

TDK-dolgozat

Czuprák Zsolt

Czuprák Zsolt
Mérnök informatikus szak
2. évfolyam

**Raj-intelligencia feladat megoldása Microsoft
Robotics Studioval**

Témavezetők:
Gurka Dezsőné Csizmás Edit
műszaki tanár
Dr. Kovács Tamás
főiskolai tanár

Kecskeméti Főiskola
Gépipari és Automatizálási Műszaki Főiskolai Kar
Informatika Tanszék
Kecskemét
2011

Tartalomjegyzék

1. A FELADAT ISMERTETÉSE	4
2. ELMÉLETI HÁTTÉR	5
2.1 A Mesterséges intelligencia	5
2.1.1 Hagyományos MI	5
2.1.2 Számítási intelligencia	6
2.2 A raj-intelligencia	6
2.2.1 Szétszéledés	7
2.2.2 Begyűjtés	7
2.2.3 Összegyűlés	7
3. AZ ÖSSZEGYŰLÉSI ALGORITMUSSEL FOGLALKOZÓ KUTATÁSI EREDMÉNYEK	8
3.1 A kiindulási algoritmus részletezése	10
3.2 Alkalmazás kiterjedéssel bíró, valós robotokra	11
4. FEJLESZTŐ KÖRNYEZET BEMUTATÁSA, SZIMULÁCIÓS KÖRNYEZET LÉTREHOZÁSA	13
4.3 Szimulációs környezet létrehozása	16
5. Kezdeti nehézségek.....	16
5.1 Robotok irányítása	16
5.2 Dashboard korlátai.....	17
6. MEGVALÓSÍTÁSOK	18
6.1 Robotok kezelése.....	18
6.2 Program szerkezete	19
7. A KIINDULÁSI ALGORITMUS MÓDOSÍTÁSA	20
8. A ROBOT SZIMULÁCIÓS ALKALMAZÁS ÉS AZ ALGORITMUSOK TESZTELÉSE-EREDMÉNYEK.....	21
IRODALOMJEGYZÉK	30

1. A feladat ismertetése

Ennek a dolgozatnak a kitűzött célja egy olyan 3D-s szimulációs alkalmazás kifejlesztése, amely alkalmas mobil robotok raj-intelligencia algoritmusainak vizsgálatára és továbbfejlesztésére. Fontos, hogy az egyedek (robotok) valóságszerűen tudják elvégezni a megfelelő feladatokat. A valóságszerű a mi esetünkben azt jelenti, hogy legyen az egyedeknek fizikai kiterjedésük, valós tömeggel rendelkezzenek és a fizika törvényei érvényesek legyenek a szimulációs környezetben. Ebben a valós szimulált környezetben mi egy alapvető raj-intelligencia algoritmus megvalósítását, vizsgálatát és továbbfejlesztését végeztük el. Ez az algoritmus a robotok egy helyre való összegyűjtését oldja meg. Ehhez a Microsoft Visual Studio 2010 fejlesztőkörnyezetet és a Microsoft Robotics Developer Studio-t használjuk. A Microsoft Robotics Developer Studio (RDS) a Microsoft Visual Studio egy kiegészítő csomagja, amit kifejezetten robot szimulációk és robotalkalmazások elkészítésére fejlesztettek ki.

A robot szimulációban alkalmazott algoritmus egyedként Pioneer 3-DX típusú robotot használtam, a robotra szerelhető lézeres távolságérzékelő miatt. A robot szimulációt C# programnyelven készítettem el. Munkámat a C# programnyelv megismerésével, az alapok elsajátításával kezdtem.

A megvalósítás első részében létrehoztuk a szimulációs környezetet, a benne valóságszerűen működő és mozgó robotokkal. Második lépésben egy szakirodalomban publikált összegyűlési algoritmust alkalmazva egyrészt teszteltük a kész szimulációs környezetünket, másrészt teszteltük az összegyűlési algoritmust valóság-hű szimulációs környezetben. Ennek tapasztalatai alapján továbbfejlesztettük az algoritmust úgy, hogy valós környezetben is működhessen.

2. Elméleti háttér

2.1 A Mesterséges intelligencia

Mesterséges intelligenciának (MI vagy AI – az angol Artificial Intelligence-ből) egy gép, program vagy mesterségesen létrehozott tudat által megnyilvánuló intelligenciát nevezzük.[9]

Több jelentésben is használják:

- A mesterséges intelligenciával táplált eszköz, emberi beavatkozás nélkül képes legyen válaszolni a környezeti változásokra (automatizáltság).

- A mesterséges intelligenciával táplált eszköz, képes legyen hasonlóan viselkedni, mint egy természetes intelligenciával rendelkező élőlény, még ha az azonos viselkedés mögött eltérő mechanizmus is húzódik meg (TI szimuláltság).

- A mesterséges intelligenciával táplált eszköz képes legyen viselkedését célszerűen és megismételhető módon változtatni (tanulás) - ez utóbbi jelentés az, ami a modern MI- kutatásban előtérbe került, és jelenleg az MI fogalmával legjobban azonosítható.

A mesterséges intelligencia-kutatást két fő területre oszthatjuk: a hagyományos MI- ra és a számítási intelligenciára.

2.1.1 Hagyományos MI

A hagyományos MI főleg a jelenlegi gépi tanulásként osztályozott módszerekből áll, amelyet a formalizmus és a statisztikai analízis jellemez. A terület a következő módszereket foglalja magába:

- Szakértő rendszerek, amelyek egy szűk szakmai területen érvényes érvelési szabályokat alkalmaznak következtetések levonásához. A működése során tényekkel, és további tények kikövetkeztetéséhez alkalmas szabályokkal operál. Az egyik legismertebb az 1970-es években létrehozott MYCIN, ami a vér betegségeinek diagnózisához adott segítséget.

- Esetalapú érvelés

- Bayes - statisztikán alapuló hálózatok

- Viselkedés alapú MI: egy moduláris módszer, MI- rendszerek kézi létrehozásához.

2.1.2 Számítási intelligencia

A számítási intelligencia az iterációs (lépésenkénti) fejlődést vagy tanulást helyezi előtérbe. A tanulási folyamat gyakorlati tapasztalatokon alapul és nem szimbolikus, koszos MI vagy puha számítási technikai módszereket használ.

- Neuronhálózatok: nagyon erős minta-felismerési képességű rendszerek.
- Fuzzy rendszerek: technikák a kockázat melletti érveléshez, amelyeket elterjedten használnak modern ipari és fogyasztói szabályozási rendszerekben.

- Evolúciós számítási technikák, amely a biológia által inspirált fogalmakat (például populáció, mutáció, a legjobb túlélése) alkalmaz egyes problémák egyre jobb megoldásához. Ezek a módszerek az evolúciós algoritmusok (például genetikai algoritmusok) és a raj-intelligencia (például hangya algoritmus) területekre oszthatók.

2.2 A raj-intelligencia

A raj-intelligencia (swarm intelligence) viszonylag fiatal tudományterület, magát a kifejezést először Gerardo Beni és Jing Wang használta 1989-ben, összefüggésben a mobil robot rendszerekkel. Az önszervező és decentralizált rendszerekben történő együttes viselkedéssel foglalkozó kutatási terület a raj-intelligencia. Ezek a rendszerek a rendszer egész viselkedését irányító vezérlők helyett, alapvetően sok természetes, egyedül primitív egyeden alapszik, amelyek a kívánt viselkedés érdekében együttműködnek. Ilyen



rendszerekkel találkozhatunk a természetben is a társas rovarok, például hangyák, méhek, termeszek, darazsak esetében, valamint egyéb állatközösségek, például madár- és halrajok esetében. A társas rovarok kolóniái már régóta foglalkoztatják a kutatókat, mégis évek óta ismeretlenek azok az alapelvek, melyek a viselkedésüket irányítják.

Az ilyen társas rendszerekre, ahol az egyedek nem túl kifinomultak jó példa a méhek, és a hangyák rendszere, együttműködve képesek egy remek fészket építeni, anélkül, hogy bármelyiküknek is lenne egy teljes elkészítési terve.

A rajintelligencia algoritmusok között a három alapvető a szétszéledés, a begyűjtés és az összegyűjtés.

2.2.1 Szétszéledés

A szétszéledés, amikor egy pontból indulnak ki az egyedek feltérképezni környezetüket. Erre jó példa a társas rajokban élő rovaroknál az élelemkeresés, ők a fészekből indulnak el feltérképezni környezetüket. Ez a feltérképezés folyamatos, még akkor is tart amikor már egyes egyedek találtak élelmet. Mivel nincs központi vezérlés a rendszerben, ezért minden egyed bejárja ugyanazt a területet legalább egyszer. A legfontosabb, hogy a feltérképezés során bárholnan visszataláljanak az egyedek a fészekbe.

2.2.2 Begyűjtés

Feltérképezés közben minden egyed bejárja véletlenszerűen ugyanazt a területet, ha az egyik egyed élelmet talál, akkor az ő állapota átvált begyűjtésre. Tipikus példa erre a hangyák begyűjtési rendszere. Az egyed útban visszafelé a fészekhez találkozik társaival, akik látván, hogy talált élelmet elkezdik követni őt. (A méheknél egy úgynevezett riszáló táncot jár a fészekben az élelemmel az egyed. A hangyák a szállító csatornán feromont helyeznek el.) Miután már többen is ugyanazon a helyen lévő élelemhalmazt gyűjtik be, azok az egyedek, akik még nem találtak semmit felfigyelnek a tömeges vonulásokra és elkezdnek ők is a tömeg felé tartani. Így egész hatékonyan, és gyorsan működik az élelem begyűjtés. Előfordulhat, hogy több élelmiszerforrást találnak, és így több szállító csatorna keletkezik.

2.2.3 Összegyűlés

Amikor a szétszéledt egyedeink végeztek a feltérképezéssel, vagy egyéb okból kifolyólag vissza kell térniük a fészekbe, vagy csak be akarjuk gyűjteni őket, akkor összegyűlésről beszélünk. A természetben ez az összegyűlés történhet feromon segítségével, vagy a környezet megjegyzésével, esetleg az útra való visszaemlékezéssel.

A mi esetünkben a szétszéledt robotokat egy pontba gyűjtjük össze úgy, hogy nincs központi navigációs rendszer, csak a robot érzékelői által „látottakra” támaszkodunk. Egy olyan algoritmus segítségével történhet mindez, amelyet ha minden egyeden lefuttatunk, akkor az egyedek egy csoportba rendeződnek össze, azaz összegyűlnek.

