Univerzális Véges Állapotú és Önstabilizáló Számítás Anonim Dinamikus Hálózatokban

Cikket írta: Giuseppe A. Di Luna1, Giovanni Viglietta

Riportot készítette: Csavar Abigél, Jurca Henriette, Nick Mónika

### Bevezetés

Az általunk feldolgozott cikk és implementált algoritmus az anonim, dinamikus hálózatokban való számítások problémájával foglalkozik. Manapság egyre nagyobb népszerűségnek örvendenek például a vezeték nélküli szenzorhálózatok, okos eszközök által alkotott hálózatok, vagy egyéb szoftveresen definiált hálózatok. Ez azt is jelenti, hogy fontosabbá vált a változó topológiájú hálózatok vizsgálata. Ennek egy változatát jelentik azok a hálózatok, ahol az ügynökök szinkron módon kommunikálnak egymással, miközben a hálózat mindig összefüggő marad, de a kapcsolatok kiszámíthatatlanul módosulnak. Különösen érdekesek az anonim hálózatok, ahol az ügynököknek nincs egyedi azonosítójuk, ezáltal megkülönböztethetetlenek.

Az eddigiekben az ilyen hálózatokban, ahol mindig összefüggő marad a hálózat, és minden ügynök kezdetben kap egy bemenetet, n ügynök esetében 2n kommunikációs kör szükséges ahhoz, hogy bármilyen bemeneti függvény, értéket kiszámíthassunk. Mivel az ügynökök nem ismerik a hálózat méretét, a legtöbb függvény helyes, explicit kiszámítása és a leállás lehetetlen. Ezeket a problémákat megoldó algoritmusok csak stabilizációra képesek, ahol minden kör után adnak egy eredményt, ami idővel “stabillá”, azaz helyessé válik. Ezek az algoritmusok egy folyamatosan növekvő adatstruktúrára, a történet fára építenek, ami korlátlan memóriát igényel és sérülékeny lehet memóriavesztés esetén.

Az általunk feldolgozott, új kutatás többek között egy általános önstabilizáló algoritmust mutat be, amely max{4n − 2h, 2h} kör alatt stabilizálódik, ahol h az ügynökök kezdeti hibás adatainak mértéke. Az eredmények javítják a korábbi, csak statikus hálózatokra vonatkozó megoldásokat, és új módszereket kínálnak a történet fák használatában. A továbbiakban részletesen bemutatjuk ezt az algoritmust, megfogalmazva a kezdeti problémát, amire megoldást kínál, majd leírva az építőelemeit és a működését. Ezután megosztjuk a saját eredményeket: hogyan implementáltuk az algoritmust, milyen méréseket végeztünk, milyen eredményeket kaptunk.

### Célok, kihívások

Az egyik elsődleges cél egy *univerzális* algoritmus kidolgozása egy *anonim* hálózatban. A hálózat csomópontjainak anonimitása miatt az algoritmus nem képes tetszőleges függvény kiszámítására. Emiatt a megoldásunkban egy univerzális algoritmusra van szükség, ami azt jelenti, hogy bármilyen olyan függvény kiszámítható a kimenetéből, ami a bemenetek százalékos eloszlásából, gyakoriságából kiszámítható és nem az egyedi input értékekben érdekelt.​ Például tegyük fel, hogy minden csomópont egy egész számot kap inputként és a cél az, hogy megismerjük a leggyakoribb értéket, vagyis a közös megegyezést. Ebből bármely más gyakoriságalapú függvény (pl. súlyozott átlag) kiszámítható lesz.

A másik megvalósítandó kihívás az *ön-stabilizáció*: ez azt jelenti, hogy hibás állapotból is képes helyreállni, garantálva a stabil működést. Ha például egy adott ügynök hibás üzeneteket küld a többi ügynöknek, azt algoritmusnak akkor is ki kell tudnia számítani a stabil eredményt. A stabilizációt szintén megnehezíti a hálózat dinamikus jellege.

Szintén kihívást jelent az adott kontextusban a *véges állapotú*ság. Bár az általunk feldolgozott algoritmusban a hálózat mérete előre ismert, a topológiája folyamatosan változik. Ennek ellenére is le kell állnia az algoritmusnak úgy, hogy csak polinomiális méretű memóriát igényel.

Természetesen mindezek mellett a cél egy olyan algoritmus kidolgozása, amely minden lehetséges bemenet esetén helyes eredményt szolgáltat.​

### Számítási modell

Ebben a fejezetben bemutatjuk a pontos számítási modellt, és a szabályokat, amelyek szerint implementáltuk az algoritmust. Ehhez először röviden pontosítjuk a cikkben használt, számunkra fontos fogalmakat.

Az algoritmus áttekintése

Dinamikus hálózatban dolgozunk, amit egy mindig *összefüggő, irányítatlan gráf* reprezentál, amely élei folyamatosan változnak. Egy előre ismert, *n méret*ű hálózatból indulunk ki, ahol a csomópontok anonim ügynököket reprezentálnak, tehát a csomópontokat nem lehet megkülönböztetni.

*Minden kör elején* létrejön egy *új kapcsolati háló* a csomópontok (ügynökök) között, ahol az élek (kapcsolatok) véletlenszerűen alakulnak ki. Az ügynökök szomszédai tehát minden körben változhatnak.​

Az algoritmus futásának kezdetén *minden csomópont kap egy bemeneti értéket*, ez az érték lesz, ami az adott csomópont belső állapotát meghatározza a futás során.​ Az általunk implementált algoritmus célja a bemeneti értékek gyakoriságának kiszámítása.

*Egy kör* a következőt jelenti: minden ügynök lekérdezi az aktuális szomszédait​. Minden szomszédnak elküldi egy üzenetben a jelenlegi állapotát. Fontos, hogy a hálózat szinkron, azaz minden ügynök egyszerre küld és fogad üzeneteket minden körben. Ezután minden ügynök frissíti a saját állapotát, vagyis végrehajtja a számára szükséges számításokat, és ad egy kimenetet. Ezek a kimenetek fogják meghatározni a végső eredményt a stabilizáció bekövetkezésekor.

Végül, ha egy adott időponttól kezdve az összes ügynök már nem változtatja meg a kimenetét, akkor azt mondjuk, hogy az algoritmus stabilizálódott. Ekkor az ügynökök aktuális kimeneteiből megkapjuk a végső eredményt.

Történet fák

Mielőtt részletesen megvizsgálnánk a számítások menetét, nézzük meg, hogy pontosan mit jelentenek a történet fák.

Egy történeti fa mindig az adott csomópont lokális nézetéből áll össze, és végigkövethető rajta a hálózat időbeli változása, dinamizmusa. Minden fának van egy vezetője, ami jelen esetben a gyökér csomópont. Ez egy kijelölt csomópont, ami nem reprezentál semmilyen speciális értéket. A többi, erre faként épülő csomópont mindig egy adott input értéket jelképez.

A történet fa *csomópontjai* egymástól megkülönböztethetetlen ügynököket reprezentálnak. Ez azt jelenti, hogy ezek az ügynökök egyrészt ugyanazt az input értéket kapták, másrészt pedig a történet fában az adott csomópontból kiindulva a fekete éleken (ld. lentebb) át a gyökérig vezető út megegyezik. A csomópontok címkéi ezen ügynökök inputját írják le.

