Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

TDK dolgozat

Anyagmozgató mobilis robot ütközésmentes irányítása

Készítette:

Mihály Szabolcs

Konzulens:

Gincsainé Dr. Szádeczky-Kardoss Emese

1 Bevezető

Az ipar gépesítésével és folyamatos fejlődésével egyre nagyobb hangsúly helyeződik az olyan folyamatok robotizálására, amelyek során anyagmozgatás történik. Már az *első ipari forradalom* előtt is nagy jelentőséggel bírt az emberek számára, hogy egy adott pontból egy másikba juttathassanak termékeket, eszközöket, illetve más tárgyakat.

Az *első ipari forradalom* során az anyagmozgatás felgyorsult a gőzgép megjelenésével, ugyanakkor alkalmas volt a szállításra a széltől függetlenül. A gőzvonat megjelenésével pedig nagy mértékben felgyorsult a szárazföldi szállítás is [1].

Az anyagmozgatás tekintetében a következő nagy előre lépésre a *második ipari forradalom* alatt került sor, amikor a gőzgépeket felváltották a belső égésű motorral hajtott járművek, valamint megjelentek elektromos hajtással rendelkező futószalagok és felvonók [2].

A *harmadik ipari forradalom* során kerültek a gyártósorra különböző munkavégzésre alkalmas robotkarok, amelyek nagyobb terheket is könnyedén mozgattak. A gyártósorokon kívül más területeken is megjelentek a robotok, mint például a raktárhelységekben a nyersanyagok betárolására, vagy a késztermékek kitárolására [3].

A jelenleg is tartó *negyedik ipari forradalom* más néven az *Ipar 4.0* a robotok illetve a automatizált eszközök közötti kommunikációt jelenti, ugyanakkor feltételezi ezek folyamatos on-line működését és a működés valós idejű (*real-time*) nyomon követését [4]. Anyagmozgatás szempontjából a valósidejűség, és az eszközök közötti kommunikáció a nagy mértékben fokozza a biztonságot, és a termelékenységet, ugyanis egy rendszer minden egyes robotja vagy automatizált eszköze az interneten keresztül információt cserél. Az információcsere révén, tudomást szereznek egymás helyzetéről, munkafázisaik állapotáról, valamint hiba esetén azt egy felügyelő szerveren keresztül jelezni tudják.

1.1 Motiváció

A harmadik ipari forradalom óta a gyártásban megjelent robotok és automatizált eszközök egyre nagyobb és nagyobb szerepet töltenek be az anyagmozgatás területén. Ilyen anyagmozgató robotok lehetnek többszabadságfokú robotkarok, amelyek bizonyos fizikai határok között képesek az anyagmozgatásra, de ilyenek például a differenciális meghajtású robotok is.

A differenciális meghajtású robotok egy gyártelephely több részlegén is megtalálhatóak: a nyersanyag betárolásától a félkész termékek gyártó sorok közötti mozgatásán keresztül egészen a kész termék kitárazásáig. Ezen részlegek mindegyikére igaz, hogy a robot nem egy szabad területen helyezkedik el, és mozog, hanem rendszerint polcok, tárolóegységek, valamint munkaállomások nehezítik meg a mozgását.

Ezen nehézségekkel számolni kell, és ebből kifolyólag nem lehet a robot segítségével egy egyenes mentén eljuttatni a mozgatni kívánt anyagot a kiindulási pontból a célpontba. Ennek a problémának a megoldására számos ütközésmentes útvonal tervező algoritmus került kidolgozásra, de ezek nagy része csak egy útvonal meghatározására alkalmas, vagyis egy kiindulási ponttól a célpontig meghatároz egy lehetséges útvonalat. Ezzel szemben a raktáregységekben jellemzően több áthelyezendő anyag található, és sűrűn előfordul, hogy különböző kiindulási pontból különböző pontokba kell juttatni a szállítmányokat.

Amennyiben több mozgatandó anyag (objektum vagy tárgy) van, úgy az egyik objektumnak a kiindulási pontból a célpontba juttatásával megváltozik a többi objektum lehetséges útvonalainak száma, illetve lehetséges útvonalainak hossza. Gondoljunk csak bele, hogy egy objektum elmozdításával új útvonalak nyílhatnak meg egy másik objektumra nézve. Ugyanakkor az objektum új helyzetébe való vitelével útvonalak záródnak el, hiszen nem hanyagolható el egyik objektum dimenziója sem.

Az előbbiekben felvázolt probléma azt vonja maga után, hogy az objektumok sorrendjére figyelni kell, ugyanis előfordulhat, hogy egy másik objektum útvonala teljesen elzáródhat és egy adott objektumot a robot nem tud majd a célpontjába juttatni. Amennyiben az útvonal hosszára optimalizáció történik, úgy egy rossz sorrenddel elzárható egy objektum legrövidebb útvonala. Egy jó sorrenddel viszont megnyitható egy objektumnak egy rövidebb útvonal is.

1.2 Célkitűzés

A dolgozatom célja egy már létező útvonalkereső algoritmus vagy algoritmus elv oly módú módosítása, bővítése, hogy képes legyen több mozgatható objektum esetén egy megfelelő sorrend meghatározására. A problémát két esetre bontom szét: az első esetben csak két objektumra valósítom meg az algoritmust, később pedig több akadályra is.

Az algoritmus az útvonalat két fő bemeneti paraméterből kell meghatározza: az egyik az összes objektumot tartalmazó bináris térkép (*.bmp* fájl), valamint a mozgatásra szánt objektumok struktúratömbje.

Az algoritmussal szemben támasztott követelmények a következők:

- Offline módban legyen futtatható, így a futásidő nem kritikus paraméter
- Az útvonaltervezés ütközésmentes legyen
  - Minden egyes útvonaltervezésnél vegye figyelembe a mozgatandó objektum dimenzióját (megfelelő mértékben tekintsen nagyobbnak minden más objektumot)→objektumnövelés (képdilatáció)
- Az útvonaltervezés során elhanyagolásra kerül az objektumokat mozgató robot dimenziója
- A bináris képen a *fekete* részeket tekintem objektumoknak, a *fehér* részeket a szabad területnek

Az algoritmus megvalósítása során a következő feltételekkel éltem:

- A mozgatható objektumok megközelítőleg négyzetek, de mindenképpen négyszögű alakzatok
- Nem fordulhat elő olyan, hogy két objektum egymásnak elzárja az egyetlen célpontjukba vezető utat
  - Ha ilyen mégis előfordulna, akkor az algoritmust fel kell készíteni arra, hogy az adott objektumot egy köztes pozícióba vigye, ahol leteszi, hogy a másik objektumot a célpontba vihesse

Az útvonaltervező algoritmuson kívül megvalósításra kerül egy felhasználói kezelőfelület (*user interface*, UI), amelyen beazonosíthatóak lesznek az objektumok. A beazonosított objektumok közül pedig ki lehet választani helyváltoztatásra szántakat a bináris térképről. A kiválasztott objektumokat egy megfelelő struktúratömbbe helyezem, amelyet az algoritmus bemenetként vár. A kezelői felületnek rendelkeznie kell a később még esetlegesen bővülő paraméterek bevitelére is. Az algoritmus lefutása után megjelenítésre kerül az útvonal az alapján, hogy mely gráfpontokon kell áthaladni a robotnak az objektumok mozgatása érdekében. Esetlegesen a későbbiekben egy szimulációval szemléltetésre kerül az objektumok mozgatása a meghatározott útvonal mentén.

A dolgozatom során bemutatásra kerülnek olyan algoritmusok, amelyek alkalmasak ütközésmentes útvonal meghatározására (2. fejezet). Ezen algoritmusok közül bizonyos kritériumok szerint kiválasztok egyet, amelyet oly módon módosítok, hogy a végén megoldást nyújtson a problémára. A kiválasztott algoritmust implementálom, oly módon, hogy a későbbiekben megfelelően skálázható legyen. A problémát két esetre bontom szét, melyek közül az egyik, az, hogy két mozgatható objektum esetére adjon megoldást a módosított algoritmus, valamint a későbbiekben több mozgatható objektumra is megoldást jelentsen. Mindkét esetben bemutatom, hogy miként kerültek megvalósításra az algoritmus, valamint az algoritmus eredményeit is szemléltetem. Végül pedig kitérek, hogy milyen hiányosságokkal rendelkezik a megvalósított algoritmus, és a jövőben miként lehetne továbbfejleszteni.

2 Ütközésmentes és legrövidebb útvonalkeresésre alkalmas algoritmusok

2.1 A*

Az A* a *legjobbat-először keresés* leginkább ismert változata, amelyet Bertram Raphael, Nils Nilsson és Peter Hart alkotott meg 1968-ban [5].

Az A* is egy gráfbejáró és útvonalkereső algoritmus, amelyet az optimális hatékonysága miatt elterjedten használnak a számítástechnikában [6]. A gráf egy adott kezdőcsúcsból kiindulva arra törekszik, hogy olyan utat találjon a célcsúcshoz, amelynek a legkisebb a költsége. Ezt úgy valósítja meg, hogy a kiinduló csúcsból fákat növeszt, és folyamatosan hozzárendel a már meglévő fához újabb éleket mindaddig, míg el nem éri a célpontot. A fő ciklus minden egyes lépésében meghatározza, hogy mely utat egészíti ki. Ezt az alapján hajtja végre, hogy veszi az ehhez az úthoz tartozó költséget, és becsüli a cél eléréséhez tartozó költséget. Matematikailag kiválasztja az

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

függvényt minimalizáló utat, ahol *n* az úton lévő következő csúcs, és $g(n)$ a kezdőcsúcstól az *n* csúcsig tartó út költsége, valamint $h(n)$ egy heurisztikus függvény, amely *n*-től a célig tartó legolcsóbb utat becsüli. Az algoritmus akkor fejeződik be, ha a kiindulópontból a célpontba vezető utat kiterjeszti, vagy ha már nincsenek kiterjeszthető utak [6].

Fontos megjegyezni, hogy a heurisztikus függvény az esetek többségében problémaspecifikus. Ha a függvény soha nem becsüli felül a cél eléréséhez szükséges tényleges költségeket, akkor a függvény megengedő, vagyis az algoritmus garantáltan az optimális utat adja meg a kiinduló-, és a célpont között [6]. Ahhoz hogy teljesüljön az elfogadó heurisztika érdemes a légvonalban vett távolságot alkalmazni, mint heurisztika, mert ez eleget tesz annak, hogy két pont között a legrövidebb távolságot becsülje [5].

2.2 Rapidly Exploring Random Trees (RRT)

Az RRT lényege, hogy véletlenszerűen mintavételezi a szabad teret [7], és ezáltal egy kiindulási pontból egy fát növeszt a szabadon bejárható térben [8] úgy, hogy egy útvonalat

tervezve a két pont között. A fa növesztése úgy történik, hogy a terjeszkedés a célpont felé tartson. Az útvonal akkor kapható meg, amikor a növesztett fa ténylegesen eléri a célpontot [8].

Az RRT-nek több változata is ismert, melyek közül az alapváltozat az, amikor a kezdeti pontból növeszt fát a célpont irányába. A másik változat, amikor a célpontot bevonva is növeszt fát a kiindulási pont irányába, de köztes pontok bevonásával egyszerre több fa is növeszthető [8].

Kezdetben adott a kiinduló pont és a célpont, amely között az útvonal meghatározásra kerül. A fa építése úgy kezdődik, hogy véletlenszerűen felvesz egy pontot a szabad térben. Ezt követően megkeresi, hogy a véletlenszerűen felvett ponthoz melyik pont esik a legközelebb a fából [8]. Meghatározásra kerül a két pontot összekötő egyenes, és amennyiben ütközésmentes útvonalat valósít meg az új egyenes, úgy a fa részévé válik.

2.3 Probabilistic Road Map (PRM)

A PRM az egyik legnépszerűbb mintavételen alapuló ütközésmentes útvonaltervező algoritmus. A PRM véletlenszerűen vesz fel a szabadon bejárható térben gráfpontokat, majd a gráfpontok között egyenesek segítségével gráféleket vesz fel úgy, hogy azok a szabadon bejárható térbe essenek. Így a csúcsok és élek révén egy gráfot épít fel. Gyorsasága abban rejlik, hogy az útvonalkeresést egy gráfbejárási problémára redukálja [9].

Hátránya az, hogy a véletlenszerű csomópontok felvétele miatt előfordulhat, hogy nem egyenletesen oszlanak el a pontok a szabadon bejárható térben. Így izolált gráfok keletkezhetnek, melyek között az átjárás nem valósítható meg [9]. Ez ellen megfelelő élhossz beállításával lehet tenni. Önmagában a PRM, alacsony gráfkihasználtsággal rendelkezik, ugyanis egy „nagy” gráfot épít fel, amit nem feltétlenül használ fel újra útvonaltervezéshez [9].

Az RRT-vel szemben a PRM egyik legnagyobb előnye, hogy amennyiben ugyanabban a környezetben kell ütközésmentes útvonalat keresni, úgy a PRM által felépített gráfban ezt egyszerűen meg lehet tenni a kiinduló-, és célpontok között. Ezzel szemben az RRT minden kezdeti pontból újabb fát kell generáljon, hogy útvonalat találjon a pontok között változatlan környezetben.

A kiindulópont és a célpont közötti útvonal tervezéséhez egy *Djikstra* legrövidebb út algoritmust használ [10].

2.4 Ant Search

A hangya kolónia alapú optimalizáló algoritmus használható legrövidebb út megkeresésére is. A hangya kolónia algoritmus (HKA) egy viszonylag új és hatékony mesterséges intelligencia, amely a hangyák viselkedését igyekszik alkalmazni optimalizációs problémák megoldására. Az alapja, hogy a kolóniában élő hangyák miként találják meg az optimális utat az élelmük, és a fészkük között egy egyedi nyomvonal kialakításával. A hangyák folyamatosan pozitív visszajelzést adnak az általuk bejárt útról feromonok révén. Az optimum megtaláláshoz ezen feromonokat frissíti minden hangya lokálisan, ami által globálisan is frissül az egyes megoldások feromonszintje, így végül egy optimális megoldás áll elő [11].

3 Megvalósítás

3.1 Felhasznált technológia

Az általam kitűzött cél megvalósítására a MathWorks által létrehozott és fejlesztett MatLab fejlesztői környezetet használtam, melynek neve a **Mat**rix **Lab**oratory szavak összekapcsolásából ered [12]. Ez a környezet saját programozási nyelvvel rendelkezik, amely nem különül el nevében a szoftver nevéhez képest. (A továbbiakban a *MatLab* a fejlesztői környezetre vonatkozik, míg a *matlab/MATLAB* a programozási nyelvre).

A matlab egy magas szintű programozási nyelv, melyet műszaki számítástechnikához fejlesztettek ki. Magába foglal különbféle számításokat és azok vizualizációját is, mindezt úgy, hogy a programozáshoz egy könnyen felhasználható környezetet biztosít, ahol a problémák és a megoldások jól ismert matematikai jelölésekkel vannak kifejezve [13]. A MatLab *C/C++* nyelvben került megvalósításra, de egyes források szerint ez kiegészül némi *Java* kóddal is. Önmagában a MatLab egy mátrix alapú programozási környezetet biztosít, de rengeteg kiegészítő könyvtárral rendelkezik, melyek esetenként probléma orientáltak, vagyis egy adott feladatra célzottan nyújtanak megoldást vagy eszközöket a megoldás megtalálására. Ezek a könyvtárak tipikusan *Perl*, *Java*, *ActiveX* vagy *.NET* nyelven kerülnek megvalósításra [12]. Ezen túlmenően a MatLab támogatja, hogy a saját nyelvén kívül, amit *M* kódnak is szoktak hívni, más nyelvet használva is lehessen függvényeket, illetve kódrészleteket írni. Ilyen nyelvek kezdetben a *C* és a *Fortran* nyelvek voltak, amiket a fejlesztő környezet a kód futtattatása során *.MEX* (MatLab executables) fájllá alakított, ami egy dinamikusan betölthető objektumfájl. Ez 2014 óta kibővült a *Python* nyelvvel is [12].

A MatLab tipikus felhasználási területei a következők:

- Matematikai problémák számítása
- Algoritmusok tervezése és fejlesztése
- Modellezés, szimuláció, prototípus tervezés
- Adatelemzés és vizualizáció
- Tudományos és mérnöki grafika
- Alkalmazásfejlesztés, magába foglalva a grafikus kezelői felületek fejlesztését is [13]

Ezen szempontokat figyelembe véve választottam ezt a fejlesztői környezetet az algoritmus megvalósítása céljából. A MatLab támogatja az objektumorientált programozást, valamint a különféle módokon megvalósítható vizualizációt. Előre megírt optimális algoritmusokkal rendelkezik különféle könyvtárakból, melyek letöltés után azonnal használhatóak. Sokrétű felhasználással bír, és megfelelő szabadság fokot ad a felhasználója kezébe.

3.2 Az alap ütközésmentes útvonaltervező algoritmus kiválasztása

Az 3.1-es fejezetben bemutatott útvonaltervező algoritmusok mindegyike statikus környezet feltételeznek az ütközésmentes útvonaltervezés során. Ezzel szemben én egy olyan algoritmust tervezek megvalósítani, amely képes változó környezetben a mozgástervezésre. Ennek elérése érdekében a PRM, RRT, A* algoritmusok közül választok ki egyet, amelyet módosítva képes lesz egy megfelelő útvonal megtalálására.

A célom az volt, hogy az alapalgoritmus futásideje kvázi invariáns legyen a térkép komplexitására. Ennek a meghatározására létrehoztam három különböző térképet, amelyeken teszteltem a MatLab beépített algoritmusait. A tesztelés során használt térképek a függelék 10.1 alfejezetében lálhatóak. Az 1-3. táblázatok a különböző komplexitású térképek esetében kapott futásidőket hasonlítja össze.

1. táblázat - Beépített MatLab PRM futásideje különböző komplexitású térképek esetén élhossz=inf (függőleges irányban a gráfot építő csúcspontok száma látható, vízszintes irányban a térkép komplexitásának változása)

PRM	Simple	Medium	Complex
50	0.06 s	0.08 s	0.08 s
150	0.7 s	0.8 s	0.7 s
250	1.7 s	1.7 s	1.9 s
400	5.8 s	2.3 s	4.8 s
500	7.3 s	7.4 s	7.4 s

2. táblázat - Beépített MatLab A futásideje különböző komplexitású térképek esetén (alapbeállítás mellet)*

A*	Simple	Medium	Complex
Default	1.03 s	1.12 s	24.3 s

3. táblázat - Beépített MatLab RRT futásideje különböző komplexitású térképek esetén (függőleges irányban a maximális élhossz szerint)

RRT	Simple	Medium	Complex
25	0.3 s	0.13 s	0.8 s
50	0.17 s	0.1 s	0.5 s
100	0.2 s	0.23 s	0.37 s
250	0.2 s	0.27 s	0.32 s
500	0.18 s	0.32 s	0.6 s

Az Ant Search (Ant Colony) nem került tesztelésre, mert a MatLab nem rendelkezik beépített ant search ütközésmentes útvonaltervező algoritmussal.

A megvizsgált ütközésmentes útvonaltervező algoritmusok közül az A* mutatta a legnagyobb futásidőbeli változást a térkép komplexitásának változásával. Habár a legrövidebb utat ez az algoritmus adja meg, de a bonyolultsága $O(b^d)$, ahol b egy csomóponthoz tartozó átlagos levelek száma (*barnching faktor*), míg d a bejárandó fa mélysége (legrövidebb út) [6]. Az RRT és a PRM viszonylag teljesítette a komplexitásra vett invarianciát. Azonban míg a PRM egy gráfot épít fel a szabad térben, addig az RRT egy fát. Az utóbbi esetében minden mozgatható objektum kiinduló pontjától a hozzá tartozó célpontba kell egy fát növeszteni, valamint minden egyes objektum célpontjától, minden mozgatható objektum kiinduló pontjához kell egy-egy fát növeszteni. Ez viszont több objektum esetén drasztikus futásidőbeli növekedést okoz. Ezzel szemben a PRM egy gráfot épít fel, mely nagyobb szabadságfokot biztosít a gráfot alkotó bármely két csúcs közötti útvonalkeresésre.

Mindezek alapján a PRM ütközésmentes útvonaltervező algoritmust választottam ki, amelyet kibővítek a célkitűzésem megvalósításához. A MatLab beépített PRM algoritmusát nem tervezem felhasználni és módosítani. A probléma megoldására egy saját PRM algoritmust implementálok, amely funkcionalitásában több lesz, mint a beépített társa.

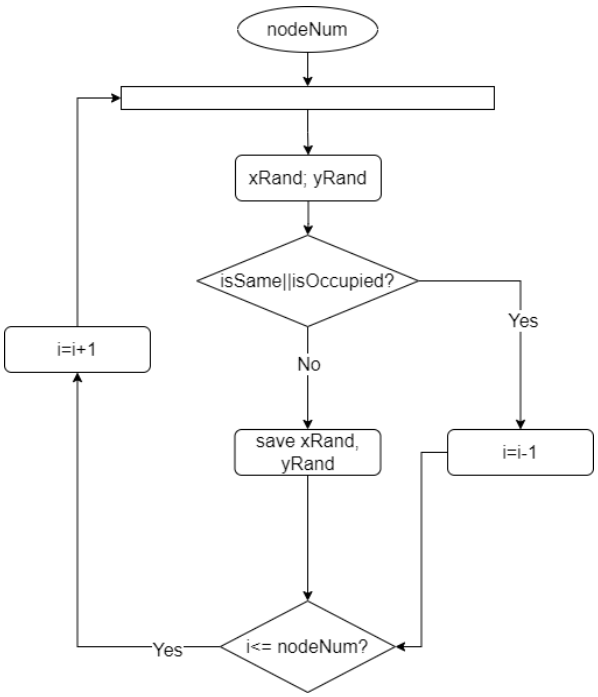
3.3 Saját PRM megvalósítása

A PRM alapvető működése, hogy véletlenszerűen gráfcsúcspontokat vesz fel a térkép szabadon bejárható terében. Majd ezen csúcspontok között éleket vesz fel, így kialakítva egy gráfot. Élet csak akkor vesz fel két csúcspont között, ha az végig a szabadon bejárható térbe esik, vagyis nem keresztez egyetlen objektumot (akadályt) sem.

Az általam megvalósított PRM-t is két nagyobb egység alkotja, amelyek közül az egyik a csomópontok felvételéért felelős, míg a másik az élek felvételét szolgáltatja. Végül ebből a két paraméterből (gráfpontok és gráfélek) felépíthető MatLab-ban az a gráf, amelyben a későbbiekben az útvonalat fogom meghatározni.

3.3.1 Gráfpontok felvétele

A gráfpontok felvételét egy függvény valósítja meg, amely a bemenetként egy térképet és a felveendő csúcspontok számát vár. Végeredményképpen egy struktúrát ad vissza, amely tartalmazza a véletlenszerűen generált pontok azonosítóját, valamint a pontok (x, y) koordinátáit. A függvény folyamatábráját az 1. ábra szemlélteti.



1. ábra - Gráfpontok felvételéért felelős függvény folyamatábrája

Első lépéként a függvény meghatározza a térkép *x* és *y* irányú kiterjedtségét ($térkép\ szélessége, térkép\ magassága$), hogy a random generált csomópontok pozíciói ezen értékeken belül helyezkedjenek el. Ezt követően egy *for* ciklusban véletlenszerűen generálja a gráfpontokat. A ciklusban a következő műveleteket hajtja végre:

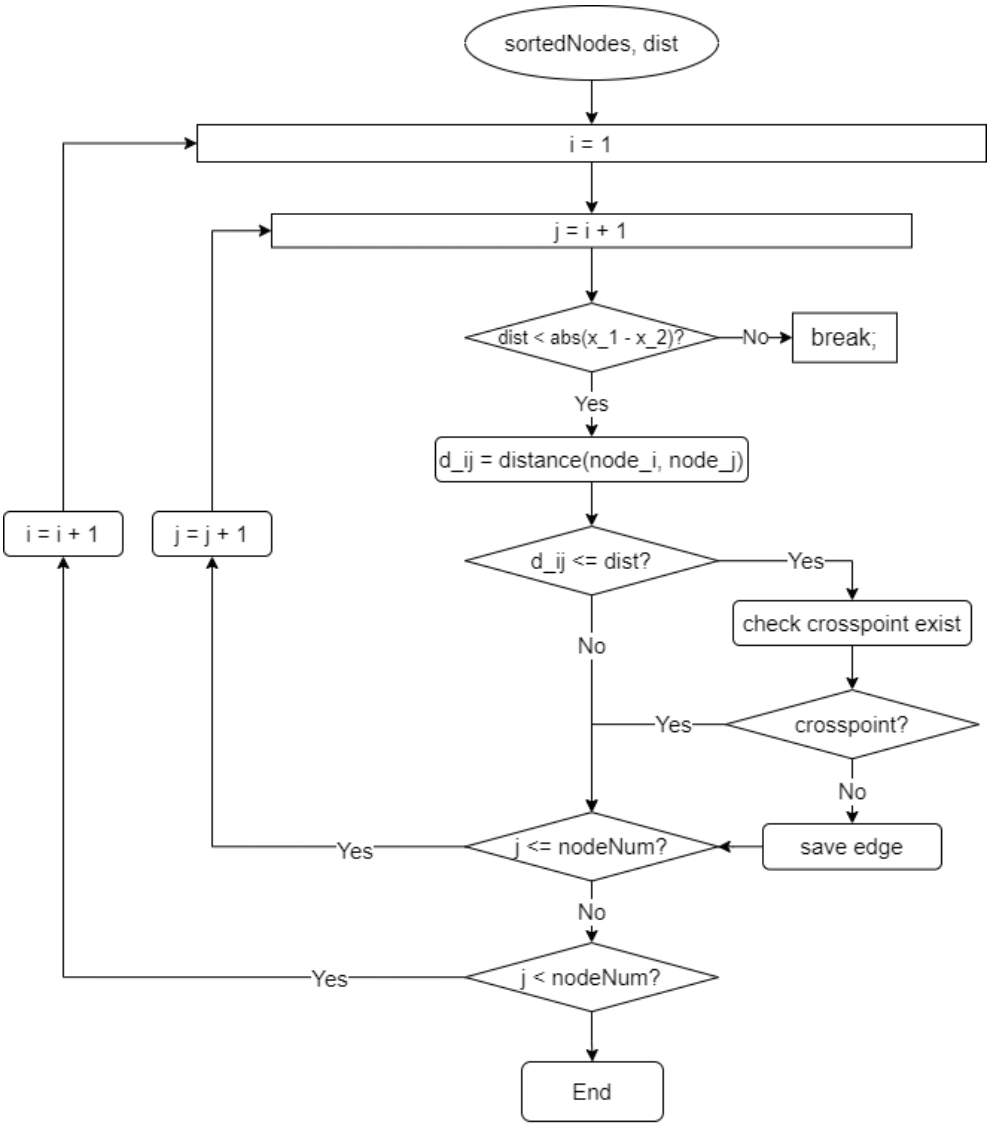
1. Generál véletlenszerű integer számokat a csúcspont *x* és *y* koordinátájának, amik a megfelelő intervallumok között kell elhelyezkedjenek; $xRand \in [1, térkép\ szélessége)$ & $yRand \in [1,\ térkép\ magassága)$.