3. Az összegyűlési algoritmussal foglalkozó kutatási eredmények

Tegyük fel, hogy egy nagy csoport katona elszórtan helyezkedik el egy ködös csatatéren, ahol a látótávolság korlátozódik, mondjuk csak 20 méterre. Például egy katona lát 3 másik katonát, de talán szem elől veszítheti őket, ha elmozdul egy kicsit eredeti helyéről. Lehetséges ilyen körülmények között a katonákat csendben egy helyre gyűjteni?

A választ és az ehhez kapcsolódó kérdéseket, egy formális modell irányította autonóm mobil robotokkal, és korlátozott látási viszonyokkal adják meg. Az utóbbi években jelentősen megnőtt az érdeklődés az elosztott autonóm robot rendszerek iránt. Az egyik fő kérdés az ilyen irányú munkákban a kooperatív viselkedés és az autonóm robotok mellett működő elosztott kontroll tanulmányozása. Az elosztott szabályozás azt jelenti, hogy minden robot teszi a saját dolgát, az adott információk és az adott algoritmus alapján és feltételezzük, hogy nincs globális navigációs rendszer.

Ando és munkatársai [1] kutatását vettük kiindulási algoritmusnak a dolgozatban. Ez egy elméleti algoritmus, amiben minden robot pontszerű. A robotok különálló processzorral rendelkeznek, minden adott időpillanatban megfigyelik saját relatív helyzetüket és az algoritmus szerint saját maguknak számolják ki az új pozíciókat. Számolás után elmozdulnak a kiszámolt pozíciókba. A robotok a kezdeti eloszlásban közös x-y koordináta rendszerben vannak. Minden robot teljes lépésciklusa három fázisból állt:

1. Körülnézés
2. Számolás és a következő cél meghatározása
3. Lépés végrehajtása

A robotok nem rendelkeznek memóriával, ami azt jelenti, hogy mindig csak a pillanatnyi felmért állapot alapján hozzák meg a döntésüket. *Ando* algoritmusában a robotok szinkronvezérlést használtak a megvalósításhoz. Ez azt jelenti, hogy a három lépésfázist mindig egyszerre hajtotta végre az összes robot amit egy központi szinkronjel segített. Az ilyen algoritmust szinkron algoritmusnak nevezzük.

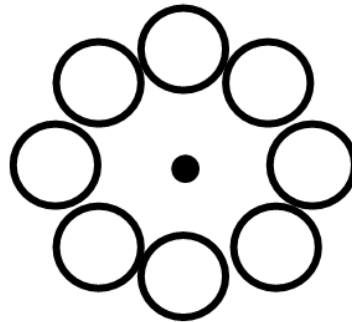
Nemrég *Flocchini* és munkatársai [2] megoldották a fenti problémát pontszerű robotokra de aszinkron algoritmussal, azaz itt nem kellett feltételezni a központi szinkronizálást. Ennek az volt az ára, hogy a robotoknak egy iránytűvel is kellett rendelkezniük, azaz a navigáció nem volt teljesen lokális.

Később *Degener* és munkatársai [3] kidolgoztak egy félig szinkron algoritmust pontszerű robotok összegyűlésére, amelyben lokális volt ugyan a navigáció, de a robotok

képesek voltak egymással fejlett kommunikációt folytatni, azaz egymás számára koordinátákat és célokat átadni.

Czyzowicz és munkatársai [4] vizsgálták a fentebb megfogalmazott problémát fizikai kiterjedéssel bíró robotok esetén. A feladatot meg is oldották kommunikáció nélkül aszinkron módszerrel, de legfeljebb négy robotra.

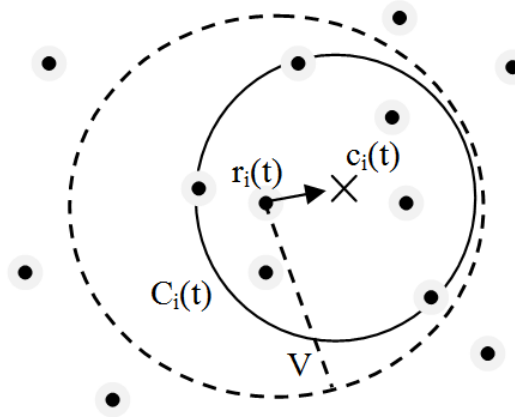
Chaudhuri és Mukhopadhyaya [5] az előzőekben leírtakhoz hasonlóan vizsgálták az azonos robotok mozgását determinisztikusan a síkon. Itt a robotok szintén nem rendelkeznek memóriával, teljes aszinkron módon mozognak és fizikai kiterjedéssel bírnak. Feltételezi, hogy a robotok teljesen átlátszóak másrészt belátják az egész síkot. Továbbá tökéletes globális navigációval rendelkezik minden robot. Így az összegyűlés viszonylag egyszerűen megoldható a robotok kiterjedése okozza. A kidolgozott algoritmusban legalább 5 robotra oldották meg az összegyűlés problémáját.



3.1 ábra: Gyűrű formába összegyűlt robotok

Cord-Landwehr és munkatársai [6] kutatásában összegyűjtenek N darab mobil robotot egyetlen pont köré, olyan közel egymáshoz amennyire csak lehetséges, azaz a robotok térkitöltése optimális az összegyűlt állapotban. A robotok kiterjedéssel bírnak, nem kommunikálnak egymással és ütközés vezéreltek valamint globális navigációval (koordináta rendszer) vannak felszerelve. A robotok szinkron módon egy központi pont felé haladnak, amennyiben nem blokkolják egymást. Ebben az esetben a robotok alkotnak egy gyűrűt (3.1 ábra) az előre meghatározott pont körül.

3.1 A kiindulási algoritmus részletezése



3.2 ábra: Egy robot-lépés kiszámítása a körülírt kör alapján

A kiindulási algoritmusunk tehát az *Ando* és munkatársai által publikált algoritmus. Ez egy elméleti algoritmus, pontszerű egyedekkel. Mivel az egyedeknek nincs kiterjedésük, ezért összegyűjthetők egy pontba. A robot két fontos szempontot tart szem előtt minden egyes lépésciklusban. (Egy lépésciklus idejét egységnyi időnek vesszük.) Az egyik szempont az, hogy a robot a t időponthoz képest a $t+1$ időpontban közelebb legyen a többi robothoz. A másik szempont, hogy amely robottársa a robot látótávolságán belül van t időpontban (V -jelű szaggatott kör a 3.2 ábrán), az $t+1$ időpontban is látható legyen számára. Először is adott t időpontban, ha a robot (r_i) nem lát más robotot saját magán kívül, akkor nem mozdul t időpontban. Ez azt jelenti, hogy nem lesz részese az összegyűlési folyamatnak, azaz az összegyűlés nem megoldható. Az összegyűlés megoldhatóságához az szükséges, hogy a robotok láthatósági gráfja összefüggő legyen. A láthatósági gráfban a pontok a robotok, és két pont akkor van összekötve, ha a két robot látja egymást. Mostantól feltételezzük, hogy a robotjaink láthatósági gráfja összefüggő, azaz a feladat megoldható. A fent említett első szempont azt jelenti, hogy a robotok törekszenek a láthatósági gráf térbeli zsugorítására. A második szempont pedig azt jelenti, hogy nem engedik, hogy a láthatósági gráf két külön gráfra szakadjon szét.

Tehát feltételezzük, hogy egy adott robot (i) a t . lépésben lát maga körül néhány robottársat. Ekkor a robot meghatározza azt a legkisebb kört, amely tartalmazza az összes látható társát és saját magát. Ez a kör $C_i(t)$ -vel van jelölve az ábrán, melynek középpontja $c_i(t)$. E felé a pont felé indul a robot. Az ilyen jellegű lépések garantálják a gráf zsugorodását minden lépésben.

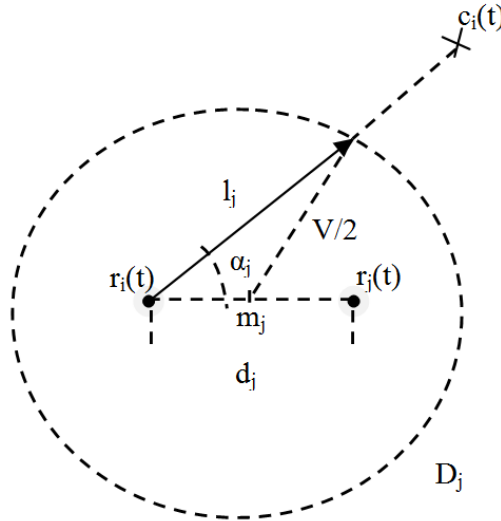
Érvényesítenünk kell azonban a második szempontot is, azaz a láthatósági gráfban egyetlen él sem szakadhat meg. Ezt az adott robot minden egyes látható robottársánál külön megvizsgálja. Vegyük az i . és j . robotot, amelyek látják egymást a t időpontban. Ha azt akarjuk, hogy ez a két robot a $t+1$ időpontban is lássa egymást, akkor az i . robot a lépése során nem hagyhatja el a 3.3 ábrán D_j –vel jelölt szaggatott kört. Ennek a körnek a középpontja a két robot közötti felezőpont, sugara pedig $V/2$, azaz a láthatósági határ fele. Ez a feltétel igaz a j . robot lépésénél is. Ilyen feltételek mellett a láthatósági gráf élei biztosan nem szakadnak meg, ez elemi geometriai eszközökkel bebizonyítható. A bizonyítás részletei a [1] számú irodalomban megtalálható.