Egy történet fában kétféle él lehet: irányítatlan, *fekete élek* és irányított, multiplicitással is ellátott *piros élek.* A *fekete élek* határozzák meg magát a fát. A fa t-edik szintjén lévő v csomópont által reprezentált ügynökök halmazának részhalmaza lesz az a halmaz, amit a t+1-edik szinten lévő, v csomópont gyerek csomópontja (fekete él szerinti) jelzépez.

A *piros élek* a csomópontok kommunikációját illusztrálják. Egy piros és és a hozzá tartozó m súly azt jelenti, hogy adott körben a gyerek csomópont által reprezentált ügynökök m db azonos üzenetet kapnak a szülő-beli ágensektől.

Fontos még megjegyezni, hogy egy adott csomópont *nézet*e a történet fában az a részfa, amely a csomóponttól kiindulva a gyökérig vett összes legrövidebb utat (fekete és piros éleket is) tartalmazza.

A gyökéren kívül van még egy kijelölt csomópont a fákban, ez pedig az úgynevezett *alsó csomópont*. Kezdetben, a fa csak a gyökérből, és az adott ügynök input értékéből áll, ezért ilyenkor egyértelműen az input értékkel címkézett csomópont az alsó. Ezután minden körben, amikor két ágens interakcióba lépett egymással és elküldte a történet fáját, a fa egy új, ezt az interakciót reprezentáló alsó csomóponttal bővül.

#### A chop művelet

Az algoritmus egy chop nevű műveletet használ, amely egy csomópont történelmi fájából eltávolítja a hibás vagy felesleges adatokat.​ A későbbiekben nyilvánvalóvá válik, hogy ez elengedhetetlen ahhoz, hogy az algoritmus előbb-utóbb stabilizálódjon. A cél tehát a fa magasságát korlátozni, és normalizálni a struktúrát.

A művelet a következőképpen végezhető el: a fából eltávolítjuk a legalsó L0 (a gyökérhez legközelebbi) szintet. Az eltávolítás magában foglalja a csomópontokat és éleket is. Ez jelképezi tehát a régi, már kevésbé releváns információkat, amelyektől ilyen módon megválunk.

Ezután a következő, L1 szintet újra összekötjük közvetlenül a gyökérrel, fekete éleket használva. Ezáltal az egész nézet egy szinttel „lejjebb csúszik”: ami eddig L1 volt, most L0, stb.

Az új nézetet jól formálttá kell tenni, ehhez végig kell menni a szinteken, és minden szinten az azonos szerkezetű rész-nézeteket össze kell olvasztani, vagyis ha találunk két izomorf csomópontot, akkor azokat összevonjuk. Izomorfnak számít minden olyan csomópont páros, amelyek megkülönböztethetetlenek, vagyis ugyanaz a nézetük a gyökérig és ugyanaz az input értékük. Az összevonás azt jelenti, hogy csak az egyik ilyen nézetet őrizzük meg a fában, és az így kapott, “összeolvadt” csomópontból kimenő piros éleket, azok multiplicitását összeadjuk. Tehát ha például egy *u* csomópont kerül törlésre, és a vele izomorf *v* marad a fában, akkor az *u* gyerekei a *v* gyerekeivé válnak (a fekete éleket nézve). Az *u*-ba érkező piros élek eltűnnek, a *v*-ből kimenő piros élek megnövekednek az *u*-ból kimenő piros élek multiplicitásával (összeadás).

Ezt az összevonást azért tehetjük meg, mert ha két csomópontnak ugyanaz a nézete a történet fában, az a fa definíciója alapján azt jelenti, hogy ugyanazokat az üzeneteket kapták meg, tehát ugyanazt is küldik majd tovább. Ez a duplikáció pedig nemcsak felesleges, de lassítaná is az algoritmust. Az összevonással sok fölös számítást megspórolunk a későbbiekben.

A chop művelet tehát „elfelejti” a nézet legkorábbi kommunikációs körét, így lehetővé teszi a hibás vagy elavult adatok törlését és rendszer öngyógyítását, vagyis az ön-stabilizációt.

#### Merge - fák összefűzése

A merge művelet célja, hogy az ügynökök által ismert fákat összehangoljuk. A művelet a gyökértől indulva megy végig sorra a fa szintjein, és először megvizsgálja az adott szinteken, hogy van-e egymásnak megfelelő izomorf csomópont páros, ahol az egyik csomópont az egyik, a másik a másik fában van. Ez két adott fa esetén azt jelenti, hogy azokat a csomópontjaikat, amelyek a két fában egymástól megkülönböztethetetlenek, azokat összevonjuk. Ezt úgy is meg lehetne fogalmazni, hogy a végső, eredmény fában csak az egyiket őrizzük meg az izomorf csomópontok közül.

Ezután az adott szinten jön létre új csomópont, ha ugyanilyen címkével (input értékkel) rendelkező gyermek nem létezik a szülő alatt​.

Az új csomópontok felvételénél a fekete éleket egyszerűen le kell másolni. Az új csomópont felvételnél és az összevonásnál is kifejezetten kell figyelni a piros élekre: az adott csomópontból kimenő piros éleket kombinálni kell, a multiplicitása kat össze kell adni.

#### Számlálási szint

Ahogyan azt az univerzális algoritmusok tulajdonságainál is átterkintettük, az eredmény kiszámítása jelen esetben a bemenetek gyakoriságának, majd eloszlásának kiszámítását jelenti. Pontosabban egy Input Frequency-nek nevezett függvényt keresünk. Az algoritmus megalkotói bizonyították, hogy ha a hálózat minden körben összefüggő, akkor maximum 2n − 2 kör után az ügynökök nézetei elég információt tartalmaznak az Input Frequency függvény kiszámításához.

A függvény 2n − 2 kör alatti kiszámítását a számlálási szint, vagyis counting level segítségével lehet elvégezni. Számlálási szintnek nevezzük a történetfa azon szintjét, ahol minden csomópontnak pontosan egy gyereke van. Egy ilyen szintről a kimenő piros élet írják le az adott körbeli interakciókat. A szinten lévő csomópontok által reprezentált ügynökök száma és a piros élek, illetve azok multiplicitása alapján lineáris egyenletrendszer írható fel. Az egyenletrenszer megoldása pedig egyben a gyakoriság is minden input értékre.

### Ön-stabilizáló algoritmus

Az ön-stabilizáció olyan algoritmikus technika, amely lehetővé teszi egy elosztott rendszer számára, hogy ideiglenes hibák (pl. memóriahibák, üzenetvesztések) után is helyreálljon, azaz újra helyes működésre álljon be, külső beavatkozás nélkül. Az algoritmus lényege tehát, hogy bármilyen kezdeti állapotból (hibás vagy véletlenszerű) indulva, a rendszer véges idő után eljut egy helyes konfigurációba. Ezután, ha nem történik újabb hiba, a rendszer helyesen viselkedik a specifikációnak megfelelően. A helyes állapot nem sérül meg újra, amíg nem történik újabb hiba.

Ezt a működést megvalósító, anonim, dinamikus hálózatban működő univerzális algoritmusok 2n kör alatt stabilizálódtak. Az általunk feldolgozott cikk ezt továbbfejleszti és a leírt algoritmus legfeljebb max{4n - 2h, 2h} idő alatt stabilizálódik, ahol h a memóriában található kezdeti hiba mértéke, n pedig az előre ismert csomópontok száma. A h paramétertől való függés elkerülhetetlen a dinamikus hálózatokban.

