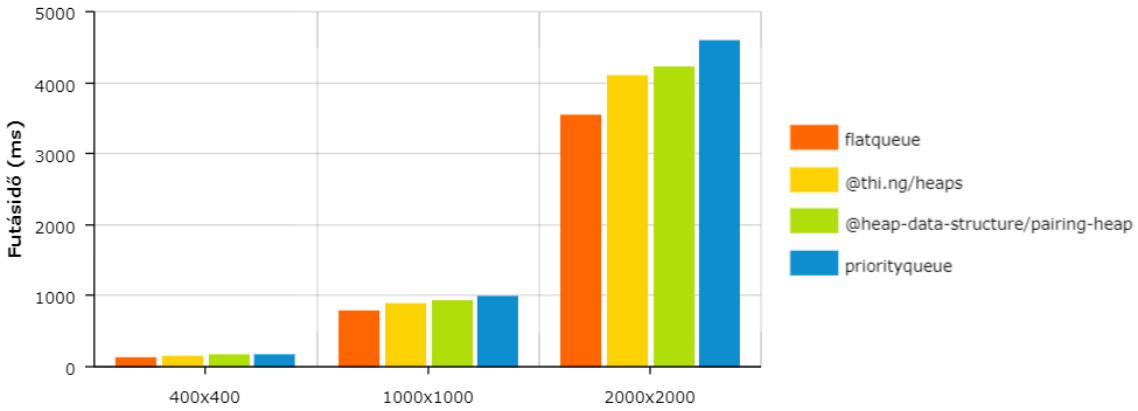
Az említett okok miatt szükségesnek véltem a gyakorlatban is megvizsgálni és összehasonlítani több különböző elsőbbségi sor implementációt és azok futásidejét, különös tekintettel az említett két fő műveletre. A vizsgálatokat JavaScript nyelv esetén végeztem el, melyek alapján végül eldöntöttem, hogy melyik adatszerkezetet használom fel majd a programban.

Az méréseket mind ugyanazon a számítógépen végeztem, ami egy Intel i5-7500 processzorral és 8 GB memóriával felszerelt Windows 10-et futtató asztali számítógép volt. A JavaScript sajátosságából adódóan minden mérés csak 1 processzor magot használhatott. A mérések során elsősorban a sebességet mértem, de mivel a Windows-on nehézkes lett volna megoldani a tényleges CPU-idő mérését, így inkább a valós futásidőt vettem alapul. Az ebből adódó pontatlanságok miatt a mérések során a számítógépen semmilyen más program nem futott (a szokásos háttérfolyamatokat leszámítva) és mindegyik mérést többször is elvégeztem, majd az eredményeket átlagolva vettem figyelembe.

Mivel rengeteg implementációt találtam az NPM adatbázisában az említett adatszerkezethez, így az összehasonlításokat két részre bontottam. Az első részben csak kisebb térképeken és kevesebbszer futtatam le a számításokat, majd azután a legjobb 4-et részletesebben vizsgáltam, nagyobb térképeken és többször ismételve a számításokat.

Maguk a számítások nagyon egyszerűek voltak: egy rögzített térkép egyik sarkából kellett az összes többi pontjához legrövidebb utat számolni, ahol azt mértem, ezt mennyi idő alatt sikerül elvégeznie a programnak.

Az első körben 400 × 400-as, 1000 × 1000-es és 2000 × 2000-es térképeket használtam, melyeken 10-szer futtattam le a számításokat. Az eredmények közül a 4 legjobbhoz tartozóakat a 4.20. ábrán lehet megtekinteni.



4.20. ábra. Az összehasonlítás első körének 4 legjobb eredménye.

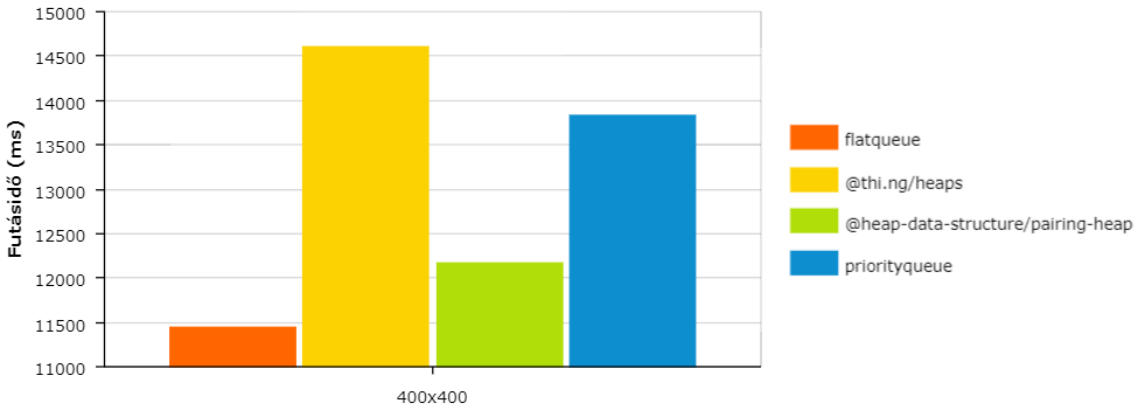
A 4 legjobb implementáció nevesítve a következő:

- 1. `flatqueue`: egy bináris fa implementáció, (https://www.npmjs.com/package/flatqueue),
- 2. `@thi.ng/heaps`: egy párosító fa implementáció, (https://www.npmjs.com/package/@thi.ng/heaps),
- 3. `@heap-data-structure/pairing-heap`: egy párosító fa implementáció, (https://www.npmjs.com/package/@heap-data-structure/pairing-heap),
- 4. `priorityqueue`: egy párosító fa implementáció, (https://www.npmjs.com/package/priorityqueue).

Ennek a 4-nek a kiválasztása elég könnyűnek bizonyult, ugyanis az első 3 esetében mind a 3 térképméret esetén egyértelmű volt a sorrend. A 4. helyezettnél már volt több lehetőség is, de végül a legnagyobb térképméret ideje alapján sikerült döntenem.

Érdekes, hogy habár a Dijkstra algoritmushoz az elméleti legjobb futásidőt a Fibonacci fa implementációja adná, addig esetünkben egy ilyen változat sem került bele a legjobb 4-be. Ennek oka lehet, például a Fibonacci fa bonyolultsága, valamint, hogy az UCS esetén kisebb elemszám mellett ilyenkor túl sok lesz a plusz konstans időigény a műveleteknél.

A második kör során a térkép méretet már 3000 × 3000-esre állítottam és 10 helyett 30 futtatásnak vettem az átlagát. Az így mért eredmények a 4.21. ábrán láthatóak.



4.21. ábra. Az összehasonlítás második körének eredményei.

Ahogy az ábráról leolvasható, az első helyezett változatlan maradt, a második helyezettből utolsó lett, míg a maradék kettő ezeknek megfelelően 1-1 helyezést javított. A mérések alapján végül úgy döntöttem, hogy a *flatqueue* implementációt fogom használni, hiszen egyértelműen az bizonyult a legjobbnak minden esetben.

5. fejezet

Paraméterek hangolása

A most következő fejezetben az implementált módszereket és azok paramétereit fogom vizsgálni, különböző bemenetek és beállítások esetében. Ezek alapján meghatározom az optimálisnak vélt paraméterértékeket, valamint az optimális értékeket felhasználva összehasonlítom a 4 módszert, majd kiválasztom a legjobban teljesítőt.

5.1. A vizsgálatok menete

A paraméterek vizsgálata során különböző tesztfuttatásokat fogok végrehajtani az adott módszert használva. Egy tesztfuttatás során minden más érték rögzített lesz, tehát csak a vizsgált paraméter fog változni, így mérhetővé válik annak hatása az eredményre. A méréseket a 4.5. fejezetben említett számítógéppel és környezettel végzem el. A tesztfuttatások során elsősorban az eredmény összköltségét vizsgáltam rögzített idő alatt, de mivel a Windows-on nehézkes lett volna megoldani a tényleges CPU-idő mérését, így ehhez inkább a valós futásidőt vettem alapul, melyet 5 percben rögzítettem. Megjegyzendő, hogy rögzített iterációszámmal is lehetett volna vizsgálni a paramétereket, viszont a módszerek összehasonlításánál már nem lett volna praktikus, lévén, hogy a módszerek 1-1 iterációja különböző időt vesz igénybe. A pontatlanságok minimalizálása végett a mérések alatt a számítógépen semmilyen más program nem futott (a szokásos háttérfolyamatokat leszámítva) és mindegyik mérést többször is (10-szer) elvégeztem, majd az eredményeket átlagolva vettem figyelembe.

Az elért eredményeket rendszerint egy-egy ábrán mutatom be. A paraméterek vizsgálatánál található ábrák mind 2 sorból állnak. Ennek oka, hogy két vizsgálatot végeztem el mindegyik paraméter esetében. Az első vizsgálatnál egy nagyobb tartományt vizsgáltam, de kevésbé részletesen, majd a következőnél, az előzőnél legjobbnak bizonyult érték egy szűkebb tartományát, de azt már részletesebben. Természetesen lehetett volna több ilyen iterációt is elvégezni, de az aktuális vizsgálatokhoz ez a közelítés megfelelőnek bizonyult. Ezeknek az ábráknak mindegyik sorában 3 diagram található. Az első diagramon a konkrét végeredmények láthatóak, ahol a kisebb érték a jobb, a másodikon a kezdeti értékhez képest javulás látható, ahol a nagyobb érték a jobb, míg