2. Megvizsgálja, hogy a generált koordinátapáros már létezik-e az eddig generált csúcsok vektorában egy erre a célra létrehozott függvény meghívásával (*isSame*), valamint, hogy a generált koordináta a térképen a szabadon bejárható területre esik-e egy másik célfüggvény függvény meghívásával (*isOccupied*).
    2.1. Amennyiben a 2. pontban említett esetek bármelyike teljesül (már létezik az adott gráfpont vagy nem a szabadon bejárható térbe esik a generált gráfpont) úgy a *for* ciklus iterátorát (i) csökkenteni kell, és a gráfpontot újra kell generálni.
    2.2. Amennyiben a véletlenszerűen generált gráfpont még nem létezik az eddig felvett gráfpontok listájában, akkor a listához történő hozzárendeléssel elmentésre kerül.
3. Ez ismétlődik addig amíg a megadott számú gráfpont (*nodeNum*) nem generálódik le.

3.3.2 Gráf éleinek felvétele

A gráfpontok között felvehető élek meghatározására egy erre a célra dedikált függvény a felelős. Bemenetként egy rendezett gráfpontokat tartalmazó listát vár, valamint a felvehető élek maximális hosszát. Végeredményképpen egy olyan listával tér vissza, amely olyan gráfpontpárokat tartalmaz, amelyek között élet kell felvenni. Ennek a függvények a folyamatábráját a 2. ábra szemlélteti.

A függvény bemenetére érkező gráfpontokat tartalmazó lista (*sortedNodes*) előfeldolgozáson esik keresztül. Az előfeldolgozás során a csúcsokat az *x* pozíciójuk szerint növekvő sorrendbe helyezem, és úgy adom át a függvénynek. Erre azért van szükség, mert így csökkenthető a függvény számításigényét, ugyanis egy elővizsgálattal leszűkíthető egy csomópontra nézve, hogy mely más pontokkal képezhet gráfélet. A leszűkítéshez az *x* koordináta szerinti távolság szolgál alapul. Amennyiben a maximális távolságon (dist) kívül esik egy csúcspont, úgy a többit már figyelmen kívül hagyja a függvény.



2. ábra - Gráf éleit meghatározó függvény folyamatábrája

A folyamatábra alapján a függvény a következő módon határozza meg a felvehető éleket:

1. Két egymásba ágyazott *for* ciklussal az összes lehetséges csúcspontpárost igyekszik megvizsgálni
    1.1. A külső ciklus a gráfpontokat tartalmazó lista első elemétől az utolsó előtti eleméig veszi a pontokat (i)
    1.2. A második ciklus pedig mindig a külső ciklus által indexelt ponthoz képest a listában elhelyezkedő következő ponttól kezdi az indexelést, és egészen az utolsó elemig megy (j)
    1.3. Ezáltal a gráfba nem kerülhet hurokél, valamint egyik él se kerül redundánsan felvételre

2. A következő lépés, hogy a kiválasztott pontok *x* koordinátája szerinti távolság ellenőrzése
    2.1. Ha nagyobb az *x* szerinti távolság, mint a megadott maximális távolság, úgy a belső ciklus befejezi az iterálást (break). Ezt azért teheti meg, mert a csúcspontok *x* szerint vannak növekvő sorrendbe helyezve.
    2.2. Ha kisebb, akkor folytatja a következő lépéssel
3. Kiszámításra kerül a két gráfpont euklideszi távolsága (distance(node_i, node_j)
    3.1. Ha nagyobb ez a távolság, mint a megengedett maximális táv (dist), akkor a következő gráfpontra lép
    3.2. Ha kisebb, akkor folytatja a következő lépéssel
4. Megvizsgálja, hogy a felvenni kívánt él átszel-e bármilyen statikus objektumot (check crosspoint exist)
    4.1. Amennyiben átszel (metsz) ilyen objektumot, úgy az az él elvetésre kerül, és a következő gráfpontra lép
    4.2. Amennyiben nincs metszés, akkor az él elmentésre, hozzárendelésre kerül az éleket tartalmazó listához (save edge)
5. Mindezt addig folytatja, míg az összes lehetséges pontpárra elvégzi.

Fontos megjegyezni, hogy olyan él nem fordulhat elő, amely teljes egészében egy akadályon belül legyen (teljes egészében a nem szabadon bejárható térbe esik), hiszen a gráfpontok úgy lettek felvéve, hogy azok mind a szabadon bejárható térben helyezkedjenek el.

4 Két akadályra történő implementáció

Ebben az esetben a térkép egy képformátumú állomány (monokromatikus bitmap), amit a MatLab beolvasás után mátrix formában tárol el. Az eltárolt mátrix minden egyes eleme a kép egy pixelének feleltethető meg. A mátrix elemei *0, 1* értékűek lehetnek, ahol a nullák szemléltetik a nem szabadon bejárható térhez tartozó pixeleket. A következő fejezetben bemutatott függvények mindegyike úgy került megvalósításra, hogy ilyen típusú térképet tudjon feldolgozni.

4.1 Kiegészítő függvények az általam implementált PRM-hez

4.1.1 Gráfpont létezésének ellenőrzése

Ezt egy függvényen keresztül valósítottam meg, amely bemenetként az eddig felvett gráfpontok listáját és az ellenőrzésre szánt koordinátapárost várja. Visszatérési értéként egy logikai értéket ad, hogy létezik-e a kérdéses *x, y* koordinátával már gráfpont a listában. Ez úgy került megvalósításra, hogy egy *for* ciklus segítségével megvizsgálom az eddig felvett összes pontot. Amennyiben egyezést találok, akkor a ciklusból kilépek és logikai *igaz* értéket állítok be a visszatérési értéknek, másképpen hamis értékkel tér vissza.

4.1.2 Gráfpont elhelyezkedésének ellenőrzése

Ezt is egy külön függvény határozza meg, amely bemenetként azt a térképet várja, amelyen ellenőrizni kell a kérdéses koordinátapárost. A térképen kívül természetesen szükség van az *x, y* koordinátákra is. Visszatérési értékként megadja, hogy a kérdéses pont a szabadon bejárható térben helyezkedik el vagy sem. Ezt úgy határozza meg, hogy a megfelelő indexű elemét lekéri a térképet reprezentáló mátrixnak. Ha az *0* értékű, akkor nem a szabadon bejárható térben helyezkedik el, és el kell majd vetni a koordinátapárost, míg *1* érték esetén megtartható a koordinátapáros.

4.1.3 Gráfél és nem a szabadon bejárható térbe eső objektum metszéspontjának ellenőrzése

Egy dedikált függvény határozza meg, hogy található-e metszés gráfél és akadály között. Bemenetként két egyenest leíró pontpárost vár, amit az alábbi képlet szemléltet.

$$L_1 = \begin{pmatrix} x_{1_0} & x_{1_1} \\ y_{1_0} & y_{1_1} \end{pmatrix}, L_2 = \begin{pmatrix} x_{2_0} & x_{2_1} \\ y_{2_0} & y_{2_1} \end{pmatrix}$$

Az L_1, L_2 azok az egyenesek, amelyek között a metszést vizsgálni kell. Az L_1 egyenes a két gráfpont között felvenni kívánt élet reprezentáló egyenes. Ennek paraméterei x_{1_0}, y_{1_0} a kiinduló gráfpont pozícióját jelentő koordináták, valamint x_{1_1}, y_{1_1} a cél gráfpont pozícióját jelentő koordináták. Az L_2 egyenes az akadályokhoz tartozó határolóvonalak egyike. Az x_{2_0}, y_{2_0} $és$ x_{2_1}, y_{2_1} azok a pontok, amelyek az említett L_2 egyenest meghatározzák. Az akadályokat határoló vonalak egy másik függvény által kerülnek meghatározásra. Ez a függvény, amely a határoló vonalakat adja vissza, minden akadályhoz négy burkoló egyenest határoz meg a négyszögű akadályok éle szerint. Ezeket a metszetkereső függvény használja fel úgy, hogy egy gráfélhez megvizsgálja az összes határolóegyenest, hogy mellyel képez metszetet.

A metszet vizsgálatára a lineáris algebra eszközeit használtam fel. A metszésponthoz tartozó P_x, P_y koordinátát a következő képlettel számoltam ki:

$$D = \begin{vmatrix} \begin{vmatrix} x_{1_0} & 1 \\ x_{1_1} & 1 \end{vmatrix} & \begin{vmatrix} y_{1_0} & 1 \\ y_{1_1} & 1 \end{vmatrix} \\ \begin{vmatrix} x_{2_0} & 1 \\ x_{2_1} & 1 \end{vmatrix} & \begin{vmatrix} y_{2_0} & 1 \\ y_{2_1} & 1 \end{vmatrix} \end{vmatrix}$$

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} = \begin{pmatrix} \begin{vmatrix} \begin{vmatrix} x_{1_0} & x_{1_1} \\ y_{1_0} & y_{1_1} \end{vmatrix}^T & \begin{vmatrix} x_{1_0} & 1 \\ x_{1_1} & 1 \end{vmatrix} \\ \begin{vmatrix} x_{2_0} & x_{2_1} \\ y_{2_0} & y_{2_1} \end{vmatrix}^T & \begin{vmatrix} x_{2_0} & 1 \\ x_{2_1} & 1 \end{vmatrix} \end{vmatrix} \\ \begin{vmatrix} \begin{vmatrix} x_{1_0} & x_{1_1} \\ y_{1_0} & y_{1_1} \end{vmatrix}^T & \begin{vmatrix} y_{1_0} & 1 \\ y_{1_1} & 1 \end{vmatrix} \\ \begin{vmatrix} x_{2_0} & x_{2_1} \\ y_{2_0} & y_{2_1} \end{vmatrix}^T & \begin{vmatrix} y_{2_0} & 1 \\ y_{2_1} & 1 \end{vmatrix} \end{vmatrix} \end{pmatrix} * \frac{1}{D}$$

A fenti képlettel számolva a megoldás során nincs figyelembe véve, hogy a pontok által meghatározott egyenesek valójában intervallumokat jelentenek, és nem végtelen hosszú

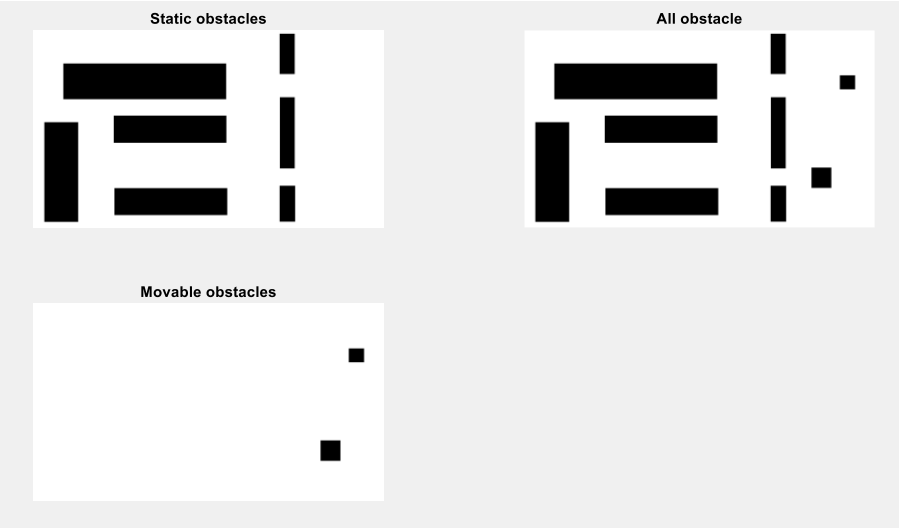
egyeneseket. A képlet mindig talált metszéspontot kivéve, ha a tökéletesen párhuzamosak az egyenese. Viszont szükséges volt megvizsgálni, hogy a metszéspont a megfelelő intervallumok között helyezkedik-e el, mert ha az intervallumon kívül esik, akkor a metszéspont a függvény szempontjából elhanyagolható.

4.2 Az útvonal tervezése

Két mozgatható akadályra történő megvalósítás során a sorrendet „hard coding”-al valósítottam meg, tehát az objektumok mozgatásának összes lehetséges esetében meghatároztam az egyes útvonalakat. Az így kapott megoldásokból a rövidebb úthosszal rendelkező került kiválasztásra. Belátható, hogy ezzel a módszerrel nem érdemes több akadályra útvonalat tervezni, ugyanis a mozgatható objektumok mozgatási sorrendje faktoriálisan növekszik. Ez azt jelenti, hogy két mozgatható akadály esetén a kiszámítandó esetek száma $2! = 2$, három akadály esetén $3! = 6$, négy akadályra már $4! = 24$ és így tovább. Annak ellenére, hogy a futásidőt nem kezelem kritikus paraméterként, hat mozgatható akadály esetén (ha egy permutáció kiszámítása 1 s) akkor az összes kiszámítása 720 s-t vesz igénybe. Ez pedig nagy időveszteség, miközben hat darab mozgatható akadály kevésnek tekinthető. A későbbiekben ezt egy mesterséges intelligenciával optimalizálom.

Az algoritmus ebben az esetben bemenetként a felépített gráfot, az elhelyezett csúcspontokat, a mozgatható objektumok azonosítóját, a statikus akadályokat tartalmazó térképet, az akadályokat és a maximális éltávolságot várja.

Ennél az esetnél még felhasználói kezelőfelület nélkül történt az algoritmus futtatása. Ebből kifolyólag a mozgatható objektumokat két térkép alapján határoztam meg. Az egyik térkép csak a nem mozgatható akadályokat tartalmazta, míg a másik az összes akadályt. Egy kizáró vagy (XOR) művelettel a két térképből egyszerűen meghatározhatóak a kívánt objektumok. Ennek a menetét a 3. ábra szemlélteti.



3. ábra - Mozgatható objektumok meghatározása XOR művelettel

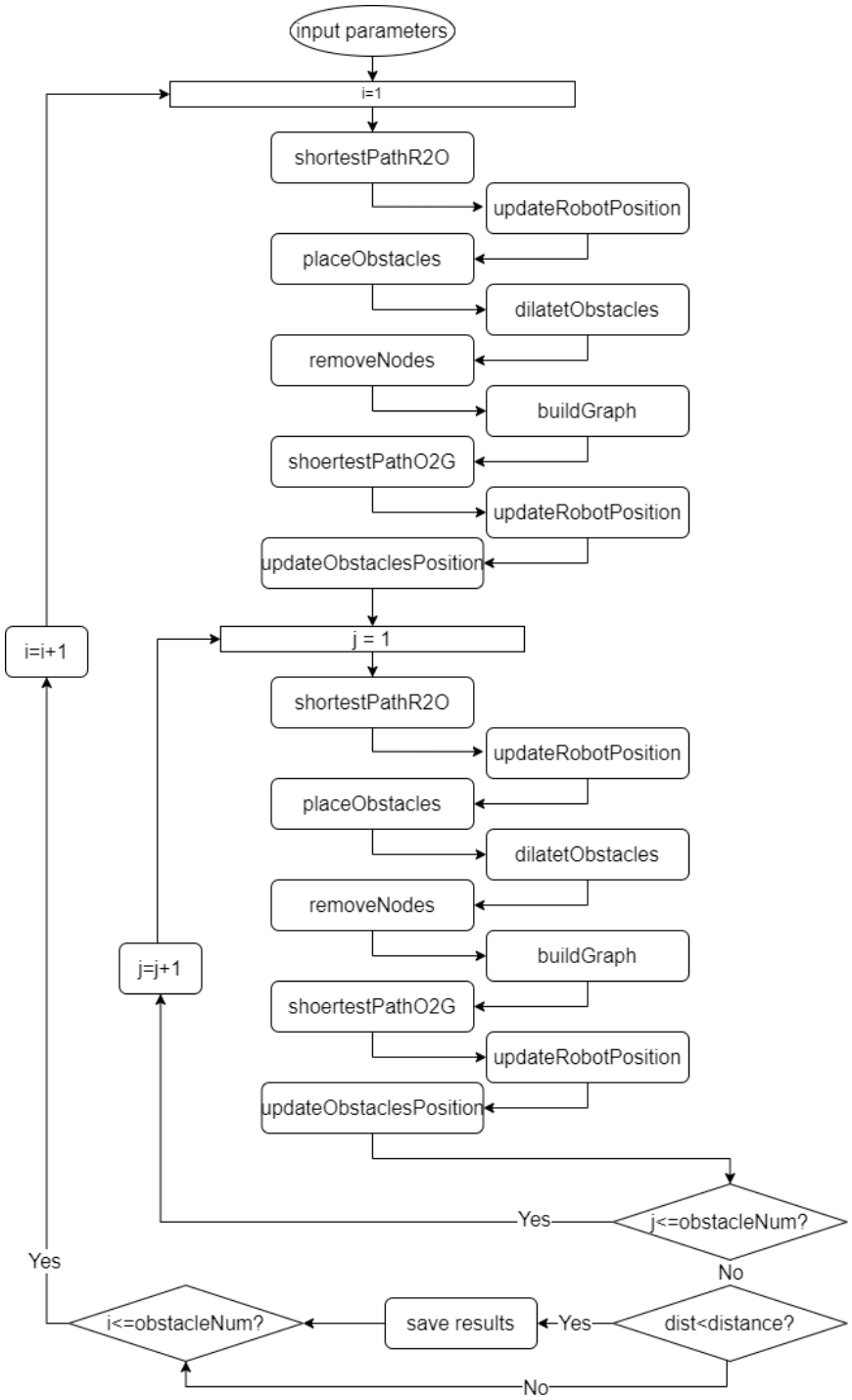
Az így kapott képen (amely csak a mozgatható objektumokat tartalmazza) egy objektumkereső függvény segítségével meghatározom a kérdéses objektumok pozícióját, valamint kiterjedtségét. Ezt követően a tömegközéppontjukhoz rendelek egy-egy pontot, amelyet hozzáadok ahhoz a pontvektorhoz, amely a gráf generálásához tartozó csomópontokat tartalmazza. Egyúttal a robot kezdeti pontjának a koordinátáit is az említett pontvektorhoz rendelem, valamint az objektumoknak a célpontját is hozzárendelem. Ezeket a pontokat a pontvektor végéhez füzöm úgy, hogy a vektorhoz először a robot kiindulópontját rendelem hozzá, majd az egyes objektumokat, végül a megfelelő sorrendben az objektumok célpontját.

Az algoritmus először a mozgatható objektumok azonosítójából kiválasztja azokat a node-okat, amelyek a kezdeti pozícióra vonatkoznak, majd a célpontokra vonatkozókat is. Az algoritmus úgy került megvalósításra, hogy nem tudja kezdetben mennyi mozgatható objektum helyezkedhet el a térképen. Ebből kifolyólag a mozgatható objektumok külön kerül meghatározásra, majd két egymásba ágyazott *for* ciklussal megvizsgálom a lehetséges mozgatási sorrendeket. A sorrendek meghatározásánál folyamatosan kerül kiszámolásra a megtett út és az, hogy milyen útvonalon juttatható el a robot a kívánt pontba.

Először kiválasztásra kerül az egyik mozgatható objektum. A robot dimenziója elhanyagolásra került, így egyszerűen a *shortestpath* függvény meghívásával határozom meg az első objektumhoz vezető utat. Elmentésre kerül az eddigi útvonal, és a megtett úthossz. Ezt követően a robot pozícióját frissítem az első objektum pozíciójára. A következő lépésben az érintetlenül hagyott mozgatható objektumot a statikus akadályokat tartalmazó térképre helyezem. Erre azért van szükség, mert a következő mozgatás során az első objektumra nézve a másik mozgatható objektum is statikus akadályként tekinthető. A mozgatni kívánt

objektumnak meghatározom a hosszabbik dimenzióját, amit strukturáló elemként használok fel, hogy a térképen lévő összes statikusnak tekinthető akadályt (beleértve a másik mozgatható objektumot is) megfelelő mértékben megnöveljem. Ezzel a művelettel elértem, hogy a dilatált akadályok révén pontszerűnek tekinthető a mozgatni kívánt objektum. Miután megnöveltem az akadályokat, meghatározom azokat a gráfcsúcspontokat, amelyek az objektumnövelés miatt már nem a szabadon bejárható térbe esnek. Ezeket a csúcspontokat elveszem az eddigi listából és újra generálom a teljes gráfot. Az újra generált gráfon meghatározom az útvonalat az objektum és az objektumhoz tartozó célpont között. Végül növelem a megtett út hosszát, bővítem a megtett útvonalat, valamint frissítésre kerül a robot pozíciója az aktuálisan áthelyezett objektum célpontjára. A mozgatható objektumokra vonatkozó információkat tartalmazó struktúrában az első objektumra vonatkozó adatokat módosítom.

A következő lépésben a második akadályra is elvégezzük az előbb említett műveleteket. A robot jelenleg az első mozgatható objektum célpontjában helyezkedik el. Ebből a pozícióból kell a második mozgatható objektumhoz juttatni. Ehhez az eredeti gráfon keressük meg az útvonalat, ugyanis a robot dimenziói elhanyagolásra kerültek. Amennyiben a robot sikeresen a második mozgatható akadályhoz jutott, úgy frissítésre kerül a pozíciója, növelésre kerül a megtett úthossz és bővítésre az útvonal. Ezt követően hasonlóan az előző esethez, elhelyezésre kerül a statikusnak tekinthető mozgatható objektum (ez az előzőekben a célpontjába juttatott objektum), egy megfelelő méretű strukturáló elemmel megnövelek minden akadályt, és eltávolítom a megfelelő csúcspontokat. Legenerálom az új gráfot és azon meghatározom az utat a célpontba, valamint frissítem a megfelelő paramétereket (az eddig megtett út hosszát, a bejárt utat). Az ehhez tartozó folyamatábrát a 4. ábra szemlélteti.

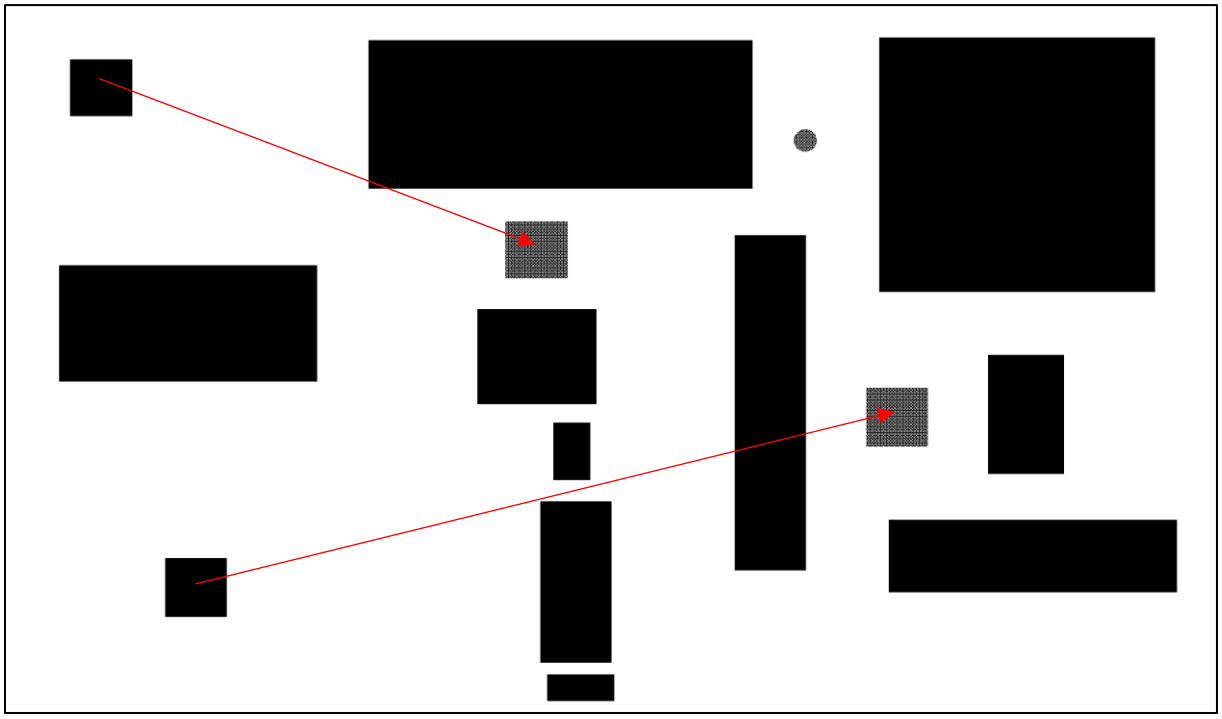


4. ábra - Az algoritmus folyamatábrája

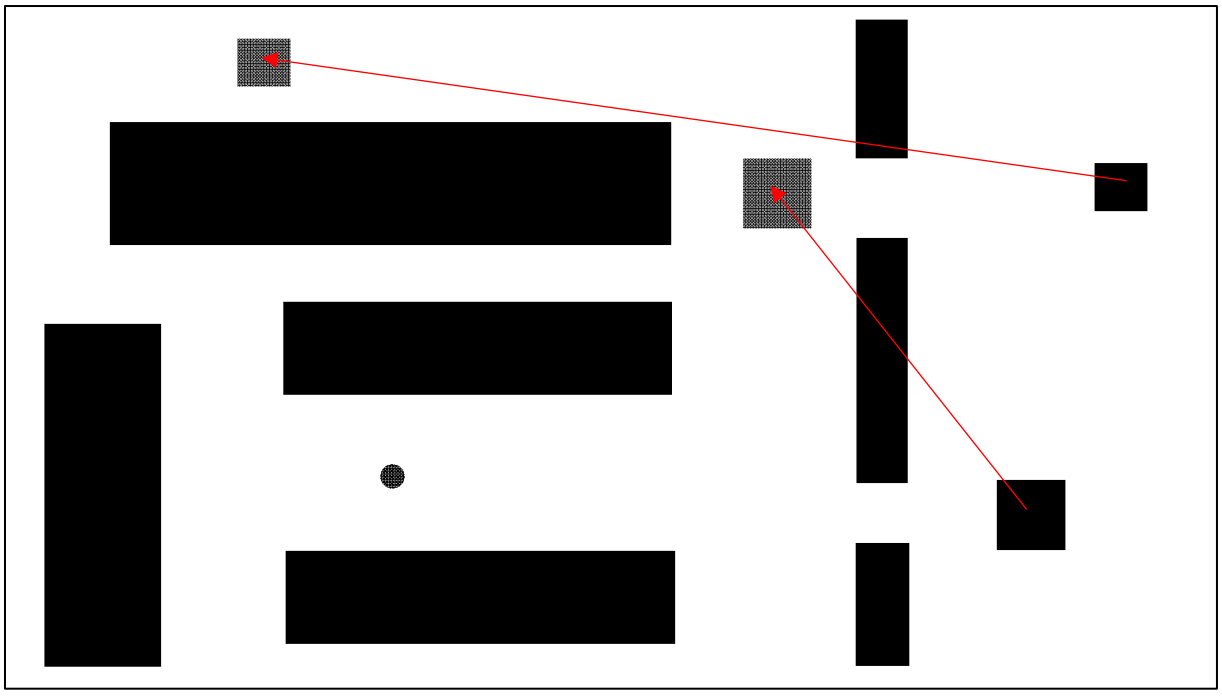
Végezetül megvizsgálom, hogy a két sorrend közül melyik rendelkezik rövidebb úttal, és azt kiválasztva a függvény visszatér a megtett úthosszal, és a gráfon bejárt útvonallal.