#### Az algoritmus működése

Egy kör kezdetekor minden csomópont elküldi a szomszédainak az általa tárolt történelmi fát, és végrehajtja a megfelelő számításokat, amit jelen esetben az ágensek main() függvénye végez el. Ez a függvény a következőket valósítja meg:

1. Átveszi a szomszédoktól azok történet fáját.​ Megnézi, hogy melyik fa magassága a legkisebb, beleszámítva a kapott fákat és a saját történetfáját is.
2. Ha a saját fája magasabb ennél a meghatározott minimum magasságnál, akkor a chop művelettel levágja.
3. Hozzáad egy új alsó csomópontot a saját fájához, ami a saját input értékét tartalmazza, és ez lesz az új alsó csomópont.
4. Végigmegy a kapott üzeneteken, és a második lépéshez hasonlóan szükség esetén levágja a fát, ami a szomszédja fájának másolata.
5. Ezáltal a saját fája és az aktuális kapott fa azonos magasságú, vagyis összevonható a merge művelettel.
6. A merge után a saját fába egy új csomópontot kell felvenni, ami az új alsó csomópont lesz, és egy piros élet kell behúzni. A piros él a szomszédtól kapott fa alsó csomópontjának megfelelő csomópontból az újonnan felvett csomópontba mutat. Ez reprezentálja az ágens és az adott szomszéd közötti interakciót.
7. Végül, hogy a stabilizáció megfelelő időn belül megvalósulhasson, ha a merge műveletek után a saját fája 2n - 1-nél magasabb, egy újabb chop műveletet hajt végre.
8. Az algoritmus akkor áll le, ha elérünk egy számlálási szintet.

Mindezek mellett fontos megjegyezni, hogy minden kör elején, még az üzenetek küldése előtt minden ágens megvizsgálja a történet fáját: ha ez nem jól formált (vagyis valamilyen hiba, hibás információ miatt elromlott), vagy 2n - 2-nél magasabb, akkor újra inicializálja. Ez azt jelenti, hogy a kezdeti, gyökér és egyetlen, a saját inputjának mefelelő csomópontból fog állni.

#### Eredmény számítása

Miután a rendszer elindul, minden csomópont fokozatosan eltávolítja a hibás vagy ideiglenes adatokat a történelmi fájából a piros élek segítségével történő összehasonlítás és konszenzus révén. Ennek eredményeként minden helyes (korrekt) csomópont történelmi fájában egy jól meghatározható számlálási szint (counting level) jelenik meg.

Ez a számlálási szint jelzi azt a mélységet, amely alatt a történelmi fák már stabilak, hibamentesek és minden szükséges információt tartalmaznak. A csomópontok ezután ezeket a fákat felhasználva ki tudják számolni, hogy az egyes bemeneti értékek hányszor fordultak elő az egész hálózatban.

Az algoritmus garantálja, hogy minden hibamentes csomópont előbb-utóbb eléri a számlálási szintet, így pontos, egyező gyakorisági eloszlást tudnak számolni.

Az algoritmus akkor áll le, amikor minden ügynök (agent) már ki tudja számolni a bemenetek gyakoriságát az alapján, hogy a történelmi fája elérte a szükséges mélységet (counting level), és azóta nem változott. Az eredmény minden stabilizált csomópontban azonos lesz, függetlenül attól, hogy azok különböző útvonalakon jutottak el az információhoz.

### Eredmények

#### Implementáció

Az algoritmust Pythonban valósítottuk meg, az alábbi könyvtárak segítségével:

* network a fák és gráfok reprezentálására
* tkinter grafikus felhasználói felület megvalósítására
* Matplotlib a gráfok kirajzolására
* numpy numerikus számításokra

Az alábbiakban bemutatjuk az általunk megvalósított osztályokat, és programelemeket.

##### Agent

Az Agent osztály egy ügynököt reprezentál a hálózatban. Minden ügynök rendelkezik egy bemeneti értékkel és egy saját történeti fával, amelyben az információkat tárolja és frissíti. Az ügynökök körkörösen üzeneteket küldenek és fogadnak a szomszédjaiktól, ezeket integrálják a saját történeti fájukba, majd bizonyos feltételek teljesülése esetén levágják a fát, hogy korlátozzák a méretét.

A fő feladata az ügynök osztálynak, hogy összegyűjtse és kombinálja a szomszédokból származó információt, majd a történeti fából egy lineáris egyenletrendszer felállításával kiszámolja a valószínűségi eloszlást a bemeneti értékekre. Az algoritmus addig fut, amíg az ügynök eléri a stabil állapotot, és kiszámítja a kimeneti értékét. Az osztály támogatja az üzenetküldést, üzenetfogadást, a történeti fa manipulációját és az eredmények kiírását.

##### History Tree

A HistoryTree osztály értelemszerűen a történet fákat valósítja meg. Ehhez egy gráf osztályt (MultiDiGraph) használ a modellezésre, és egy gyökér, illetve az input értéket tartalmazó csúcsból alló, fát inicializál kezdetben. Az osztály támogatja új csúcsok és élek hozzáadását, piros (különleges jelentésű) élek kezelését, valamint két fa összeolvasztását (merge) a csúcsok és élek közti megfeleltetések alapján. Emellett lehetőséget biztosít a fa vágására (chop), valamint tartalmaz ellenőrző műveleteket, amelyek biztosítják, hogy a fekete (alapértelmezett) élek irányítottsága és kapcsolódása ne sértse a hierarchiát. A megvalósítás támogatja az élek többszörös előfordulásának (multiplicitásának) kezelését a fa fejlődése során.

##### GraphViewer

A GraphViewer osztály egy vizuális szimulációs felületet valósít meg, amely az ügynökök közötti kapcsolatok és azok időbeli alakulásának szemléltetésére szolgál. A felhasználó meghatározhatja, hány ügynök vegyen részt a szimulációban, valamint hányan kezdjenek egy adott kezdeti állapottal, azaz 0-ás bemenettel. Ezután a rendszer automatikusan előállítja a szükséges kezdeti struktúrákat és megnyit egy grafikus felületet, ahol a kapcsolatok és döntési fák jól látható módon jelennek meg.

A képernyő több részre van osztva, ahol egyszerre jelennek meg az aktuális és a korábbi ügynök-kapcsolatok, valamint ezek döntési történetei fákként ábrázolva. A felület görgethető elemeket is tartalmaz, így nagyobb vagy összetettebb struktúrák is áttekinthetően megjeleníthetők. A gráfok vizuális elrendezése automatikusan történik, az egyes elemek egymáshoz viszonyított helyzetét a belső logika határozza meg.

A felhasználó egyszerű gombokkal vezérelheti a szimulációt: új folyamat indítása, lépésenkénti haladás és kilépés is elérhető. Minden lépés után a megjelenített gráfok frissülnek, tükrözve a változásokat, például új kapcsolatok létrejöttét vagy döntési utak bővülését. A rendszer végigköveti, hogyan alakul a kommunikáció és az információáramlás az ügynökök között, miközben vizuálisan megmutatja a kapcsolati mintázatok és döntési hierarchiák fejlődését.

A szimuláció végén egy összegző ablak jelenik meg, amely tájékoztatást ad az elért eredményekről. Az egész alkalmazás célja, hogy interaktív és szemléletes módon segítse az ügynökalapú rendszerek viselkedésének megértését, különös tekintettel a kapcsolati hálózatok dinamikájára és az ebből kialakuló történeti struktúrákra.