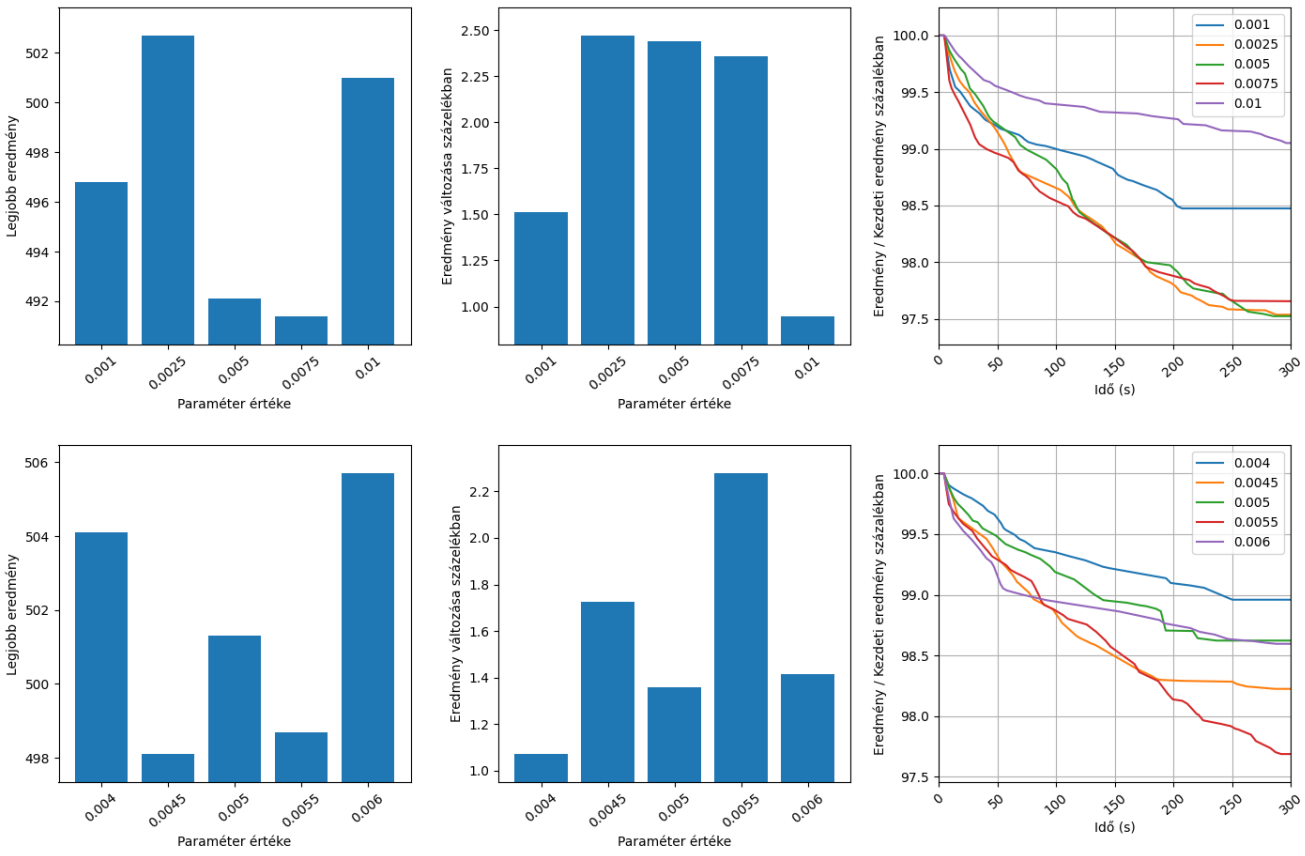
a harmadikon az eredmény alakulása látható az idő függvényében, ahol a meredekebben csökkenő görbe a jobb. Azért készítettem 3 diagramot is, mert önmagukban nem tökéletesek. Az első diagram értékeit nagyban befolyásolhatják a generált kezdeti megoldások. A második diagram ezen már javít valamelyest, de ott meg olyan probléma lehet, hogy nem látható mennyire gyorsan közelíti az adott végeredményt az aktuális paraméterezés. Ezt a harmadik diagramon már könnyedén le lehet olvasni, de ott meg az előző adatokat nem lehet feltétlenül könnyedén kivenni.

5.2. ACO paramétereinek hangolása

Ebben az alfejezetben a hangyakolónia optimalizációs módszer (ACO) paramétereit fogom vizsgálni.

5.2.1. A feromonok párolgási együtthatója

Ez a paraméter a módszer ρ értékének feleltethető meg. Ez kontrollálja, hogy milyen gyorsan párologjanak a feromonok, vagyis, hogy milyen gyorsan felejtsék el az előzőekben megszerzett információt a hangyák.



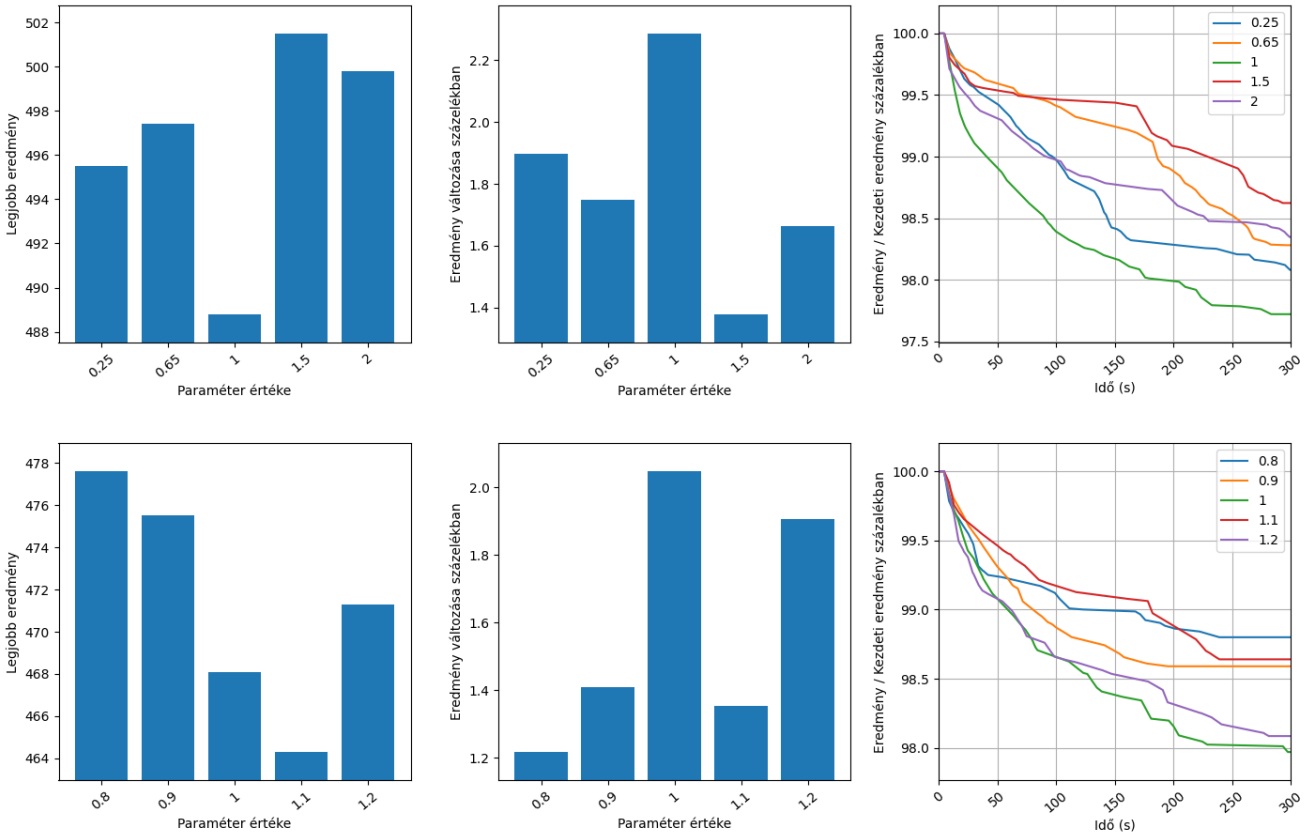
5.1. ábra. Az ACO vizsgálata a feromonok párolgási együtthatójának tekintetében.

Az első körben a 0.0025, 0.005 és a 0.0075 értékek a százalékos változás és a görbék meredeksége tekintetében elég hasonló eredményt értek el, viszont a tényleges végeredmények tekintetében a 0.0025 már nem teljesített annyira jól. Végül úgy döntöttem, hogy a 0.005 értéket választom a második körhöz, mivel százalékos változásban ez volt a jobb.

A második kör során mindhárom ábra alapján egyértelműen a 0.0055 érték lett a legjobb, így ezt neveztem ki az optimális paraméterérték becslésének.

5.2.2. A feromonok vonzósági együtthatója

Ez a paraméter a módszer α értékének feleltethető meg. Ez kontrollálja, hogy mekkora hatást gyakoroljon a feromon a hangyák választásaira.



5.2. ábra. Az ACO vizsgálata a feromonok vonzósági együtthatójának tekintetében.

Az első körben egyértelműen látható, hogy az 1 érték teljesített a legjobban mind a 3 ábra szerint, így azt választottam a második körhöz.

A második kör során csak a végeredmény tekintetében nem lett a legjobb az 1 érték, de a százalékos változás és a görbe meredekségénél akkora a különbség, hogy ezt bőven ellensúlyozza, így végül az 1 értéket választottam az optimális paraméterérték