5 Eredmények két akadályra

A megvalósított algoritmust két különböző, akadályokkal rendelkező térképen teszteltem. A térképek az 5. és 6. ábrákon láthatóak.



5. ábra - Tesztelésre készített első térkép



6. ábra - A tesztelésre készített második térkép

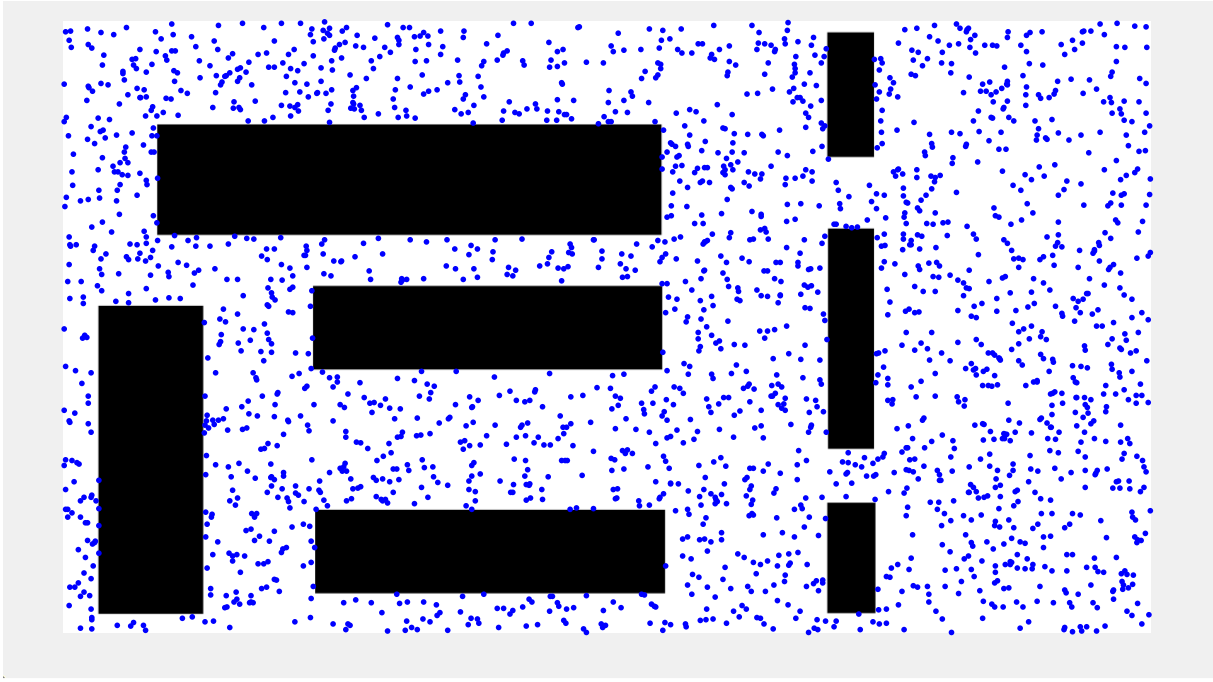
A mozgatható objektumok az ábrákon látható fekete objektumok, melyekből egy piros nyíl indul ki. A mozgatható objektumok célpontjai a szürke akadályok, melyekbe egy-egy piros nyíl mutat. A piros nyilak szemléltetik, hogy honnan hova kellett eljuttattatni az adott objektumot. A fekete-szürke kör pedig a robot kiindulási pozícióját jelöli.

A két térkép két különböző speciális esetet vizsgál meg. Az első térképen úgy kell mozgatni az objektumokat, hogy azok közül az egyik elzárja a másik egyetlen útvonalát, amelyen keresztül a célpontba érhet. Ezért csak egy lehetséges sorrendben valósítható meg az objektumok mozgatása. A második térkép úgy került kialakításra, hogy az egyik objektumnak a célpozíciója a másik objektum legrövidebb útját zárja el. Viszont, ha a gráf úgy kerül generálásra, hogy az alsó résen nem férhet át az objektum, akkor amennyiben rossz sorrendben kerülnek mozgatásra az objektumok, úgy a felső mozgatható objektum egyetlen útját is elzárhatja. Ellenkező esetben, ha a gráf úgy kerül felvételre, hogy az alsó résen is átmozgatható a felső akadály, akkor a két lehetséges sorrend közül mind a kettő jó megoldást tud adni. A két lehetőség közül az egyik mindig rövidebb útvonallal fog rendelkezni, és ezt a rövidebb útvonallal rendelkező sorrendet kell visszaadnia az algoritmusnak.

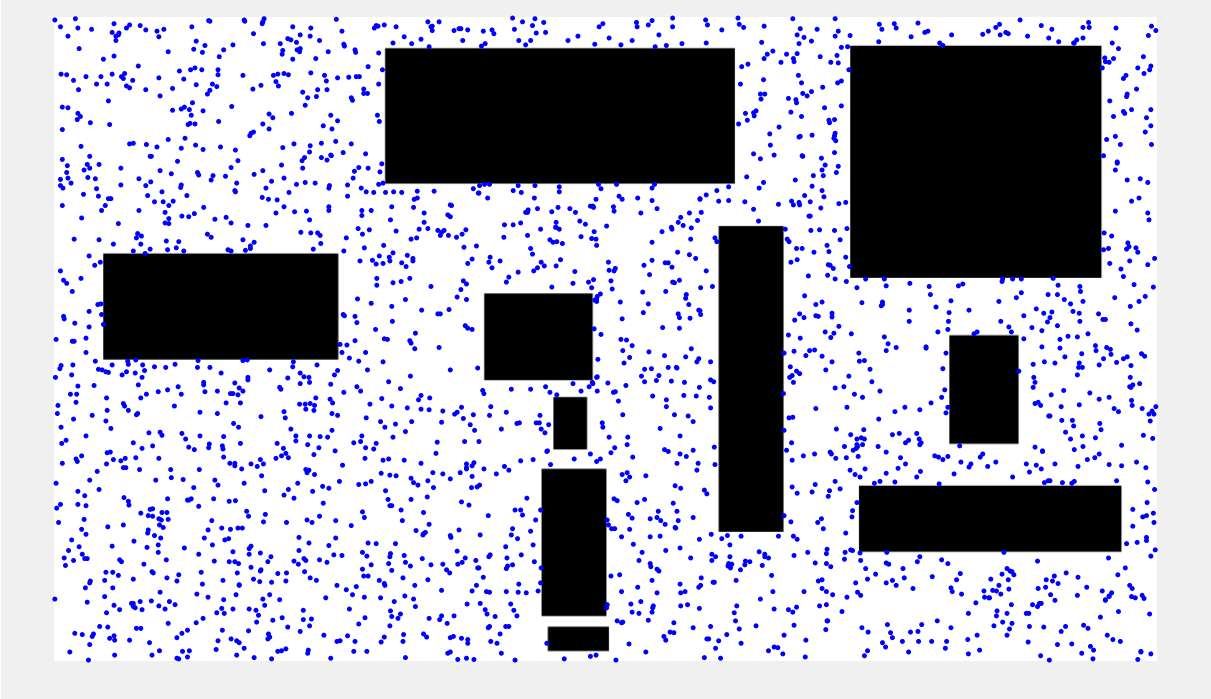
Mielőtt a kész algoritmus bemutatásra kerülne, a következő alfejezetekben a két akadály mozgatásához tartozó egyes függvénykomponensek eredményei kerülnek bemutatásra.

5.1 Random csúcspontok generálása

A 7. és 8. ábra a gráfpontok felvételét szemlélteti két akadály esetében a korábban bemutatott két térképen.



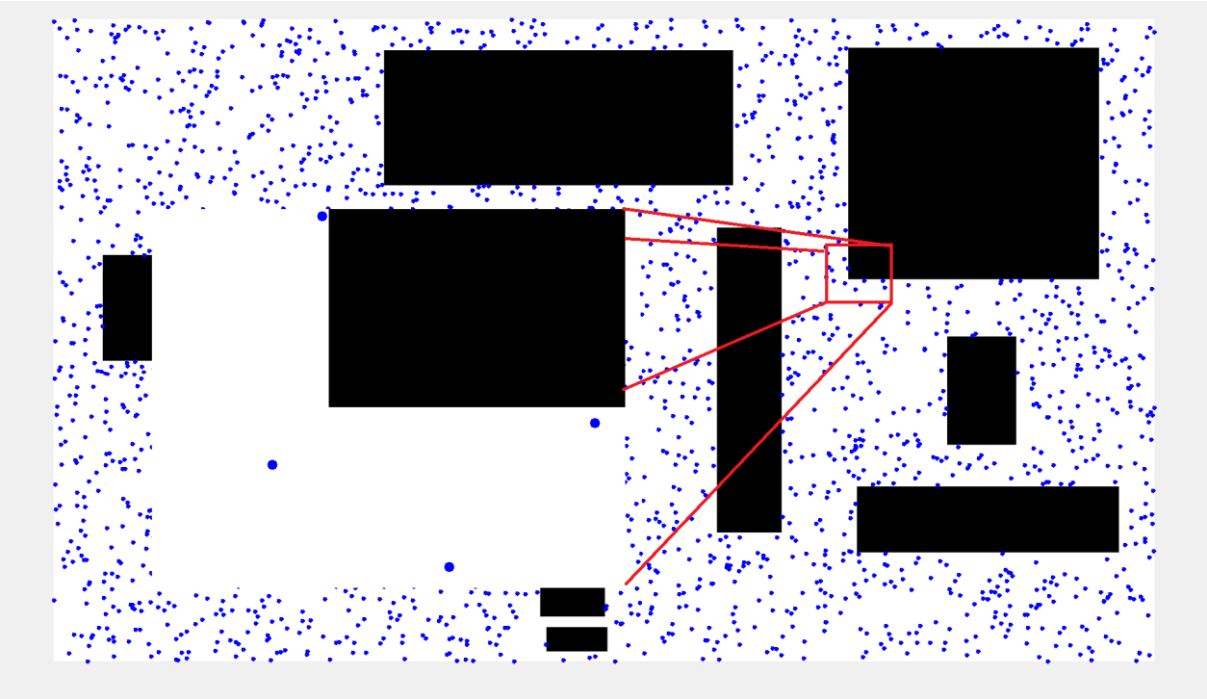
7. ábra - Első térképen felvett 2250 darab csomópont



8. ábra - Második térképen felevett 2250 darab csomópont

A két ábrán látható, hogy a felvett csomópontok mind a szabadon bejárható térbe esnek. A generált random koordináták egyenletes eloszlást követnek, ugyanis megközelítőleg egyenletesen fedik le a 2D-s teret (kivéve a nem szabad területeket). Esetenként tapasztalható kivétel, ahol ugyanis az emberi szem úgy érzékeli, mintha egy-egy csomópont a szabadon bejárható tér és egy objektum határán helyezkedne el. Ez azért van, mert a térképek viszonylag

nagy felbontásúak voltak (1000x666 pixel). Ha belenagyítunk a térképbe, akkor egyértelműen látható, hogy a csomópontok nem esnek objektum által határolt területen belülre. Ezt a 9. ábra szemlélteti.

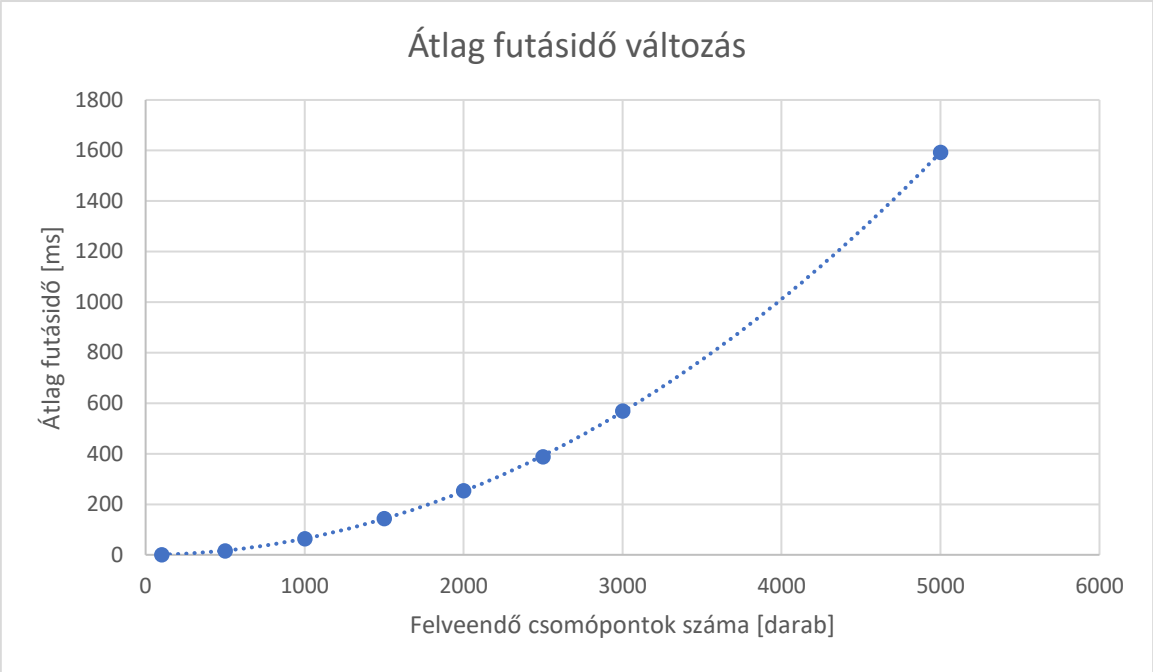


9. ábra - Csomópontok kiemelése az objektum és a szabadon bejárható tér határán

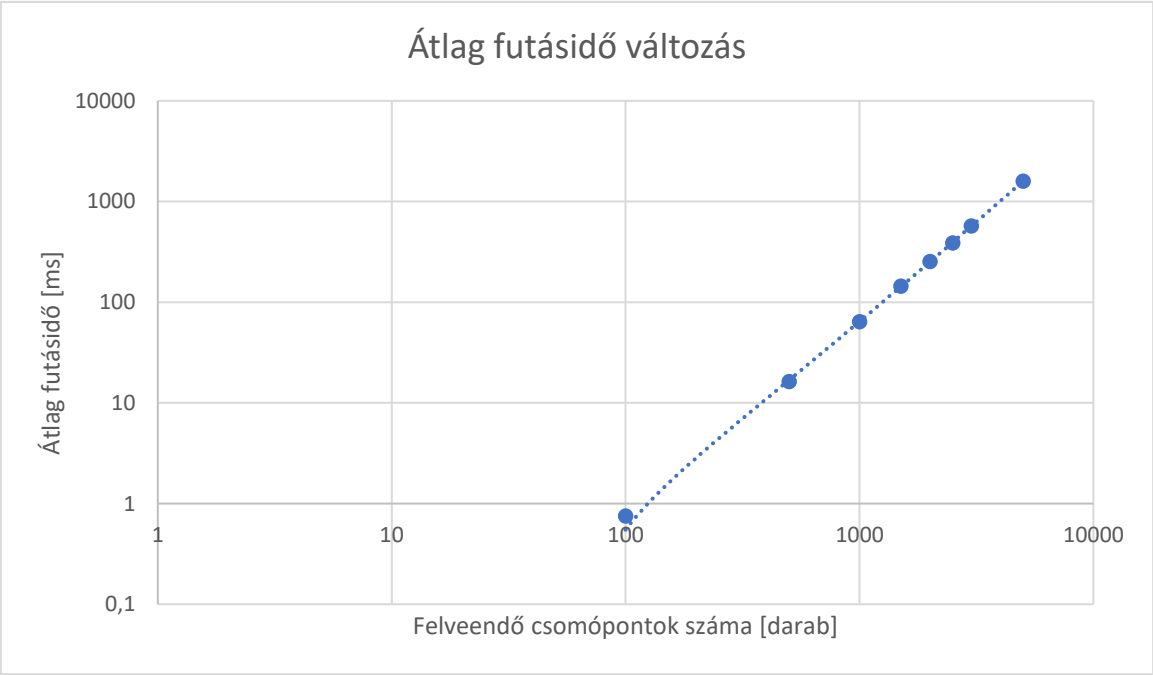
A következőkben a csomópontok felvételéhez szükséges átlagos futásidőt szemléltetem, különböző számosságú pont felvétele esetén.

4. táblázat - Futásidő változása a felveendő csomópontok számától függően

Csomópontok száma	100	500	1000	1500	2000	2500	3000	5000
Átlagos futásidő [ms]	0.75	16.28	63.99	143.77	253.46	397.74	568.67	1592.43



10. ábra - Futásidő változása lineáris skálázással (lin-lin)



11. ábra - Futásidő változása logaritmikus sálázással (log-log)

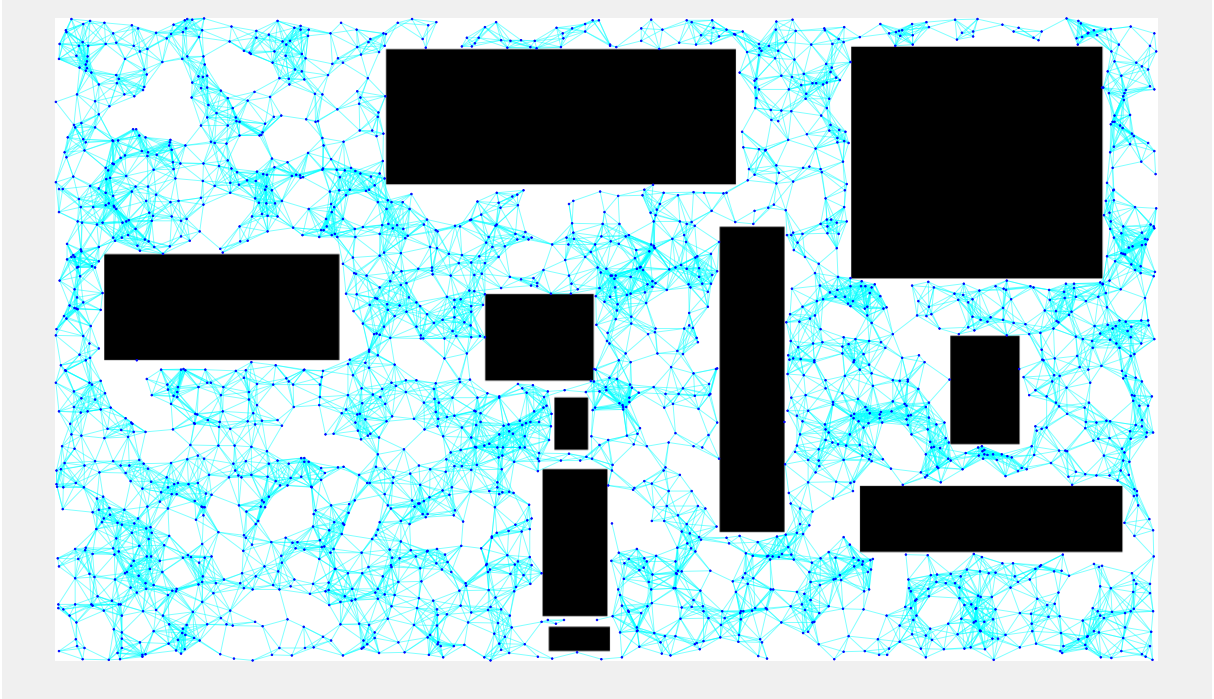
A fenti két ábra jól szemlélteti, hogy a csomópontok felvételéhez tartozó átlagos futásidő a felveendő pontok számosságával nem lineárisan növekszik. Ez azzal magyarázható, hogy több csomópont felvétele esetén, egyre nagyobb listával kell összevetni az új véletlenszerűen generált csomópontot, ami nagyobb számítási szükségletet eredményez.

A futásidő csökkentése érdekében, amikor egy új gráfpont meglétét vizsgálom a már meglévő listában, akkor csak abban az esetben iterárlok végig a meglévő pontokon, amennyiben nem talál egyezést a függvény. Amennyiben egyezést talál, úgy egy flag-gel jelzem, hogy volt egyezés, és az iterációból egy *break* utasítással kilépek. Ezzel a megoldással egyezés esetén csökken a számítási szükséglet, de romlik az algoritmus futásidejének determinisztikussága.

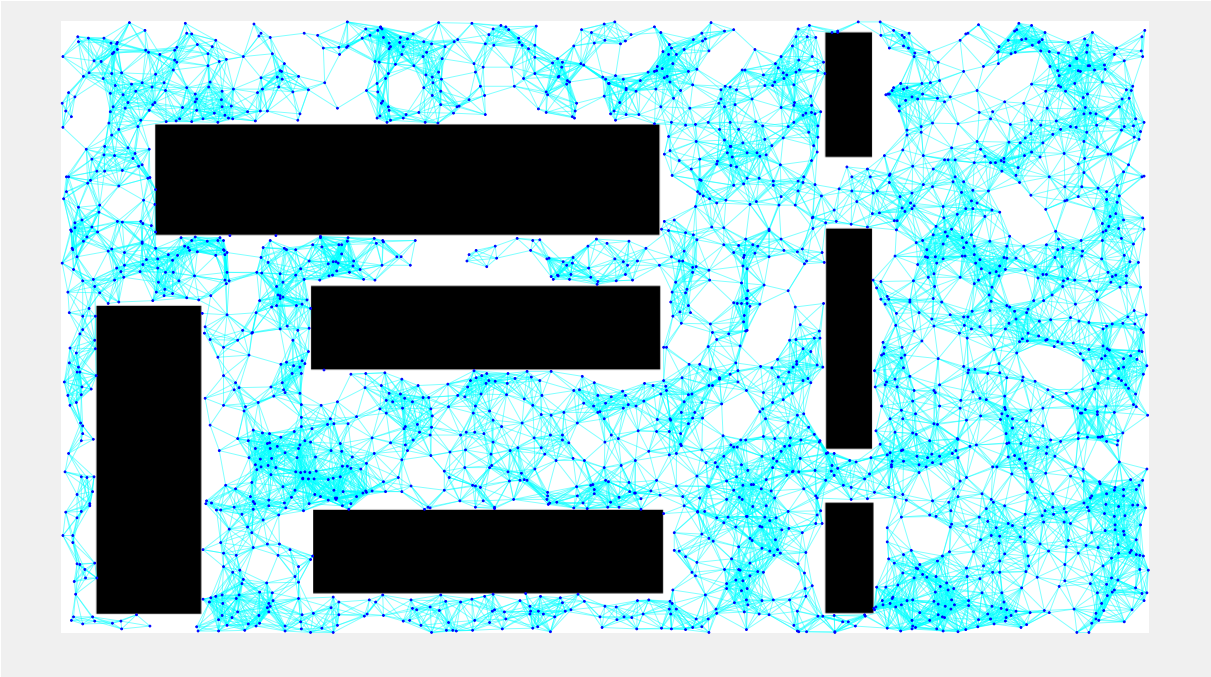
Következtetésképpen azt vonhatjuk le, hogy nem feltétlenül érdemes túl sok csomóponttal dolgozni, mert az algoritmus szemszögéből nézve egy kisebb feladat a pontok generálása.

5.2 Gráf éleinek felvétele

A gráfot alkotó csomópontok felvétele után az éleket kell felvenni, amit egy erre dedikált függvény valósít meg. Ezt a függvényt is teszteltem az ebben a fejezetben említett két térképre. A futásidő tesztelésnél szükséges volt, hogy külön táblázatokra válasszam szét, mert ebben az esetben két paramétert kell hangolni. A térképek kiterjedtsége 1769x1032 pixel volt mindkét esetben. Ez az adat azért fontos, mert befolyásolja, hogy mekkora a minimális élhossz, amivel érdemes az algoritmust futtatni.



12. ábra - Az első térképen generált gráf, 2250 darab csomóponttal 50-es élhosszal



13. ábra - A második térképen generált gráf. 2250 darab csomóponttal 50-es élhosszal

A következő táblázatok szemléltetik a futásidő alakulását, különböző élhosszok, és csomópontszámok mellett. A táblázatok fejlécében található *n* paraméter az adott teszteléshez tartozó csomópontok számát reprezentálja. Azokban az esetekben, amelyeknél a csomópont-élhossz párosítás „értelmetlen" értékű, ott egy „-" -t tettem. Értelmetlen érték azt jelenti, hogy az adott dimenziójú térkép esetén nem generálódik egy megfelelő gráf. Ez jellegzetesen a kevés csomópont, és kis élhossz párosítások esetén fordul elő.