##### Algorithm

A SimulationApp osztály felelős az algoritmus irányításáért. Létrehozza az ügynököket az előre megadott n db bemeneti érték alapján, majd minden körben egy új, összefüggő véletlen gráfot generál, amely az aktuális hálózati topológiát reprezentálja. A generált gráf esetében paraméterként megadható az élek generálásának valószínűsége. Az osztály gondoskodik a körök megfelelő elindításáról, meghívja az ügynökök main() metódusát, és addig indít újabb köröket, amíg nem teljesül a leállási feltétel. A szimuláció addig folytatódik, amíg minden ügynök el nem éri a stabil állapotot, a lépések eredményeit naplózza a későbbo feldolgozás céljából.

##### Main

A main rész feladata a grafikus megjelenítő elindítása, amely a GraphViewer osztály példányosításával történik meg.

#### Tesztek

Az egyes osztályokat az implementálás közben és után is igyekeztünk tesztelni, a helyes működést több különböző példára is megvizsgálni. Ilyenek voltak a cikkben szemléltetett, egyes műveleteket illusztráló példák, de emellett természetesen saját magunk által kitalált példa gráfokra is teszteltük a megvalósított függvények, algoritmust felépítő elemek működését.

Ennek megfelelően a legtöbb tesztesetet a history tree osztályban valósítottuk meg, ahol több különböző topológiájú és méretű gráfra is kipróbáltuk mind a merge, mind a chop műveleteket. Emellett még külön teszteket futtattunk ezen algoritmusok izoláltabb részeire, például a piros élek multiplicitásának megfelelő kezelésére, a gyökérig vezető utak megkeresésére, a fa magasságának meghatározására. Ezekben a tesztekben sokat segített a megfelelő vizualizációja is a fáknak, így az eredmények egyszerűen összehasonlíthatóak az elvárásainkkal.

Az agent osztály esetében a tesztjeink a leállási feltételre irányultak, vagyis a számlálási szint megfelelő meghatározását és az input frequency függvény kiszámítását próbáltuk ki előre generált gráfokon, és hasonlítottuk össze az eredményt a gráf alapján vett saját számításainkkal.

A külön egységek tesztelése után következhetett az integráció, ahol már a teljes algoritmust teszteltük. Ahhoz, hogy megvizsgálhassuk az eredményeket, és a futási időt (pontosabban a leállásig eltelt köröket) összehasonlíthassuk a cikkben bizonyított elmélettel, majd következtetéseket vonhassunk le, több szimulációt is végeztünk. Ezeket a következő bekezdésben mutatjuk be.

#### Szimulációk

Ahogyan azt fentebb is részleteztük, a végső cél ugyanaz: a bemeneti értékek gyakoriságát meghatározni. Ahhoz, hogy konkrét szimulációkat végezhessünk egy adott környezetben, fontos meghatározni a lehetséges bemeneti értékek halmazát.

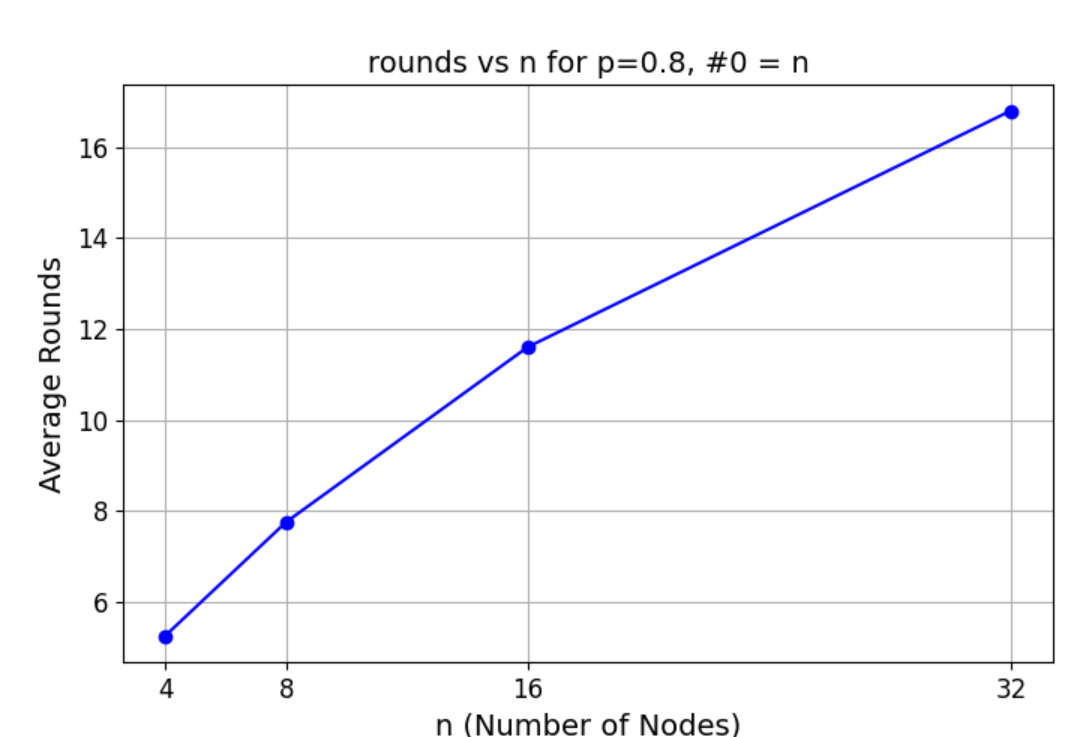
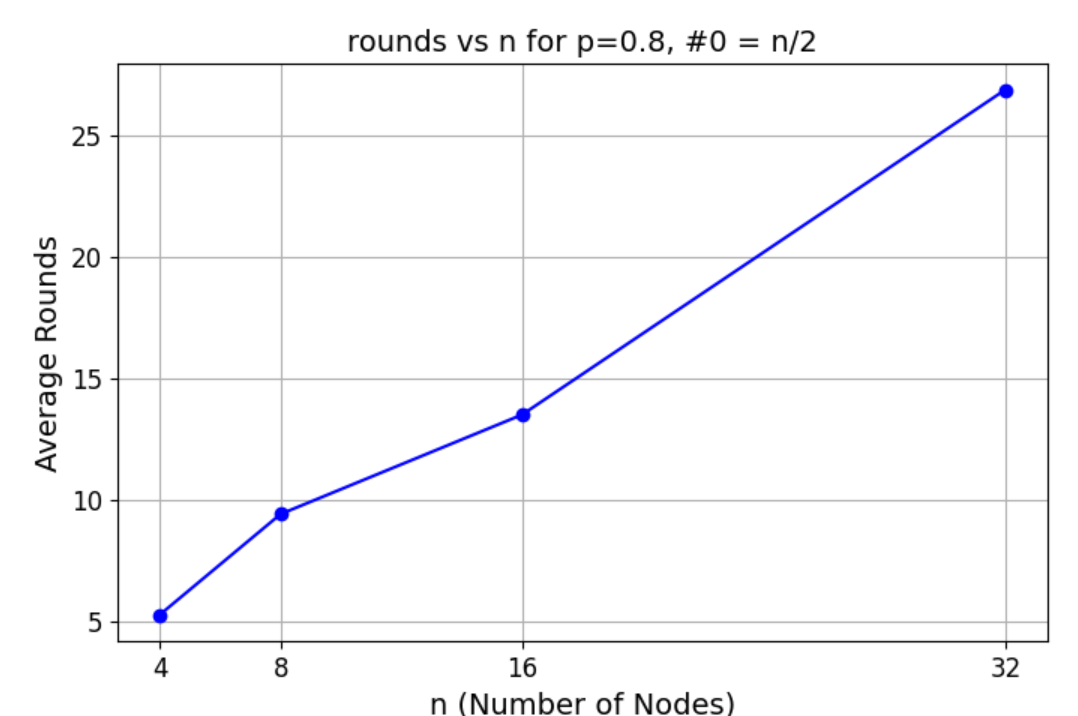
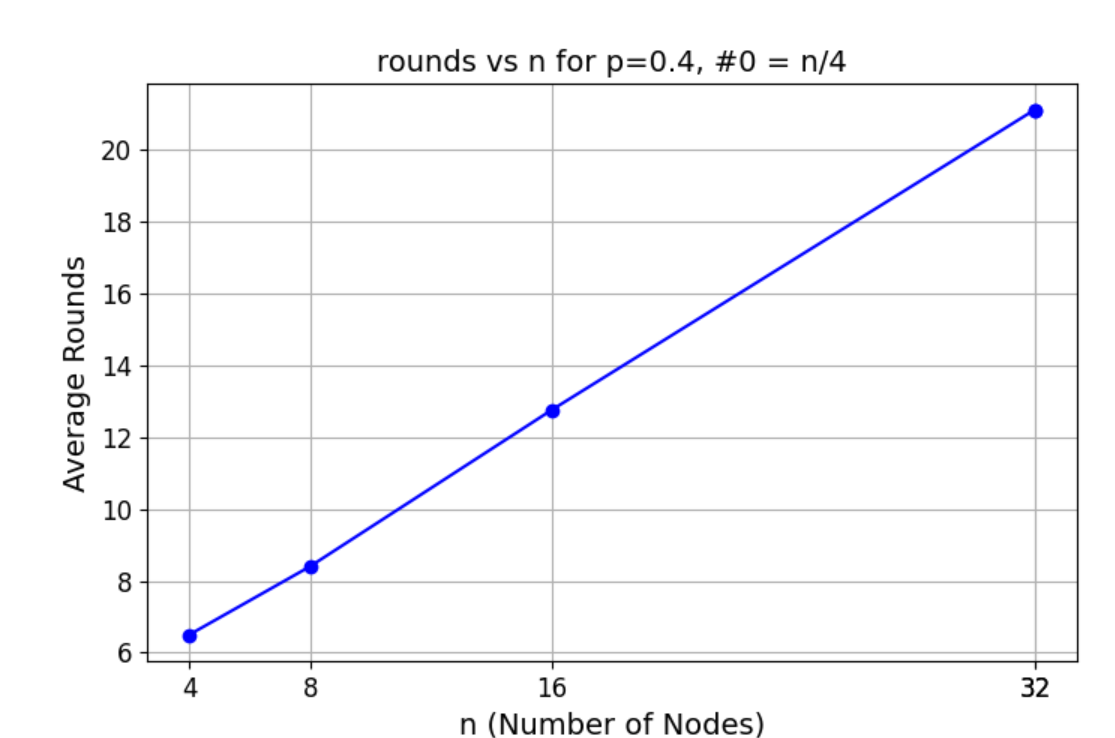
Bár az általunk megvalósított algoritmus képes lenne kettőnél több input értéket is kezelni az ágensek esetén, annak érdekében, hogy átfogóbban tesztelhessük magát a működést a rendelkezésre álló idő alatt, a tesztjeinket és szimulációinkat csak kétféle input értékkel végeztük el. Érdemesnek tartottuk egy jól definiált kontextus meghatározását, amelyben a méréseket elvégezzük, így arra az esetre fókuszáltunk, amikor minden ágens 0 vagy 1 értéket kap meg inputként. Innentől kezdve csak ezeket az eseteket mutatjuk be.