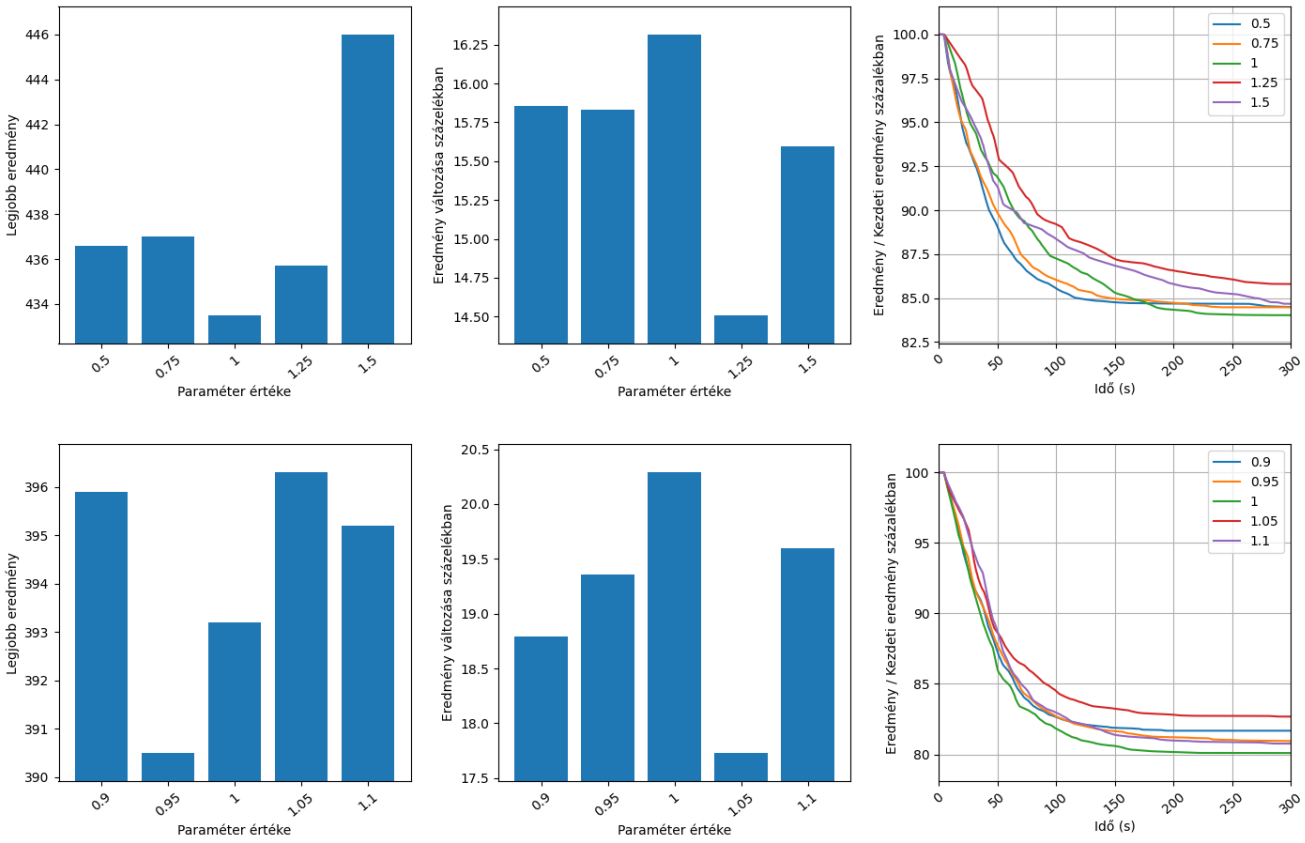
becslésének.

5.3. PSO paramétereinek hangolása

Ebben az alfejezetben a részecskeraj optimalizációs módszer (PSO) paramétereit fogom vizsgálni.

5.3.1. A tanulási együttható

Ez a paraméter a módszer L értékének feleltethető meg. Ez kontrollálja, hogy mekkora mértékben tanuljanak a részecskék.



5.3. ábra. A PSO vizsgálata a tanulási együttható tekintetében.

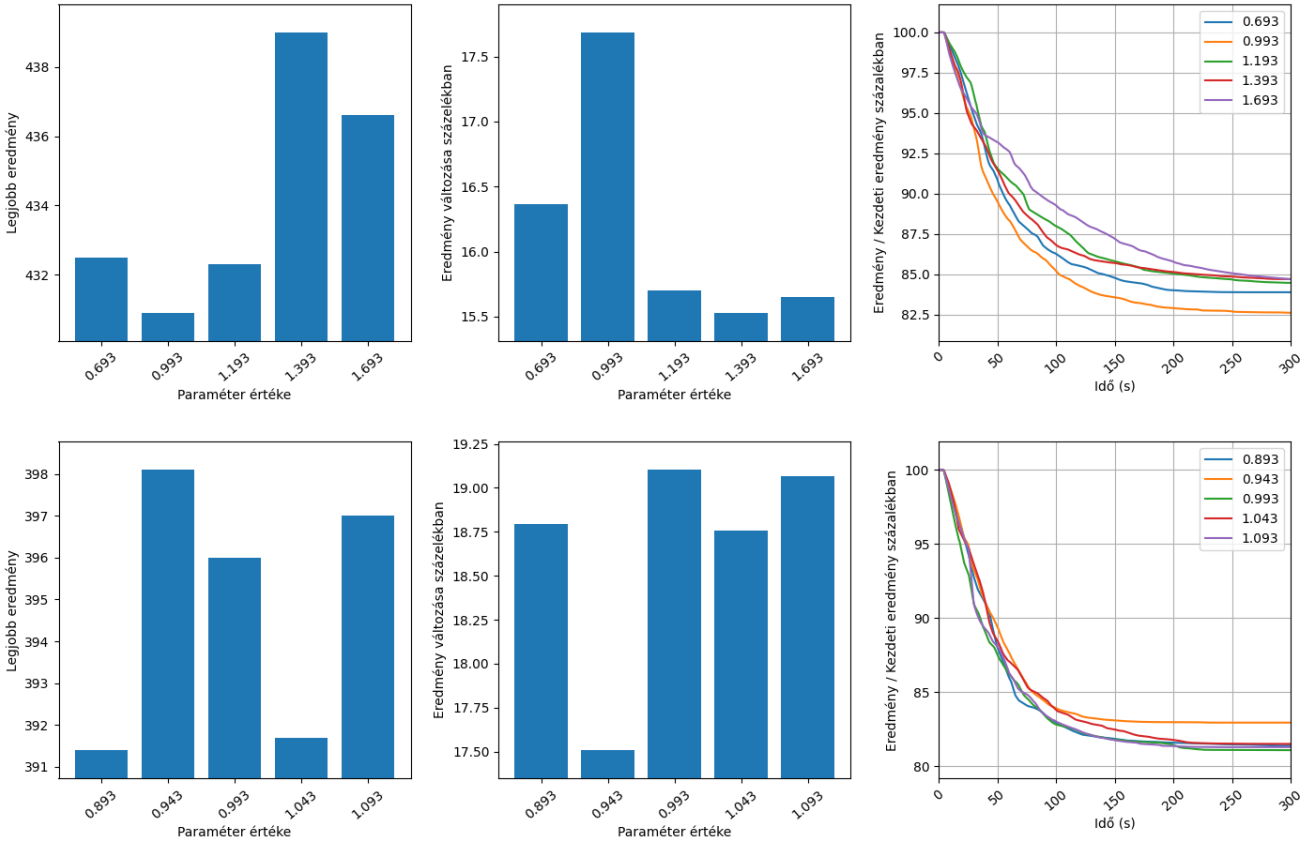
Az első kör során az 1 érték bizonyult a legjobbnak. A görbe alakulása alapján szóba jöhetne még a 0.5 és a 0.75 érték is, habár ezek kezdetben meredekebbek, de sokkal hamarabb ellaposodnak, így végül hosszabb távot nézve rosszabb eredményt kapnánk velük.

A második kör során az első diagramot leszámítva megint az 1 érték teljesít a legjobban, viszont arányaiban nem nagy a különbség ott sem, így az optimális paraméternek

végül ezt választottam.

5.3.2. A csoport legjobb pozíciójának együtthatója

Ez a paraméter a módszer Φ_g értékének feleltethető meg. Ez kontrollálja, hogy mekkora mértékben vegyék figyelemebe a csoport legjobb pozícióját a részecskék.



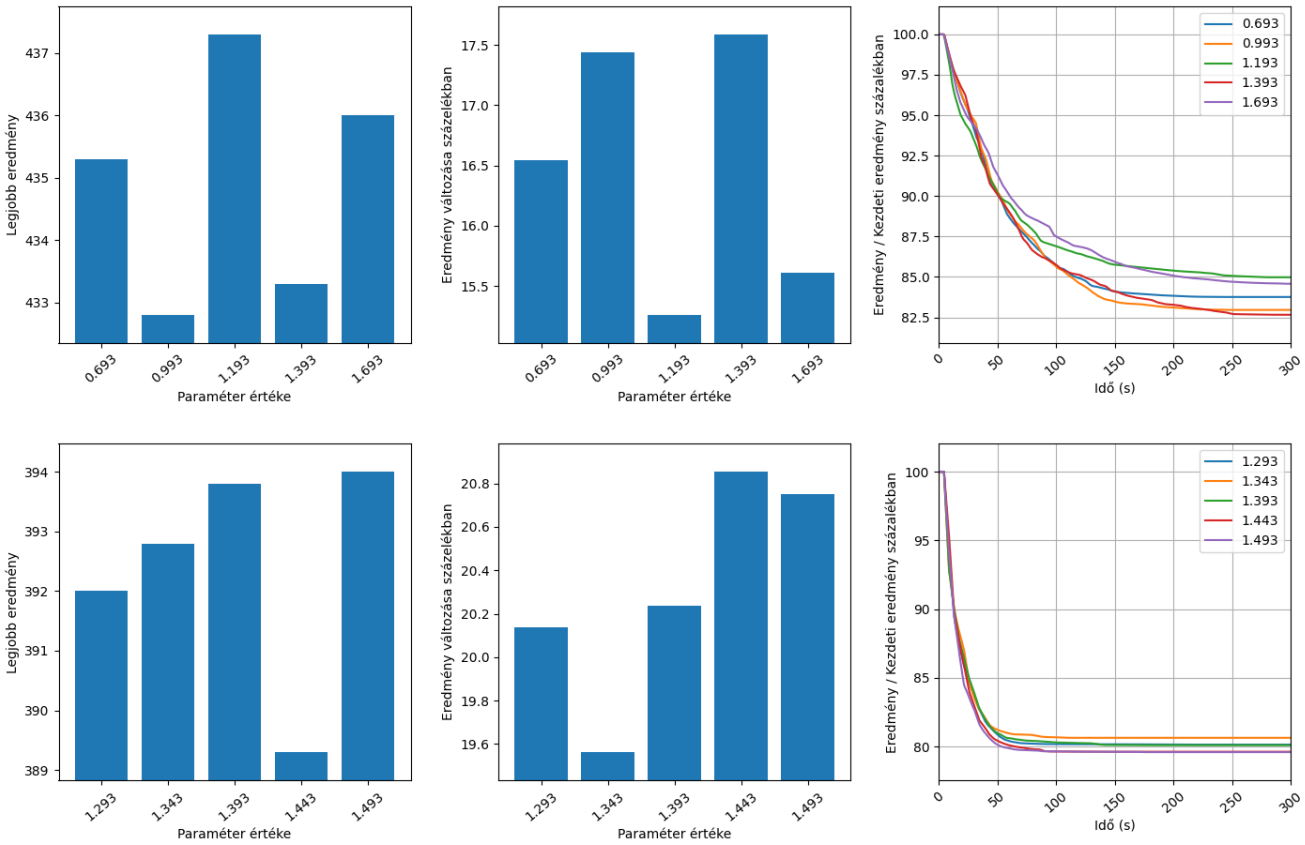
5.4. ábra. A PSO vizsgálata a csoport legjobb pozíciójának együtthatójának tekintetében.