5. táblázat - Futásidő változása az élhosszok függvényében 100 darab csomópont esetén

Élhosszok (n = 100)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	-	-	0.0668	0.205	0.408	0.753

6. táblázat - Futásidő változása az élhosszok függvényében 500 darab csomópont esetén

Élhosszok (n = 500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	-	0.901	1.658	5.387	9.713	18.998

7. táblázat - Futásidő változása az élhosszok függvényében 1000 darab csomópont esetén

Élhosszok (n = 1000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	1.076	3.623	6.935	20.924	38.928	75.687

8. táblázat - Futásidő változása az élhosszok függvényében 1500 darab csomópont esetén

Élhosszok (n = 1500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	2.368	7.759	15.902	45.693	90.187	171.886

9. táblázat - Futásidő változása az élhosszok függvényében 2000 darab csomópont esetén

Élhosszok (n = 2000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	4.216	14.198	27.416	81.367	157.051	312.663

10. táblázat - Futásidő változása az élhosszok függvényében 2500 darab csomópont esetén

Élhosszok (n = 2500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	2.048	6.604	22.591	43.154	130.066	254.08	480.634

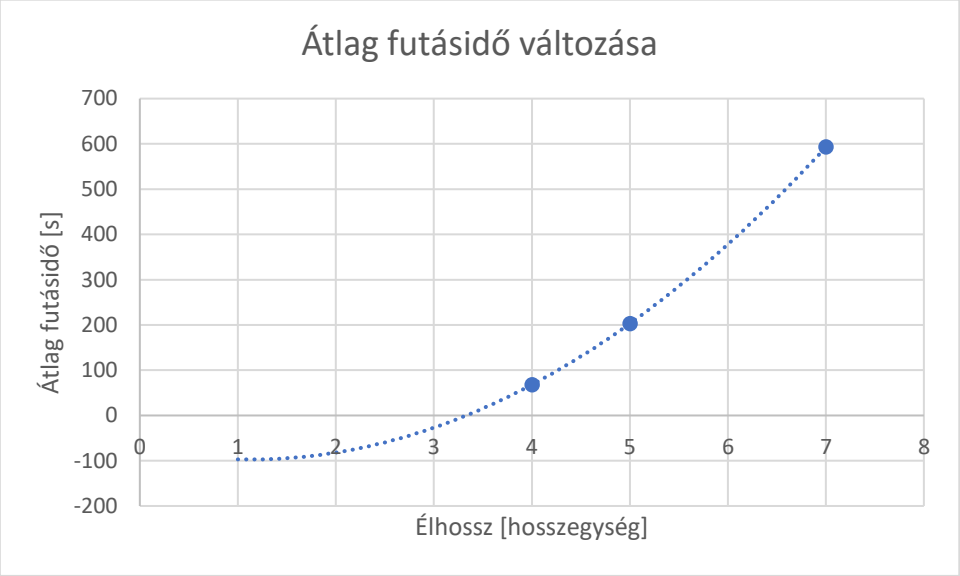
11. táblázat - Futásidő változása az élhosszok függvényében 3000 darab csomópont esetén

Élhosszok (n = 3000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	2.923	9.348	31.025	62.187	186.578	360.026	686.281

12. táblázat - Futásidő változása az élhosszok függvényében 5000 darab csomópont esetén

Élhosszok (n = 5000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	8.149	26.326	88.783	174.685	516.103	1007.193	1910.468

A fenti táblázatokban megfigyelhető, hogy az élhosszok növelésével nagy mértékben növekedett a futásidő is. A futásidőt az előző fejezethez hasonlóan itt is szemléltetem a 14. ábra-n, de csak 2500 darab csomópont esetén.



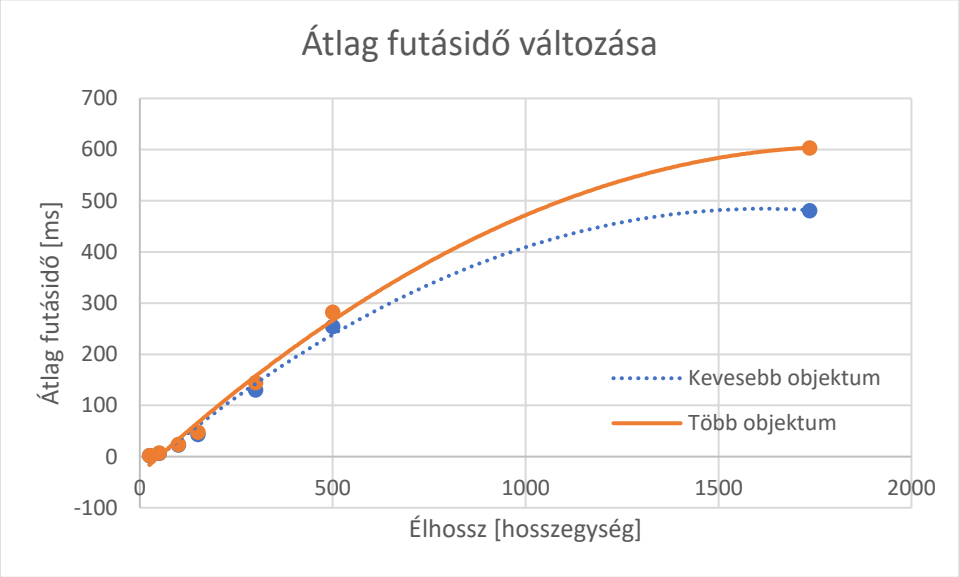
14. ábra - Futásidő változása az élhosszúság szerint 2500 darab csomópont mellett (lin-lin)

Jelen esetben egy másodrendű függvénnyel közelítettem. Az így kapott görbe jól megközelíti a számolt átlagértékeket.

Mivel két különböző térképen végeztem el a gráf felvételének futásidejét, ezért egy fontos megfigyelésre jutottam. A két térkép különbözik akadályainak számában is, így az amelyiken több akadály található több akadályt körülhatároló egyenest is tartalmaz. Gráfél felvételénél pedig ezekkel az egyenesekkel vizsgálok metszéspontot egy iterációval. Amennyiben több egyenessel kell megvizsgálnom a metszéspontot, úgy hosszabb lesz a futásidő. Ez kis számosságú csomópont mellett nem feltűnő eltérés, de nagyobb számosságúnál már igen. Ezt szemlélteti a következő táblázat és a 15. ábra.

13. táblázat - Több akadály esetén az élhossz függvényében vett futásidők 2500 darab csomópont mellett

Élhosszok (n = 2500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	2.128	7.073	24.147	47.953	144.439	282.486	603.028



15. ábra - Futásidőváltozás összehasonlítása 2500 darab csomópont mellett különböző számosságú akadályok esetén

A továbbiakban az akadályok számától való függőséget szeretném kiküszöbölni, amennyiben ez lehetséges.

Végül egy összehasonlítottam az általam megvalósított PRM, és a MatLab beépített PRM-e között. Futásidőre való tekintettel a következő eredményeket kaptam azonos térkép, csomópontszám és élhosszra véve.

14. táblázat - Az általam megvalósított és a beépített PRM összehasonlítása

	Saját PRM futásideje [s]	Beépített PRM futásideje [s]
n=50, *dist*=inf	0.439	0.0673
n=100, *dist*=inf	2.061	0.193
n=200, *dist*=inf	3.648	1.004
n=300, *dist*=inf	11.328	3.683
n=400, *dist*=inf	18.376	6.861
n=500, *dist*=inf	21.279	7.937

Az látható, hogy kis csomópont mellett relatív nagy a futásidőbeli eltérés a beépített PRM javára, de a csomópontok számának növelésével a relatív eltérés csökken. Habár az abszolút eltérés folyamatosan növekszik. Ez viszont nem jelent akkora problémát, ugyanis offline futtatható algoritmus tervezése a cél. Ennek ellenére fontos volt szem előtt tartani, hogy az algoritmus futása a perces nagyságrendbe essen. Ebből kifolyólag a továbbiakban is az általam megvalósított PRM-t használtam fel, mert a skálázhatóbban került implementálásra, mint a MatLab beépített PRM-je.

5.3 Az útvonal tervezése

Ebben a fejezetben a két mozgatható akadályra elkészült algoritmus kerül bemutatásra, az algoritmus lépései, valamint az eredményei alapján. Az algoritmus az 5.3.2 fejezetben került bemutatásra, viszont első lépésként a paraméterek előkészítése kerül bemutatásra

5.3.1 Paraméterek előkészítése

Mivel a két mozgatható objektumos esetben nem valósítottam meg kezelői felületet, a szükséges paramétereket kóddal olvastam be, vagy deklaráltam. A szükséges paraméterek a következők voltak:

- Az összes akadályt tartalmazó térkép
- A statikus akadályokat tartalmazó térkép

- Robot kiindulási pozíciója
- Mozgatható objektumok célpozíciójának koordinátái
- PRM-hez tartozó paraméterek (csomópontok száma, maximális élhosszúság)

A szükséges paraméterek feldolgozása a következőféleképpen történt. Először az akadályokat tartalmazó térképen kerestem meg az objektumokat és azok tulajdonságait egy struktúrába tároltam el. A struktúra a következőket tartalmazta minden objektumról

- Az objektum hosszúsága
- Az objektum szélessége
- Az objektum tömegközéppontjának koordinátái
- Az objektum bal alsó sarkának koordinátái

Ezt követte a gráfot alkotó csomópontok felvétele, amelynek a bemutatása a 5.1 fejezetben található, és a hozzá tartozó eredményeket az 5.1 fejezetben tárgyaltam. A csomópontokat egy struktúrába tároltam, amely tartalmazta minden egyes pont *(x,y)* koordinátáját, valamint a csomópont azonosítóját (*ID*).

A következő lépésben a csomópontokhoz rendeltem a robot kiindulópontjának a koordinátáját, mint egy gráfpont. Majd a rendelkezésre álló két térképpel egy *kizáró vagy* (XOR) műveletet alkalmazva meghatároztam a mozgatható objektumokat. Az így kapott képen (ami csak a mozgatható objektumokat tartalmazta) ismét objektumkeresést hajtottam végre, aminek eredményeképpen megkaptam a mozgatható objektumokhoz tartozó tulajdonságokat (hosszúság, szélesség, középpont koordináta, bal alsó sarok koordináta).

Miután meghatározásra kerültek a mozgatható objektumok kiindulópontjainak koordinátái, ezeket is hozzárendeltem csomópontként a gráfpontokat tartalmazó struktúratömbhöz. Ezt követően az egyes objektumok célpontjához tartozó koordinátákat is a csúcspontokat tartalmazó struktúratömbhöz rendeltem. Ezzel a lépéssel elértem azt, hogy a gráf biztosan tartalmazni fogja a robot kiindulási pozícióját, illetve az mozgatható objektumok kiinduló-, és célpozícióját. Tehát amennyiben a gráfot alkotó élek megfelelően kerülnek meghatározásra, a szükséges pontok biztosan elérhetőek lesznek.

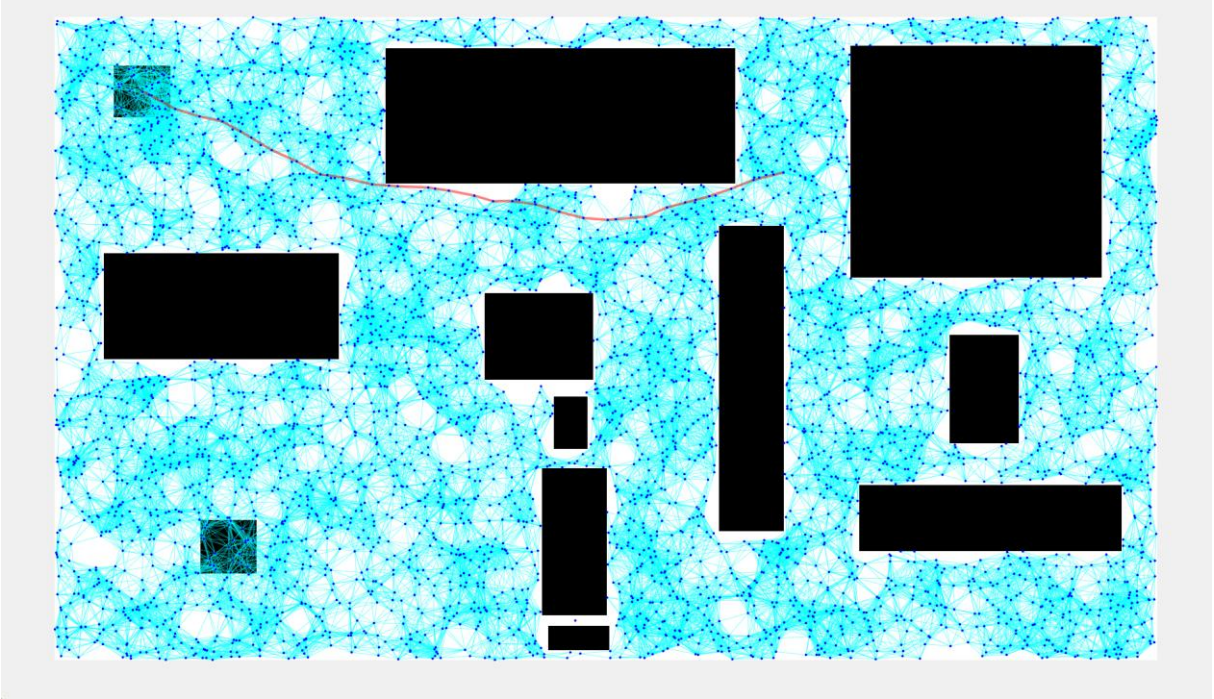
A következőkben a gráf élei kerültek meghatározásra a 5.2 fejezetben bemutatott függvény révén. A gráfot egy MatLab függvény segítségével építettem fel. Ezt követően az útvonaltervező függvény került meghívásra, amely meghatározta két akadályra az ütközésmentes legrövidebb útvonalat. A függvény bemenetként a következőket kapta:

- A felépített gráfot
- Az elhelyezett gráfpontok struktúratömbjét
- A mozgatható objektumokhoz tartozó gráfpontok azonosítóját
- A statikus akadályokat tartalmazó képet
- A mozgatható objektumok tulajdonságait tartalmazó struktúrát (hosszúság, szélesség…)
- Maximális élhosszat

5.3.2 Útmeghatározás lépései és eredményei

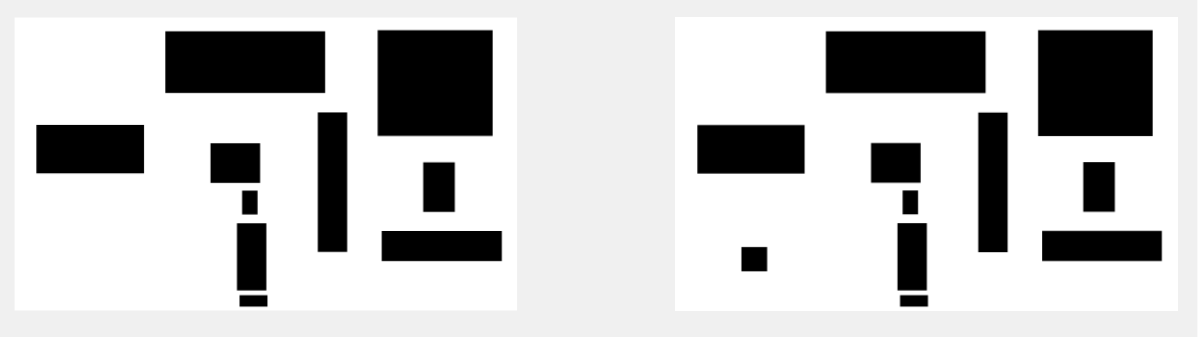
Az 5.3.1 fejezetben bemutatott algoritmus lépéseit, annak szemléltetéseit, illetve eredményeit tárgyalom ebben a fejezetben. Az algoritmust 4500 db csomóponttal, 40-es maximális élhosszal futtattam. A szemléltetést csak az egyik térképpel végzem el.

Első lépésként a robot és az első akadály között kerül meghatározásra a legrövidebb útvonal. A függvény a statikus akadályokat tartalmazó térképpel dolgozik. A 16. ábrán a mozgatható objektumok csak azért láthatóak, mert így áttekinthetőbb, hogy honnan hova mozgott a robot.



16. ábra - Első lépés, Robot és az első akadály közti út meghatározása

A következő lépésnél frissítettem a robot aktuális pozícióját az elért objektum pozíciójára. Mivel a függvény a statikus akadályokat tartalmazó térképpel dolgozik, ezért az elért objektum mozgatásához figyelembe kell venni a másik akadály pozícióját. Ezt úgy valósítottam meg, hogy egy rajzoló függvény révén a statikus akadályokat tartalmazó térképre rajzoltam a másik mozgatható objektumot. A 17. ábra szemlélteti, hogy a térképet rajzolás előtt és után.

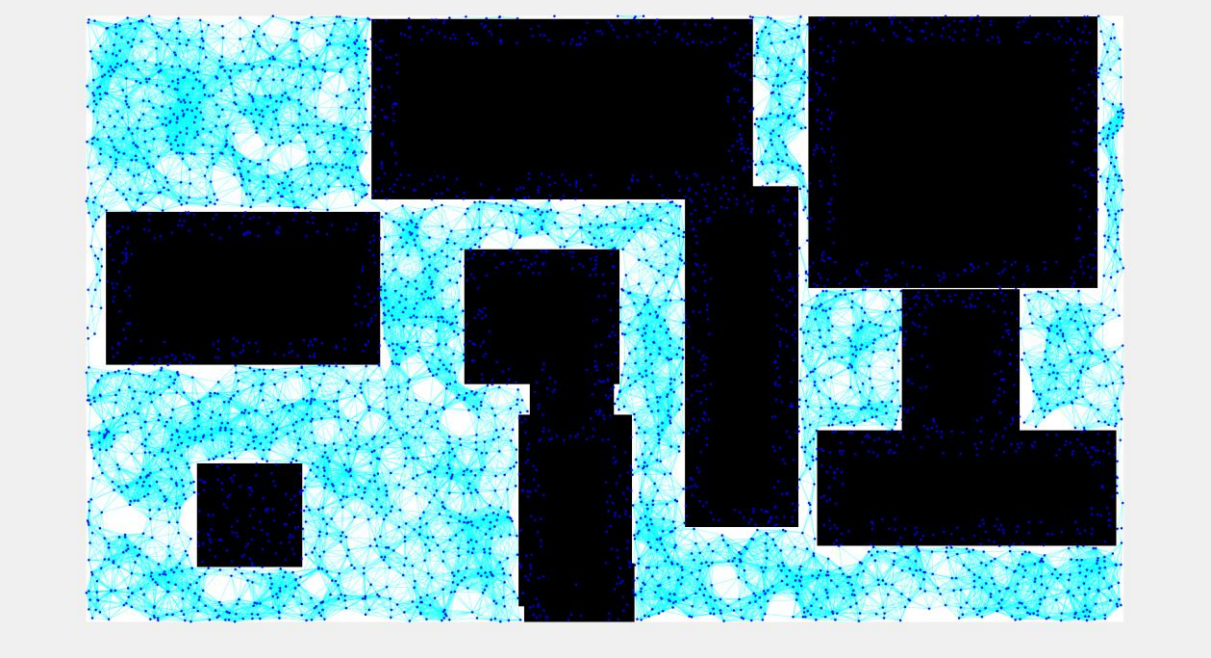


17. ábra - A mozgatandó objektum szögéből a statikus akadályok rajzolás előtt (bal), és rajzolás után (jobb)

Ezt követően az elért objektum mozgatásához figyelembe kellett vennem az objektum fizikai kiterjedtségeit, mivel ütközésmentes pályatervezés a célom. Ezt úgy tudtam figyelembe venni, hogy megvizsgáltam a mozgatandó objektum maximális kiterjedtségét és az alapján meghatároztam egy struktúráló elemet, amellyel minden statikusnak tekinthető akadályt dilatáltam. Ennek a lépésnek a végrehajtásával a 18. ábrán látható térképet kaptam.

18. ábra - Statikus akadályok kiterjedtségé dilatálás előtt (fekete) és dilatálás után (zöld)

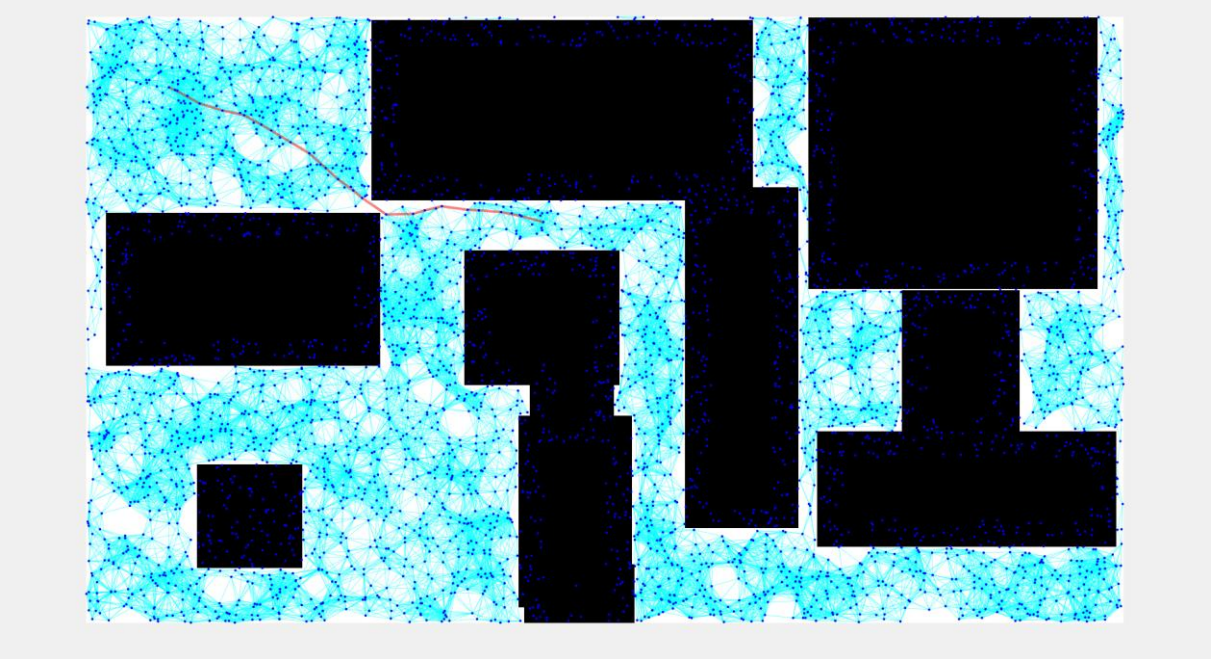
Az új méretű statikus akadályokat tartalmazó térképen meghatároztam, hogy mely eddigi gráfpontok esnek a szabadon bejárható téren kívül, és ez alapján újra felépítettem a gráfot.



19. ábra - A kevesebb csomóponttal rendelkező gráf

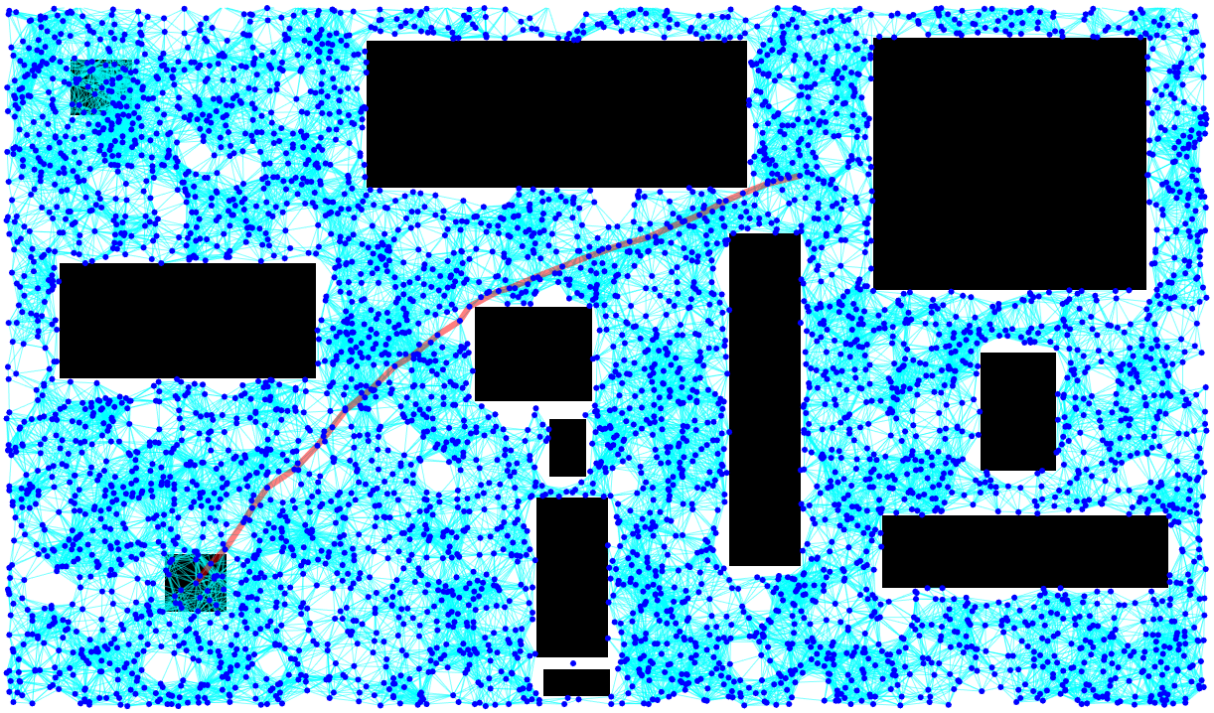
Annak ellenére, hogy továbbra is látszanak azok a csomópontok, melyek a dilatált statikus objektumok által határolt területre esnek, élek nem vezetnek oda, így

elhanyagolhatónak tekinthetőek. A következő lépésben az objektumot a célpontjába kell juttatni. Az így talált utat a 20. ábra szemlélteti.