Tehát a robot legfeljebb l_j távolságot mehet az eredeti célja felé (lásd ábra). Az l_j távolságot az alábbi képlettel számolhatjuk ki:

$$l_j = (d_j/2)\cos \alpha_j + \sqrt{(V/2)^2 - ((d_j/2)^2(\sin \alpha_j)^2)} \quad (3.1)$$

Ezt minden egyes j . társra ki kell számítani és kiválasztani a legkisebbet. Ez lesz a legszigorúbb korlát a lépés során. Tehát a lépés hossza

$$\text{Lépéshossz} = \min \{ \text{Eredeti távolság}, \min(l_j) \} \quad (3.2)$$



3.3 ábra: Egy adott robot lépéshosszát korlátozó feltétel

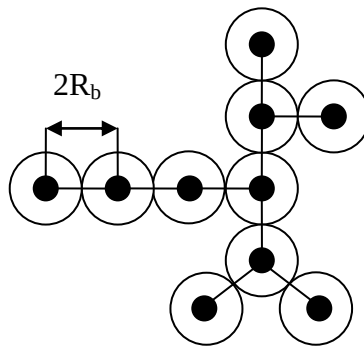
3.2 Alkalmazás kiterjedéssel bíró, valós robotokra

A kiindulási algoritmusban szereplő egyedek pontszerűek, mi ezt továbbfejlesztettük valódi kiterjedéssel bíró (R sugarú) egyedekre. Annak érdekében, hogy a robotok ne

ütközzenek, és ne lökjék fel egymást, be kell vezetni egy úgynevezett ütközésvédelmet. Azaz nem engedhetjük egymáshoz közel őket, és bevezetjük a biztonsági távolság (R_b) fogalmát. Ez azt jelenti, hogy minden esetben a

$$\text{Robot-Robot távolság} > R_b \quad (3.3)$$

feltételnek teljesülnie kell. Ennek az a következménye, hogy nem megoldható az egy pontba összegyűlés. Akkor mit jelent az egy helyre való összegyűlés ebben az esetben? Minimum követelményként tűztük ki, hogy a biztonsági zónájukkal érintkező robotok gráfja legyen teljes és összefüggő. Ez azt jelenti, hogy ha összekötjük azokat a robotokat, amelyek egymástól $2R_b$ távolságra vannak, akkor egy olyan gráfot kapunk, amely tartalmazza az összes robotot és összefüggő. (3.4 ábra)

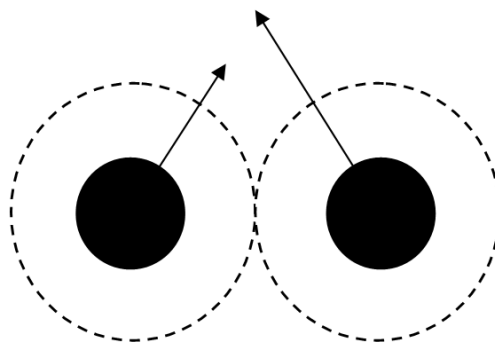


3.4 ábra: Egy összefüggő teljes gráf

A minimum követelményen felül azt is kitűztük célként, hogy ez a gráf minél tömörebb legyen, azaz a robotok ütközési zónái minél jobban töltsék ki a gráf által elfoglalt területet. Egy eredményt akkor nevezünk jobbnak, ha a térkitöltése jobb.

S. G. Chaudhuri és K. Mukhopadhyaya 2010-ben [5], továbbá A.C.-Landwehr és munkatársai 2011-ben [6] részletesen foglalkoztak a robot-gráf térkitöltésének problémájával. Az általuk javasolt algoritmusok tökéletes térkitöltést eredményeznek, de a robotoknak globális navigációval kell rendelkezniük a megvalósításhoz, ezért az ő módszerüket a jelen munkában nem alkalmazhatjuk, mivel mi globális navigáció nélkül kívánjuk megoldani a feladatot.

A másik következménye a robotok kiterjedésének a holtpont (Deadlock) problémája. A kiszámolt célhoz tartás közben a robotok keresztezhetik egymás útját (3.5 ábra), így ütközés veszélynek vannak kitéve, amit a biztonsági sugár bevezetése old meg.



3.5 ábra: Egymás útját keresztező robotok. A fekete rész a robot kiterjedése, a szaggatott kör a biztonsági zónája a robotnak.

Ez azt jelenti, hogy két robot nem közelítheti meg egymást $2R_b$ –nél jobban, ezért azonnal megállnak, amikor érzékelik robot társuk közelségét. A biztonsági távolság bevezetése egy olyan problémát vet fel, hogyha két robot által kiszámolt célponthoz vezető út keresztezi egymást t időpillanatban, akkor a két útvonal $t+1$ időpillanatban is keresztezni fogja egymást. Ezért a robotok egymás mellett állva, úgymond összetapadva várakoznak. Várakozásuk egészen addig tart, amíg az egyik robot tervezett útja meg nem változik úgy, hogy már nem keresztezi a másik robot útvonalát. Ez az esemény kiszámíthatatlan ideig tarthat, történhet pár ciklus után, de az is elképzelhető hogy végleg így maradnak. Ezt hívjuk holtpont, azaz angolul deadlock problémának. Ennek a problémának a megoldása a kiindulási algoritmusban nem szerepel, hiszen ott pontszerű robotok vannak. Ebben a dolgozatban a szimulációs algoritmus kidolgozásánál a későbbiekben kitérünk a probléma megoldására.

4. Fejlesztő környezet bemutatása, szimulációs környezet létrehozása

Az elmúlt években már készült a Microsoft Robotics Studio –val foglalkozó TDK dolgozat Robotszimuláció C# nyelven, Microsoft Robotics Developer Studio használatával címen [7]. Ebben a dolgozatban írója részletesen bemutatja a fejlesztői környezet leírását.

4.1 Fejlesztő környezet bemutatása

A Microsoft Robotics Developer Studio (RDS) egy Windows-alapú, szolgáltatásorientált futási idejű összetevőt, vizuális eszközöket, példa programokat és dokumentációt tartalmazó fejlesztőkörnyezet. Az RDS egy vizuális programnyelvet is

tartalmaz, amely megkönnyítheti a tanulók illetve a hobbi-programozók munkáját. A programmal 3D-s szimulációs környezet is létrehozható.[8]

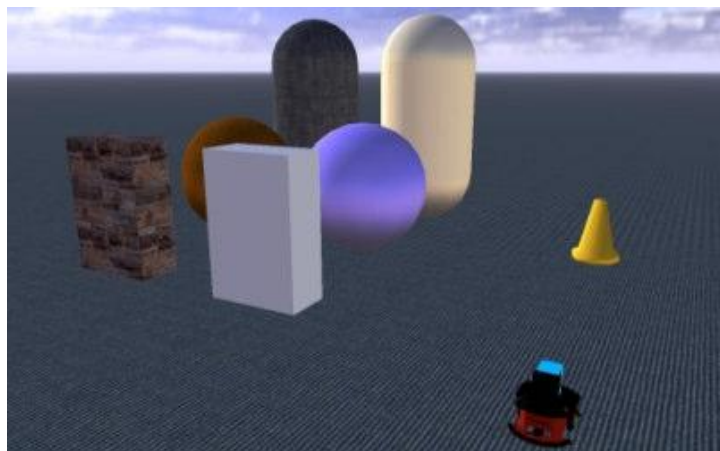
4.2 Támogatott nyelvek

Az RDS a .NET keretrendszert használja, ezért a Microsoft Visual Studio fejlesztő környezetet kell használnunk az alkalmazásaink elkészítéséhez. Bármely .NET keretrendszer által támogatott nyelvet használhatjuk a fejlesztéshez. A példaprogramok és a gyakorlatok nagy része C#-ban vagy Visual Basic .NET-ben áll rendelkezésünkre. Az RDS támogatja az objektumorientált, nyílt forráskódú licenccel rendelkező Python szkript nyelvet is. A rendszer tartalmaz egy vizuális nyelvet is, a Visual Programming Language-et (VPL).[8]

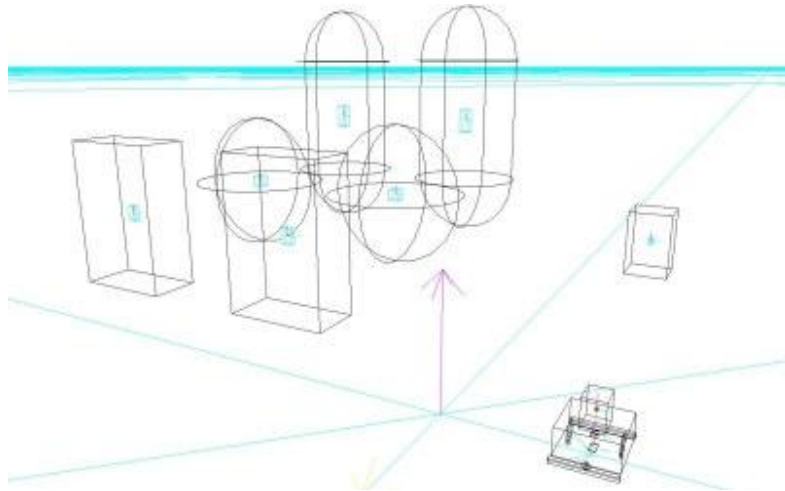
4.3 Szimulációs környezet

Az RDS tartalmazza a Vizuális Szimulációs Környezet (Visual Simulation Environment, VSE) eszközt, melynek segítségével robotalkalmazást készíthetünk anélkül, hogy bármilyen robot eszközzel rendelkezniénk. A VSE Microsoft DirectX 9-et és Microsoft XNA-t használ a térbeli világ megalkotásához. DirectX 9-et és Microsoft XNA-t használ a térbeli világ megalkotásához, az AGEIA PhysX grafikus motor segítségével.

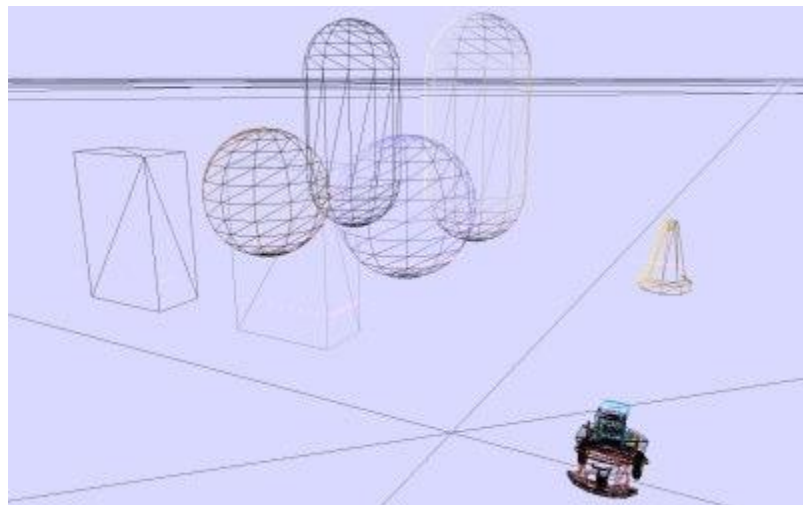
A szimulált világ négy különböző mód valamelyikében jeleníthető meg: vizuális (4.1 ábra), drótváz (4.3 ábra), fizikai (4.2 ábra) vagy e három kombinációja (4.4 ábra). A vizuális mód úgy jeleníti meg a környezetet, mint ahogyan az a való életben is kinézne. Az egyszerű, drótváz megjelenítéssel csupán a tárgyak körvonala jelenik meg. A fizikai nézet szintén egy egyszerűbb megjelenés, ami megmutatja, hogyan kezeli a tárgyakat a *fizikai motor*, azaz a szimulált környezetben a fizikai kölcsönhatásokat modellező interfész.



