
**Budapesti Műszaki és Gazdaságtudományi Egyetem**
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Maróti Árpád

# DINAMIKUS ABLAK ALAPÚ NAVIGÁCIÓS ALGORITMUSOK VIZSGÁLATA VALÓS ROBOTON

KONZULENS

Kiss Domokos

BUDAPEST, 2012

# Tartalomjegyzék

# Összefoglalás

Mobil robotok esetében az egyik legösszetettebb megoldandó feladat a robot navigációjának megvalósítása. Egy jó navigációs algoritmusnak ütközés nélkül, biztonságosan el kell juttatnia a robotot a kiindulási pontból a célpontba, lehetőség szerint minél gyorsabban.

Ebben a dolgozatban egy ilyen algoritmus fejlesztése kerül bemutatásra. Munkám célja egy olyan megoldás kifejlesztése volt, melyet a tanszék Eurobot nemzetközi autonóm robotversenyre készített robotján fel lehet használni.

Munkámat a szakirodalom megismerésével kezdtem, mely alapján már el tudtam dönteni, hogy milyen megoldások jöhetnek szóba jelen esetben. Választásom egy olyan algoritmus kifejlesztésére esett, mely a dinamikus ablak megközelítésre épül.

Ezt követően hozzáláttam a fejlesztéshez. A dolgozat jelentős része ennek folyamatát írja le. A harmadik fejezetben először ismertetem a pálya és a robot paramétereit tároló objektumokat. Ezután az ütközésdetektálást, majd a dinamikus ablak módszer megvalósítását kerül tárgyalásra.

A negyedik fejezetben a lokális jellegű navigációs algoritmusok lokális minimum problémájának kiküszöbölését célzó megoldás ismertetése olvasható.

Az ötödik fejezet az általam készített szimulátor működését mutatja be, majd a valós roboton futtatott tesztek eredménye következik.

# 1 A feladat

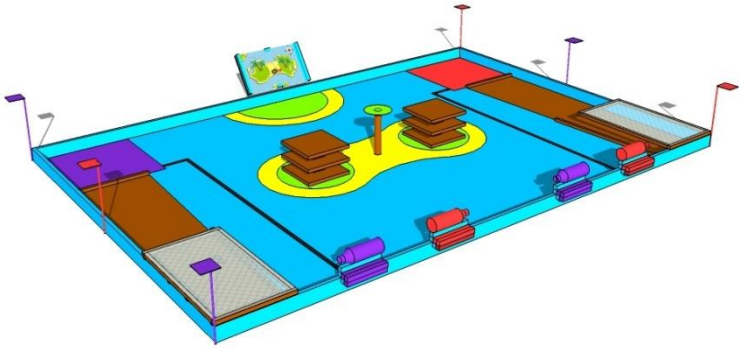
Mobil robotok szoftverének tervezése során az egyik leglényegesebb megoldandó feladat a navigáció. Navigációs algoritmus alatt azt a szoftveres megoldást értjük, ami meghatározza, hogy a robot „A” pontból „B” pontba milyen útvonalon jut el. Ehhez szükséges figyelembe venni a robot környezetéről szerzett információkat, illetve a robot felépítéséből és dinamikai tulajdonságaiból adódó korlátait. Egy navigációs algoritmus akkor tekinthető működőnek, ha képes a robotot ütközésmentesen irányítani, illetve amennyiben létezik a célponthoz vezető útvonal, megtalálja azt. Két működő navigációs algoritmus közül a gyorsabbat érdemes előnyben részesíteni, ugyanis amennyiben a robot hamarabb ér a munkavégzési pontba – azaz kevesebb ideig van úton – több munkát tud elvégezni adott idő alatt.

Feladatom az volt, hogy a tanszéken az Eurobot versenyre készülő robothoz tervezzek és implementáljak navigációs algoritmusokat. Amennyiben sikerül elkészíteni egy olyan algoritmust, melynek segítségével az Eurobot versenyen hatékonyan tud mozogni a robot, a jövőben a csapat felhasználná azt. Természetesen egy ilyen robot túlságosan nagy értéket képvisel ahhoz, hogy teszteletlen navigációs algoritmust futtassunk rajta, szükséges volt egy olyan program elkészítése is, amiben szimulálni lehet a robotot, illetve az algoritmus működését.

Ezen kívül feladatom volt egy olyan program készítése a roboton található Nokia táblaszámítógépre, mely képes a robot aktuális állapotát kijelezni, ezzel segítve a fejlesztés menetét.