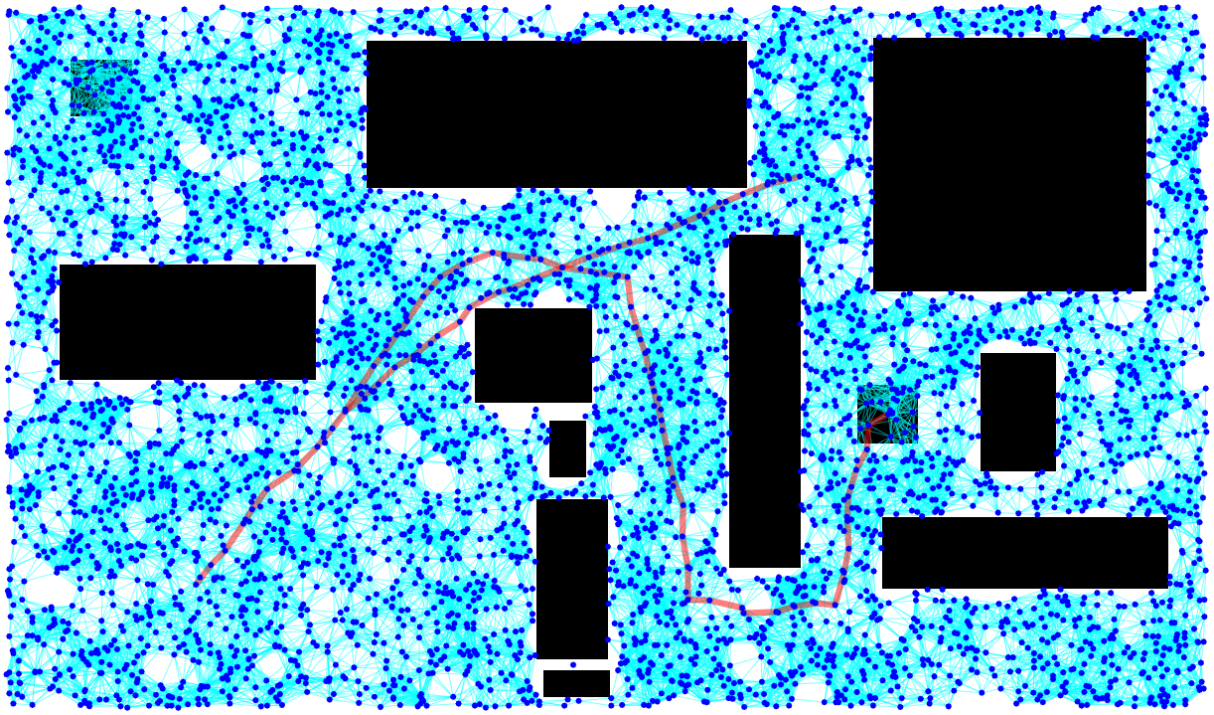


20. ábra - Objektum célpontba való mozgatása

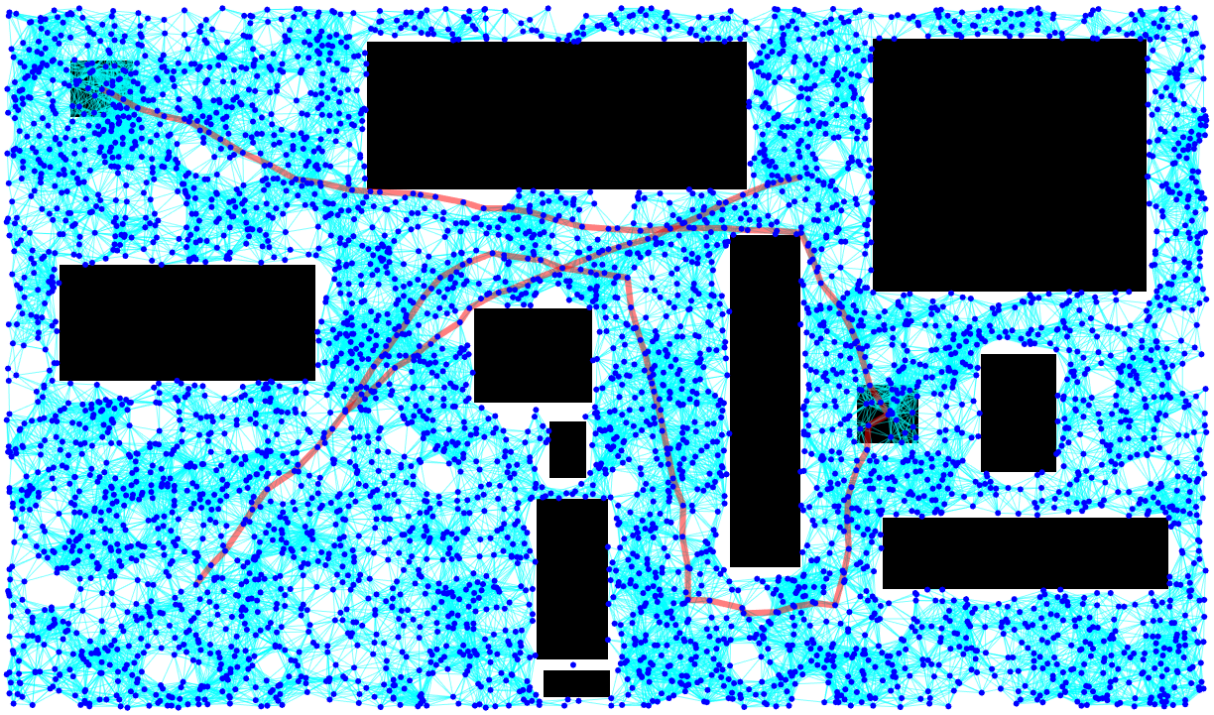
Az objektum célpontjába juttatását követően, úgy a másik objektumhoz kell eljuttatni a robotot. Amennyiben ezt sikerül megvalósítani, úgy az előbbiekben bemutatott lépéseket megismételve kapjuk meg az eredményt. Az eddig bemutatott ábrák alapján amennyiben az aktuális térképen ebben a sorrendben mozgatom az objektumokat, úgy elzárom a másik objektum egyetlen lehetséges útvonalát. Ezért az algoritmusom azt fogja jelezni, hogy ebben a sorrendben mozgatva nincs lehetséges útvonal. A következő sorrend szerint, viszont jó megoldást fog adni. A legjobb útvonalat visszaadó futást a 21-24. ábrák szemléltetik.



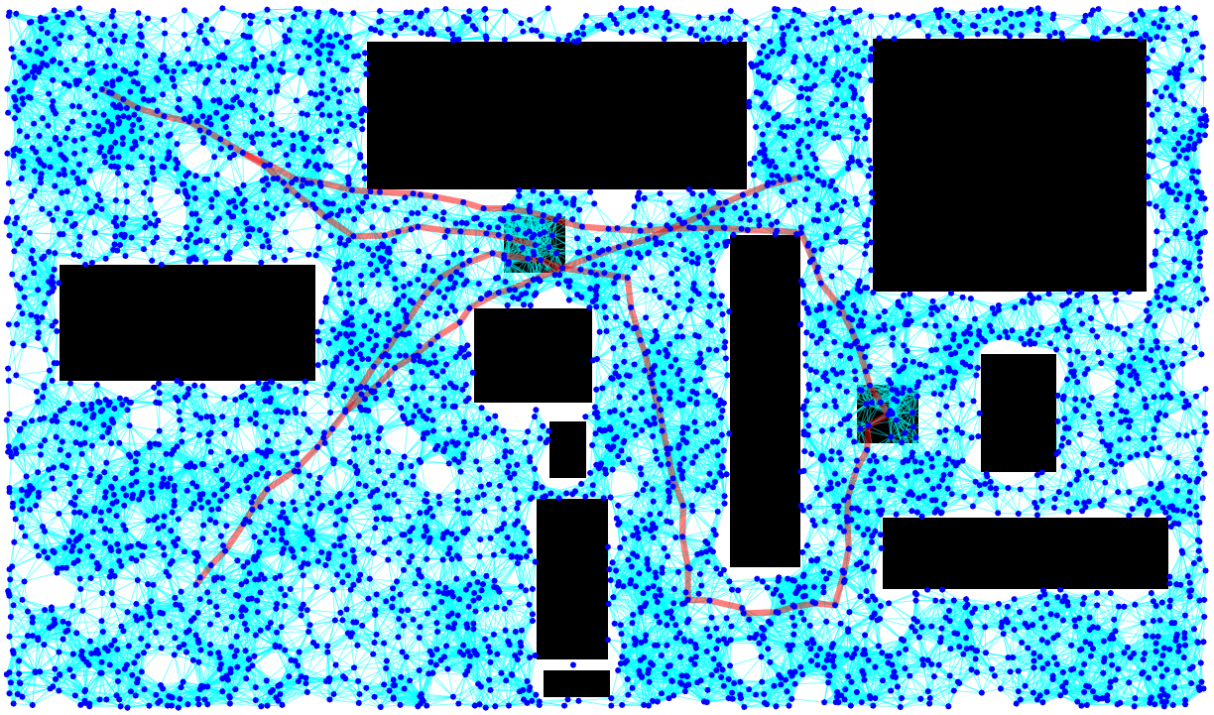
21. ábra - A robot és az először mozgatandó objektum közötti út



22. ábra - Az először mozgatandó objektum célpontba való mozgatása



23. ábra - Az első mozgatandó objektum célpontjától a második mozgatandó akadály kiindulópontjába juttatása a robotnak



24. ábra - A második mozgatandó objektum célpontjába való juttatása

Az algoritmusom tökéletesen megoldja a felvetett problémát két mozgatható objektumra. Viszont a probléma megoldásához szükséges idő még nem került tárgyalásra. Ebből kifolyólag teszteltem az útvonaltervezéshez szükséges futásidőt. A futásidőteszt különböző PRM paraméterek (gráfpontok száma, maximális élhossz) mellett került

megvalósításra, de figyelembe kellett venni azt is, hogy milyen valószínűséggel talál útvonalat az algoritmus, amit az algoritmus hatásfokának lehet tekinteni.

Az 5.2 fejezetbenbemutatott eredmények alapján nem érdemes kevés csomóponttal és kis élhosszúsággal dolgozni, ugyanis a jelenlegi térkép méretei mellett az algoritmus nem tud megfelelő méretű gráfot felvenni. Ez alapján a következő paraméterkombinációk mellett teszteltem az algoritmus működését:

15. táblázat - Az algoritmus teljesítménye

	$n = 300$ $d = 500$	$n = 300$ $d = inf$	$n = 400$ $d = 400$	$n = 500$ $d = 300$	$n = 500$ $d = 500$	$n = 1000$ $d = 200$	$n = 1000$ $d = 400$	$n = 2000$ $d = 100$	$n = 2000$ $d = 200$	$n = 2500$ $d = 50$
η [%]	45	75	100	85	100	100	100	100	100	100
$t_{alg_{avg}}$ $[s]$	12.2	24.3	16.65	18.25	24.57	30.76	68.35	39.19	107.37	18.74
$t_{total_{avg}}$ $[s]$	18.5	37.16	25	27.15	35.68	42.53	103.22	54.79	153.73	21.54

A táblázatban használt rövidítések a következőket jelentik:

- n – gráfot alkotó csomópontok száma
- d – maximálisan megengedett élhosszúság
- η – az algoritmus hatásfoka (20 egymás után történő futtatás során milyen valószínűséggel talált utat)
- $t_{alg_{avg}}$ – az útkereső függvény átlag futásideje
- $t_{total_{avg}}$ – a teljes algoritmus átlag futásideje

A 15. táblázatban látható, hogy bizonyos paraméterek mellett milyen eredményt ért el az algoritmus. Habár az a cél, hogy 100% hatásfokot érjen el, de ez elérhetetlen bizonyos paraméterek mellett. A 100%-os hatásfokot a beépített PRM se tudja garantálni, ha olyan paramétereket kap bemenetként. Futásidő és hatékonyság alapján a sok csomópont-rövid élhossz párosítás adta legjobb eredményt. Ezt veszem alapul a több mozgatható objektumos megvalósítás során is. Mielőtt tárgyalásra kerülne a kettőnél több mozgatható objektumos megvalósítás, megvizsgálásra kerül, hogy milyen futásidő prediktálható a jelenlegi ismeretek alapján

Három mozgatható objektum esetében hat különböző sorrend alakítható ki. A robotnak először három objektum közül kell választania. Ha a megfelelőt a célpontba juttatta, akkor már csak kettőből, és így tovább. Ez egy kombinatorikai probléma és faktoriális függés van a lehetséges sorrend és az objektumok száma között. Az előző fejezetben látható volt, hogy a futásidő jelentős részét a gráf éleinek felvétele teszi ki. A 13. táblázatban található 2500 darab csomópont és 25-ös élhosszúságú paraméterek mellett kapott futásidő felét véve egy kombináció kiszámítására szükséges futásidő meghatározható. Ez az érték $\sim 9.37s$. Ha 5 mozgatható objektumra akarjuk meghatározni az összes kombináció kiszámításához szükséges időt az akkor $5! * 9.37s = 120 * 9.37s = 1124.4s$. Ez megközelítőleg 18.75 percet jelent, ami futásidő tekintetében soknak számít.

A több mozgatható objektumos megvalósítás során célom, hogy javítsak a futásidőn, valamint egy mesterséges intelligencia felhasználásával optimalizáljam a sorrend kiválasztást.

6 Több akadályra történő implementáció

Több akadályos megvalósítás során célkitűzés volt az, hogy egy kezelői felület révén a felhasználó szabadon tudja megadni a térképet, amellyel az algoritmusnak dolgoznia kell. A kezelői felület által a felhasználó szabadon kiválaszthatja a mozgatható objektumokat, valamint az objektumok célkoordinátáit is megadhatja. Ugyanígy a robot kiindulásának pozíciója is szabadon megadható a felhasználó által. A 6.1 alfejezetben a felhasználói felület kerül bemutatásra, majd pedig a 6.3 alfejezetben a több akadályos megvalósítást tárgyalom.

6.1 Kezelői felület (UI)

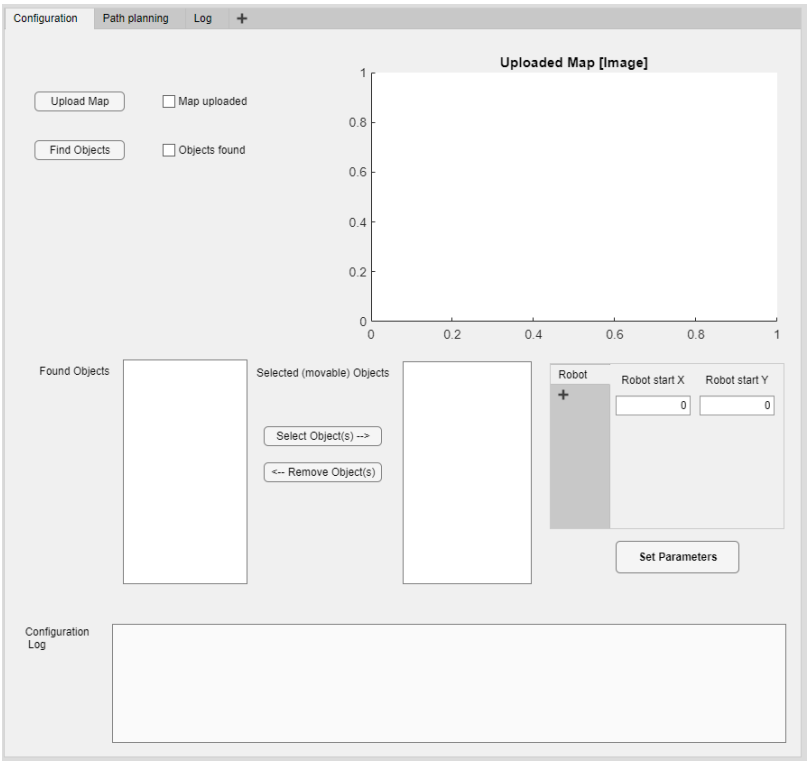
Amint már említésre került, a kezelői felülettel az a cél, hogy a felhasználónak nagyobb szabadkezet biztosítsak az algoritmus futásához szükséges paraméterek és konfigurációk megadására. Erre a célra a MatLab App designer tool-ját használtam fel. Azért esett erre a választásom, mert könnyed skálázhatósággal rendelkezik, valamint a létrehozott front-end és back-end között a kommunikáció megvalósítása nem ütközik problémába. A felhasználói felülettel kapcsolatban úgy tűztem ki a célokat, hogy lehetőség legyen:

- Térképet feltölteni
- A feltöltött térképen bejelölni a mozgatható objektumokat
- A robotnak megadni a kiindulási helyzetét
- A mozgatható objektumoknak megadni a célpontjuk koordinátáit
- Lefuttatni az algoritmust, és az eredményét megjeleníteni

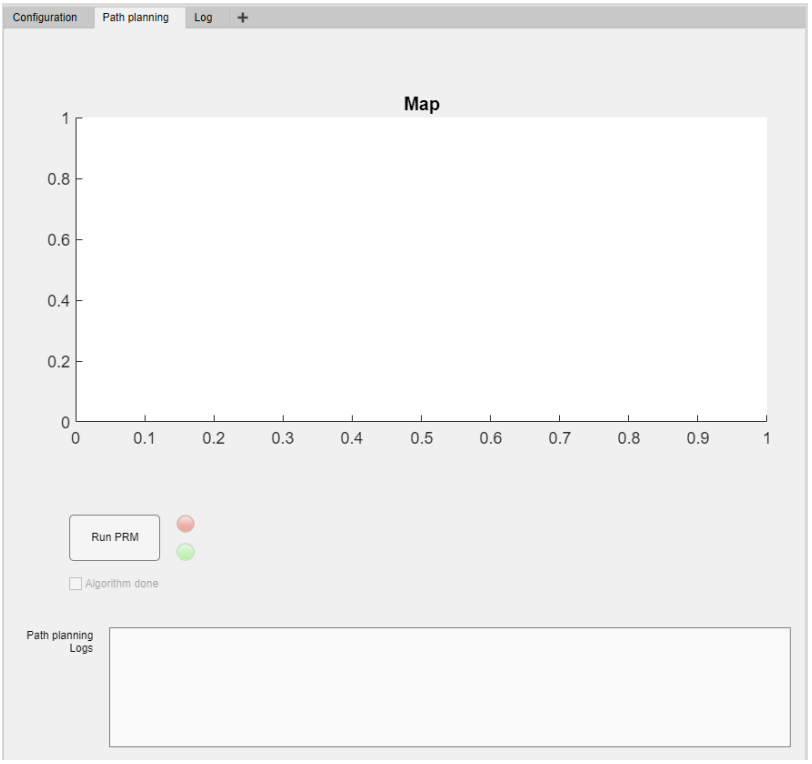
6.1.1 Kezelői felület külalakja

A UI három nagyobb fő füllel rendelkezik melyek a következők: *Configuration, Path planning, Log*. Az utolsó fül, a *Log*, szerepe, hogy a felhasználó számára információt biztosítson arról, hogy a kezelő felület alkalmazása során milyen fontosabb lépések történtek. Emellett visszajelzést ad egy-egy paraméter beállításáról és változtatásáról. A *Configuration* fülön van lehetőség feltölteni a térképet, kiválasztani a mozgatható objektumokat, valamint megadni az objektumok célpontjának koordinátáit és a robot kiindulási pozícióját. A *Path planning* fül

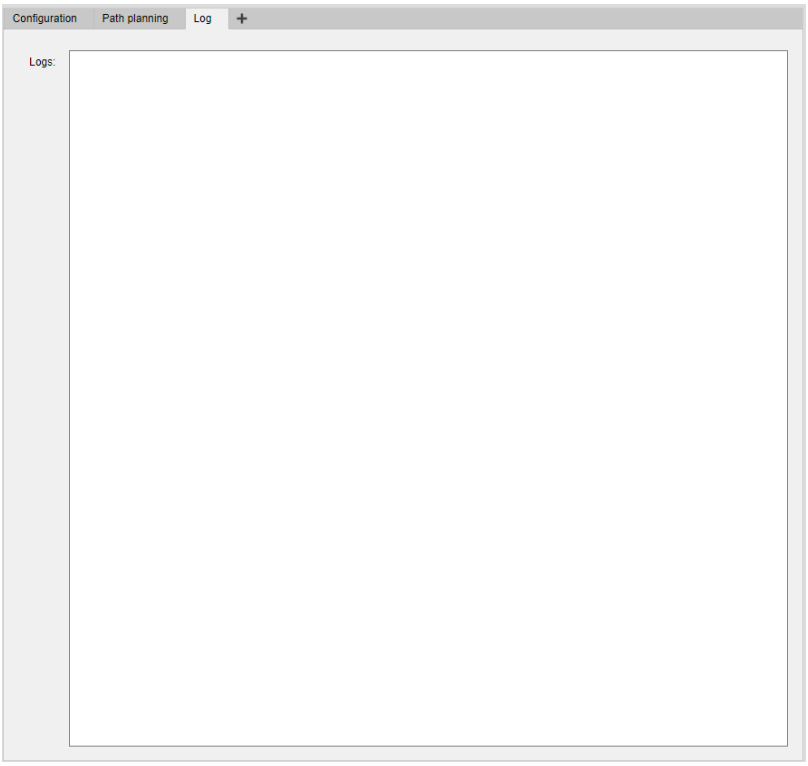
pedig az algoritmus futtatását teszi lehetővé. Futás után ezen a fülön kerül megjelenítésre az eredmény. A kezelő felületet a következő ábrák szemléltetik.



25. ábra - A felhasználói felület Configuration füle



26. ábra - A felhasználói felület Path planning füle



27. ábra - A felhasználói felület Log füle

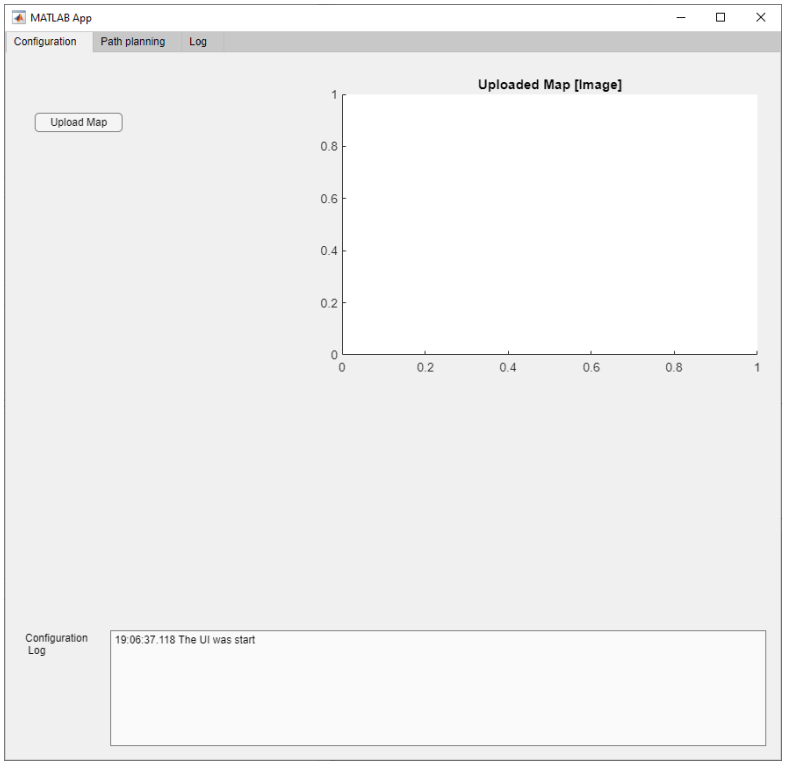
6.1.2 A kezelői felület megvalósítása és működése

A feladat megvalósítása terén a kezelői felületnek két fontosabb eleme van, a *Configuration* és a *Path planning* fül. A *Log* fülön belül csak informatív üzenetek kerülnek megjelenítésre, ezért a következőkben ezt nem részletezem.

A *Configuration* fülön belül történik a fontos paraméterek megadása, amelyekre az algoritmusnak szüksége van a futáshoz. Ezen a fülön a következő lépéseket lehet megvalósítani:

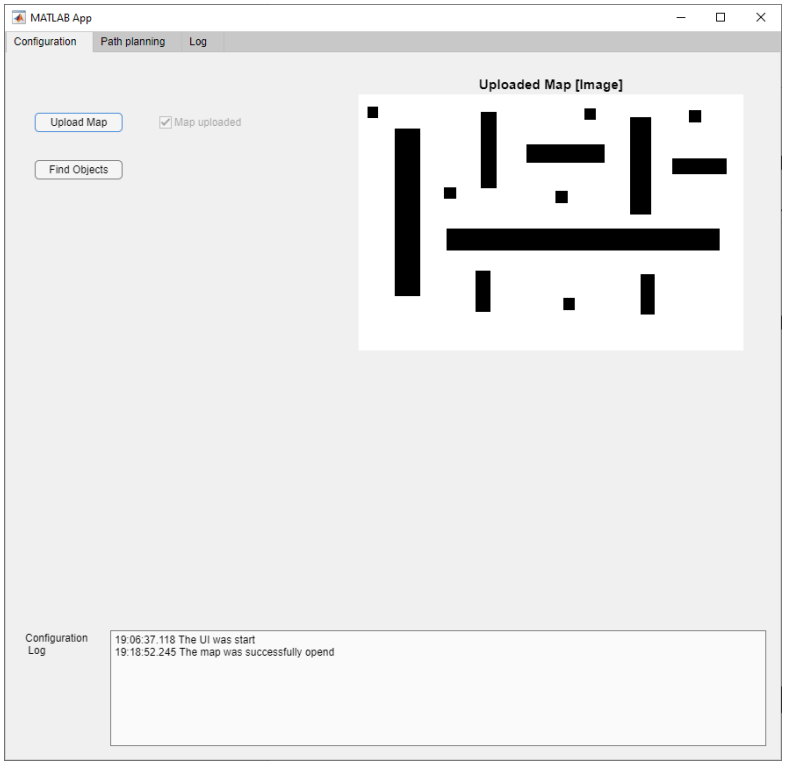
- Térkép feltöltése → megjelenítésre is kerül a térkép
- A térképen objektumok keresése
- A keresés során megtalált objektumok közül a mozgathatók kiválasztása, illetve elvétele
- A kiválasztott mozgatható objektumokhoz meghatározni a célpont koordinátáit
- A paraméterek elmentése

Mivel a hibakezelés nem került még megvalósításra, ezért a kezelői felületen néhány elem indításkor deaktiválva van, vagy el van rejtve, és nem lehet szerkeszteni. A következő ábra az elindítás pillanatába látható kezelői felületet szemlélteti.