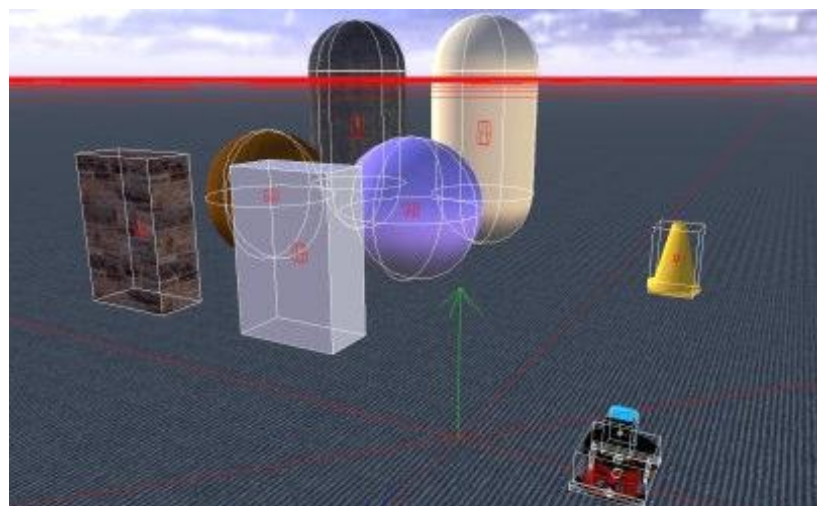
4.1 ábra: Vizuális nézet



4.2 ábra: Fizikai nézet (optimálisabb nyomtatási kép érdekében a kép negatívja)



4.3 ábra: Drótvázás nézet



4.4 ábra: Kombinált nézet

4.3 Szimulációs környezet létrehozása

A VSE -ben minden objektum egy-egy entitás, minden entitást egy függvénnyel kell létrehoznunk és az entitást típusától függően kell paramétereznünk a függvényt.

A *talaj* létrehozásához a függvénynek egy képet kell átadnunk paraméternek, amit a program majd az X és Z tengelyek által meghatározott síkot kitapétázza. Itt kell beállítanunk a talaj tapadási tulajdonságait is.

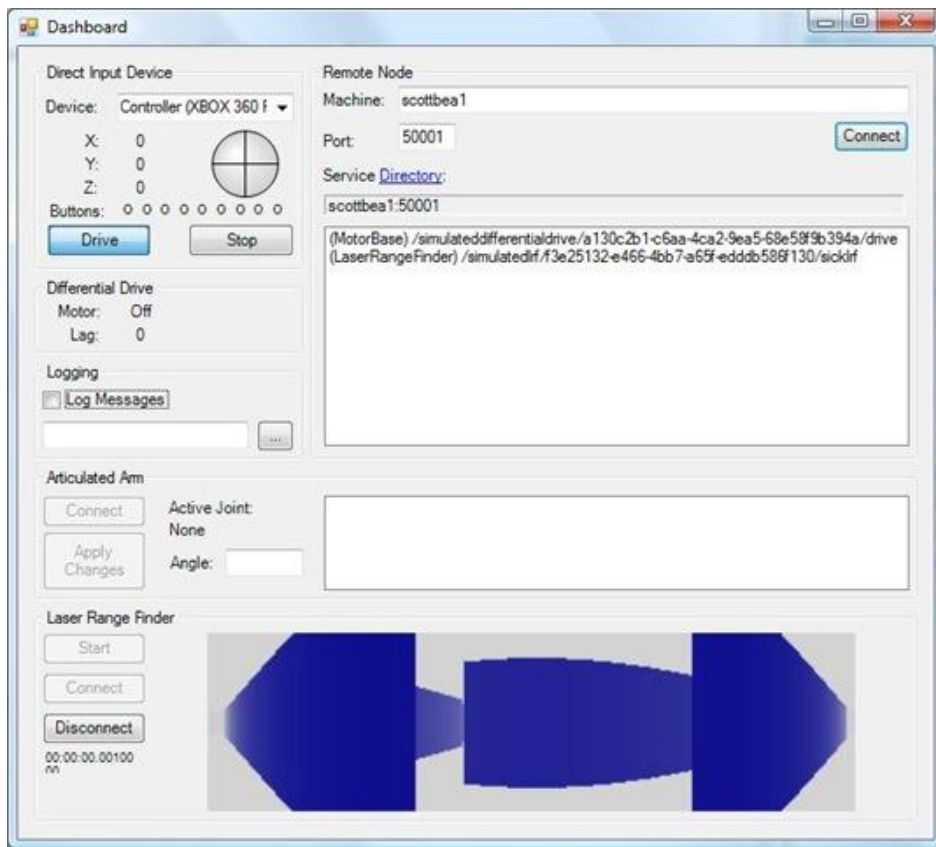
Az égbolt megalkotásához egyetlen függvényre van szükségünk, amely paraméterként két képet vesz át, az egyiket az ég területére széthúzza, a másikat az X tengelyre illeszti annyiszor, hogy azt teljesen lefedje. Ez a kép szín átmenetet biztosít az égbolt és a talaj között.

A kamera automatikusan bekerül egy alapértelmezett helyre, előre meghatározott pontra irányítva. Függvénnyel megadhatjuk a kezdőpozícióját illetve azt, hogy koordináta-rendszerünk melyik pontjára nézzen.

5. Kezdeti nehézségek

5.1 Robotok irányítása

A kezdetekben nehézséget okozott, hogy az RDS példa programjaiban csak olyan programok szerepelnek, amelyekben a robot irányítása nem kódból történik. Ezekben a példa kódokban egy *Dashboard* nevű felhasználói interfészen keresztül kapcsolódhatunk, és irányíthatjuk a robotokat. A *Dashboard*-on megtalálható minden, ami a robotok irányításához és az érzékelők használatához szükséges. Ebben az ablakban egy botkormány segítségével irányíthatjuk az épp felcsatlakozott robotunkat. Az ablak alsó felében pedig kirajzolódik a robot érzékelője által látott kép (5.1 ábra). Mivel csak ez az egy olyan kód található, ami a robotok irányításával foglalkozik, ezért ezt a *Dashboard* -ot kezdtük el mélyebben tanulmányozni.



5.1 ábra a Dashboard interfész felülete

5.2 Dashboard korlátai

A továbbiakban a *Dashboard*-on használt kattintásokat, és egyéb mozgásokat próbáltuk program kódból szimulálni. Megfigyeltük, hogy a *Dashboard*-on kattintáskor valójában a programkódban mi is történik. Leköveltük a folyamatokat, értelmeztük a program sorokat, programrészeket, majd ezeket a sorokat felhasználva elkészítettük a robot irányítását végző programrészletet.

Ez az irányítás mód jónak tűnt, mert sikerült egy olyan kódot létrehozni, amely felcsatlakozik a robotra és a lézeres távolság érzékelőre, és a kódból sikeresen lehetett irányítani robotunkat. A probléma akkor jelentkezett, amikor kibővítettük a programot több robotra. A sorozatos egymás utáni kattintásokat és mozgásokat nem bírta olyan gyorsan feldolgozni a *Dashboard*, mint ahogy a program és az algoritmus igényelte. Legtöbb esetben ez a megoldás a teljes program lefagyását okozta. Tehát olyan megoldást kellett találnunk, ami nem veszi igénybe folyamatosan a *Dashboard* erőforrásait.

6. Megvalósítások

6.1 Robotok kezelése

A robotok létrehozása után, nagyon fontos lépés a robotok megfelelő kezelése. A robotok komponenseivel *kapukon* idegen szóval *portokon* keresztül kommunikálhatunk, a kapuhoz *Post* metódussal küldhetünk üzenetet, adatot. Ha valamilyen üzenet érkezik a kapuhoz, az irányítók mindegyike meghívódik, és megvizsgálja, hogy létre kell-e hozni egy taszkot. Ez az úgynevezett *aktív sor*. Ha nincs irányító, az adat a kapu várakozási sorába kerül, ezt nevezzük *passzív sornak*. Ilyenkor a *Test* metódus segítségével kell kivenni az adatokat. Ha több üzenet is érkezik, arra *PortSet*-et kell használni. Ilyenkor mindegyikhez egy-egy külön *Port* objektum keletkezik.

Irányító létrehozásához, készítenünk kell egy osztályt, amit a *DriveControlEvent* osztályból kell származtatni. Ezt az elkészített osztályt a *DriveControlEvents* osztály *PortSet* paramétereinél kell feltüntetni. Az elkészített irányító osztályt, funkcionalitását tekintve kell beállítanunk. Ez az irányító a *Post* metóduson érkező üzenetet kapja meg az aktív sortól.