A szimulációk során az alábbi paraméterek értékeivel játszottunk:

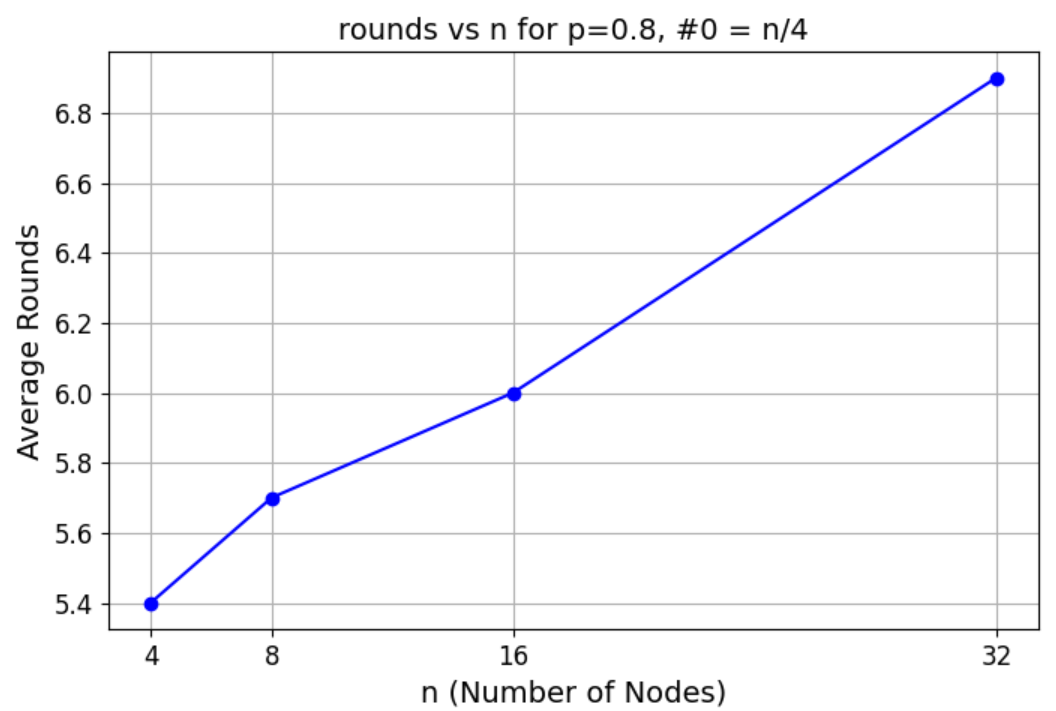
* n: az ágensek száma
* #0: a 0 bemeneti értékek száma (értelemszerűen az 1-es bemeneti értékek száma: #1 = n – #0)
* p: az egyes körök elején generált gráfban az élek behúzásának valószínűsége
* h: a hibás inputok, vagy egyéb hibák száma az egyes ágensek esetében