## 1.1 A pálya

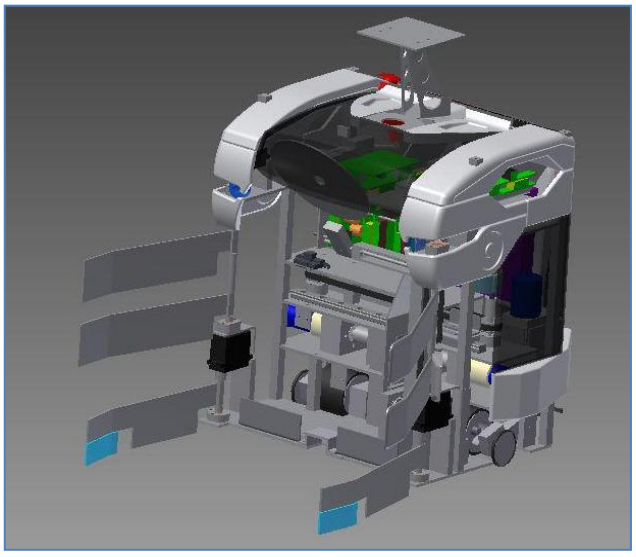
Az Eurobot verseny pályája előre ismert (évről évre változik), 2 méter széles és 3 méter hosszú terület. A rajta található akadályok helye fix, és ez a verseny során nem is változik. A pályán azonban található a saját robotunkon kívül egy vagy két ellenfél robot is, melyek pozíciója folyamatosan változik, tehát a navigációs algoritmust muszáj felkészíteni a környezet változására.



**1. Ábra: A 2012-es Eurobot verseny pályája**

## 1.2 A robot

Az irányítandó robot 33 cm széles és 29 cm hosszú, körülbelül 50 cm magas. Az elején található két kinyitható kar, melyek segítségével a versenyen összegyűjtendő pénzérméket jelképező CD lemezeket tudja terelgetni. Ezen kívül nyitott állapotban le tud hajtani középen egy kart, melynek végén egy tapadókorong található (egy szivattyúval lehet benne vákuumot létrehozni). Ezzel a karral képes az előtte található érmet megfogni, és a roboton található tárolóba helyezni.



**2. Ábra: A tanszéken készülő robot**

### 1.2.1 A robot fizikai korlátai

A robot differenciál hajtású. A két hajtott kereket, melyek egy tengely mentén találhatók, egy-egy motor hajtja. Ezen motorok szabályzóinak sebesség-alapjelét fogja

előállítani az általam készített navigációs algoritmus. Látható, hogy az ilyen felépítésű robotoknak két irányítható szabadsági fokuk van, ugyanis a robot pillanatnyi mozgásállapotát kizárólag a két kerék sebessége határozza meg. Azonban a robot pozíciójának meghatározásához három adatra van szükségünk: a középpontjának koordinátáira, illetve a robot orientációjára. Tehát a robot mozgástere három szabadsági fokú. Amennyiben a mozgástér szabadsági foka nem egyezik meg az irányítható szabadsági fokok számával, nem holonomikus robotról beszélünk.

A navigációs algoritmusnak tehát szem előtt kell tartania, hogy a robot aktuális helyzete meghatározza, hogy milyen irányú gyorsító erőt lehet éppen kifejteni a rá. (A robot pillanatnyi sebességvektora mindig merőleges a kerekeinek tengelyére.)

### 1.2.2 A robot dinamikai korlátai

A robotra érvényesek a két kereket hajtó motor mechanikai és elektronikai korlátozásai, azaz a kerekek nem foroghatnak bármekkora fordulatszámmal. A kerekek gyorsulásának nagysága is korlátozott, egyrészt a motorok által kifejthető maximális erő által, másrészt a kerék és a talaj közötti tapadás által. Könnyen belátható, hogy a robot megcsúszása nem megengedhető, ugyanis ebben az esetben nem tudjuk a robot mozgását teljes mértékben irányítani.

Amennyiben a robot nem egyenes pályán halad, hat rá centrifugális erő. Amennyiben ez az erő túl nagy, a robot egyik kereke elemelkedhet a talajtól, mely szintén nem megengedhető.

# 2 Navigációs algoritmusok

## 2.1 Globális és lokális algoritmusok

A navigációs algoritmusokat alapvetően két típusba sorolhatjuk. Globális navigációs algoritmusok közé soroljuk azokat a megoldásokat, melyek a robot teljes mozgásterének pillanatnyi állapotát használják fel működésükhöz, azaz a teljes környezetüket ismerik. Lokális navigációs algoritmusok azok az algoritmusok, melyek csak a robot közvetlen környezetének ismereteire építenek.