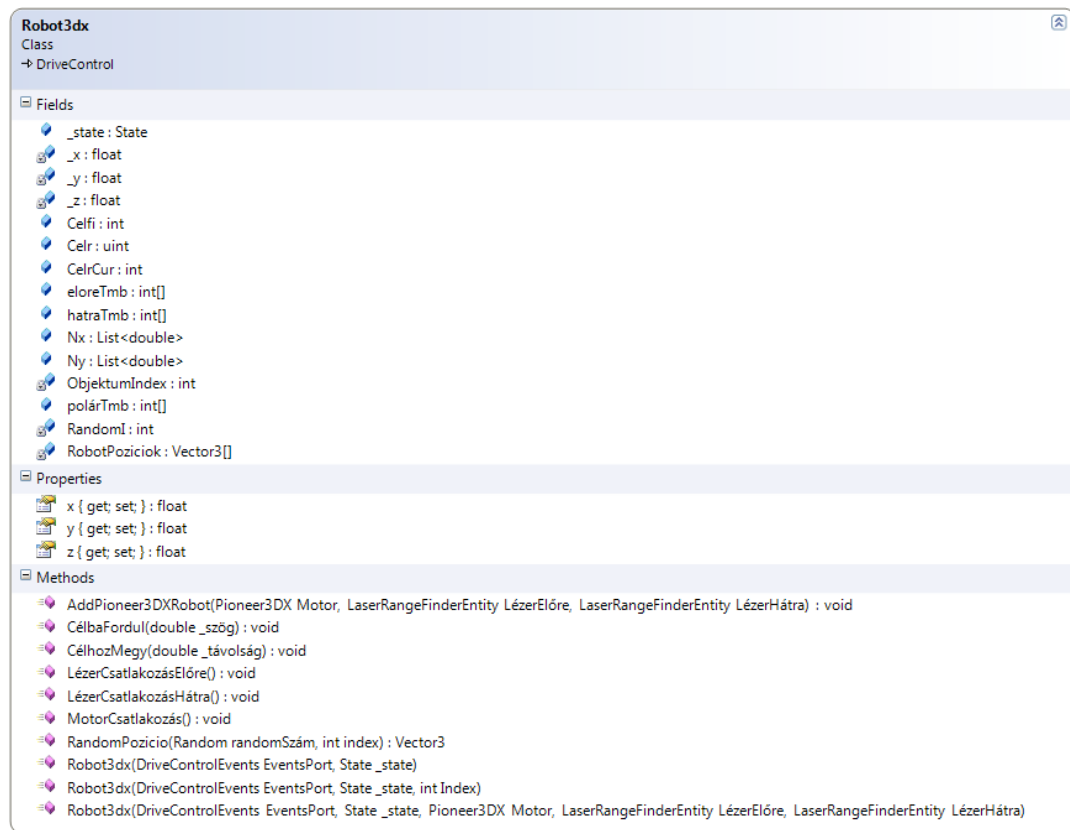
Létrehoztunk egy elfordulás kezelő irányítót, ami az aktív sortól kap egy adattagot, amibe szerepel a robot által kiszámolt elfordulás szöge fokban. Az irányítóban létre kell hozni egy elfordulás kérés objektumot, aminek két adattagjának adunk értéket. Az üzenetben érkező elfordulás fokát, valamint az elfordulás erősségét állíthatjuk be ebben az objektumban. Az elfordulás erősségének nem adhatunk túl nagy értéket, mert akkor a kerekek elcsúsznak a talajon, azaz pontatlan lesz a fordulás. Az inicializált objektumot az úgynevezett *_drivePort* -nak adjuk át, ami a beállított adatokat elküldi a robot komponenseinek. A *_drivePort* a program és a szimulált robot között oldja meg a kommunikációt. Szükség volt még egy távolság kezelő irányítóra is, ami hasonlóképpen működik, mint az elfordulás kezelő. A távolság kezelőben beállíthatjuk, hogy a robot mekkora távot tegyen meg, mekkora sebességgel. Ha túl nagyra állítjuk a sebességet, akkor a megállításkor a robot meg fog billenni.

Azért, hogy az ütközéseket elkerüljük, ennek a távolság kezelő irányítónak a munkáját meg kell szakítani. Ezt úgy értük el, hogy készítettünk egy robot mozgását megállító irányítót, ami ha két robot túl közel kerül egymáshoz, akkor megállítja a robotokat.

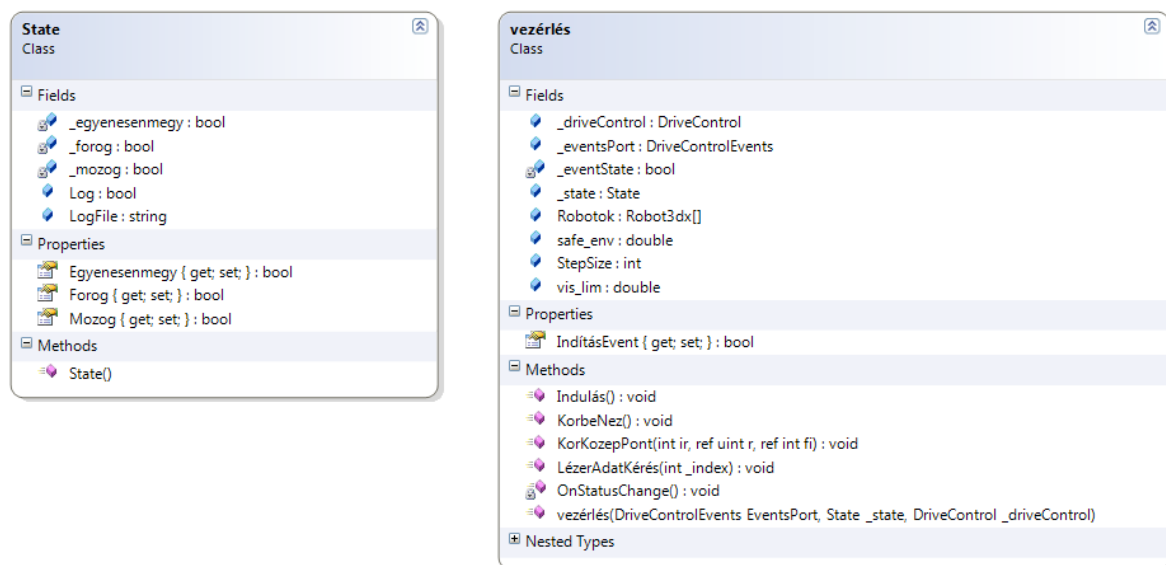
6.2 Program szerkezete

A programot öt osztályra bontottuk fel, az első az alap osztály (*pioneer3dxService*), amelyből indul az egész szimuláció. Ebben az osztályban található a szimulációs környezet létrehozásához szükséges metódusok, valamint az aktív sor is. Következőekben létrehoztunk egy olyan osztályt (*DriveControl*), amiben a *Dashboard* vezérlő elemeit gyűjtöttük össze. Az osztályban készítettük el a robot érzékelői által látottak fogadására képes metódust, valamint az aktív sorhoz szükséges irányítók definiálását. Minden robotot külön objektumként kell kezelni, amelyeknek vannak saját tulajdonságai és metódusai. Létrehoztunk egy *Robot3dx* (6.1 ábra) osztályt, amibe összegyűjtöttük a robot egyed tulajdonságait, a robot érzékelőkkel való felszerelését és a robotmozgató illetve állapotlekérdező metódusait. Ebben az osztályban tároljuk a robot érzékelői által látott aktuális adatokat. Szükség volt még egy robotok állapotát tároló osztályra (*_state*, 6.2 ábra), amiben eltároljuk a robotok éppen aktuális mozgási állapotát, hogy a későbbiekben meg tudjuk állapítani a mozgási folyamatot.

Az összes objektum könnyebb kezelése érdekében, létrehoztunk egy központi vezérlő osztályt (6.2 ábra), amiben elérjük az eddigi összes osztály metódusait és adatait. Ebben az osztályban implementáltuk a tovább fejlesztett algoritmust.



6.1 ábra: *Robot3dx* osztály szerkezete

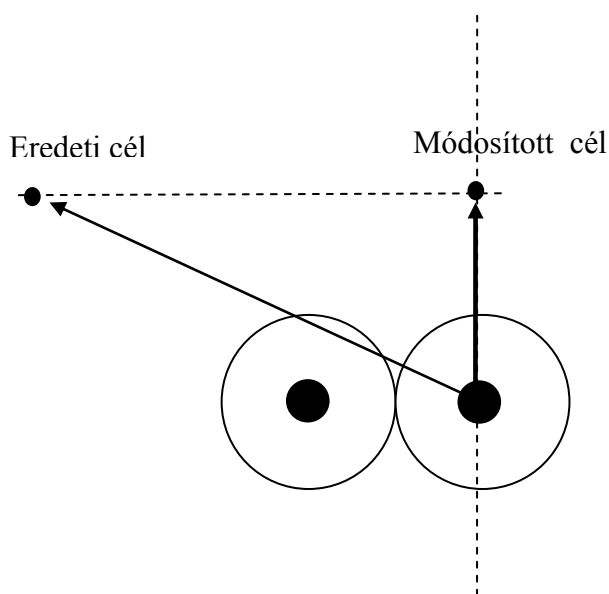


6.2 ábra: State osztály és a vezérlés osztály szerkezete

A szimulációs környezetben egyszerűen lekérdezhető a robotok térben való elhelyezkedése, amit a *Pioneer3DX* osztály példányán keresztül érhetünk el akár futási időben is. Ez az osztálypéldány magát egy robotot valósít meg, amelyről a pozíción kívül sok egyéb állapotot, tulajdonságot lekérdezhetünk.

7. A kiindulási algoritmus módosítása

Ahogy már említettük a robotok kiterjedése miatt jelentkeznek a holtpon (deadlock) problémája. Erre egy viszonylag egyszerű megoldást javasolunk ebben a munkában.