28. ábra - Kezelői felület indítás után

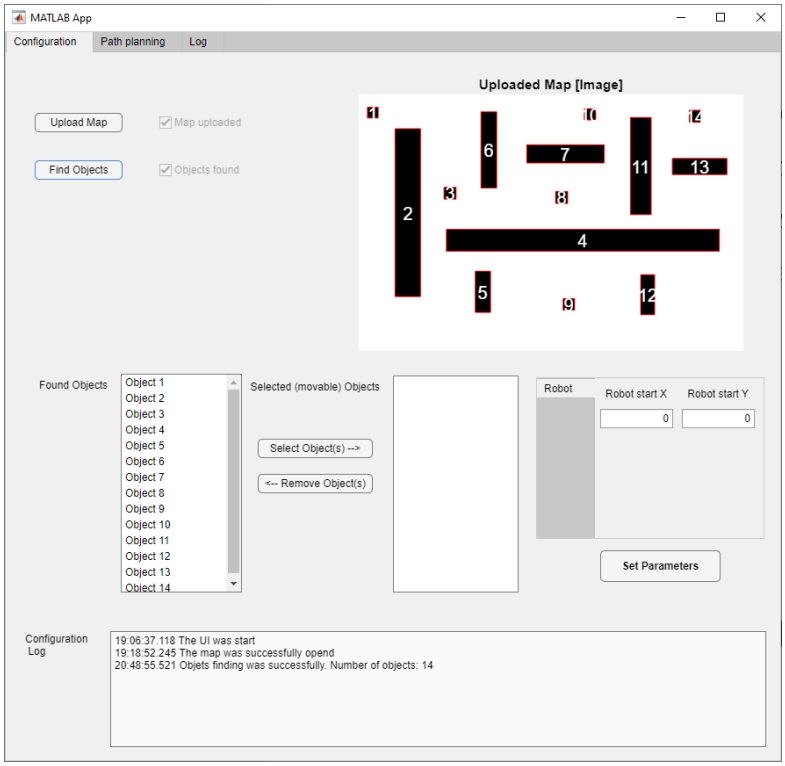
Első lépésként a térkép kerül kiválasztásra az *Upload Map* gomb segítségével. Miután kiválasztásra került a kívánt térkép a kezelői felület megjeleníti azt az *Uploaded Map [Image]* plot-ban. Ezzel egyidőben elérhetővé válik a *Find Objects* gomb, és megjelenik egy checkbox az *Upload Map* gomb mellett, ezzel jelezve, hogy a térkép sikeresen feltöltődött. Ezt a következő ábra szemlélteti.



29. ábra - Kezelői felület a térkép kiválasztása után

A *Find Objects* gomb megnyomásával a feltöltött térképen meghívódik egy objektumkereső függvény. A függvény által meghatározott objektumokat egy-egy számozott címkével látom el, ezáltal egyértelműen beazonosítható minden objektum. Ezután megjelenítésre kerül két szövegmező (text area), melyek közül az egyik a *Found Objects* míg a másik a *Selected (movable) Objects*. A *Found Objects* szövegmező tartalmazza a függvény által megtalált objektumokat az azonosítójuk alapján (*Object 1, Object 2, …*). A *Selected (movable) Objects* szövegmező kezdetben üres, de később itt kerülnek megjelenítésre azon objektumok azonosítója, amelyek kiválasztásra kerültek mozgatás céljából. Ezek közül egyik se szerkeszthető tulajdonságú, hanem a két mező között található gombokkal módosítható a tartalmuk. Ez a két gomb a *Select Object(s)* és a *Remove Object(s)*. Amint a nevük is mutatja az egyik az objektumok kiválasztására, míg a másik az objektumok elvételére szolgál. Erről a későbbiekben lesz még szó. A következő elem, ami megjelenik a *Find Objects* gomb

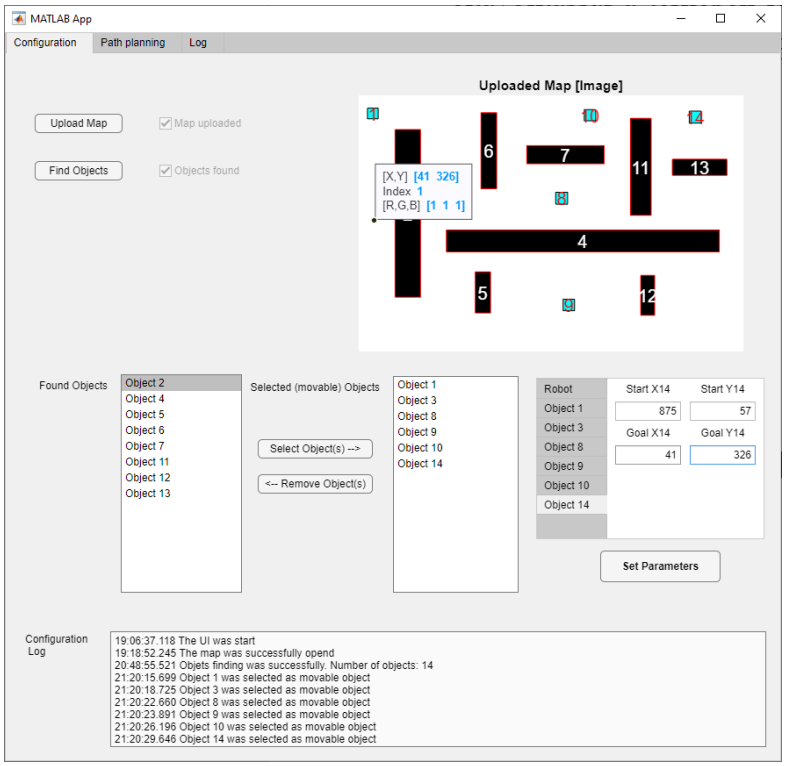
megnyomása után az egy olyan TabGroup, amely kezdetben csak egy tabot tartalmaz. Ez az egy tab a robot kezdőpontjának koordinátáihoz tartozó inputmezővel rendelkezik. A későbbiekben ez is részletesebben kerül bemutatásra. Jelenleg a kezelői felületen a következőket láthatjuk az alábbi ábra szerint.



30. ábra - Kezelői felület az objektumok meghatározása után

A következő lépés a mozgatható objektumok kijelölése (meghatározása). A *Found Objects* listában található az összes objektum az azonosítójukkal, amit a függvény a feltöltött térképen talált. Ezen listából a megfelelőket egyesével kijelölve a *Select Object(s)* gomb megnyomásával lehet áthelyezni (kiválasztani) a *Selected (movable) Objects* listába. A kiválasztás során a kiválasztott objektum a megjelenített térképen kiemelésre kerül (megváltozik az objektumot kitöltő szín). Ezzel egyidőben egy új tab generálódik a TabGroup-hoz, amely a kiválasztott objektumra vonatkozóan tartalmaz információkat. Ilyen információk az objektum kiindulási helyzetének a koordinátái, valamint a célpontjának a koordinátái. Az előbbi a kiválasztás során kiszámításra kerül egy számmezőben. Ennek a mezőnek az értéke nem módosítható. Az utóbbit pedig a felhasználónak kell megadnia. A térképet úgy jelenítettem meg, hogy az egérrel rákattintva a kattintás helyének a koordinátáit jelenítse meg. Ezáltal könnyedén meghatározható a kívánt célpozícióhoz tartozó koordináták értéke.

Amennyiben egy kiválasztott objektumot el kell távolítani, úgy a *Remove Object(s)* gombbal eltávolítható a *Selected (movable) Objects* listából a kiválasztott objektum. Az eltávolítással a térképen lévő kiemelése is megszűnik azzal, hogy az adott objektum kitöltő színe ismét fekete lesz. Ezzel egyidőben a TabGroup-ból is törlésre kerül minden eddigi adatával együtt. A következő ábra szemlélteti a mozgatható objektumok kiválasztását és célpontjuk megadását a kezelői felületben.

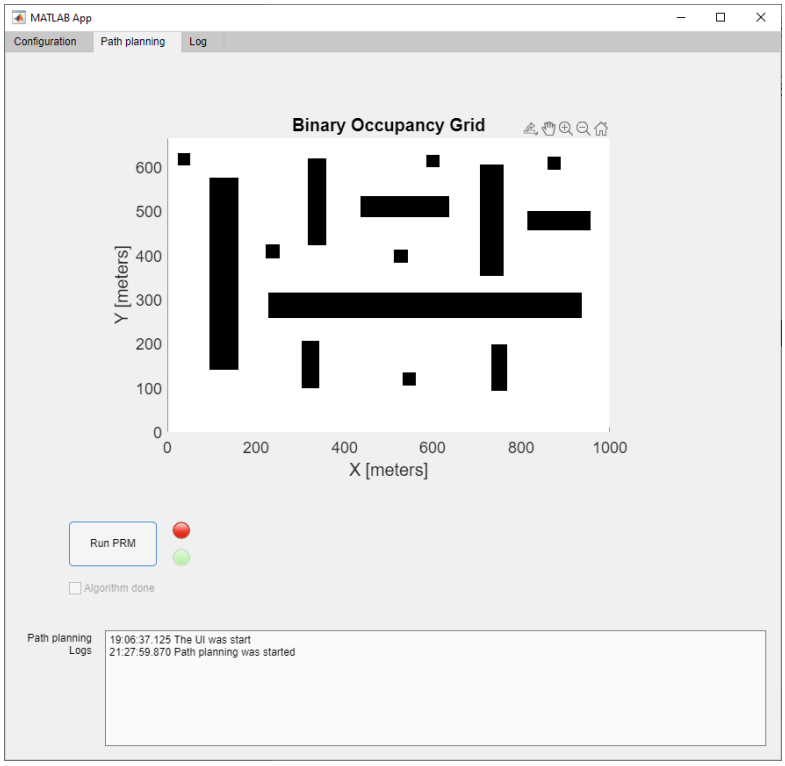


31. ábra - Kezelői felület a mozgatható objektumok kiválasztása és célpontjuk megadása után

Ha megadásra kerültek a célpontok koordinátái és a robot kiindulási pozíciója, úgy a *Set Parameters* gombbal elmentésre kerülnek egy struktúratömbbe a szükséges paraméterek. Ha mentést követően törlésre szánunk egy objektumot a mozgatható objektumok listájából, akkor újra el kell menteni. Különben az algoritmus az utolsó mentett struktúratömbbel fog lefutni.

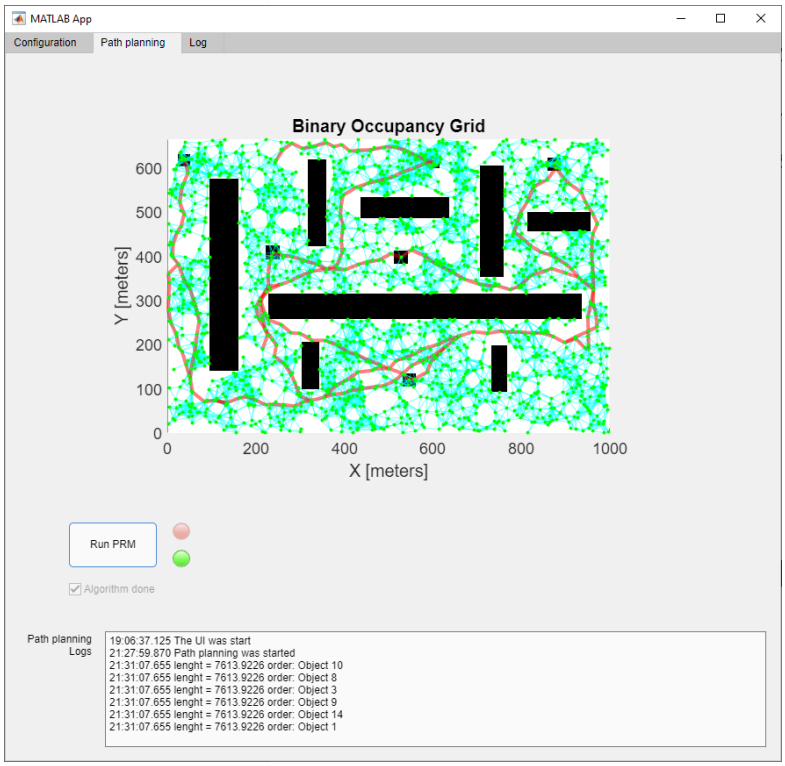
A kezelői felület következő része az útvonaltervezés futtatása. Ez a fül egy térképmegjelenítésre alkalmas elemből és egy futtatásra alkalmas gombból áll, ugyamakkor státuszinformációt szolgáltató LED-ekkel és logolásra alkalmas szövegmezővel is rendelkezik. Kezdetben a térképet jeleníti meg, de itt már nem képként, hanem Occupancy Grid-ként. Azért van erre szükség, mert az algoritmus nem képként dolgozza fel a térképet, hanem Occupancy Grid-ként, és minden hozzá kapcsolódó elem ennek megfelelően van paraméterezve. A

következő ábra szemlélteti a kezelői felület *Path planning* fülét a mozgatható objektumok kiválasztása és elmentése után.



32. ábra - Kezelői felület Path planning füle kezdetben

A *Run PRM* gomb megnyomásával meghívásra kerül az útvonaltervező algoritmus. Annak sikeres lefutása után a térképre rajzolódik a felvett gráf és az útvonal, amelyen a robot végig haladva az összes objektumot a kívánt célpontba juttattja. A 33. ábra szemlélteti a sikeres pályatervezés eredményét.



33. ábra - A sikeres pályatervezés eredménye

Az útvonalat a pirossal kiemelt élek jelentik, amik néhol egybe folynak, a megjelenítés során. A szövegmezőben kerül szemléltetésre a megtett út hossza, ami jelen esetben 7613.92 hosszegység, valamint, hogy milyen sorrendben kerültek mozgatásra az objektumok.

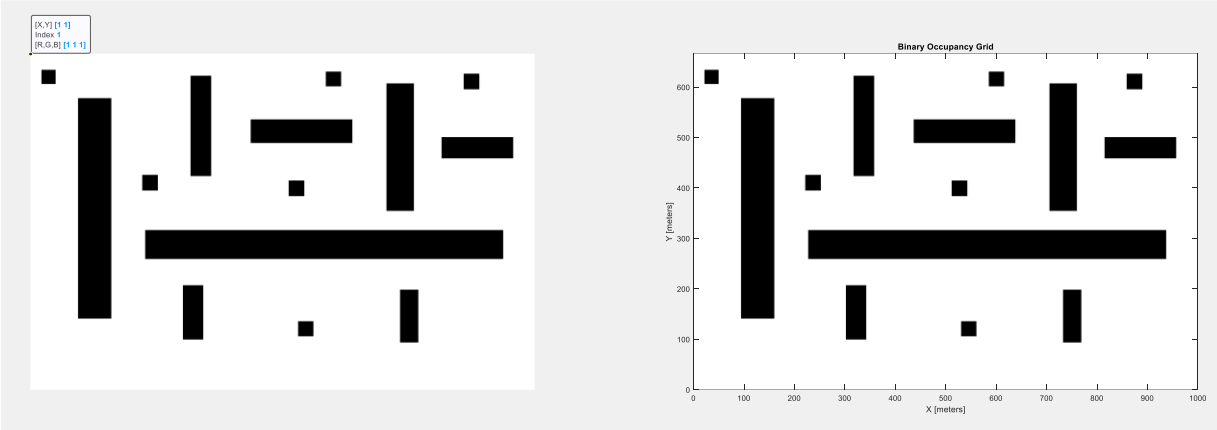
6.2 Alapmódosítások a kétobjektumos megoldáshoz képest

A kétobjektumos megvalósítás során a térképet képként kezeltem. Ezzel szemben a MatLab beépített PRM-je az Occupancy grid-ként kezeli a térképet, amelyen ütközésmentes útvonalat kell meghatároznia. Emiatt döntöttem, úgy, hogy megvizsgálom, milyen hatással rendelkezik az algoritmusra, ha a térképet én is Occupancy grid-ként kezelem. Az ehhez tartozó eredmények a következő fejezetben kerülnek tárgyalásra. Jelen fejezetben az Occupancy grid-re való áttérés miatti változtatásokat fogom bemutatni.

6.2.1 Gráfpontok felvétele

A gráfpontok generálása az eddigiektől csak annyiban tér el, hogy egy koordinátatranszformáció szükséges a csomópontok pozíciójára, ugyanis a megjelenítés során

az Occupancy grid origója máshol helyezkedik el, mint a kép megjelenítése során. Ezt szemlélteti a következő ábra.



34. ábra - Azonos térkép képként (bal) és Occupancy grid-ként (jobb) való megjelenítése

Míg a képként való megjelenítés eseten a bal felső sarokban helyezkedik el az (1,1)-es origó, addig az Occupancy grid esetén a bal alsó sarokban és (0,0)-ás értékű. A megvalósítás során végül nem transzformációt használtam, hanem direktben az Occupancy grid-en generáltam a csomópontokat.

A gráfpontok felvételéhez szükséges *isSame* és az *isOccupied* függvények is csak annyiban kerültek változtatásra, hogy megfelelően tudják kezelni az Occupancy grid-et.

6.2.2 Gráfélek felvétele

A gráfélek felvétele esetén se kellett nagy változást eszközölni. A legnagyobb változtatás az volt, hogy nem egyenesek metszetvizsgálatával határoztam meg azt, hogy egy él áthalad-e egy akadályon. A gráfélet lineáris interpolációval határoztam meg a két csomópont által. Ezen a gráfélen bizonyos lépésközzel végighaladva ellenőriztem, hogy az Occupancy grid-en detektálható-e „foglalt” terület az egyenes mentén.

A futásidő csökkentése érdekében a gráfpontokat *x* koordinátájuk szerint rendeztem növekvő sorrendbe. Ezáltal nem kell megvizsgálni minden lehetséges kombinációt, csupán azokat, amelyek a maximális távolság szerint egy adott intervallumba esnek.

6.2.3 Meghatározott gráfél ellenőrzése

A gráfélek felvételének alapkoncepciója változatlan a 3.3.2 fejezetben tárgyaltakhoz képest, csupán a *checkCrossPoint* függvény került újragondolásra. Eddig két egyenes közti metszéspont keresésével lett meghatározva, hogy az adott él felvehető vagy sem. Ez most úgy módosult, hogy a két gráfpont alapján egy egyenest interpolálok, amit egy lineáris függvényként tekintek. Ezt a függvényt a következőféleképpen lehet általánosan felírni:

$$f(x) = ax + b$$

Lineáris algebra segítségével az a, b paramétereket a következő módon lehet meghatározni a két gráfpont alapján:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

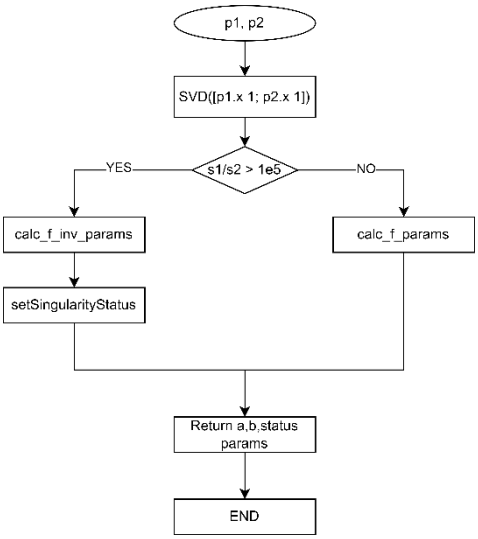
(x_1, y_1) az egyik gráfpont koordinátái és az (x_2, y_2) a másik gráfpont koordinátái. Előfordulhat, hogy a gráfél közel függőlegesnek tekinthető, ilyenkor az a paraméter közel ∞ értékű, és szingularitás miatt nem használható további számításokra az így kapott paraméter. Amennyiben hasonló fordul elő, úgy a függvény inverzét számolom ki:

$$f^{-1}(x) = a'x + b'$$

Ilyenkor hasonló módon kerül kiszámításra az a' és b' paraméter az x és az y felcserélésével.

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} y_1 & 1 \\ y_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

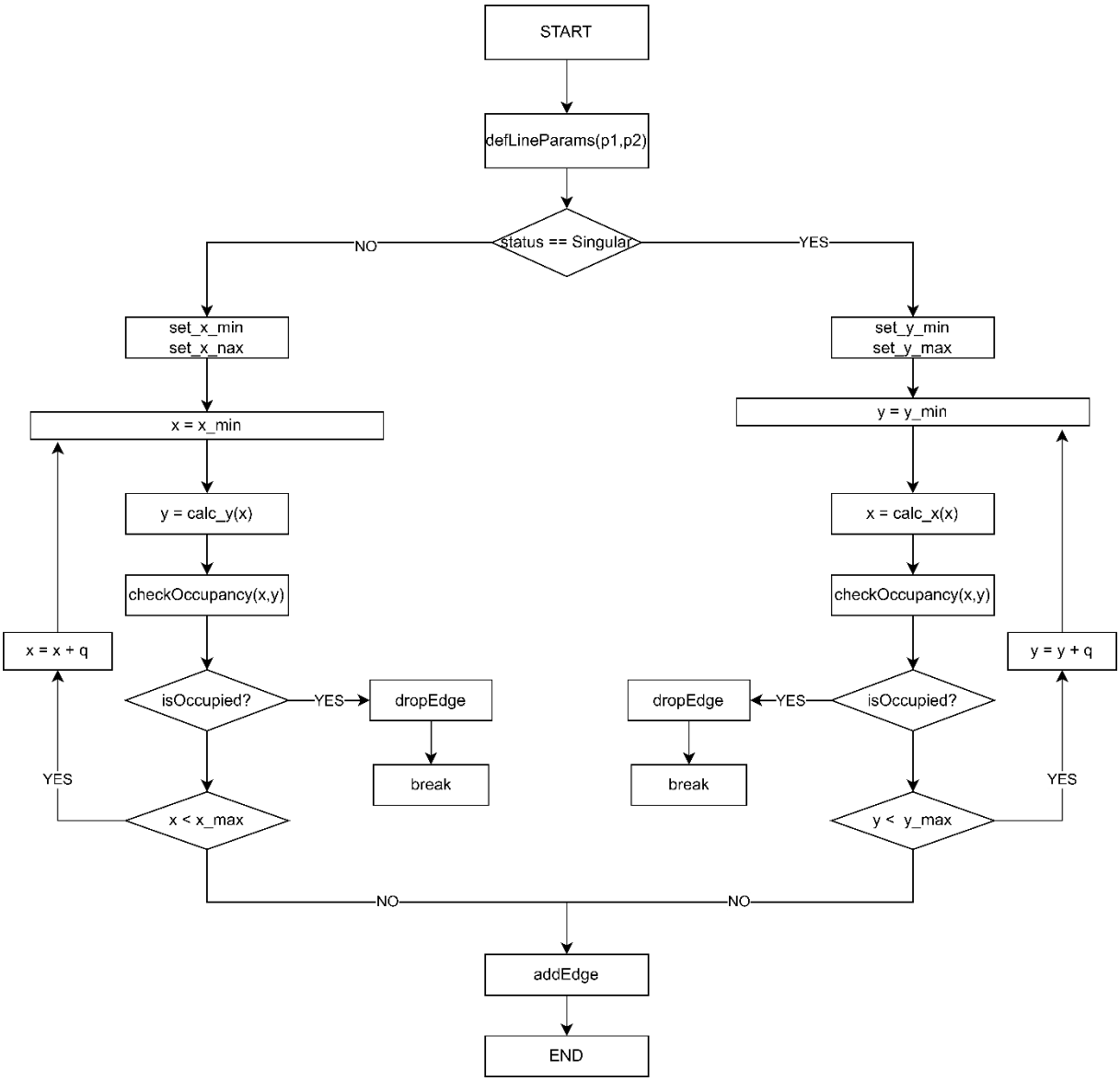
A szingularitás eldöntéséhez a függvény az $\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}$ mátrix SVD felbontását számolja ki, és amennyiben az s_1/s_2 értéke 10^5-nél nagyobb, úgy szingulárisnak tekinti. Ebben az esetben nem az $f(x)$-nek kerül kiszámításra az a és b paramétere, hanem az $f^{-1}(x)$-nek az a' és b' paraméterei, valamint a visszatérési értékei között lesz egy státusz jelző, aminek az értéke *Singular*. A következő ábrán látható a paraméterek meghatározásának folyamatábrája.



35. ábra - A paraméterek meghatározásának folyamatábrája (defLineParams)

A paraméterek meghatározása után kerül megvizsgálásra, hogy az adott él mutat-e metszést statikus akadállyal. Ez a szingularitástól függően kerül megvizsgálásra. Amennyiben a mátrix nem szinguláris, úgy meghatározásra kerül a kezdeti *x* és a végső *x* érték, amelyek között fix lépésenként az *y*-t kiszámolva lekérdezésre kerül az adott koordináta Occupancy grid szerinti foglaltsági állapota. Amennyiben foglalt értéket ad a lekérdezés, úgy az él elvetésre kerül, különben a gráfhoz rendelődik hozzá.

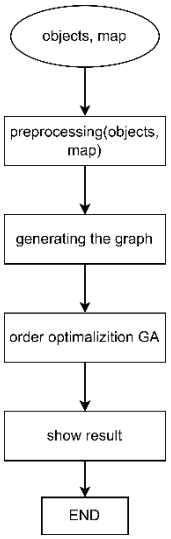
Amennyiben a mátrix szinguláris, úgy meghatározásra kerül a kezdeti *y* és a végső *y* érték. Az előzőekhez hasonlóan kiszámításra kerül az *x* értéke. Ezt követően pedig a kapott koordinátára lekérdezem a foglaltsági állapotot. Ekkor vagy hozzárendelődik a gráfhoz az él vagy elvetésre kerül. Az ehhez tartozó folyamatábra alább látható.



36. ábra - Egy gráfél hozzárendeléséről döntő függvény folyamatábrája

6.3 Több objektumos útvonal tervezése

Több objektum esetén mesterséges intelligenciát használtam, hogy ezáltal csökkenteni tudjam, az ellenőrzésre kerülő mozgatható objektumok sorrendjének kombinációit. Az algoritmust megvalósító fő függvény folyamatábrája a következő ábrán látható.



37. ábra - A fő függvény folyamatábrája

Első lépésként a felhasználói felülettől érkező bemeneti paraméterek kerülnek feldolgozásra, ahogyan azt a folyamatábra is szemlélteti. Ezt követően a gráf kerül létrehozásra, amely segítségével megvalósul az ütközésmentes pályatervezés. Genetikus algoritmust alkalmazva, mint mesterséges intelligencia, kiszámításra kerül az optimális sorrend. A genetikus algoritmus pedig az általa meghatározott legjobb sorrenddel tér vissza, ami megjelenítésre kerül, jelen esetben egy *plot*olással.

6.3.1 Bemeneti paraméterek feldolgozása

A felhasználói felülettől érkező paramétereket fel kell dolgozni egyrészt azért, mert az algoritmus Occupancy grid-el dolgozik, míg a kezelői felület képként tekint a térképre. Másrészt azért, mert a sorrendoptimalizáláshoz szükséges néhány módosítás a térképen és a mozgatható objektumokat tartalmazó struktúratömbön.