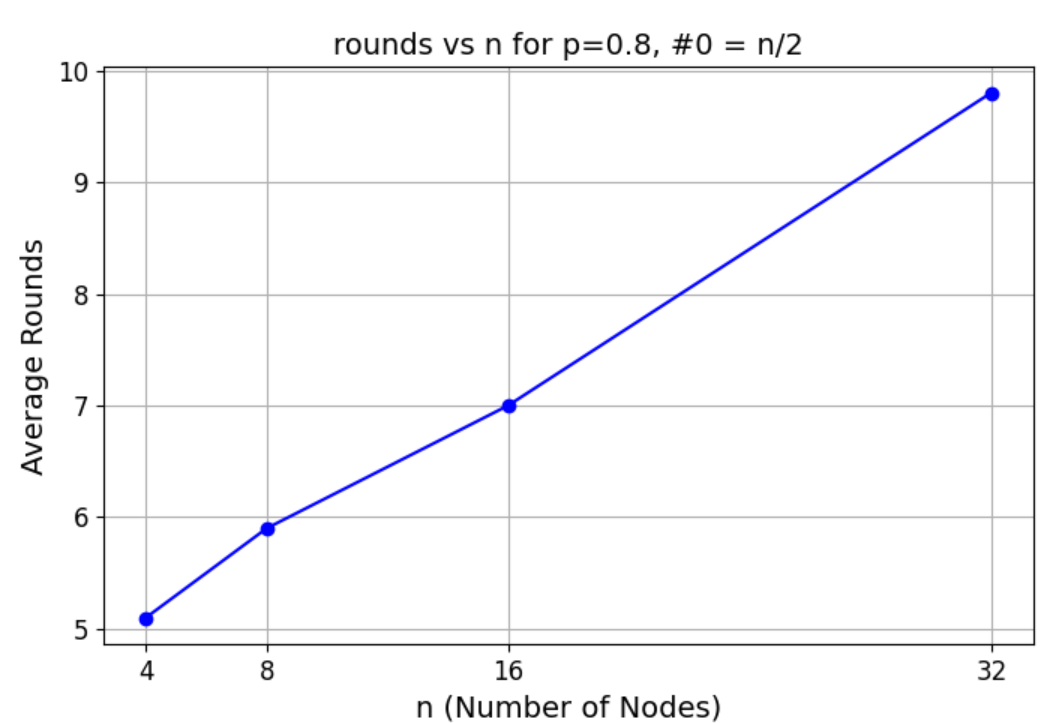
A fenti paraméterek függvényében azt viszgáltuk meg, hogy hány kör alatt állt le az algoritmus. Ezek alapján a következő eredményeket kaptuk:

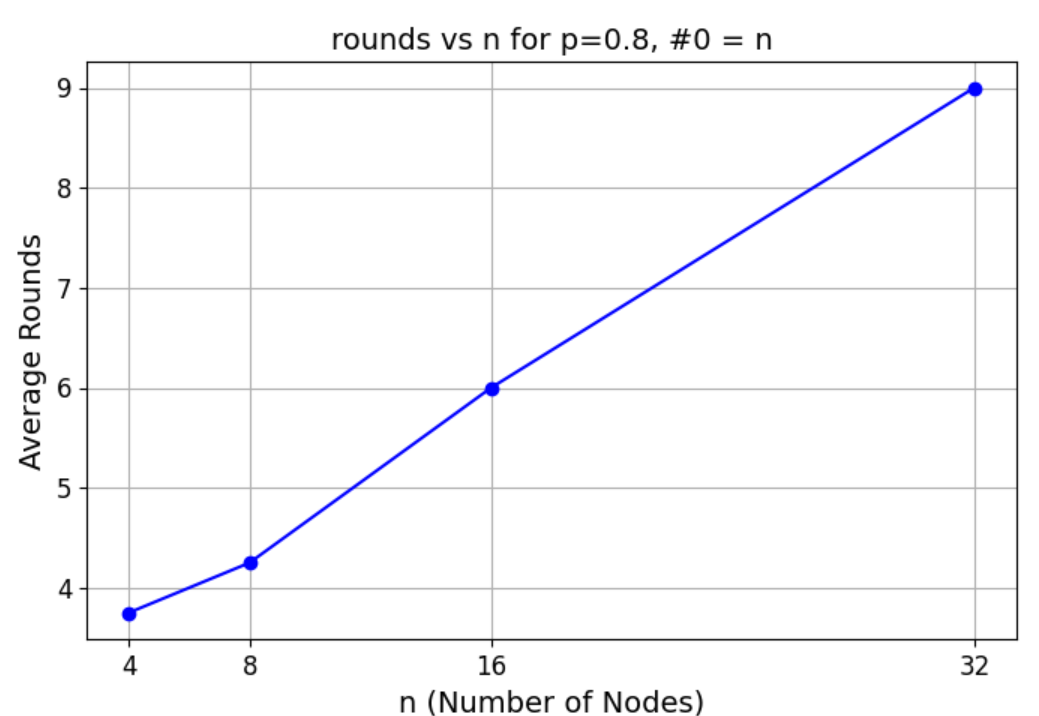
Az alábbi három képen látható, hogy p = 0.4-es valószínűség esetén megnéztük először azt az esetet, amikor a #0 az n értékének legfeljebb negyede, majd amikor az n értékének fele, végül pedig amikor ugyanannyi 0 van, mint n (vagyis mindenhol ugyanaz a bemenet):