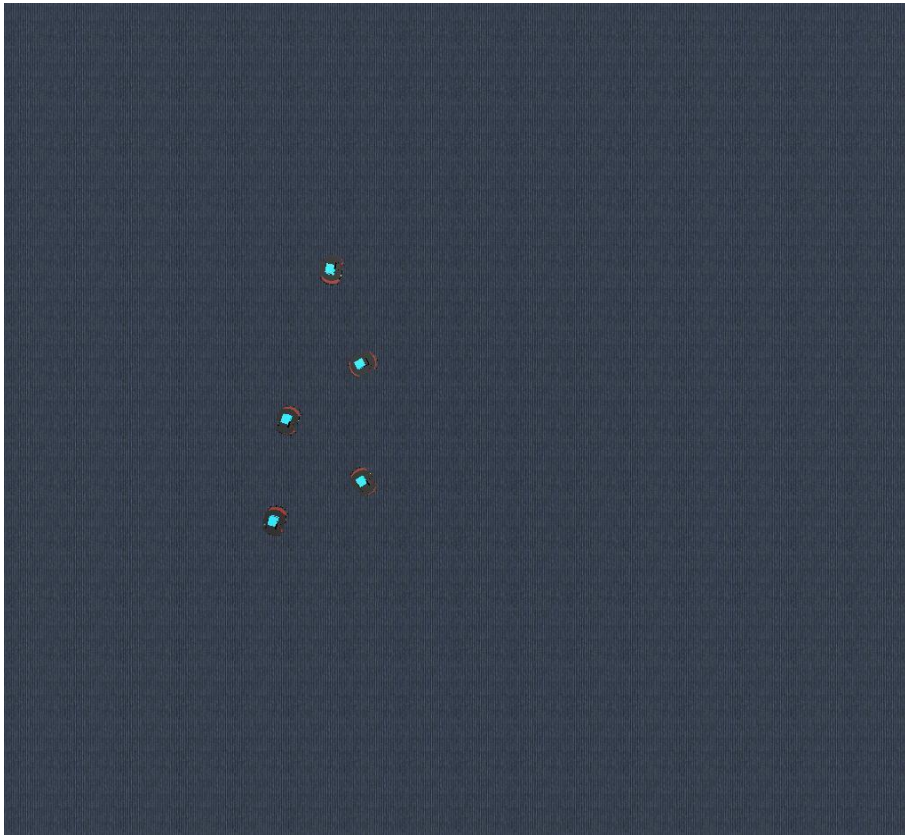
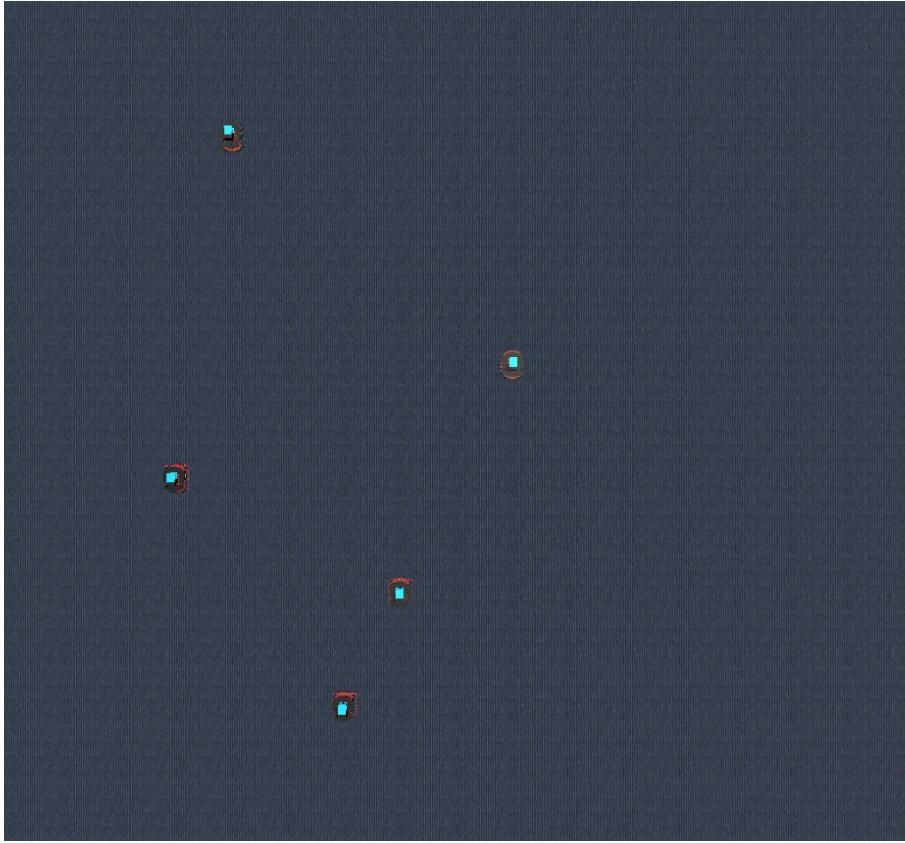


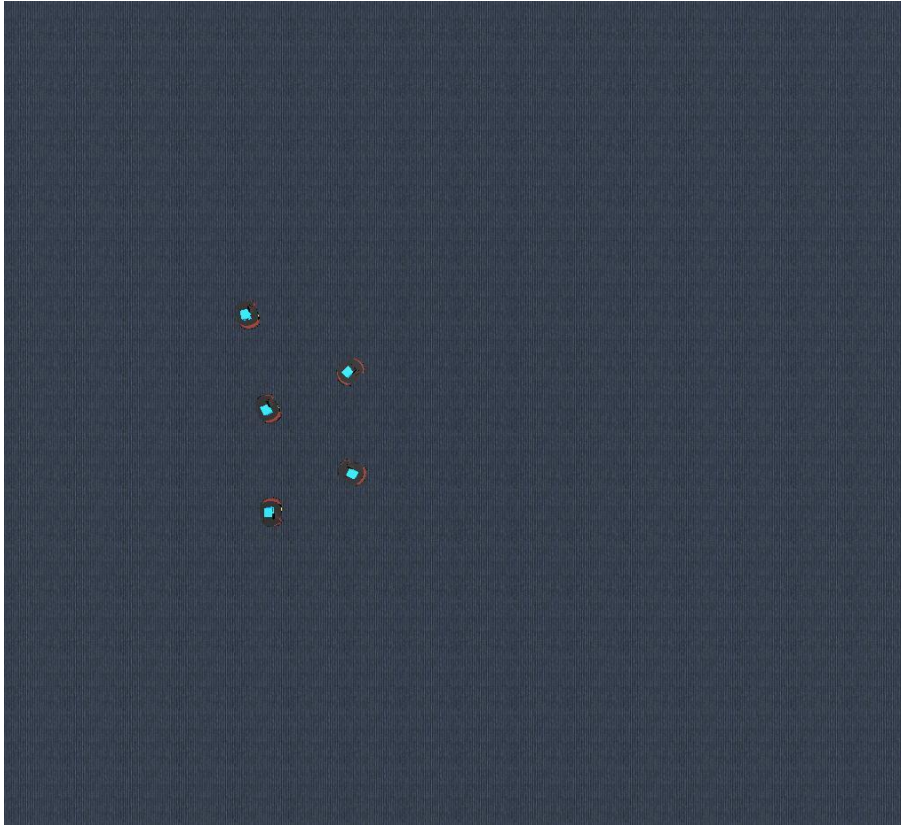
7.1 ábra: A blokkolási probléma megoldása

A módosító megoldás lényege az, hogy egy robot nem tud tovább haladni az útjában álló másik robot miatt, akkor azzal próbálkozik, hogy a társa biztonsági zónájának érintője mentén mozdul el. A megoldás részletezése a 7.1-es ábrán látható. A robot eredeti célvektorát nem tudja követni, helyette ennek a vektornak azt a vetületét fogja követni, amely a másik robot biztonsági zónájára érintőleges. Ilyen módon a robot kimozdulhat a holtpontról. Ez a módszer csak akkor működik, ha a robotot egyetlen másik robot blokkolta. Ha két blokkoló robot is van, akkor az érintő irány elmozdulás sem lehetséges, azaz az ilyen jellegű holtpontot ez a módszer nem tudja megoldani.

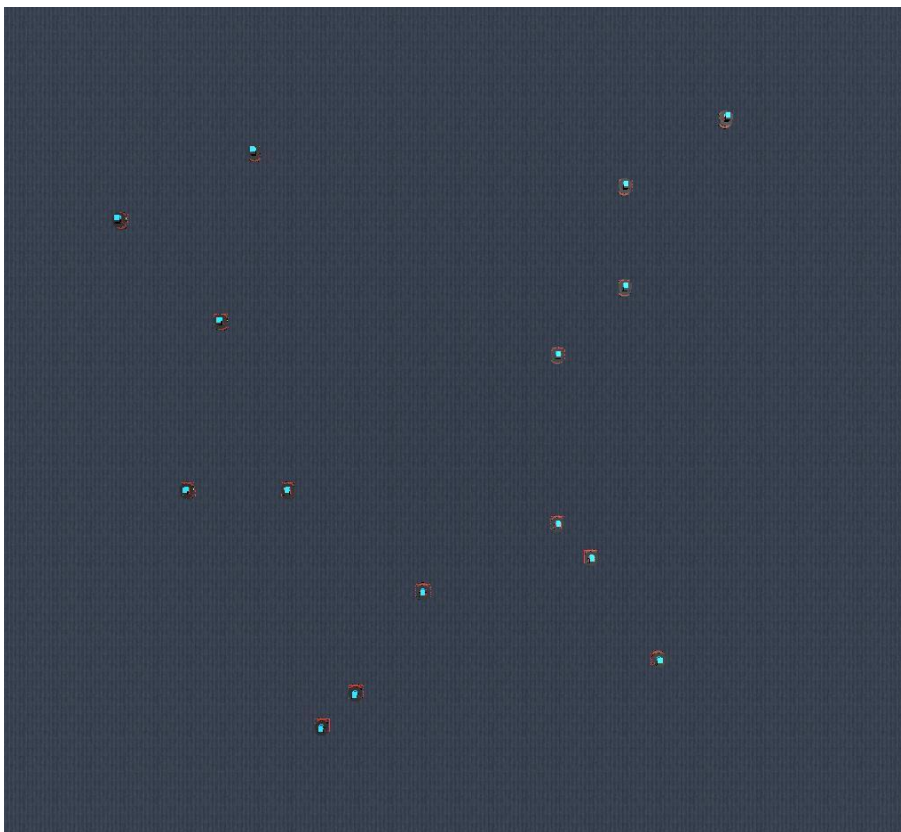
8. A robot szimulációs alkalmazás és az algoritmusok tesztelése- eredmények