Az első körben egyértelműen a 0.993 érték bizonyult a legjobbnak minden esetben, így azt vittem tovább.

A második körben már sokkal szorosabb volt a verseny. A görbék a 0.893, 0.993 és a 1.093 esetében szinte egymásra illeszkednek. Végül a 0.993 értéket választottam az optimális értéknek, ugyanis a százalékos változás tekintetében hangyányival, de ez teljesít a legjobban, valamint a második helyezetthez képest, a végeredmény tekintetében is jobban teljesít.

5.3.3. A részecske legjobb pozíciójának együtthatója

Ez a paraméter a módszer Φ_p értékének feleltethető meg. Ez kontrollálja, hogy mekkora mértékben vegyék figyelembe a korábbi legjobb pozíciójukat a részecskék.



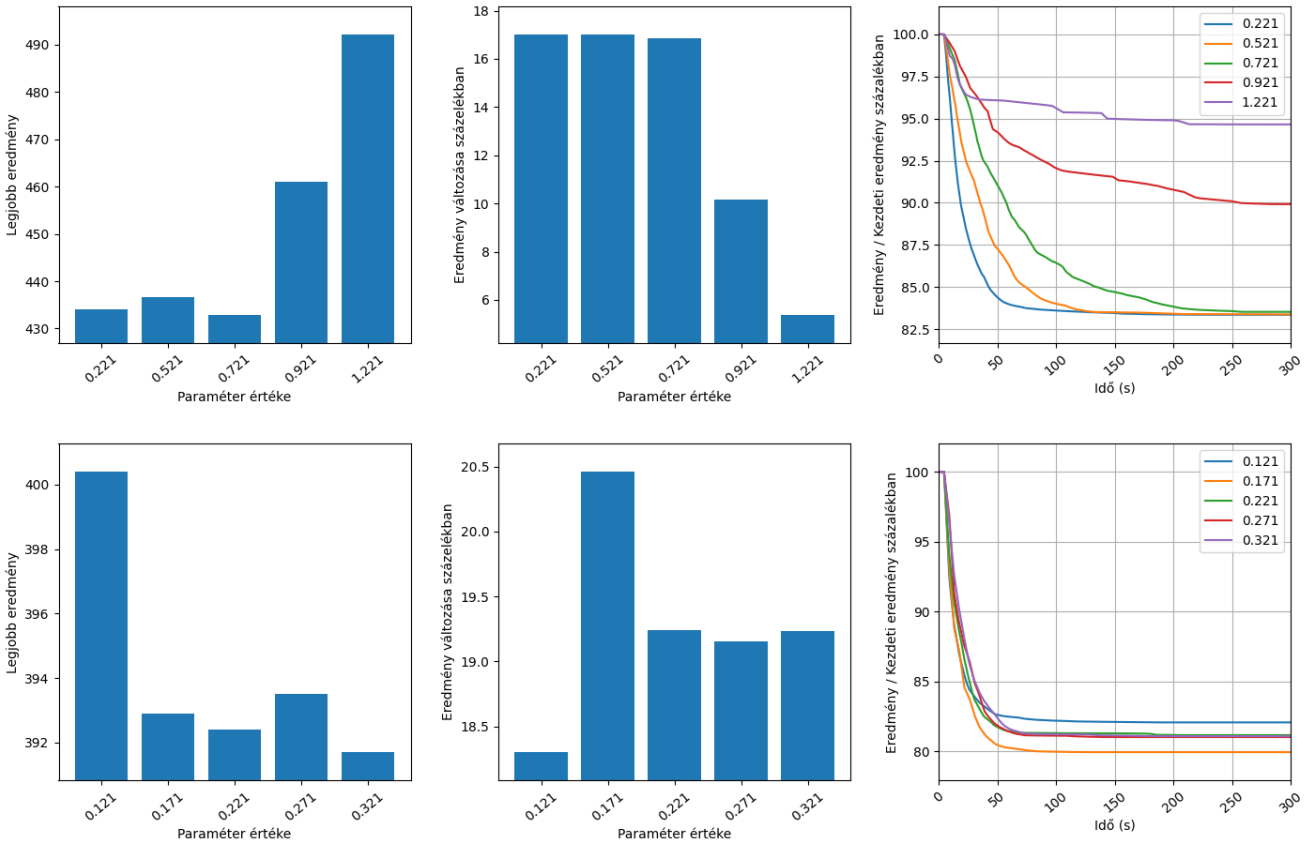
5.5. ábra. A PSO vizsgálata a részecske legjobb pozíciójának együtthatójának tekintetében.

Az első körben a 0.993 és az 1.393 elég szoros eredményeket produkáltak, végül a jobb százalékos változás miatt az 1.393 értéket választottam.

A második kör során a görbék megint nagyon egymásra esnek, a többi ábra alapján viszont könnyedén kijelenthető, hogy az 1.443 érték a legjobb, így azt választottam az optimális értéknek.

5.3.4. Az előző mozgási vektor együtthatója

Ez a paraméter a módszer w értékének feleltethető meg. Ez kontrollálja, hogy mekkora hatása legyen az előző elmozdulási vektornak az újra.



5.6. ábra. A PSO vizsgálata az előző mozgási vektor együtthatójának tekintetében.

Az első kör során a 0.221, 0.521 és a 0.721 elég szoros eredményeket produkált az első két diagram alapján, viszont a harmadik ábrán jól látható, hogy a 0.221 sokkal meredekebben közelíti ugyan azt az értéket, így azt választottam a következő körhöz.

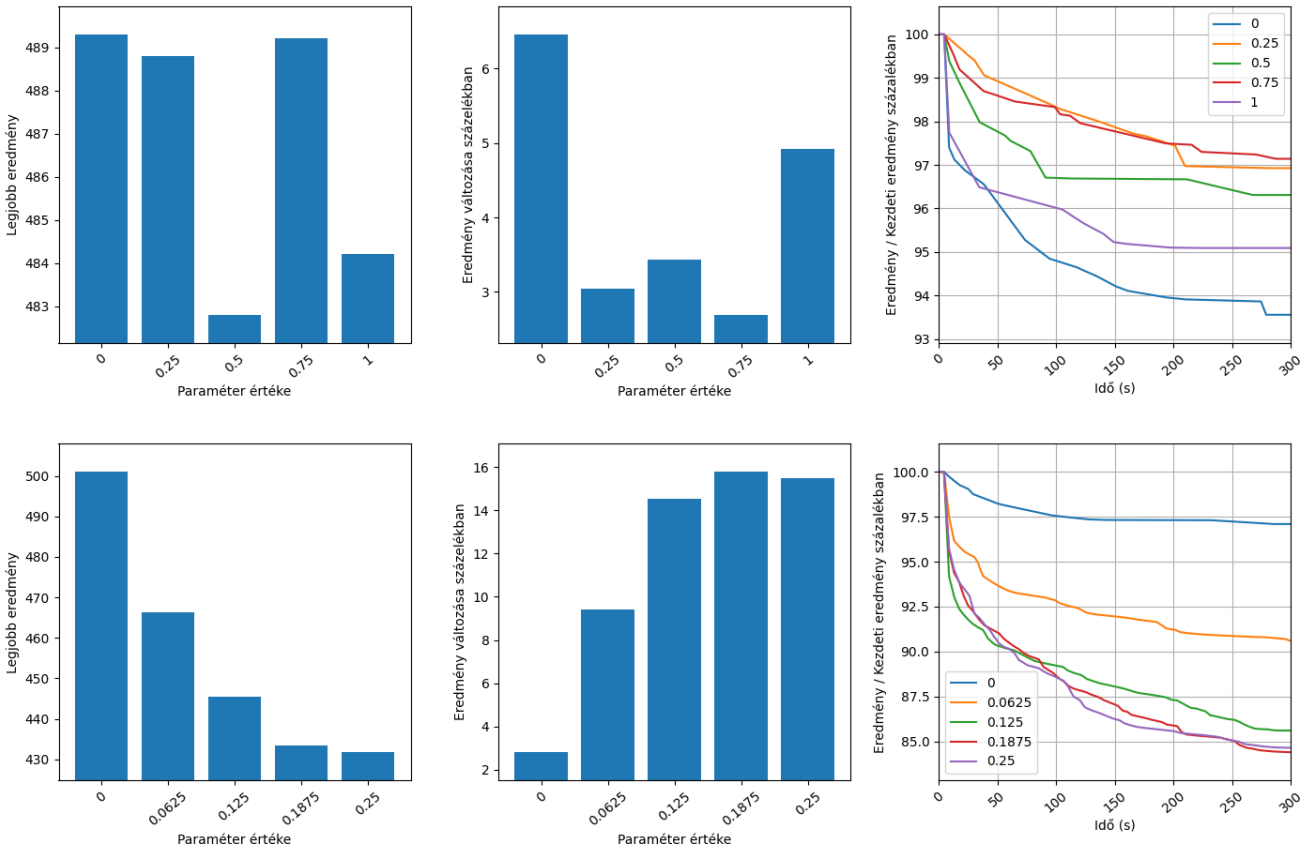
A második körben már könnyebb dolgom volt, ugyanis jól kitűnik, hogy a 0.171 a görbe meredeksége és a százalékos változás tekintetében is a legjobb volt, míg a végeredmény szempontjából csak kicsivel maradt le, így végül ezt választottam az optimális paraméternek.

5.4. AFSO paramétereinek hangolása

Ebben az alfejezetben a mesterséges halraj optimalizációs módszer (AFSO / AFSA) paramétereit fogom vizsgálni.

5.4.1. A raj méretének paramétere

Ez a paraméter a módszer δ értékének feleltethető meg. Ez kontrollálja, hogy mekkora mérete legyen a létrejövő halrajoknak.