A feldolgozás során a következő lépéseket kell végrehajtani:

- A bemenetként érkező objektum struktúratömb tartalmazza a robot kiindulási pozícióját, amit külön kell választani az objektumoktól
- A kapott térkép alapján létre kell hozni egy mozgatható objektumok nélküli változatot, valamint egy olyan változatot, melyen az objektumok mind a kiindulási, mind a célpontjukban is megjelennek
- A térképeket Occupnacy grid-é kell konvertálni
- A mozgatható objektum struktúratömbben a koordinátákat transzformálni kell

- A gráfpontokat le kell generálni a mozgatható objektumokat nem tartalmazó térkép alapján
- A generált gráfpontokat *x* koordinátájuk szerint sorba kell rendezni, majd meg kell határozni a felveendő gráféleket a mozgatható akadályokat nem tartalmazó térkép alapján, valamint a kiinduló és a célpontban is meglévő térkép alapján is
- Létre kell hozni a két gráfot
- Ezután a sorrendoptimalizálás következik

Mivel a robot csak kiindulási pozícióval rendelkezik, de a kezelői felülettől az objektumokat tartalmazó struktúratömbben érkezik, ezért külön kellett választani.

Azért kell a térképből két másik változatot is létrehozni, mert a gráfpontokat a mozgatható objektumokat nem tartalmazó térképen veszem fel, de a kezdeti gráfot a „megkettőzött” mozgatható objektumos térkép szerint generálom. A mozgatható objektumokat tartalmazó struktúratömböt pedig kiegészítem két újabb tulajdonsággal. Az egyik tulajdonság azon gráfpontok azonosítóját tartalmazó vektor, melyeket a mozgatható objektumok kiindulási helyzetében „takar”. A másik tulajdonság azon gráfpontok azonosítóját tartalmazó vektor, melyeket a mozgatható objektumok a célpontjuk pozíciójában fog „takarni”. Ez a későbbiekben kerül felhasználásra, mikor az útvonalat számolja ki az algoritmus.

Mivel az algoritmus a térképeket Occupancy grid-ként kezeli, ezért a két új térképet ennek megfelelően kell konvertálni. Ezen túlmenően a mozgatható objektumokra vonatkozó koordinátákat ennek megfelelően kell transzformálni.

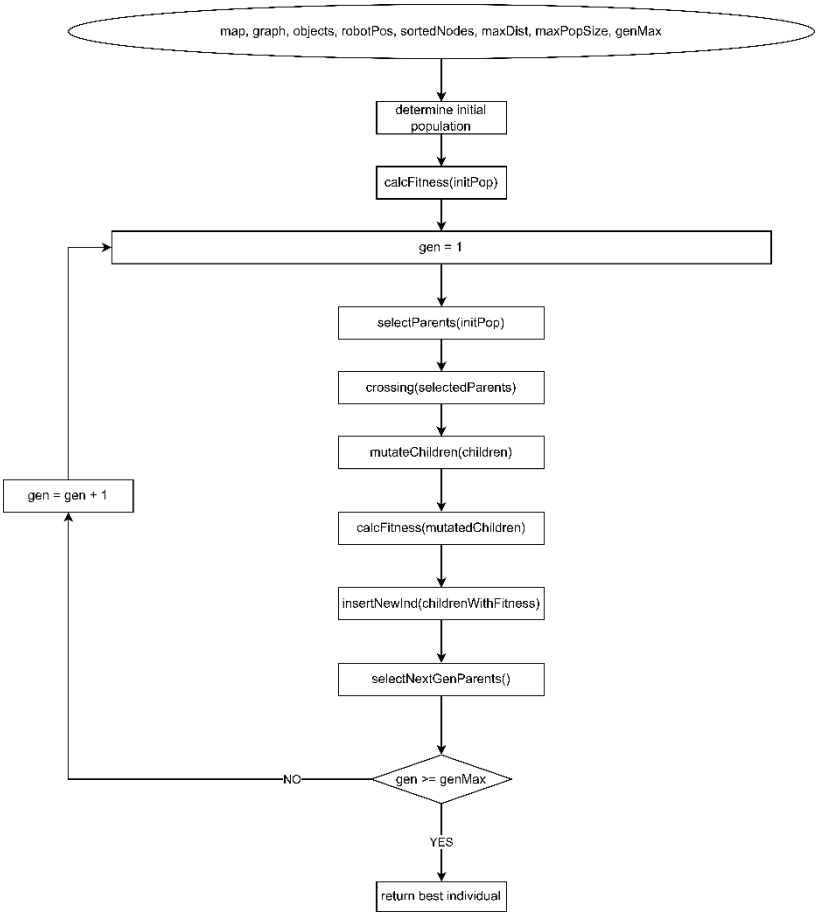
Felvételre kerülnek a gráfpontok és *x* koordinátájuk szerint növekvő sorrendbe teszem, valamint meghatározom azokat a csomópontokat, amelyek a mozgatható objektumokat tartalmazó struktúratömb két új tulajdonságának eleget tesz. A gráfot alkotó élek is felvételre kerülnek, de két különböző esetben is. A csak statikus akadályokat tartalmazó térkép alapján felvett élek alapján képzett gráf a meghatározott útvonal szemléltetése céljából van. Ezzel szemben, a célpontokban is elhelyezett objektumos térképen felvett élek alapján képzett gráf az útvonal meghatározására szolgál.

Ezt követően kerül a sorrendoptimalizálást megvalósító genetikus algoritmus meghívásra.

6.3.2 Sorrendoptimalizálás genetikus algoritmussal

A sorrendoptimalizációra genetikus algoritmust választottam. Azért esett erre a megoldásra a válaszom, mert a genetikus algoritmus, olyan optimalizációra alkalmas mesterséges intelligencia, amely képes globális optimum megtalálására.

Bemenetként szüksége van arra az Occupancy grid-re, amely a statikus akadályokat, és a mozgatható objektumokat tartalmazza, az utóbbit csak a kiinduló helyzeteikben. Szüksége van arra a gráfra, amelyet úgy generáltam, hogy figyelembe vettem a mozgatható objektumok kezdeti és célpontjukban elfoglaló helyet. Továbbá kell még a mozgatható objektumokat tartalmazó struktúratömb, valamint a rendezett gráfpontok listája, és a maximálisan felvehető élhosszak. Végezetül a robot kezdeti pozíciórája, a maximális populációra egy generáción belül, valamint hány generációt kell fusson az algoritmus. A következőben a genetikus algoritmus folyamat ábrája kerül szemléltetésre.



38. ábra - A genetikus algoritmus folyamat ábrája

A genetikus algoritmust alkotó függvények és a hozzájuk tartozó leírás a következő alfejezetekben kerülnek bemutatásra.

6.3.2.1 Kezdeti populáció létrehozása

A genetikus algoritmusom kezdetben nem rendelkezik egyetlen egyeddel sem, ezért szükséges, hogy létrehozzak néhány egyedet. Az egyedek génjét annyi kromoszóma alkotja, ahány objektumot kell mozgatni. Minden egyes kromoszóma egy-egy objektumot jelöl. A kromoszómák génben elfoglalt helyzetükkel, pedig a kódolt objektum sorrendjét jelöli. Tehát, ha adott az alábbi gén 7 darab kromoszómával, akkor a robot először a *2*-es számú objektumhoz megy és viszi azt el a célpontjába, majd a *6*-os számú objektumhoz és így tovább, míg az *5*-ös számú objektumot is nem juttattja a célpontjába.

2	6	7	1	3	5

39. ábra - 7 kromoszómát tartalmazó gén

A kezdeti populáció létrehozása úgy történik, hogy megvizsgálom, hány objektum kerül mozgatásra, és annyi egyedet hozok létre ahány objektumot kell mozgatni. Azért, hogy a populáció diverzitását növeljem, úgy hozom létre a kezdeti egyedeket, hogy mindegyik objektumazonosító egyszer egy egyed génjének első kromoszómája legyen. A további kromoszómákat génen belül véletlenszerűen veszem fel. Erre egy pszeudó-random szám generátort használok a MatLab beépített függvényei közül. Hét mozgatható objektum esetén kezdeti populációra példaként szolgál a következő táblázat, ahol a sorok az egyes egyedeket jelentik, az oszlopok az egyedekhez tartozó egyes kromoszómákat.

16. táblázat – 7 egyedes kezdeti populáció példa

	Chrom 1	Chrom 2	Chrom 3	Chrom 4	Chrom 5	Chrom 6	Chrom 7
Ind 1	1	3	5	7	6	4	2
Ind 2	2	4	6	7	5	3	1
Ind 3	3	6	7	4	2	1	5
Ind 4	4	2	6	1	3	5	7
Ind 5	5	7	3	2	4	6	1
Ind 6	6	3	2	4	1	5	7
Ind 7	7	5	3	1	6	4	2

A kezdeti populáció egyedeire kiszámításra kerül a fitnesz értékük, ami egy „jósági” faktornak felel meg, és elkezdődik az optimalizációi. Az algoritmus következő lépése a 6.3.2.2 fejezetben tárgyalt szülő egyedek kiválasztása.

6.3.2.2 Szülő egyedek kiválasztása

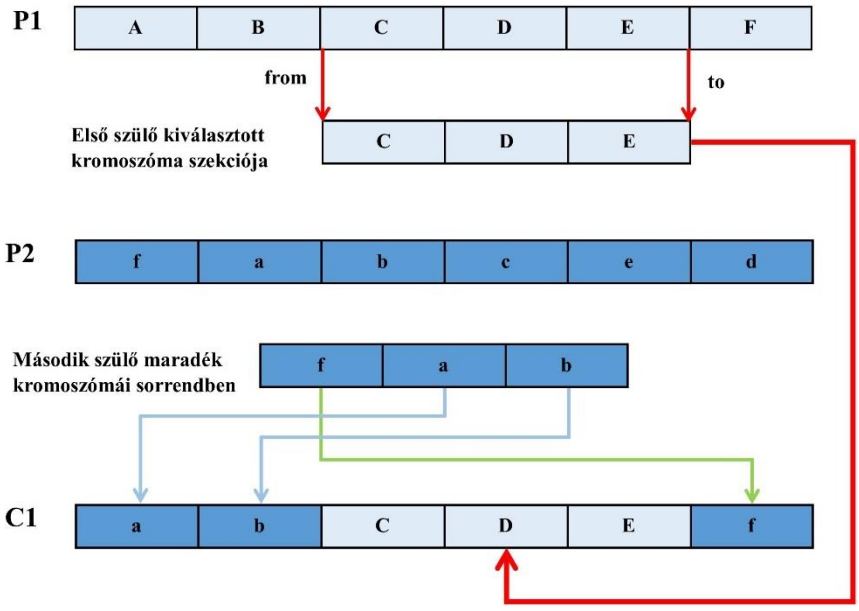
A genetikus algoritmus számára fontos kiválasztani, hogy a meglévő populációból mely egyedek kerülnek kiválasztásra, hogy új egyedek képződjenek. A kiválasztás során 3 párt választok ki a következő szempontok alapján. Kiválasztom a legjobb és a második legjobb fitneszértékkel rendelkező egyedet, melyek egy párt alkotnak. Ezt követően a legjobb egyed mellé kiválasztok véletlenszerűen a populáció többi tagjából egyet, amely nem lehet a második legjobb fitneszértékkel rendelkező, és ez alkotja a második párt. Végezetül a harmadik párt két véletlenszerűen kiválasztott egyed alkotja, melyek közül egyik se lehet a legjobb fitneszértékkel rendelkező egyed.

Ezzel a kiválasztási stratégiával az volt a célom, hogy biztosítsam a lehetőséget, hogy az utód a legjobb tulajdonságokkal rendelkező szülőktől kapott kromoszómákkal egy jobb fitneszértékű egyed lehessen. De amennyiben egy lokális optimum felé tartana a populáció, úgy a véletlenszerűen választott szülők révén esetleg a globális optimum felé tartó egyed is létre jöhessen.

6.3.2.3 Új egyedek létrehozása

Új egyed létrehozásánál a szülő egyedek kromoszómáit kell felhasználni. Mivel kombinatorikai problémára keresem a megoldást, ezért a leggyakrabban használt utódképzési módszerek nem voltak használhatók. Az új egyed képzésére a következő megoldást találtam.

Veszem a két szülőegyedet. Az első szülő (P1) esetén véletlenszerűen generálok két *pointer*-t, amelyek kiválasztják az öröklésre szánt kromoszómaszekciót. Ebben a szekcióban található kromoszóma értékeket elveszem a második szülő (P2) génjéből. Az elvétel után egy megfelelő hosszúságú kromoszómaszekciót kapok a második szülőtől. Az új egyedet (C1) úgy képzem, hogy az első szülőtől vett kromoszómaszekciót a megfelelő helyre beillesztem a gyerek génjébe, majd az üresen maradt kromoszómahelyeket feltöltöm a következők szerint. A gyerek génjéből hiányzó első *n* darab kromoszóma a második szülő maradék kromoszómájából az utolsó *n* darab kromoszóma értékét veszi fel. Hasonlóan a gyerek génjéből hiányzó utolsó *m* darab kromoszóma a második szülő maradék kromoszómájából az első *m* darab kromoszóma értékét veszi fel. Ezt szemlélteti a 40. ábra.

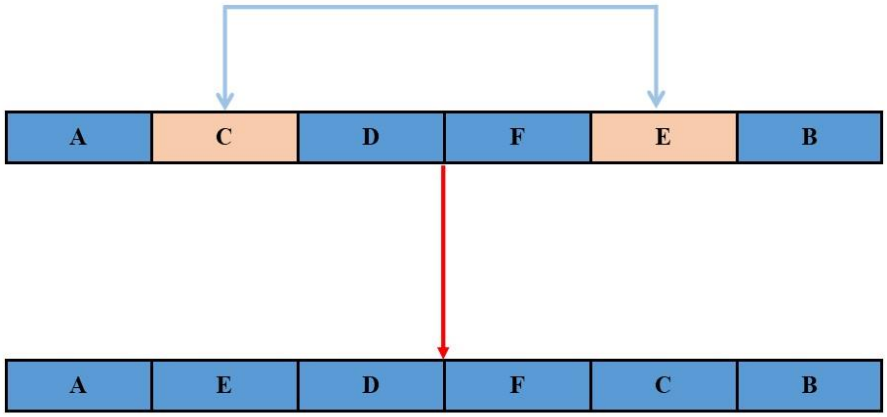


40. ábra - Új egyed létrehozásának folyamata

Ugyanezt elvégezem, úgy is, hogy a *P2*-es szülőt tekintem az első szülőnek és a *P1*-es szülőt pedig a második szülőnek. Így az utódok képzésénél egy szülőpáros két utódod hoz létre. Ezzel is az egyedek diverzitásának elérése a cél.

6.3.2.4 Egyedek mutációja

A diverzitás elérésében szerepe van a véletlenszerű mutáció alkalmazásának is. A mutációt úgy valósítottam meg, hogy két véletlenszerűen kiválasztott kromoszómát az egyed génjében kicseréltem. A mutáció bekövetkezésének valószínűségét 2.5%-ra választottam. A következő ábra szemlélteti a mutációt egy egyeden belül.



41. ábra - A mutáció megvalósítása

6.3.2.5 Fitneszfüggvény megvalósítása

A genetikus algoritmus legfontosabb függvényének tekinthető, hiszen ez alapján kerül meghatározásra, hogy egy egyed milyen jósági tényezővel rendelkezik. Ez a függvény számolja ki az adott kombinációhoz tartozó útvonal hosszát, valamint magát az útvonalat is. A függvény az összes egyedre kiszámolja a fitnesz értéket egyesével egy *for* cikluson belül.

Egy egyedre a következőféleképpen történik az útvonal meghatározása. Bemenetként szükség van:

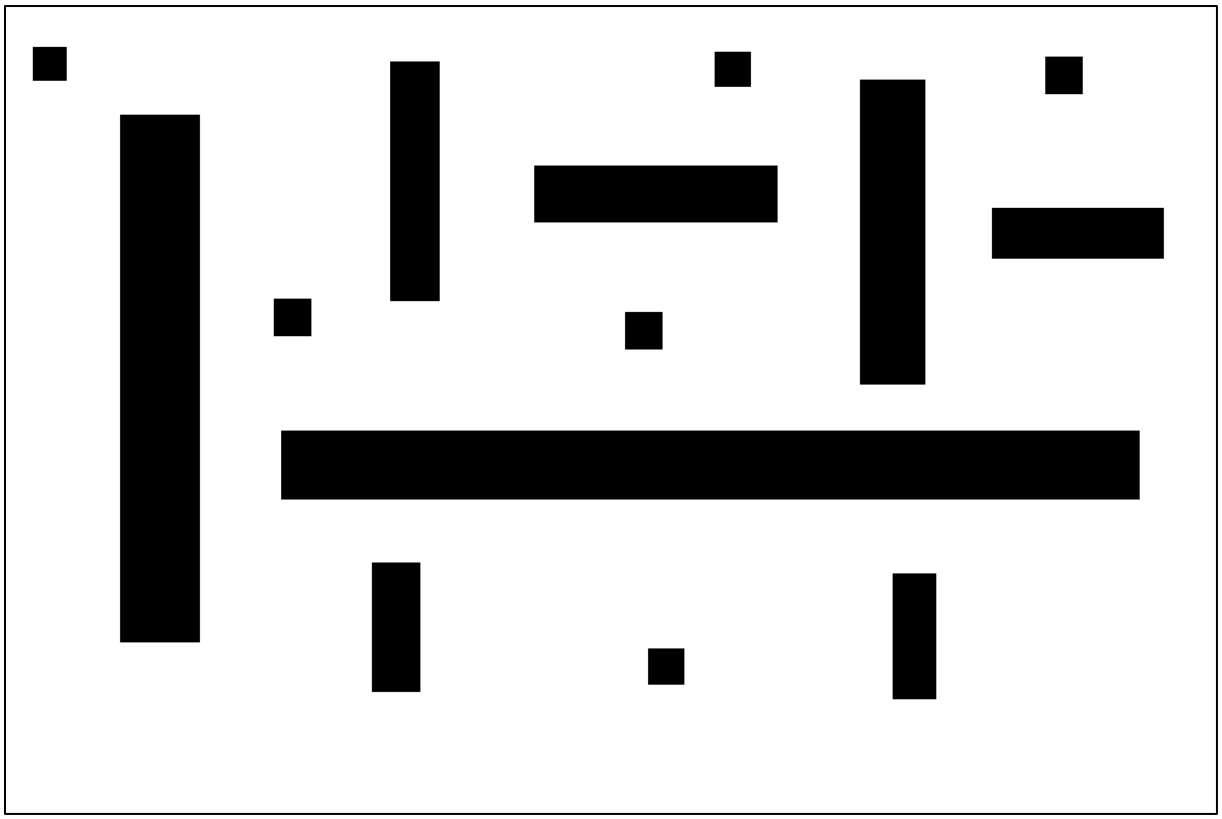
- A mozgatható objektumokat kezdeti pozíciójukban tartalmazó térképre
- A felvett gráfra (ez az a gráf, ami a megduplázott objektumokkal rendelkező térképen került felvételre)
- A mozgatható objektumok mozgatási sorrendjére
- Az összes objektum tulajdonságait tartalmazó objektumstruktúra tömbre
- A robot kezdeti pozíciójára
- A rendezett gráfpontokat tartalmazó tömbre
- A maximális élhosszra

A függvény visszatérésként a teljes útvonalat, és a megtett útvonal hosszát adja vissza. Első lépésként, mivel az ütközésmentes útvonal számításakor szükséges az objektumok áthelyezését is figyelembe vennem, ezért a térképről készítek egy másolatot. A második lépés, hogy a megadott sorrenden végig kell haladni és egyesével kiszámolni a kívánt objektumhoz vezető utat, valamint az objektum célpontjába vezető utat. Minden egyes számolás után eltárolásra kerülnek az addig megtett útvonalak. Ez közel azonos a két mozgatható objektumos esetben bemutatott megvalósítással.

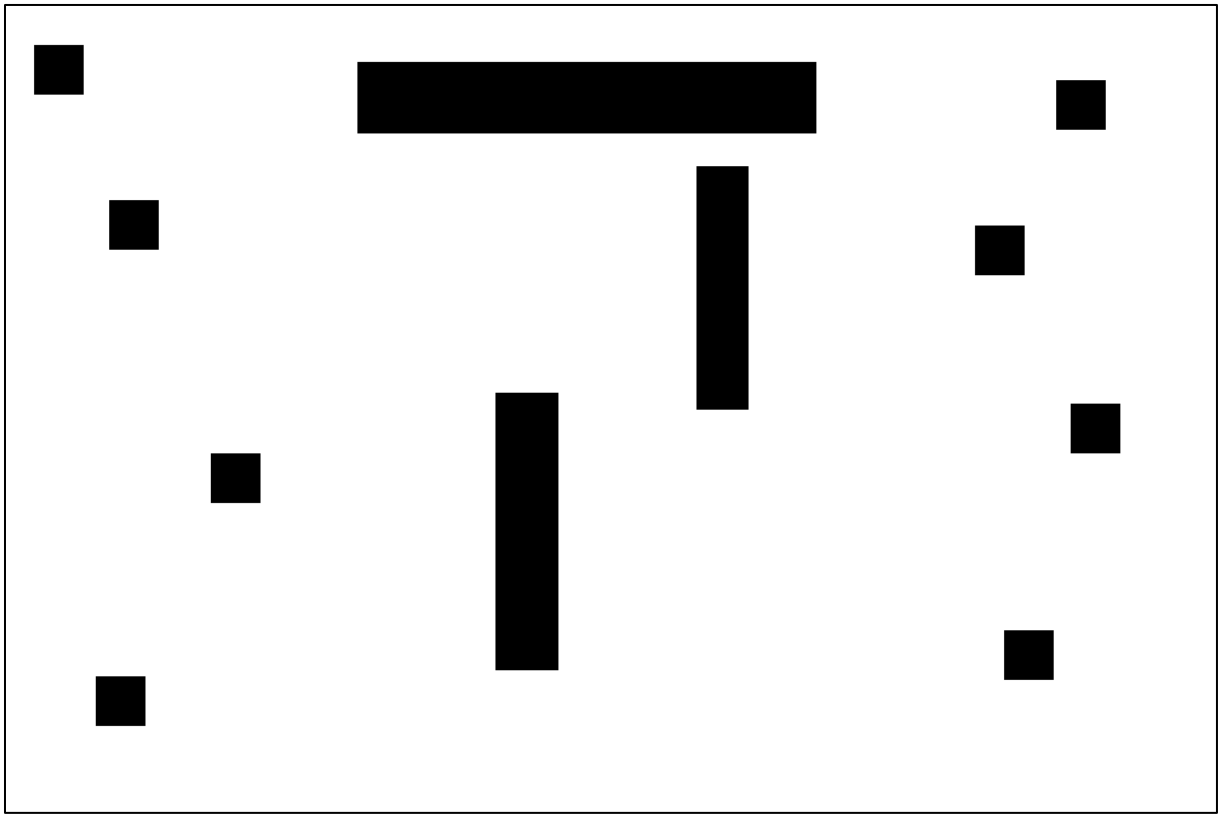
A legnagyobb eltérés az útvonal meghatározás során, hogy nem generálom mindig újra a gráf éleit, és ezáltal a gráfot. Ez úgy került kiküszöbölésre, hogy mindig eltárolásra kerültek azok a csomópontok, melyeket a mozgatható objektumok kezdeti-, illetve célpontjukban elfedtek. Az így eltárolt csomópontok két csoportra osztódtak. Az egyik csoportba azok a csomópontok kerültek, melyeket a mozgatható objektumok aktuálisan elfedtek, a másik csoportba pedig azok a csomópontok, melyek aktuálisan is a szabadon bejárható térbe estek. Minden egyes útvonaltervezés során a szabadon bejárható térbe eső csomópontokat, és a hozzájuk tartozó éleket a gráfhoz rendeltem. Az objektumok dilatálásával a szabadon bejárható térből kieső csomópontokhoz tartozó élek elvetésre kerülnek. Ezáltal a futásidő csökkenését lehetett elérni, mert nem egy új gráfot kellett kiszámolni, hanem csak egy részgráfot, és azt hozzárendelni a már meglévő gráfhoz.

7 Eredmények több akadályra

A több mozgatható objektum esetén is teszteltem az algoritmus egyes komponenseit. Ezt egyrészt azért is tettem, hogy a működésnek a sikerességéről meggyőződhessem, valamin, hogy a térkép Occupancy grid-ként való kezelése a futás időben javulást ér el vagy sem. A teszteket a következő két térkép által végeztem. Jelen esetben nem kerül bejelölésre, hogy melyik mozgatható objektum melyik célpontba kell jusson. Ez utóbbi csak az algoritmus működésének tesztelésekor kerül bemutatásra. A 42. ábra esetén jól látható, hogy a térkép jóval zsúfoltabb, mint a 43. ábra esetén. Azért valósítottam meg ezt a két esetét, mert elvárom az algoritmustól, hogy viszonylag bonyolultabb környezet mellett is megfelelő eredményt szolgáltasson.



42. ábra - A tesztelésre használt első térkép több mozgatható objektum esetén



43. ábra - A tesztelésre használt második térkép több mozgatható objektum esetén

Mivel jelen esetben részleges módosítást eszközöltem a térkép kezelésével, valamint a PRM-t megvalósító függvények közül az élfelvevést módosítottam, ezért újra teszteltem a gráfpontok felvevéséért és az élek meghatározásáért felelős függvények futásidejét. Ezt követően pedig összevettem a fentebb említett megvalósítással.