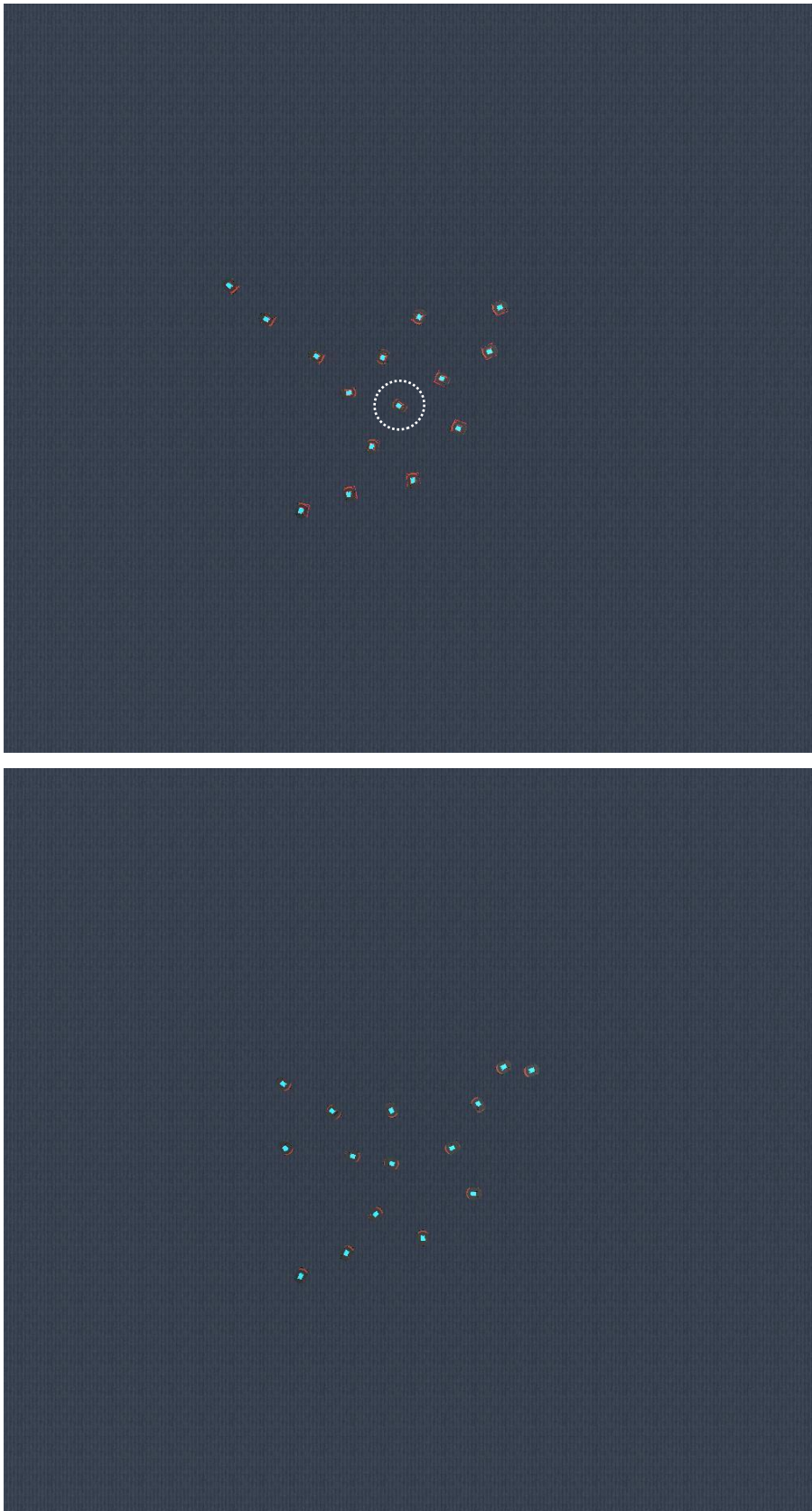
A szimulációs alkalmazást lefuttattuk az eredeti és a módosított algoritmussal is, különböző méretű robotrajok mellett. A rajok egyedszáma a különféle méréseknél 5-10-15-20 volt. Elvégeztünk pár mérést 30 egyedszámra is, de a kevés mérési szám miatt nem vontuk bele a statisztikába az eredményeket. A robotokat kezdetben véletlenszerűen helyeztük el egy 20x20 méteres pályán. A robotok átmérője kb. 80 cm. A biztonsági sugarat $R_b=1$ méterre választottuk. A szimuláció futása alatt minden robot rendszeresen lekérdezte a két lézeres távmérőjét, hogy megállapíthassa a szomszédjai helyét. A lépésciklusok elején kiszámították a tervezett elmozdulások irányát, és távolságát, a korábban részletezett algoritmusok alapján. Azonban a lézeres távmérő a mozgások közben is rendszeresen figyelték, hogy az ütközés elkerülés biztosítva legyen. Ha valamelyik robot azt érzékelte, hogy közelebb került egy másik robothoz, mint $2R_b$, az éppen végrehajtott lépést azonnal megszakította. Az eredményeket 8.1, 8.2, 8.3-as ábrán láthatjuk. Az eredeti és a módosított algoritmust ugyanolyan kezdőfeltételek mellett indítottuk el, a 8.1-es ábrán látható a kiindulási állapot valamint az összegyűlés eredményei mindkét algoritmusra 5 egyedszámmal. A 8.2-es ábrán láthatóak az eredmények 15 a 8.3-as ábrán pedig 30 robotra. Jól látható, hogy mindkét algoritmus működik kiterjedéssel bíró robotokkal valósághű szimulációban. Az is megfigyelhető, hogy a módosított algoritmus, kisebb helyre tudja összegyűjteni a robotokat, vagyis az összegyűlés térkitöltése jobb.



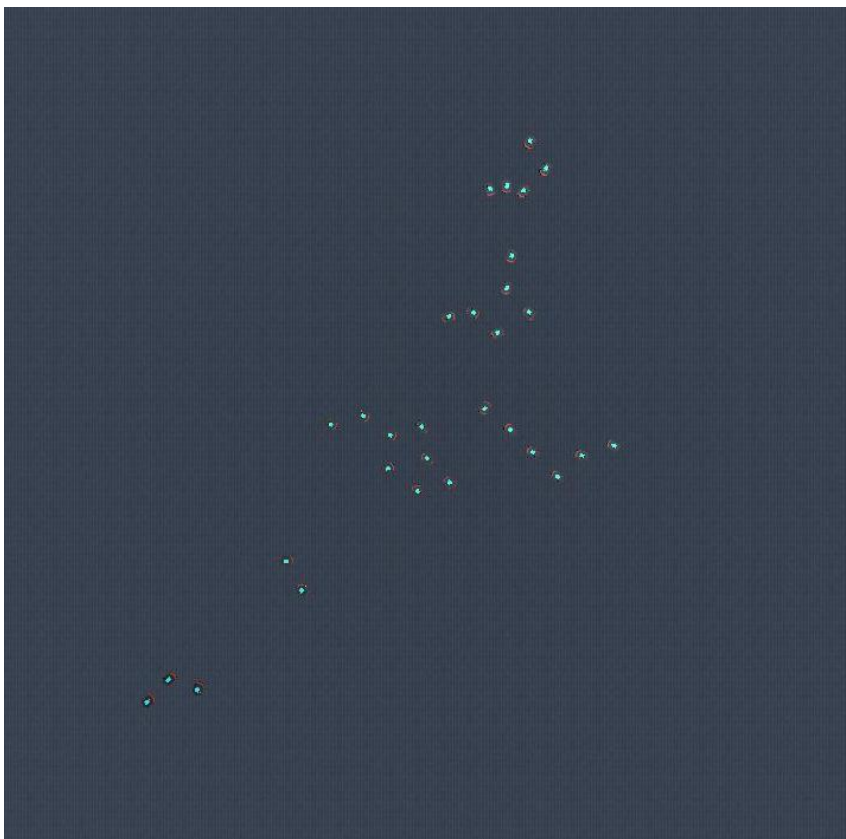
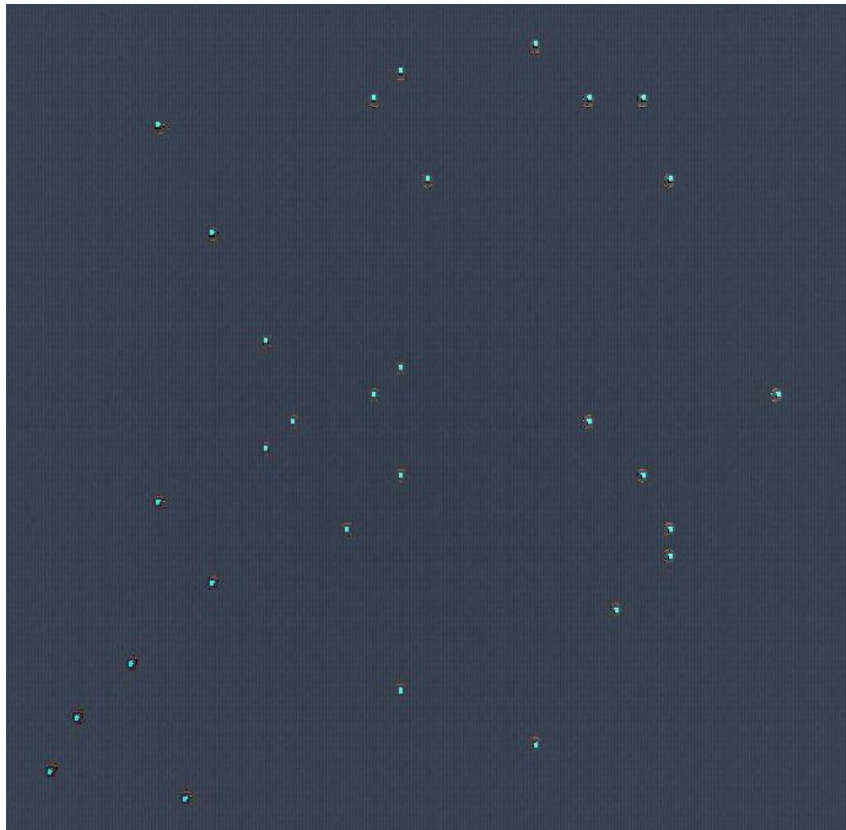


8.1 ábra: A képeken a kiindulási állapot (első kép), az eredeti algoritmus alkalmazásával kapott állapot (második kép), valamint a módosított algoritmus eredménye (harmadik kép) látható 5 robotra.





8.2 ábra: A képeken a kiindulási állapot (első kép), az eredeti algoritmus alkalmazásával kapott állapot (második kép), valamint a módosított algoritmus eredménye (harmadik kép) látható 15 robotra, az egyik képen fehér szaggatott körrel látható a robot biztonsági zónája.