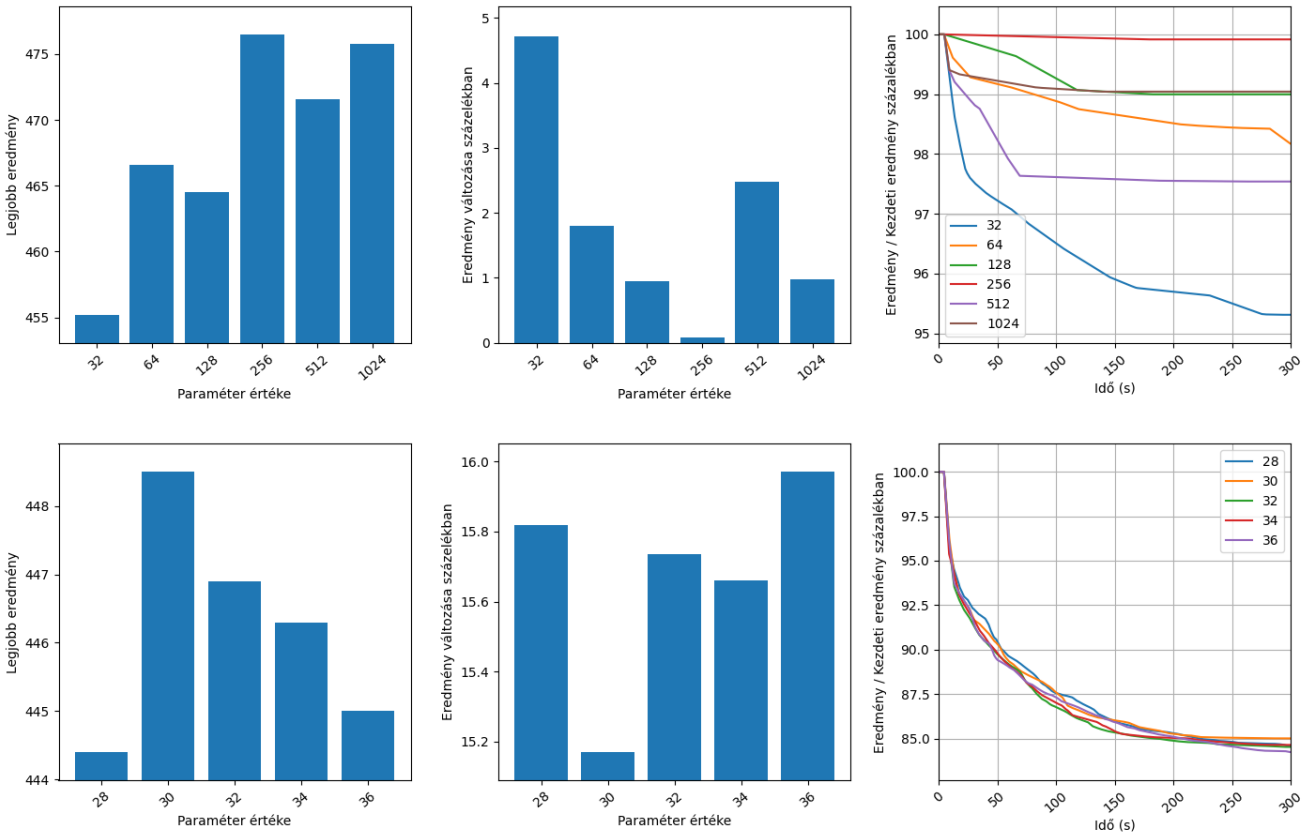
5.7. ábra. Az AFSO vizsgálata a raj méretének paraméterének tekintetében.

Az első kör során egyértelműen a 0 érték bizonyult a legjobbnak. Ez olyan szempontból problémás, hogy ez azt jelentené, hogy nincs is szükség a módszerben a csoportos viselkedésekre. Ezért ezt újra megvizsgáltam, de ugyanezt az eredményt kaptam, szóval végül a 0-t vittem tovább a következő körbe.

A második körben, mivel a paraméter értéke nem lehet negatív, így az előbb kiválasztott 0 érték nem a tartomány közepére, hanem a szélére került. Itt szerencsére már nem a 0 lett a legjobb. Gyakorlatilag csak hajszálnyi különbség látható a 0.1875 és a 0.25 értékek között, de végül a jobb százalékos változása miatt, a 0.1875 értéket választottam az optimális paraméternek.

5.4.2. A maximális lépésköz paramétere

Ez a paraméter a módszer *Step* értékének feleltethető meg. Ez kontrollálja, hogy maximálisan mekkorát léphetnek a halak egy mozgási akció során.



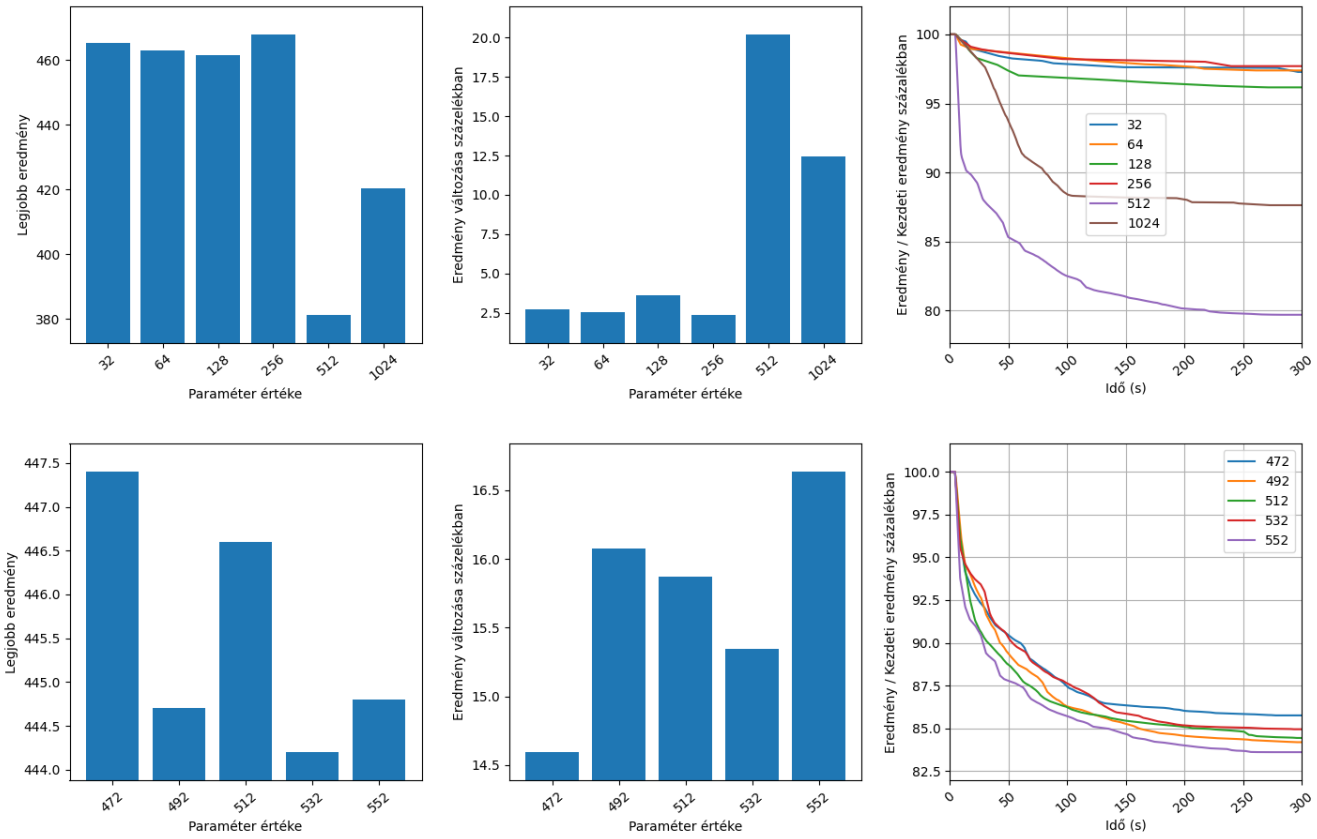
5.8. ábra. Az AFSO vizsgálata a maximális lépésköz paraméterének tekintetében.

Az első körben egyértelműen a 32 érték bizonyult a legjobbnak mind a 3 diagram alapján, így azt választottam a második körhöz.

A második kör során a görbék között nagy különbség nem látható, de a százalékos változás alapján kijelenthető, hogy a 36 az optimális paraméterérték.

5.4.3. Az észlelési távolság paramétere

Ez a paraméter a módszer *Visual* értékének feleltethető meg. Ez kontrollálja, hogy maximálisan mekkora távolságban tudják észlelni a környezetüket a halak.



5.9. ábra. Az AFSO vizsgálata az észlelési távolság paraméterének tekintetében.

Az első körben egyértelműen látható, hogy kiugróan a legjobb lett az 512, melyet így kiválasztottam a következő körhöz.