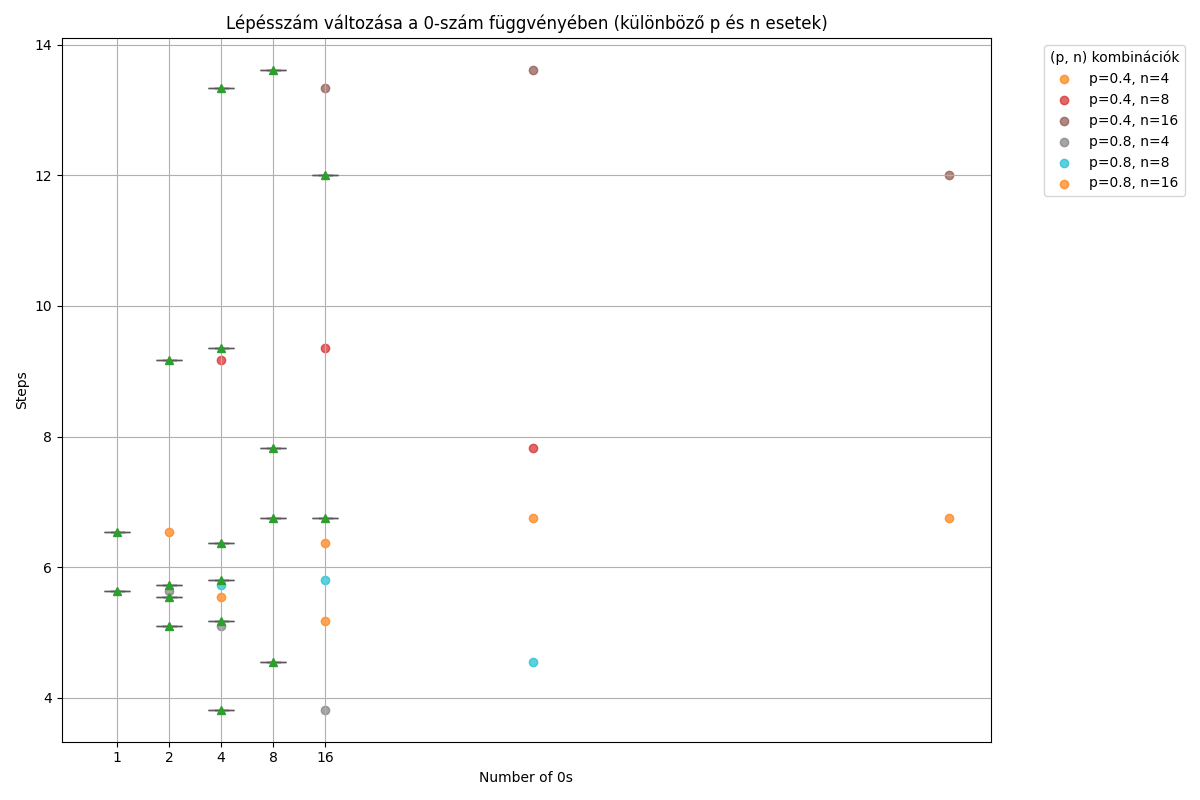
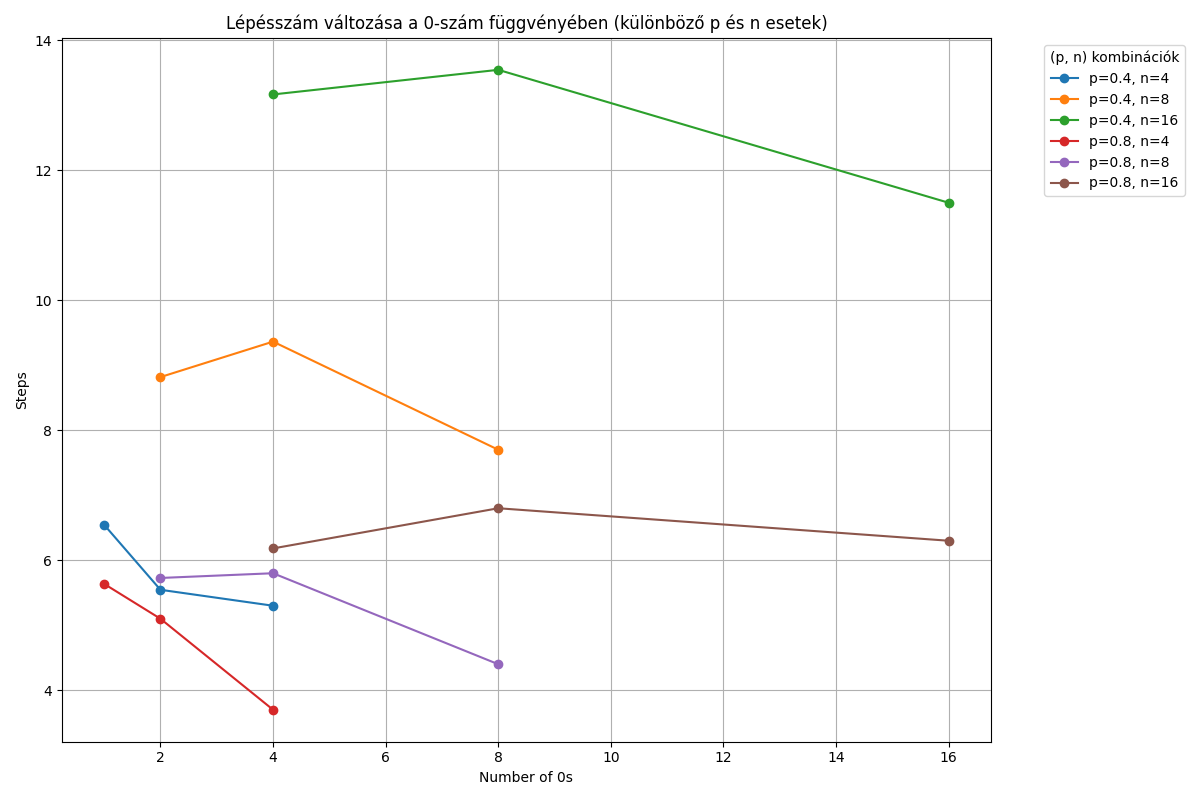
A követkető három képen látható, p = 0.8-as valószínűség esetén először az az esetet, amikor a #0 az n értékének legfeljebb negyede, majd amikor az n értékének fele, végül pedig amikor ugyanannyi 0 van, mint n (vagyis mindenhol ugyanaz a bemenet):





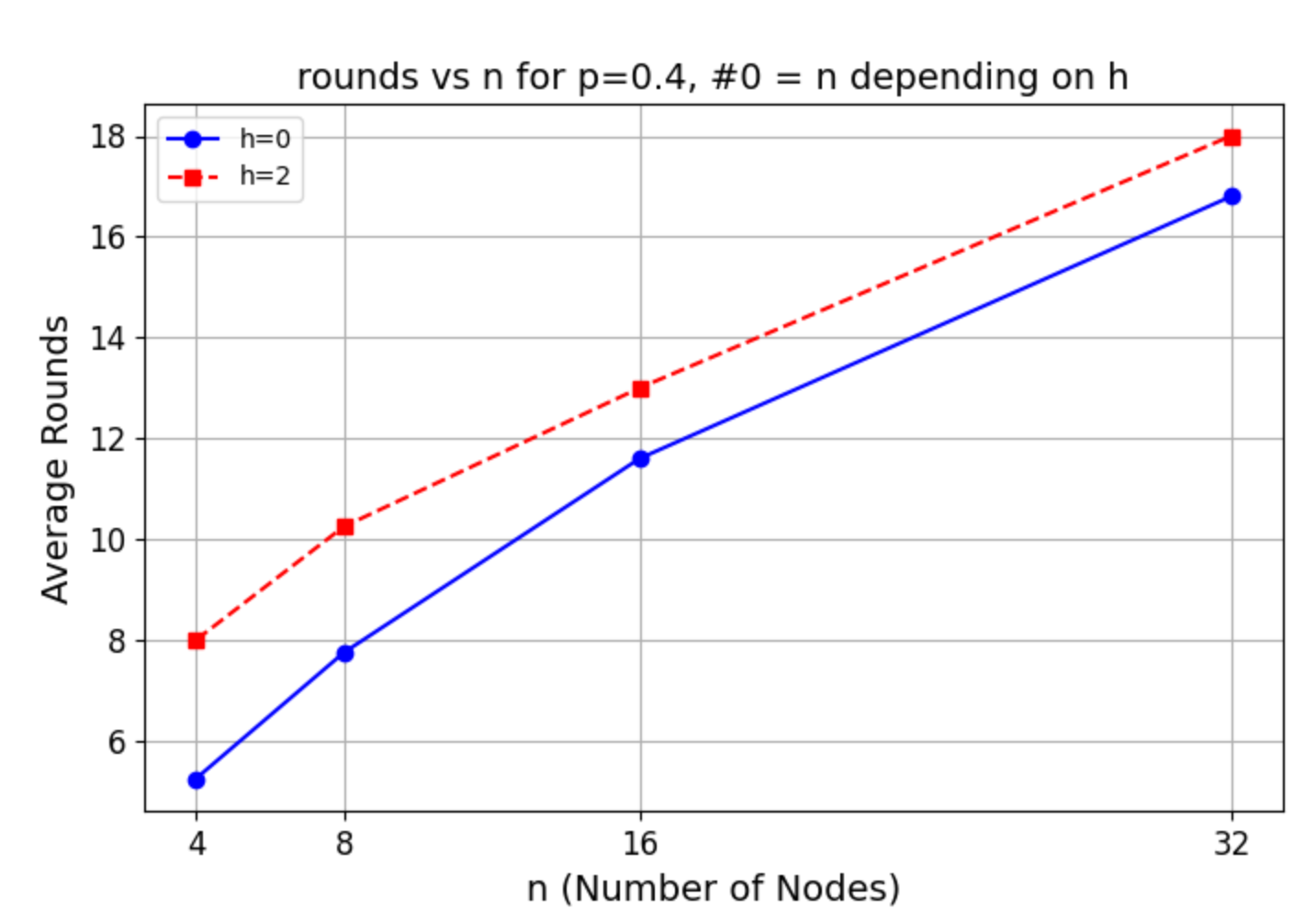


Ezek az ábrák segítenek szemléltetni, hogyan változik az eloszlás különböző arányú 0-k és nem-0 értékek mellett:

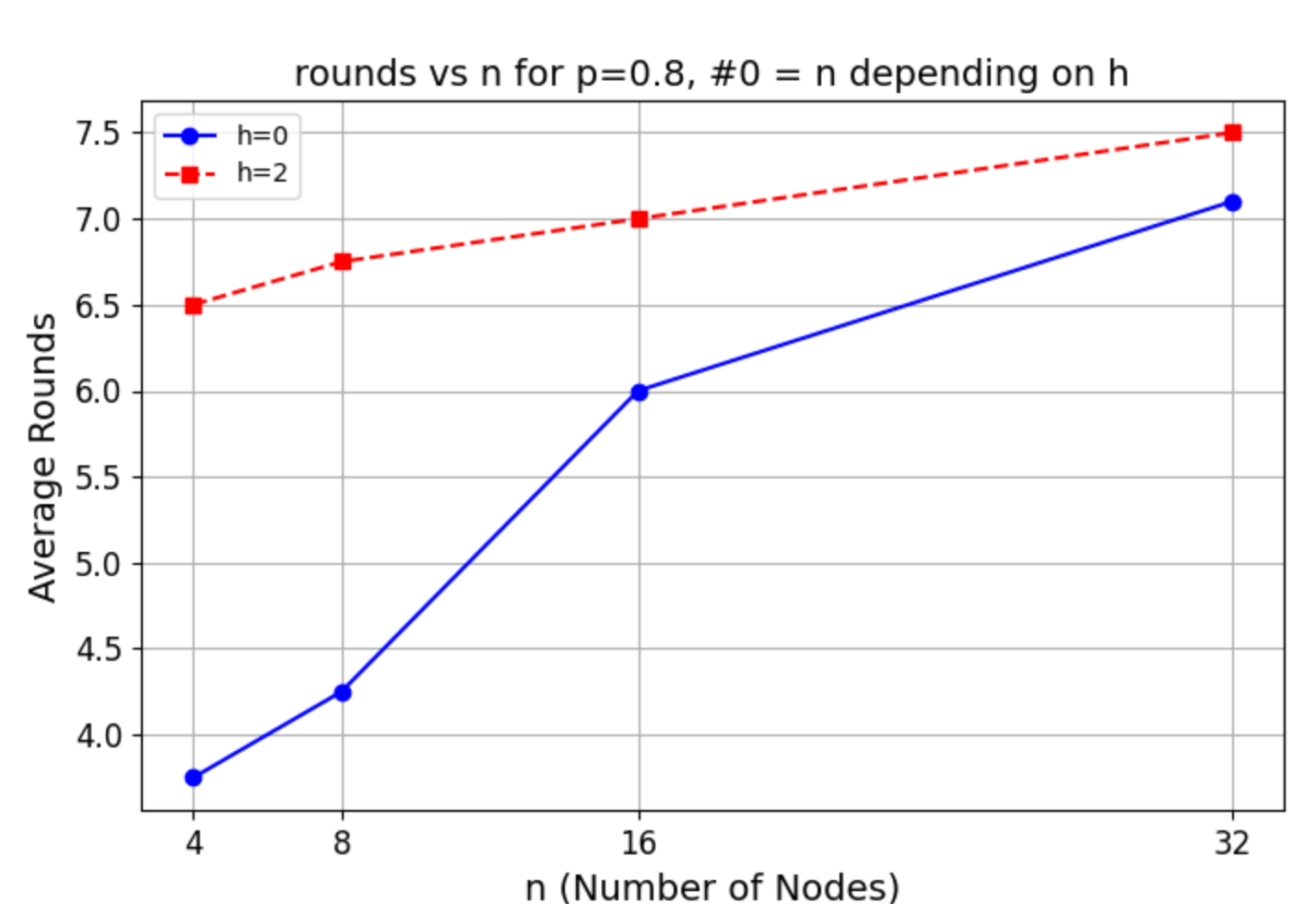


Végül azt is teszteltük, hogy a hibák száma hogyan befolyásolja az algoritmust. Ezért mesterségesen generáltunk adott számú hibákat (vagyis h értékeket). Ez azt jelentette, hogy h db ágensnek elrontottuk az input értékét, és megfigyeltük, hogy az így hibássá vált üzenetekkel együtt hány kör alatt kerül stabil állapotba az algoritmus.

Ezt ábrázolja az alábbi diagram, ahol p = 0.4, a 0-k száma egyenlő n-el, és a kék vonal a hibák nélküli átlagos leállási időt (körök számát) jelképezi, a piros vonal pedig azokat a leállási időket, amikor időközben két agent hibás értékeket küldött.



Ugyanez látható az alábbi képen, annyi különbséggel, hogy p=0.8.



Mindezek alapján azt állapítottuk meg, hogy a szükséges körök száma egyértelműen megnő, azonban továbbra is lináris tendenciát követ n-hez képest, és belefér abba a futásidőbe, amit az algoritmustól elvártunk.

A kisebb gráfok esetén észrevehető egy nagyobb kiugrás a hibamentes futási időhöz képest, ez értelemszerűen amiatt van, mert kisebb n-hez viszonyítva a h=2 is nagy hibaaránynak számít, azonban ahogy n növekszik, ez egyre elenyészőbb. Ez látható a diagramokon is.

Elvégeztük azt az összehasonlítást is, hogy hogyan befolyásolja az algoritmus lefutási idejét az ágensek száma, és arra jutottunk, hogy minél több ágens vesz részt a rendszerben, annál lassabban áll le az algoritmus. Ennek oka, hogy több ágens esetén megnő az üzenetváltások száma, valamint több különböző történelmi fát kell összehangolni, ami lassítja a konvergenciát. A piros élek – amelyek a konszenzus kialakulását segítik – lassabban jelennek meg és mélyülnek, így az algoritmus csak később tudja meghatározni a bemenetek gyakoriságát minden csomópontban. Ugyanakkor, azt konstatáltuk, hogy az algoritmus gyorsabban áll le, ha kevesebb a hiba a rendszerben. Ennek oka, hogy a hiba jelenléte azt eredményezi, hogy a csomópontoknak több időre van szükségük ahhoz, hogy megszűrjék a hibás adatokat a történelmi fákból, és kialakuljon egy közös, konzisztens számlálási szint.

#### Összefoglalás

A bemutatott algoritmus egy univerzálisan ön-stabilizáló megközelítést alkalmaz, amelyben minden csomópont saját történelmi fát épít a korábbi interakciók alapján. Ezek a fák piros élek segítségével kapcsolódnak össze, amelyek az ügynökök közötti információcserét és összehasonlítást reprezentálják. Az algoritmus lényege, hogy az ismételt fák összeolvasztása (merge) során a hibás vagy nem egyező adatok automatikusan kiesnek, így egyre mélyebbre nyúló, megbízható struktúra alakul ki. A folyamat során kialakul egy úgynevezett számlálási szint, amely alatt az adatok már tiszták és stabilak – ezek alapján számolja ki minden csomópont a bemenetek gyakoriságát. A rendszer célja, hogy tetszőleges kezdeti állapotból indulva – akár hibás, inkonzisztens vagy hiányos adatok mellett – minden korrekt (nem hibás) csomópont végül ugyanazt a helyes gyakorisági eloszlást számolja ki.