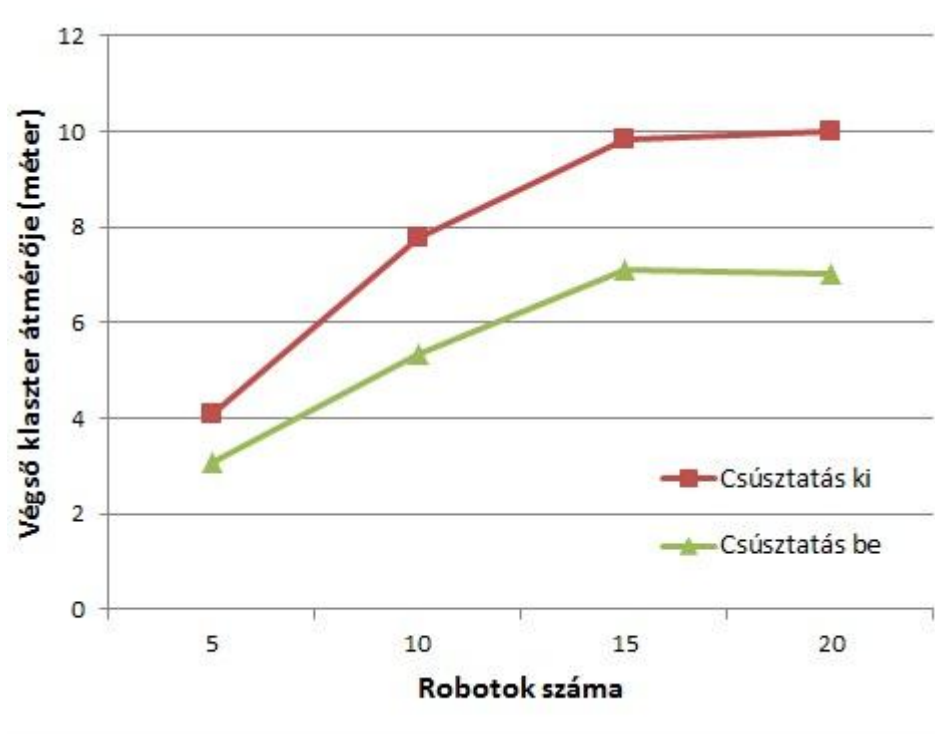
8.3 ábra: A képeken a kiindulási állapot (első kép), az eredeti algoritmus alkalmazásával kapott állapot (második kép), valamint a módosított algoritmus eredménye (harmadik kép) látható 30 robotra.

A 8.1-es ábrán láthatóak az összegyűlési eredmények 5 robotra, ugyanazon kezdőfeltételekkel indítva az eredeti algoritmust és a módosított algoritmust. Látható, hogy itt nem sokban tér el a két eredmény, amiből következtethető, hogy az algoritmus módosítás kis egyedszámú rajokra nem okoz nagy változást az eredeti algoritmushoz képest.

A 8.2-es ábrán 15 egyedre láthatóak az eredmények. Ennél az egyedszámnál már látható különbség a két algoritmus között. Az eredeti algoritmus összegyűlési eredményén látható a bal felső negyedben egy kinyúló, nagyjából egyenes szakasz, ahol a robotok blokkolták egymást, és ezért nem tudtak jobban összegyűlni. Ugyanilyen kezdő feltételek mellett, a módosított algoritmus eredményében ez a kinyúló szakasz már nem található, az algoritmus kisebb területre tudta összegyűjteni a rajt. A 8.3-as ábra egy 30 robotból álló rajon végzett szimulációt mutat (szintén eredeti és módosított algoritmussal). Látható, hogy az eredeti algoritmus ennél a kísérletnél nem tudta megoldani az összegyűlési feladatot,

mivel a végső érintkezési gráf nem teljesíti a 3.2 szakaszban megfogalmazott minimum követelményt, azaz nem összefüggő (az érintkezést a biztonsági zónára vonatkoztatva értjük). Ezek szerint kellően nagy, esetünkben 30 robotból álló rajnál, előfordulhat, hogy az eredeti algoritmus nem működik kiterjedéssel bíró robotokból álló rajnál. Viszont jól látható a 8.3-as ábrán, hogy az általunk módosított algoritmus egy sokkal tömörebb, és ami még fontosabb, összefüggő csoportba gyűjtötte össze a robotrajt. Ezek alapján a szimulációk alapján úgy tűnik, hogy a módosított algoritmus előnye kis elemszámú rajnál még nem mutatkozik meg, de nagy elemszámú rajnál már jobban teljesít, mint a kiindulási algoritmus.

A fenti megállapítást nagyszámú kísérlettel is teszteltük. $N=5,10,15,20$ robotszámok esetén 10 darab szimulációs kísérletet végeztünk el minden egyes N értékre. A 6.2-es alfejezetben elmondottak alapján a szimulációs rendszerből kinyerhetők a robotok pozíciói a szimuláció bármely szakaszában. Ezeket a pozíciókat felhasználva, kiszámítottuk minden egyes kísérlet végső klaszterének legnagyobb átmérőjét, azaz a két legtávolabbi robot távolságát. Ezt számítást elvégeztük mind a kiindulási algoritmus összegyűlési eredményeire, mind az általunk fejlesztett algoritmus eredményeire. Az így kapott eredményeket átlagoltuk, majd egy grafikonon ábrázoltuk (8.4. ábra).



8.4 ábra: A két algoritmus eredményeinek összehasonlítása.

A grafikonon szereplő *Csúsztatás ki* kifejezés, az eredeti (Ando-féle) algoritmus eredményeit jelöli (piros vonal), míg a *Csúsztatás be* kifejezés (zöld vonal), pedig az általunk módosított algoritmus eredményeit jelöli. A grafikonon az összegyűlt robotraj átlagolt legnagyobb átmérőjét ábrázoltuk a robotok számának függvényében.

A grafikonon jól látszik, hogy a módosított algoritmus kisebb területre gyűjti össze a fizikai kiterjedéssel bíró robotokat, mint az eredeti algoritmus.

Térkitöltő alakzatoknál a klaszter által elfoglalt terület (T) és a klaszter átlagos átmérője (D) a következő arányban áll egymással:

$$T \sim D^2 \quad (8.1)$$

azaz

$$D \sim T^{1/2} \quad (8.2)$$

A mi esetünkben a klaszter által elfoglalt terület arányos a robotok számával (N), azaz térkitöltő esetben:

$$D \sim N^{1/2} \quad (8.3)$$

Ez azt jelenti, hogy elméletileg egy négyzetgyök függvényt kell kapnunk, ha a D -t az N függvényében ábrázoljuk.

A bemutatott grafikon valóban gyökfüggvényre hasonlít, de egyelőre a kevés mérési eredmény és a túl nagy szórás miatt ezt nem tudjuk bizonyítani. Ezért további nagy robotszámokkal elvégzett mérésekre van szükség. (A grafikonon látható adatpontok esetében a szórás a 0.4-1.3 tartományban mozog.)

Mivel az általunk kifejlesztett szimulációs program egy olyan valóság-hű robot szimulációs fejlesztői környezeten alapszik, amelyet műszaki és tudományos körökben is hiteles szimulációnak fogadnak el (Microsoft Robotics Developer Studio), megállapíthatjuk, hogy ezek az összegyűlési algoritmusok az általunk szimulált robot típus esetében a valóságban is kivitelezhetőek. A szimulációs környezetünk segítségével egyéb összegyűlési algoritmusok is kiválóan tesztelhetőek lennének, továbbá egyéb raj-intelligencia algoritmusok tesztelésére is alkalmas. A jelenlegi munka alapjául szolgálhat további raj-intelligencián alapuló kutatásoknak is.

További terveink között szerepel még, olyan teszt szimuláció elvégzése, amelyben nem csak robotok, hanem szilárd átlátszatlan akadályok is vannak a pályán. Az ilyen akadályok ugyanúgy blokkolják a robotok mozgását, ahogyan a robotok is egymást, viszont a pozíciójuk nem változik. Könnyen elképzelhető, hogy bizonyos akadály sűrűség illetve elhelyezkedés mellett, nem minden összegyűlési algoritmus működik, vagy akár megoldhatatlanná is válhat a feladat.

Kezdetben kisszámú akadályt tartalmazó pályán tervezzük a tesztelést, azt feltételezzük, hogy ilyen feltételek mellett az általunk módosított algoritmus sokkal jobban fog működni, mint az eredeti mivel ennél a robotok rendelkeznek azzal a képességgel, hogy az esetleges egyszerűbb akadályokat megkerüljék.

További érdekes kutatási terület az aszinkron összegyűlési algoritmusok vizsgálata a kifejlesztett szimulátorral, mivel a szimulációs rendszerben a robotok külön objektumként működnek. Ezért az aszinkron algoritmusok is könnyen tesztelhetők ezzel a rendszerrel.

Irodalomjegyzék

- [1] H. Ando, I. Suzuki, M. Yamashita
„Formation and Agreement Problems for Synchronous Mobile Robots with Limited Visibility”
Syp. of Intelligent Control (1995) o. 453-460
- [2] P. Flocchini, G. Prencipeb, N. Santoroc, P. Widmayer
„Gathering of asynchronous robots with limited visibility”
Theoretical Computer Science 337 (2005) o. 147 – 168.
- [3] B. Degener, Barbara K., F. M. Heide,
„A Local $O(n)$ Gathering Algorithm”
SPAA'10, ACM (2010) o. 217-223
- [4] J. Czyzowicz, L. Gasieniec, A. Pelc
„Gathering few fat mobile robot sin the plane”
Theoretical Computer Science 410 (2009) o. 481-499
- [5] S. G. Chaudhuri and K. Mukhopadhyaya
„Gathering Asynchronous Transparent Fat Robots”
INCDIT (2010), LNCS 5966, o. 170-175
- [6] A. Cord-Landwehr, B. Degener, M. Fischer, M. Hüllmann, B. Kempkes, A. Klaas, P. Kling, S. Kurras, M. Martens, F. Meyer, C. Raupach, K. Swierkot, D. Warner, C. Weddemann, and D. Wonisch,
„Collisionless Gathering of Robots with an Extent”,
Sofsem, LNCS 6543, (2011) o. 178-189
- [7] Nép Gábor
„Robotszimuláció C# nyelven, Microsoft Robotics Developer Studio használatával”
KEFO-GAMF Kari TDK Konferencia (2009)

[8] Sara Morgan: Programming Microsoft Robotics Studio, Microsoft Press, 2008, 1. fejezet.

[9] http://hu.wikipedia.org/wiki/Mesters%C3%A9ges_intelligencia