7.1 Csomópontok felvétele

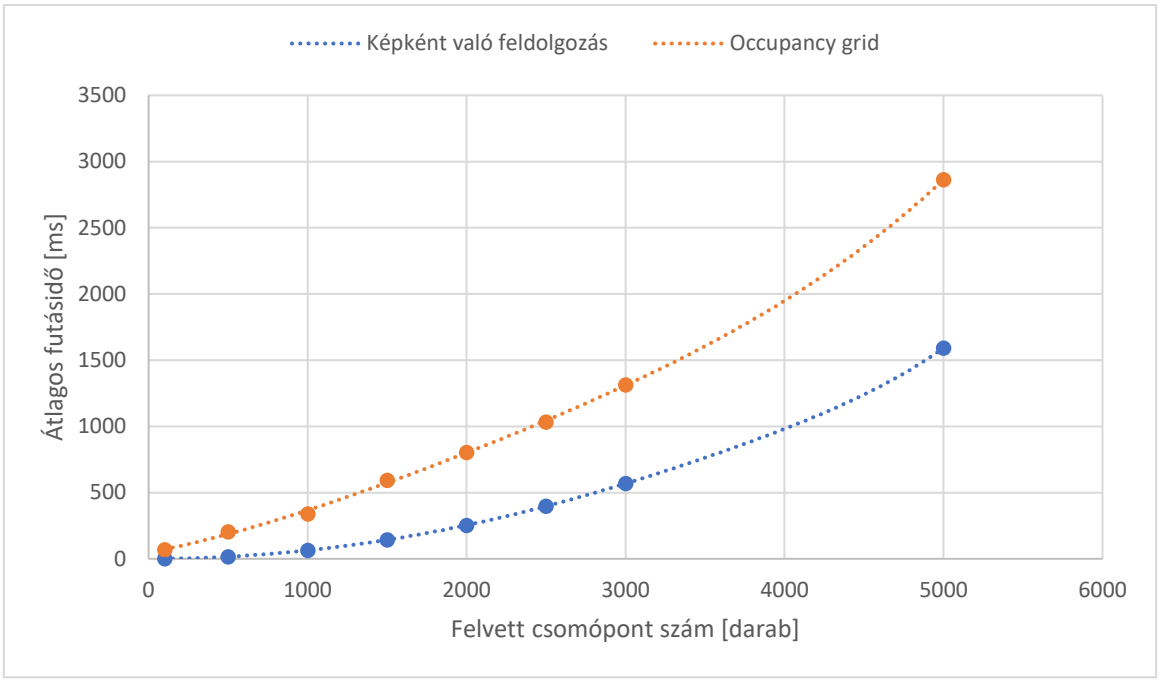
Jelen esetben nem szemléltetem, hogy a csomópontok felvétele után az egyes pontok hogyan fedik le a szabadon bejárható teret. Ezt tökéletesen szemlélteti a 7. ábra és a 8. ábra, amik az 5.1-es fejezetben találhatóak. Most főként csak a futásidőre térek ki részletesebben. Az átlagos futásidőket azonos mintavétellel végeztem, mint az 5.1-es fejezetben, valamint a teszteket azonos csomópont számokra végeztem el.

17. táblázat - Futásidők Csomópontok felvételére Occupancy grid alkalmazása mellett

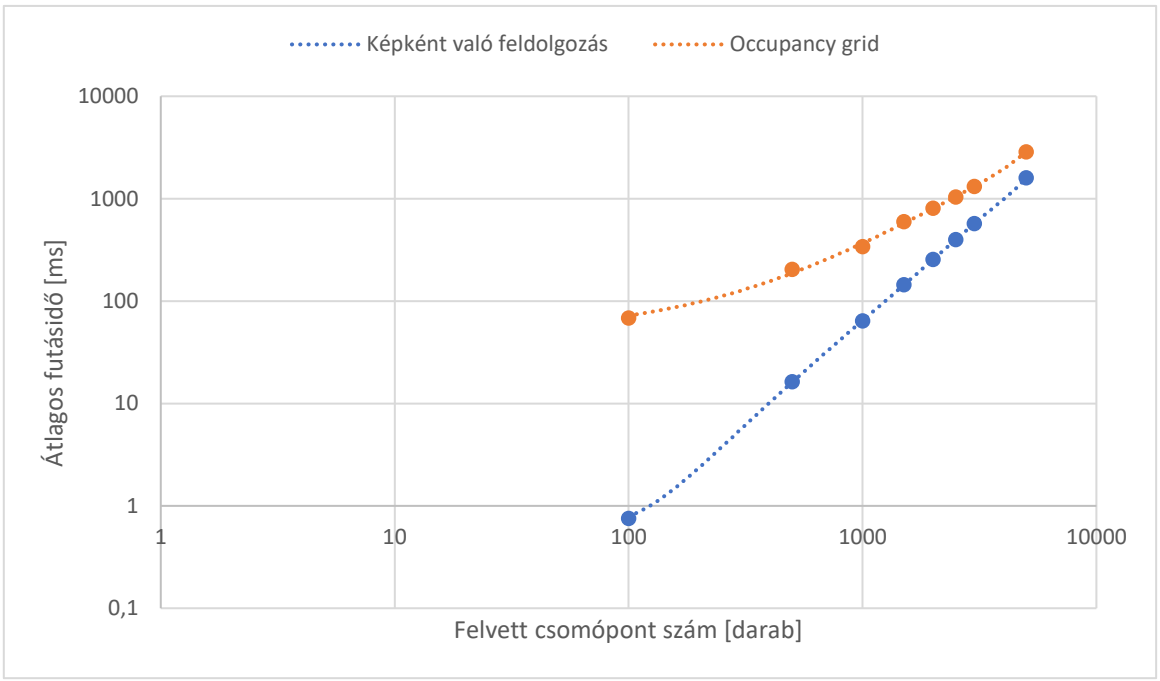
Csomópontok száma	100	500	1000	1500	2000	2500	3000	5000
Átlagos futásidő [ms]	68.15	203.26	339.86	593.76	802.48	1033.17	1314.36	2862.97

Sajnálatos módon azt tapasztaltam, hogy a csomópontok felvétele nagyságrendekkel több időt igényel Occupancy grid alkalmazásával, mint a térkép képként való feldolgozása esetén. Ezt a tendenciát a következő grafikon a 44. ábra-n is jól szemlélteti. Az a tendencia figyelhető meg, hogy a csomópontok számának növelésével az abszolút eltérés folyamatosan növekedett, de mindeközben a relatív eltérés csökkent. Egy adott csomópont mellett a relatív eltérés minden bizonnyal megközelíti az 1-et. Ez viszont számomra nem volt cél, sőt a csomópontszám akkorának adódik, ahol az egyet megközelítheti a relatív eltérés, hogy a futásidő drasztikusan megnövekszik.

A következtetés az, hogy amennyiben nem érhető el megfelelő mértékű javulás a gráfélek meghatározása során, úgy nem kifizetendő, hogy áttértem Occupancy grid alkalmazására. Erre a kérdésre ad választ a következő rész.



44. ábra - Az Occupancy grid (narancs) és a kép (kék) esetén a csomópontok felvételéhez szükséges átlagidő (lin-lin)



45. ábra - Az Occupancy grid (narancs) és a kép (kék) esetén a csomópontok felvételéhez szükséges átlagidő (log-log)

7.2 Gráfélek felvétele

A gráfélek felvételének hatékonyságára úgy tudok választ találni, ha azonos térképen alkalmazom mind a két megvalósítást (a térkép Occupancy grid-ként és képként való

tekintését). A tesztelést a 42. ábra által szemléltetett térképen végeztem el, és a következő eredményeket kaptam.

18. táblázat - Futásidő változása az élhosszak függvényében 100 darab csomópont esetén

Élhosszak (n = 100)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	-	-	0.752	4.523	14.653	40.728

19. táblázat- Futásidő változása az élhosszak függvényében 500 darab csomópont esetén

Élhosszak (n = 500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	-	-	14.967	93.583	300.367	898.938

20. táblázat - Futásidő változása az élhosszak függvényében 1000 darab csomópont esetén

Élhosszak (n = 1000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	-	-	23.145	78.493	587.647	1576	> 1h

21. táblázat - Futásidő változása az élhosszak függvényében 1500 darab csomópont esetén

Élhosszak (n = 1500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	1.136	7.809	46.651	134.403	827.595	2668	> 1h

22. táblázat - Futásidő változása az élhosszak függvényében 2000 darab csomópont esetén

Élhosszak (n = 2000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	2.045	14.157	87.012	251.834	1297	>1h	>1h

23. táblázat - Futásidő változása az élhosszak függvényében 2500 darab csomópont esetén

Élhosszak (n = 2500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	3.368	23.7896	344.762	943.786	>1h	>1h	>1h

24. táblázat - Futásidő változása az élhosszak függvényében 3000 darab csomópont esetén

Élhosszak (n = 3000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	4.645	32.284	201.365	946.357	>1h	>1h	>1h

25. táblázat - Futásidő változása az élhosszak függvényében 100 darab csomópont esetén

Élhosszak (n = 5000)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	13.178	252.175	806.946	>1h	>1h	>1h	>1h

A táblázatokból az olvasható ki, hogy azonos számú csomópontok mellett az élhosszak növelésével drasztikusan megnövekszik a futásidőigény. Ahhoz, hogy meg lehessen állapítani, hogy a módosítások sikeresek voltak vagy sem, össze kell hasonlítanom az 5.2 fejezetben bemutatottal, ugyanolyan térkép mellett. A fentebbi tábláztok alapján a 1500 darab gráfpontos vizsgálat során jöttek ki elfogadható futásidők, ebből kifolyólag ennek megfelelően hajtom végre a tesztet. A teszt eredményeit a 26. táblázat szemlélteti.

26. táblázat - futásidő változása az élhosszak függvényében 1500 darab csomópont esetén – a képként történő kezelés mellett

Élhosszak (n = 1500)	25	50	100	150	300	500	inf
Átlagos futásidő [s]	7.821	26.998	51.324	142.674	264.975	376.247	597.354

A két táblázat alapján (21. táblázat és 26. táblázat) látható, hogy javulást sikerült elérnem a futásidő tekintetében, amennyiben Occupancy grid-ként kezelem a térképet, de csak kis élhosszak esetén. Amennyiben nagyobb hosszúságú éleket engedek meg a gráfban, úgy az Occupancy grid rosszabb futásidővel rendelkezik. Ezek alapján nem lehet kijelenteni, hogy melyik megoldás rendelkezik jobb futásidővel. Ha az élek viszonylag rövidek, akkor Occupancy grid-ként érdemes feldolgozni a térképet, másképpen képként érdemes feldolgozni.

Végeztem egy másik tesztelést is a két gráfél meghatározó függvénnyel is. A teszt arra tért ki, hogy melyik megoldás futásideje változik nagyobb mértékben a térképen található objektumok számossága függvényében. Az Occupancy grid-ként való kezelés során azt tapasztaltam, hogy futásidő pár század másodperces szórással rendelkezett. Ezzel szemben a képként való kezelés során pedig akár nagyságrendbéli különbségek is jelentkeztek a futásidőben. Viszont a két megoldást összevetve azt tapasztaltam, hogy kis objektumszámosság (max. 7-8 darab) esetében a képként való feldolgozás némileg jobb futásidővel rendelkezik. Ezzel szemben, ha több objektum van a térképen, akkor az Occupancy grid-ként való feldolgozás nagyságrendekkel jobb futásidővel rendelkezik.

7.3 Az útvonaltervezés

Az útvonaltervezés során olyan megkötéssel kellett élnem, hogy a felvett gráfot alkotó élek hossza nem lehetett nagyobb, mint a mozgatható objektumok közül a legkisebb alapterületűnek a kisebbik dimenziójának a kétszerese. Erre azért volt szükség, mert másképpen lehetnek olyan élek a gráfban, amelyek nem kerülnek elvetésre az objektumok dilatálása után. Az élek hosszának maximalizálásával kiküszöbölhetővé vált ez a probléma.

A genetikus algoritmust megvalósítása során létrehozásra került egy look-up table, amelyben az addigi egyedek eltárolásra kerültek. Ezáltal a futásidő csökkenést vártam el, ugyanis az ismétlődő egyedek esetén nem kell újraszámolni a fitnesz értéküket. Tesztek révén

15 generáció és 10 egyedes maximális populáció mellett 2250 darab csomópont és 35-ös élhossz mellett ez a megoldás az algoritmus futásidejéből átlagosan ~100 másodperces javulást eredményezett mind a két térképen. Ez azt jelenti, hogy hat mozgatható objektum esetén 370 másodpercről 260 másodpercre csökkent az átlagos futásidő. Hét mozgatható objektum esetén 425 másodpercről 310 másodpercre csökkent az átlagos futáidő.

A genetikus algoritmus populációjának legjobb egyedét folyamatosan monitorozva minden generációban az a tendencia jelentkezet, hogy 7-10 generáció alatt a legjobb fitnesz értékű egyed előáll. A look-up table révén meghatározható, hogy átlagosan hány egyed jön létre a genetikus algoritmus futtatása során. Hat mozgatható objektum esetén átlagosan 65 darab különböző egyed került létrehozásra. Hét mozgatható objektum esetén pedig 75 darab különböző egyed került létrehozásra. Ezen adatok ismeretében egy egyed utáni útvonal kiszámításának ideje hat objektum esetén ~4.2 másodperc, míg hét objektum esetén ugyanúgy ~4.2 másodperc. Ezt az értéket összehasonlítva a két objektum mozgatásával jelentős futásidőbeli javulás volt elérhető.

Ezek alapján kijelenthető, hogy sikeresen megvalósításra került a megoldandó problémára egy prototípus algoritmus. A dolgozatban bemutatott rendszer rengeteg továbbfejlesztési lehetőséget rejt még magában.

8 További fejleszthetőségek

Több mozgatható objektum mellett az algoritmus számára nem engedhetem meg, hogy a gráf éleinek hossza nagyobb lehessen, mint a legkisebb objektum legkisebb kiterjedtségének kétszerese. Ez azért van, mert az élek elvételét úgy valósítottam meg, hogy a megnövelt objektum által lefedett csomópontokhoz tartozó éleket dobtam el. Ezt a továbbiakban fejleszteni kell, hogy ütközésmentes útvonalat lehessen tervezni bármilyen hosszúságú élek esetén

A genetikus algoritmust kell még úgy módosítani, hogy az alkalmas legyen kevesebb számú mozgatható objektum esetén is optimalizálni. Jelenleg az algoritmus négynél nagyobb számú mozgatható objektum esetén tud biztosan lefutni. Ez annak a következménye, hogy az egyedek keresztezését megvalósító függvény nincsen felkészítve kis kromoszóma számú gének kezelésére.

A genetikus algoritmust tovább kell fejleszteni, hogy amennyiben felismerhető, hogy egy objektumot semmiképpen nem juttatthatunk a célpontjába mindaddig, míg egy másikat nem mozgatunk el, akkor azt ismerje fel. Amennyiben sikeresen felismert egy ilyen helyzetet akkor az egyedek között tiltásra kerüljön, az ilyen kombináció.

A genetikus algoritmus kezdeti populációjának meghatározására érdemes lenne még bizonyos feltételek meghatározása. A térképen egy előfeldolgozás révén intuitíven meghatározni bizonyos kombinációkat, amelyek potenciálisan jó megoldáshoz vezetnek, és ezeket a megoldásokat beilleszteni a kezdeti populációba. Így várhatóan hamarabb megtalálható lenne az optimumút, és a futásidő is csökkenthető.

A gráfot alkotó élek és csomópontok felvételének gyorsítása mindenképpen csökkentené az algoritmus futásidejét. Jelenleg a csomópontok felvétele, illetve a gráfélek generálása is egy-egy *for* ciklus révén került megvalósításra. Ez azt jelenti, hogy mindaddig nem kerül felvételre egy másik gráfpont, gráfél, amíg az előző nem lett meghatározva. Erre egy optimális megoldás lenne az ilyen feladatok párhuzamosítása. Így a jövőben ezeket a részeket párhuzamosítani tervezem.

A kezelői felület jelenleg egyszeri futtatást biztosít, ugyanis nem rendelkezik hibakezeléssel. Ezt is fejleszteni kell a jövőben, hogy egymás után akár többszöri futtatást is meg lehessen valósítani a felületen keresztül, akár különböző térképek esetén is.

9 Bibliográfia

[1] Haradhan, “The First Industrial Revolution: Creation of a New Global Human Era,” *J. Soc. Sci. Humanit.*, vol. 5, no. 4, pp. 377–387, 2019.

[2] R. Engelman, “The Second Industrial Revolution, 1870-1914,” *U.S. History Scene*. https://ushistoryscene.com/article/second-industrial-revolution/

[3] R. N. Khan *et al.*, “The third industrial revolution Impact industrial and engineering data,” 2009.

[4] J. Rymarczyk, “Technologies, opportunities and challenges of the industrial revolution 4.0: Theoretical considerations,” *Entrep. Bus. Econ. Rev.*, vol. 8, no. 1, pp. 185–198, Mar. 2020, doi: 10.15678/EBER.2020.080110.

[5] K. Podobni and A. Nagy, “Legrövidebb útkereső algoritmusok,” 2009.

[6] Wikipedia, “A* search algorithm.” https://en.wikipedia.org/wiki/A*_search_algorithm (accessed May 21, 2022).

[7] I. Noreen, A. Khan, and Z. Habib, “A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms,” *IJCSNS Int. J. Comput. Sci. Netw. Secur.*, vol. 16, no. 10, pp. 20–27, 2016, [Online]. Available: http://cloud.politala.ac.id/politala/1. Jurusan/Teknik Informatika/19. e-journal/Jurnal Internasional TI/IJCSNS/2016 Vol. 16 No. 10/20161004_A Compar ison of RRT, RRT and RRT - Smart Path Planning Algorithms.pdf

[8] D. Kiss, “Pályatervezési és mozgásirányítási algoritmusok fejlesztése mobil robotokhoz Tartalomjegyzék,” 2014.

[9] Q. Li, Y. Xu, S. Bu, and J. Yang, “Smart Vehicle Path Planning Based on Modified PRM Algorithm,” *Sensors*, vol. 22, no. 17, 2022, doi: 10.3390/s22176581.

[10] A. Muhammad, N. R. Hasma Abdullah, M. A. H. Ali, I. H. Shanono, and R. Samad, “Simulation Performance Comparison of A*, GLS, RRT and PRM Path Planning Algorithms,” *2022 12th IEEE Symp. Comput. Appl. Ind. Electron. ISCAIE 2022*, pp. 258–263, 2022, doi: 10.1109/ISCAIE54458.2022.9794473.

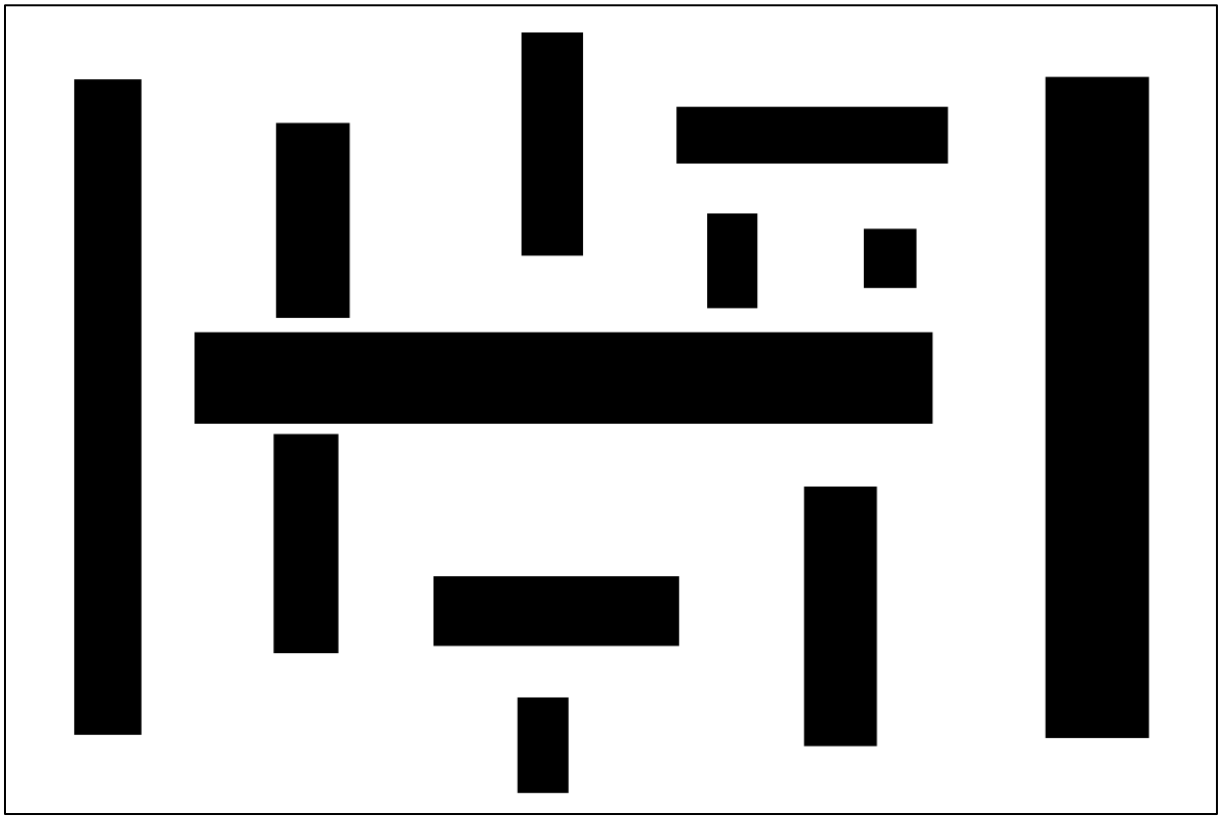
[11] C. T. Su, C. F. Chang, and J. P. Chiou, “Distribution network reconfiguration for loss reduction by ant colony search algorithm,” *Electr. Power Syst. Res.*, vol. 75, no. 2–3, pp. 190–199, 2005, doi: 10.1016/j.epsr.2005.03.002.

[12] “MatLab Wikipedia EN.” https://en.wikipedia.org/wiki/MATLAB (accessed May 21, 2022).

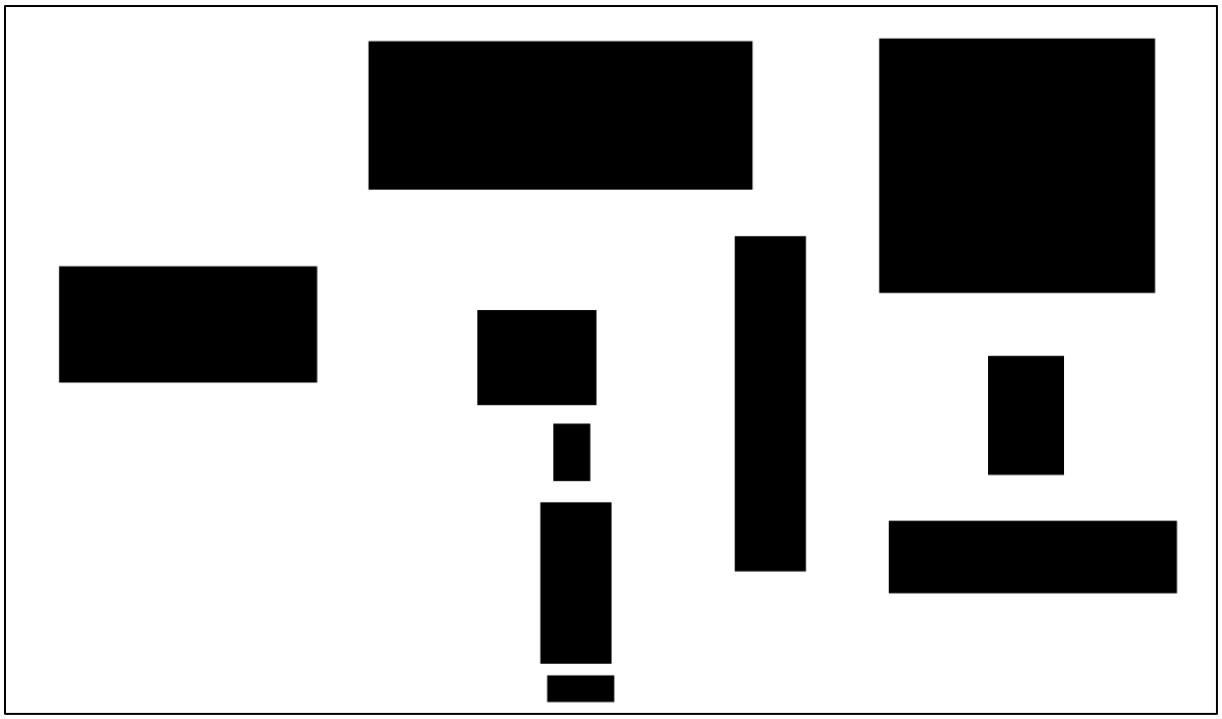
[13] “What is MATLAB.” https://cimss.ssec.wisc.edu/wxwise/class/aos340/spr00/whatismatlab.htm (accessed May 21, 2022).

10 Függelék

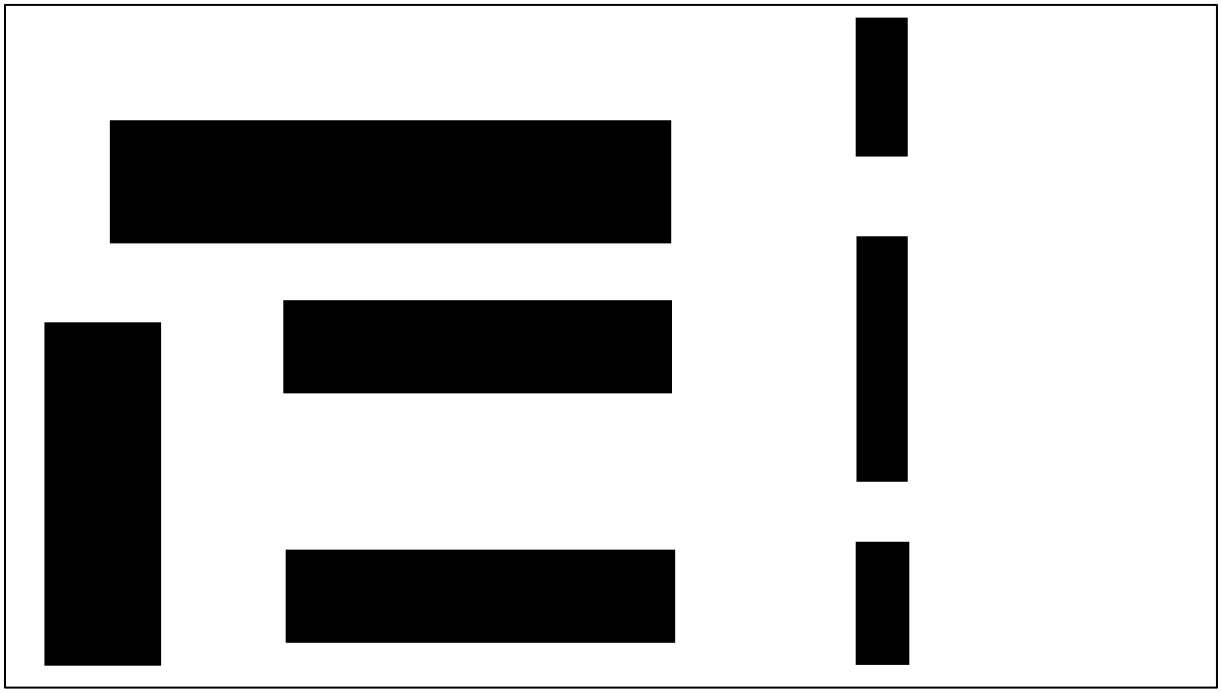
10.1 Szimulációhoz használt térképek



ábra 46 - Komplex teszt térkép



ábra 47 - Közepes teszt térkép



ábra 48 - Egyszerű teszt térkép