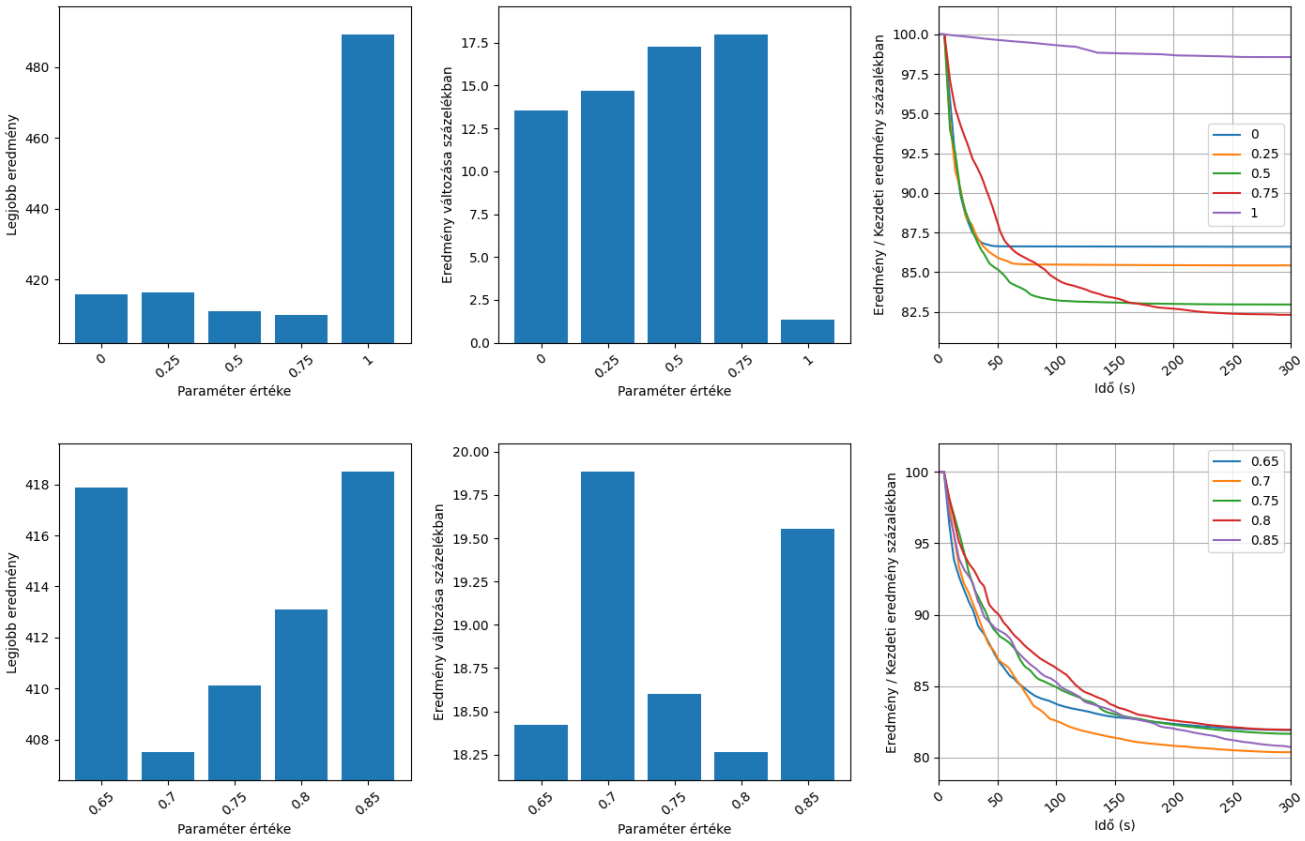
A második kör során már szorosabbak voltak az eredmények, de viszonylag még mindig egyértelműen megállapítható, hogy az 552 teljesített a legjobban, így azt választottam az optimális paraméterértéknek.

5.5. EBA paramétereinek hangolása

Ebben az alfejezetben a mesterséges denevérekkel működő optimalizációs módszer (EBA) paramétereit fogom vizsgálni.

5.5.1. A PulseRate paraméter

Ez a paraméter a módszer *PulseRate* értékének feleltethető meg. Ez kontrollálja a csoportosulásos mozgás gyakoriságát.



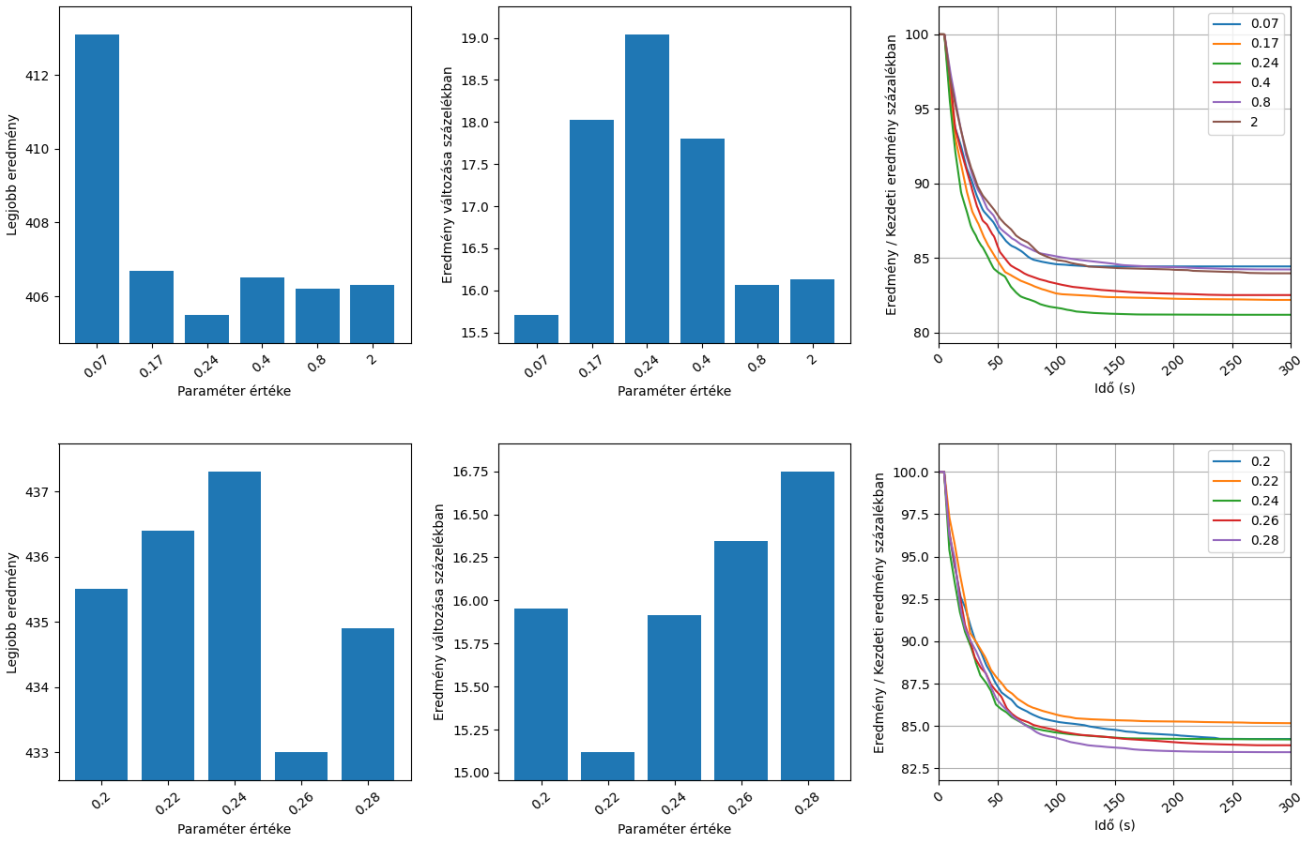
5.10. ábra. Az EBA vizsgálata a PulseRate paraméter tekintetében.

Az első kör során a 0.5 és a 0.75 hasonló eredményeket produkáltak. A görbe eleje alapján a 0.5 bizonyult jobbnak, de a végére már a 0.75. Mivel hosszabb távon gondolkodom, így inkább a 0.75 -t választottam a következő körhöz.

A második körben, habár nagyon nagy különbségek nem voltak az eredmények között, minden ábra alapján a 0.7 bizonyult a legjobbnak, így végül azt választottam az optimális paraméterértéknek.

5.5.2. A véletlen lépés távolságának paramétere

Ez a paraméter nem feleltethető meg a módszer egyetlen paraméterének sem, ugyanis ez az eredeti változatban egy konstans érték, mégpedig 0.17. Viszont úgy gondoltam, hogy érdemes lehet megvizsgálni, hogy biztosan a 0.17-e a legjobb.



5.11. ábra. Az EBA vizsgálata a véletlen lépés távolságának paraméterének tekintetében.

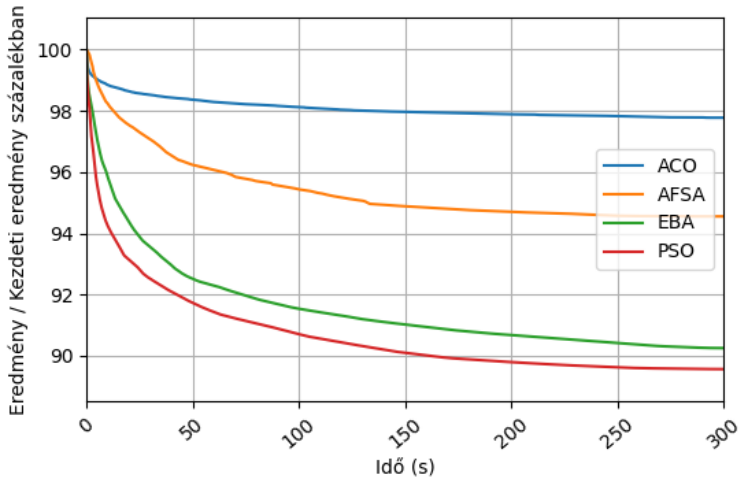
Már az első kör során bebizonyosult, hogy nem feltétlenül a 0.17 a legjobb érték. Ugyanis esetünkben egyértelműen a 0.24 bizonyult a legjobbnak, melyet ki is választottam a következő körhöz.

A második körben már sokkal szorosabb volt a verseny, csak nagyon kis mértékben különböztek az eredmények. Végül a százalékos változás és a görbe alakulása alapján a 0.28-at választottam az optimális paraméterértéknek.

5.6. A módszerek összehasonlítása

Miután meghatároztam minden módszerhez az optimális paraméterértékeket, összehasonlítottam őket, hogy kiderüljön melyiket is lenne érdemes használni az aktuális probléma esetén. Ahhoz hasonlóan, mint a paraméterek esetén, itt is tesztfuttatásokat végeztem, melyek eredményét végül összegeztem. Ezek az összegzett eredmények láthatóak az 5.12. ábrán. Itt az összegzés azt jelenti, hogy az összes vizsgált esetet átlagolva vettem figyelembe. Természetesen külön-külön, egy adott beállítás esetére is megtekinthetőek a görbék alakulásai, de ezeket, azok számossága miatt, már csak a

CD mellékleten helyeztem el.



5.12. ábra. A módszerek konvergencia görbéjének összehasonlítása.

A ábráról jól leolvasható, hogy általános esetben a PSO és esetleg még az EBA teljesít nagyon jól. A többi módszer kevésbé jó eredményeket produkál, de persze egy konkrét beállítás esetén elképzelhető, hogy másképp alakulnak a görbék.

Ezek alapján én a PSO módszert ajánlanám a korábbiakban felvetett probléma megoldására, mivel nagy valószínűséggel azzal érjük el a legjobb eredményt. A többi módszer kipróbálását kísérletezgetés erejéig azért még érdemesnek tartom vizsgálni.

Megjegyzendő, hogy a legjobb módszerrel se nagyon értünk el 20%-nál nagyobb javulást a kezdeti megoldáshoz képest, így csodákat nem kell várni tőlük. Ahogy az kicsit várható is volt egy ekkora méretű problémánál, valószínűleg csak lokális optimumokat tudunk találni, de véleményem szerint, garantálni ennek az ellenkezőjét hasonló futásidő mellett szinte lehetetlen lenne, legalábbis a tudomány jelen állása szerint.

6. fejezet

Összegzés

A dolgozat megírása során alkalmam adódott közelebbről is megismerkedni az optimális úthálózatokhoz kötődő minimális Steiner-fa problémájával. Ekkor szembesültem a ténnyel, hogy egy ilyen egyszerűnek tűnő probléma mennyire bonyolult lehet, hiszen az említett problémáról kiderült, hogy NP-nehéz. A megoldásához először megvizsgáltam az elterjedt heurisztika alapú módszereket, melyek viszonylag jó eredményeket adtak hasonló feladatok esetében, de többségében nem voltak képesek elérni az optimális eredményt.

A hatékonyabb megoldások érdekében elkezdtem tanulmányozni az ágens alapú heurisztikákat, amelyek így a dolgozat fő témakörét adják. Megvizsgáltam számos ágens alapú optimalizáló módszert, valamint a Steiner problémára való adaptálásához 3 keretrendszert. Ezek kapcsán igyekeztem áttekinteni a szakirodalmat, melyek során számos újdonságnak tekinthető módszert is kipróbáltam, mint például az AFSO és az EBA módszereket. Mivel új módszerekről van szó, ezért a szakirodalomban fellelhető hiányosságokra, és az alkalmazásuk során tapasztalt teljesítmény problémákra szintén kitértem a dolgozatban.

Az említett módszerek tesztelésére készítettem egy böngészőben futtatható alkalmazást. Ezt az Angular keretrendszer segítségével valósítottam meg, TypeScript programozási nyelven. Az alkalmazás lehetőséget ad a dolgozatban részletezett 4 ágens alapú módszer különböző paraméterekkel történő vizsgálatára. Az alkalmazás kapcsán számos optimalizálási lehetőséget megvizsgáltam, melyek közül néhány, meglepő eredményt produkált. Ilyen volt például a JavaScript esetén a mátrixkezelés, amelynél a kód szintjén optimalizálatlan változat bizonyult a leggyorsabbnak, valamint a legrövidebb útszámító algoritmusok esetén, amelyeknél hatalmas különbségek adódtak a számítási időben.

Az elkészült alkalmazást természetesen lehetne még tovább fejleszteni, például több algoritmus implementálásával a magasságmező generálásához, a térképen víznek jelölt részek kihagyásával az útkeresésből, az ágensek átlós mozgásának implementálásával, párhuzamosítási lehetőségek megvizsgálásával, az optimalizációs eljárásokból további kilépési feltételek implementálásával, valamint a kód gyorsításával WebAssembly segít-

ségével.

A dolgozat végén számos példát láthatunk az alkalmazás használatára vonatkozóan, tesztfuttatások formájában. Az eredmények segítségével megbecsültem a módszerek paramétereinek optimális értékét, valamint össze is hasonlítottam őket. Végezetül konklúzióként az eredmények alapján meghatároztam, hogy mely módszert érdemes a leginkább használni az általam elkészített alkalmazásban.

7. fejezet

Summary

By this thesis work, I had an opportunity to study the topic of optimal networks in detail, which are closely related to the problem of Steiner trees. In the meantime, I have recognised the fact, that the problem, which seems to be simple at the first sight is fairly complex, so called NP-hard. Firstly, I have reviewed the commonly used heuristics, which provide acceptable solutions for similar problems. Unfortunately, most of them were unable to obtain the optimal result.

In the aim of more efficient solutions, I have started to examine the agent based heuristics, as the main topic of the thesis work. There were investigated many of the agent based optimisation methods, furthermore 3 frameworks for adapting them to the Steiner problem. Their literature has been reviewed in the work, some of the novel methods have been tried, for instance the AFSO and the EBA methods. In the case of new methods, I have mentioned some deficiencies of the literature and the performance problems of the heuristics.

For the experimentation and measurements, I have designed and implemented a web application. It uses the Angular framework and has been written in the TypeScript programming language. The application makes available the analysis of the mentioned 4 agent based method by various parametrisation. I have studied many optimisation options, and some of them resulted in unexpected outcomes. For instance, in the case of matrix implementation in JavaScript, where the unoptimised source code resulted in the best running time. Moreover, for the shortest path calculations, where large differences have been revealed in the calculation time.

Naturally, there have been remained possibilities for further improvements. For instance, by implementing other algorithms for generating the height maps, by ruling out the water parts of the map from the pathfinding stage, by implementing diagonal steps for the agents, by examining the possibilities of parallelisation, by adding further exit conditions for the optimisation methods or by improving the performance by using WebAssembly.

At the last part of the work, we can find many examples of the usage of the implemented application as test runs. Based on these results, the optimal parameters of the

methods have been estimated and compared. Finally, I have provided a guide, which is based on the achieved results, for selecting the proper method in the presented application as a conclusion.

Irodalomjegyzék

[1] *Térkép*,
https://hu.wikipedia.org/wiki/T%C3%A9rk%C3%A9p,
Wikipedia, Internet (2021).

[2] Red Blob Games:
Making maps with noise functions,
https://www.redblobgames.com/maps/terrain-from-noise/,
Internet (2021).

[3] Jordan Mecom:
Procedural Terrain Generation: Diamond-Square,
http://jmecom.github.io/blog/2015/diamond-square/,
Internet (2021).

[4] Alain Fournier, Don Fussell, and Loren Carpenter:
Computer rendering of stochastic models,
Commun. ACM, 25, 6 (1982), 371–384.

[5] Gavin S P Miller:
The definition and rendering of terrain maps,
SIGGRAPH Comput. Graph., 20, 4 (1986), 39–48.

[6] Paul Martz:
Generating Random Fractal Terrain,
https://web.archive.org/web/20060420054134/http://www.gameprogrammer.com/fractal.html#diamond,
Internet (2021).

[7] Nick O'Brien:
Diamond-Square Algorithm Explanation and C++ Implementation,
https://medium.com/@nickobrien/diamond-square-algorithm-explanation-and-c-implementation-5efa891e486f,
Internet (2021).

[8] Fralonra:
Zatlas - Online random heightmap generator,

https://fralonra.github.io/zatlas/,
Internet (2021).

[9] Fralonra:
Diamond-square heightmap,
https://github.com/fralonra/ds-heightmap,
Internet (2021).

[10] Christopher Ewin:
Visualization of the Diamond Square Algorithm,
https://en.wikipedia.org/wiki/Diamond-square_algorithm#/media/File:Diamond_Square.svg,
Internet (2021).

[11] Dreyfus, S.E. and Wagner, R.A:
The steiner problem in graphs,
Networks, 1 (1971), 195-207.

[12] Ljubić, I:
Solving Steiner trees: Recent advances, challenges, and perspectives,
Networks, 77 (2021), 177-204.

[13] Bern, Marshall W.; Graham, Ronald L:
The shortest-network problem,
Scientific American., 260, 1 (1989), 84–89.

[14] Ivanov, Alexander O. and Alexey A. Tuzhilin:
Minimal Networks: The Steiner Problem and Its Generalizations,
CRC press (1994).

[15] Cmglee:
Minimum Steiner trees of (shortest networks connecting) vertices of regular polygons with N = 3 to 8 sides,
https://en.wikipedia.org/wiki/File:Regular_polygon_Euclidean_Steiner_tree.svg,
Internet (2021).

[16] Felner, Ariel:
Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm,
Proc. 4th Int'l Symp. on Combinatorial Search (2011).

[17] Mehlhorn, Kurt; Sanders, Peter:
Algorithms and Data Structures: The Basic Toolbox,
Springer (2008), 191-216.

[18] Mehlhorn, Kurt; Sanders, Peter: *Algorithms and Data Structures: The Basic Toolbox*,
Springer (2008), 217-232.

[19] Clemens Gröpl, Stefan Hougardy, Till Nierhoff, Hans Jürgen Prömel:
Approximation Algorithms for the Steiner Tree Problem in Graphs,
Humboldt Egyetem, Berlin (2001).

[20] Hiromitsu Takahashi, Akira Matsuyama:
An approximate solution for the Steiner problem in graphs,
Mathematica Japonica, 24 (1980), 573-577.

[21] J. Plesnik:
Heuristics for the Steiner problem in graphs,
`https://core.ac.uk/download/pdf/82609861.pdf`,
Discrete Applied Mathematics, 37/38 (1992), 451-463.

[22] Pawel Winter, J. MacGregor Smith:
Path-Distance Heuristics for the Steiner Problem in Undirected Networks,
Algorithmica, 7 (1992), 309-327.

[23] L. Kou, George Markowsky, and L. Berman:
A fast algorithm for Steiner trees,
Acta Informatica, 15 (1981), 141-145.

[24] J. Plesnik:
A bound for the Steiner problem in graphs,
Mathematica Slovaca, 31 (1981), 155-163.

[25] Choukhmane El-Arbi:
Une heuristique pour le problème de l'arbre de Steiner,
RAIRO. Recherche Opérationnelle, 12 (1978), 202-212.

[26] V.J. Rayward-Smith:
The computation of nearly minimal Steiner trees in graphs,
Internat. J. Math. Ed. Sci. Tech., 14 (1983), 15-23.

[27] V.J. Rayward-Smith, A. Clare:
On finding Steiner vertices,
Networks, 16 (1986), 283-294.

[28] Magnús M. Halldórsson, Shuichi Ueno, Hiroshi Nakao, Yoji Kajitani:
Approximating Steiner trees in graphs with restricted weights,
`https://www.ru.is/faculty/mmh/raunvis/Papers/netw.pdf`,
Networks, 31 (1998), 283-292.

[29] Sabzian, Hossein; Shafia, Mohammad Ali; Bonyadi Naeini, Ali; Jandaghi, Gholam Reza; Sheikh, Mohammad Javad:
A Review of Agent-based Modeling (ABM) Concepts and Some of its Main Applications in Management Science,
Iranian Journal of Management Studies, 11, 4 (2018), 659–692.

[30] Niazi, Muaz; Hussain, Amir:
Agent-based Computing from Multi-agent Systems to Agent-Based Models: A Visual Survey,
Scientometrics, 89, 2 (2011), 479–499.

[31] Szécsényi Hajnalka:
MI2 előadás jegyzet,
`http://www.inf.u-szeged.hu/~jelasity/mi2/2008/Szecsenyi_Hajnalka-jegyzet.pdf`,
SZTE TTIK Informatikai Intézet (2008).

[32] Yoann Kubera, Philippe Mathieu, Sébastien Picault:
Everything can be Agent!,
9th International Conference on Autonomous Agents and Multiagent Systems (2010).

[33] Danny Weyns, Andrea Omicini, James Odell:
Environment as a first class abstraction in multiagent systems,
Autonomous Agents and Multi-Agent Systems, 14 (2006), 5–30.

[34] Reinhard Laubenbacher, Franziska Hinkelmann, Matt Oremland:
Agent-Based Models and Optimal Control in Biology: A Discrete Approach,
Mathematical Concepts and Methods in Modern Biology (2013), 143–178.

[35] Shelby M. Scott, J. Casey E. Middleton, Erin N. Bodine:
An Agent-Based Model of the Spatial Distribution and Density of the Santa Cruz Island Fox,
Integrated Population Biology and Modeling, Part B (2019), 3–32.

[36] Petra Heijnen, Émile J.L. Chappin, Igor Nikolic:
Infrastructure Network Design with a Multi-Model Approach: Comparing Geometric Graph Theory with an Agent-Based Implementation of an Ant Colony Optimization,
Journal of Artificial Societies and Social Simulation (2014).

[37] Shima Kashef, Hossein Nezamabadi-pour:
A new feature selection algorithm based on binary ant colony optimization,
The 5th Conference on Information and Knowledge Technology (2013).

[38] Murilo Zangari de Souza, Aurora Trinidad Ramirez Pozo:
Multiobjective Binary ACO for Unconstrained Binary Quadratic Programming,
Brazilian Conference on Intelligent Systems (BRACIS) (2015).

[39] Peter Kuhn, Jannik Luboeinski, Laura Martin, Marius Schneider:
Ants Simulation,
`https://itp.uni-frankfurt.de/~gros/StudentProjects/Applets_2014_AntsSimulation/ants.htm`,
Goethe University Frankfurt (2014).

[40] Mesude Bicak:
Agent-Based Modelling of Decentralized Ant Behaviour using High Performance Computing, PhD thesis
`https://etheses.whiterose.ac.uk/1392/2/Bicak%2C_Mesude.pdf`,
The University Of Sheffield (2014).

[41] Marco Dorigo:
Optimization, Learning and Natural Algorithms, PhD thesis,
Politecnico di Milano, Italy (1992).

[42] Wen-Liang Zhong, Jian Huang, Jun Zhang:
A novel particle swarm optimization for the Steiner tree problem in graphs,
IEEE Congress on Evolutionary Computation (2008).

[43] Kennedy, J.; Eberhart, R:
Particle Swarm Optimization,
Proceedings of IEEE International Conference on Neural Networks, IV (1995), 1942–1948.

[44] Kennedy, J:
The particle swarm: social adaptation of knowledge,
Proceedings of IEEE International Conference on Evolutionary Computation (1997), 303–308.

[45] Shi, Y.; Eberhart, R.C:
A modified particle swarm optimizer,
Proceedings of IEEE International Conference on Evolutionary Computation (1998), 69–73.

[46] Aiman H. El-Maleh, Ahmad T. Sheikh Sadiq M. Sait:
Binary particle swarm optimization (BPSO) based state assignment for area minimization of sequential circuits,
Applied Soft Computing, Volume 13, Issue 12 (2013), 4832-4840.

[47] Mojtaba Ahmadieh Khanesar, Mohammad Teshnehlab, Mahdi Aliyari Shoorehdeli:
A novel binary particle swarm optimization,
Mediterranean Conference on Control & Automation (2007), 1-6.

[48] Elliackin M. N. Figueiredo, T. Ludermir:
Topologies of the PSO,
`https://www.semanticscholar.org/paper/Effect-of-the-PSO-Topologies-on-the-Performance-of-Figueiredo-Ludermir/7ac77af9b663f6f3ae65fe2c158121b613fe1797/figure/0`,
Internet (2021).

[49] Quing Liu, Tomohiro Odaka, Jousuke Kuroiwa, Haruhiko Shirai, Hisakazu Ogura:
An Artificial Fish Swarm Algorithm for the Multicast Routing Problem,
IEICE Transactions on Communications, E97.B(5) (2014), 996–1011.

[50] Xuan Ma, Qing Liu:
An Artificial Fish Swarm Algorithm for Steiner Tree Problem,
IEEE International Conference on Fuzzy Systems (2009).

[51] Mehdi Neshat, Ghodrat Sepidnam, Mehdi Sargolzaei, Adel Najaran Toosi:
Artificial fish swarm algorithm: a survey of the state-of-the-art, hybridization, combinatorial and indicative applications,
Artificial Intelligence Review, 42(4) (2012), 965–997.

[52] Li LX, Shao ZJ, Qian JX:
An optimizing method based on autonomous animals: fish-swarm algorithm,
Systems Engineering - Theory & Practice, 22(11) (2002), 32–38.

[53] Reza Azizi:
Artificial Fish and the Environment,
`https://www.semanticscholar.org/paper/Empirical-Study-of-Artificial-Fish-Swarm-Algorithm-Azizi/cce58c87b595abcc9e6d351861bb5cd2ffac2d4c/figure/0`,
Internet (2021).

[54] Xin-She Yang:
Nature-Inspired Metaheuristic Algorithms,
University of Cambridge, Luniver Press (2010).

[55] Xin-She Yang:
A New Metaheuristic Bat-Inspired Algorithm,
Studies in Computational Intelligence (2010), 65–74.

[56] Pei-Wei Tsai, Jeng-Shyang Pan, Bin-Yih Liao, Ming-Jer Tsai, Vaci Istanda:
Bat Algorithm Inspired Algorithm for Solving Numerical Optimization Problems,
Applied Mechanics and Materials (2011), 148-149, 134–137.

[57] MDN Web Docs:
TypedArray,
`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray`,
Internet (2021).

[58] MDN Web Docs:
Using Web Workers,
`https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`,
Internet (2021).

[59] Baeldung:
Comparison Between Uniform-Cost Search and Dijkstra's Algorithm,
`https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras`,
Internet (2021).

[60] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford:
Introduction to Algorithms (3rd ed.),
MIT Press and McGraw-Hill (2009).

[61] Brodal, Gerth S:
Worst-Case Efficient Priority Queues,
Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (1996), 52–58.

Az internetes források utolsó ellenőrzése: 2021.11.14

CD melléklet tartalma

A dolgozathoz mellékelt lemezen egy `Dolgozat` nevű jegyzékben a következő fájlok találhatóak.

- A témakiírás Word-dokumentum és PDF formátumban (`Szacsuri-Norbert-G0XXTC-diplomamunka-2-msc-GEIAL536M-kiiras.docx`, `Szacsuri-Norbert-G0XXTC-diplomamunka-2-msc-GEIAL536M-kiiras.pdf`).
- A dolgozat LaTeX forráskódja (`dolgozat.tex`).
- A dolgozat PDF formátumban (`dolgozat.pdf`).
- A magyar és angol nyelvű összefoglaló LaTeX és PDF formátumban (`osszegzes.tex`, `osszegzes.pdf`, `summary.tex`, `summary.pdf`).

A lemez `Program` nevű jegyzékében található a dolgozathoz elkészített program forráskódja és futtatható változata, valamint az egyéb használt forráskódok. Ezek az alábbi jegyzékekbe lettek szétosztva.

- `Alkalmazas forditott` jegyzék, ebbe az alkalmazás lefordított változata került.
- `Alkalmazas forrasok` jegyzék, ebbe az alkalmazás forrásfájljai kerültek.
- `Egyeb forraskodok` jegyzék, ebben a diagramok készítéséhez és az alkalmazás optimalizációjához készített forráskódokat találhatjuk meg.

A forrásfájljok és a fordított alkalmazás használatához szükségünk lesz a `Node.js` és az `NPM` programokra, valamint aktív internet kapcsolatra.

Az alkalmazás fordított változatának futtatásához a fájlokat egy webszerverre kell felmásolnunk. Amennyiben nincs ilyen a kezünk ügyében, az adott mappából az `npx http-server --port 5050 --silent -o` paranccsal tudunk ideiglenes létrehozni egyet. Az előbbi parancs elméletileg automatikusan meg is nyitja az alkalmazást az alapértelmezett böngészőnkben, amennyiben mégsem történne így, a `http://localhost:5050` címen érhetjük azt el.

Az alkalmazást a forrásokból történő futtatásához először a `UI` jegyzékből az `npm install` parancsot kell kiadnunk, amivel letöltjük az alkalmazás függőségeit, majd az `npm start` paranccsal fejlesztői módban elindítjuk az alkalmazást. Lehetőség van az alkalmazás optimalizált változatát is elkészíteni, melyet az `npm run build` paranccsal tudunk elérni. Ennek hatására létrejön egy `dist` jegyzék, ami a webalkalmazás statikus fájljait fogja tartalmazni. Ezeket a fájlokat a korábban leírt módon tudjuk felhasználni

Az egyéb forráskódok az `npm install` parancs és a `package.json`-ben található scriptek, valamint a `node <jsfile>` parancs segítségével futtathatóak.

A lemez `Meresek` nevű jegyzékében a dolgozat során elkészített mérések eredményei találhatóak meg, melyek az alábbiak szerint lettek tovább bontva:

- `Parameter vizsgalatok` jegyzék, ebbe kerültek a vizsgált módszerek különböző paraméterezésével mért eredményei.
- `Modszerek osszehasonlitasa` jegyzék, ebbe kerültek a vizsgált módszerek összehasonlítása során mért eredmények.
- `Optimalizalasok` jegyzék, ebbe kerültek az alkalmazás optimalizálása során készített mérések eredményei.

Az előbb említett jegyzékek mindegyike tartalmaz még egy `Adatok` jegyzéket, amibe a konkrét mért adatok, valamint egy `Diagramok` jegyzéket, amibe az adatok alapján elkészített diagramok kerültek.