Amennyiben a robot mozgástere nagyon ritkán változik, és a robot számára rendelkezésre áll a mindenkori abszolút pozícióinformáció, előnyösebb egy globális algoritmus használata. Ezek nagy előnye, hogy lehetőség van az optimális út megtalálására, illetve globális mivoltából adódóan képes eldönteni, hogy egy adott célpontba el lehet-e jutni. Azonban ezeknek a jó tulajdonságoknak ára van, ugyanis egy ilyen algoritmus változó környezetben nagyon nagy számítási kapacitást igényel, mert a terület bármely pontján bekövetkező változás miatt újra kell számolni az egész modellt.

Szinte kizárólag lokális algoritmusok jöhetnek szóba abban az esetben, ha a robot előre nem ismert környezetben mozog. Ilyen algoritmusra van szükség akkor, ha csak a roboton találhatóak olyan szenzorok, melyek segítségével „lát”, azaz fel tudja térképezni a környezetét. A lokális algoritmusok előnye az, hogy változó környezetben is jól tudnak működni, illetve a robot pályájának területével nem nő a feldolgozandó információk mennyisége, azaz az algoritmus futásideje. Az ilyen típusú megoldások fontos hátulütője az, hogy nem garantálható, hogy a robot megtalálja az utat a célponthoz. Előfordulhat, hogy robotunk bent ragad egy lokális minimumban, és megáll célba érés előtt.

Ennek a problémának a kiküszöbölésére be szoktak vezetni egy globális jellegű kiegészítő információt [2], melynek segítségével a robot el tudja hagyni a lokális minimumot, és meg tudja találni az utat a célponthoz. Azonban ekkor sem garantált, hogy az ideális, leggyorsabban teljesíthető útvonalon fog közlekedni a robot.

## 2.2 Példa globális algoritmusra

A robot pályáján található akadályokat vegyük körül olyan sokszögekkel, melyek oldalai mentén a robot még éppen elfér. A sokszögek csúcsai legyenek egy gráf pontjai, az oldalai legyenek ennek a gráfnak az élei.

Ezután határozzuk meg akadálypáronként azokat a csúcspontokat, melyek kölcsönösen láthatóak, és az őket összekötő szakaszokat szintén vegyük fel a gráfba. Az így elkészített gráfot hívják láthatósági gráfnak (visibility graph).

Ha a kiindulási pontot, a végpontot és az azokból látható pontokba vezető éleket felvesszük a gráfba, akkor az útvonaltervezés egy egyszerű gráfbéli legrövidebb út problémára redukálódik. Erre jó módszer Dijkstra algoritmusa, melynek számításigénye a pontok számától ($V$) és az élek számától ($E$) függően:

$$O(|V| * \log(|V|) + |E|)^1 \qquad 2\text{-}1$$

Az élek száma lényegesen lecsökkenthető, ha eltávolítjuk azokat az útvonalakat, melyek meghosszabbított egyenese metszené az akadályokat. Egyszerűen belátható, hogy ezek az élek a legrövidebb útvonalakban soha sem szerepelnek.

Ezen megoldás nagy előnye, hogy mindig megtalálja a legrövidebb utat a kezdőpont és a célpont között. Ellenben közel sem biztos, hogy ez az út a leggyorsabb is egyben. Ugyanis csak abban az esetben garantált az ütközésmentes haladás, ha a robot soha nem tér le az élekről. Ebből következik, hogy minden pontban meg kell állni, irányba fordulni, és azután folytatni a haladást. (Ha nem képes a robot egy helyben megfordulni, az tovább bonyolítja a helyzetet.) Amennyiben fékezés során nem termeljük vissza az energiát, ez a megoldás nem is energiahatékony.

Az algoritmus számításigénye azonban akkor nő meg igazán, amikor mozgó objektumok jelennek meg a pályán. Ez mindig a gráf valamilyen szintű újraszámolását jelenti. Egyszerűbb esetben csak azt kell ellenőriznünk, hogy a mozgó objektum (például az ellenfél robotja) metszésbe kerül-e valamelyik, az aktuális útvonalunkban szereplő éllel. Ha igen, akkor ezt az élt ki kell vennünk a gráfból, és újra útvonalat kell tervezni. Bonyolultabb helyzetben, ha az akadályok is mozognak, előfordulhat, hogy a

[1] Dijkstra eredeti algoritmusa $O(|V^2|)$ komplexitású, de kis módosítással elérhető a feltűntetett komplexitás.

teljes láthatósági gráfot is újra kell számolni. Ebben az esetben már nem éri meg globális navigációs algoritmus használata.

## 2.3 A dinamikus ablak megközelítés

A dinamikus ablak megközelítés [1] egy olyan akadályelkerülési stratégia, melyre nagyon sok (főként lokális) navigációs algoritmus épül. Ciklikus lefutású, nagy előnye, hogy figyelembe veszi a robot dinamikai tulajdonságait.

### 2.3.1 A sebességtér

Sebességtérnek nevezzük a robot által felvehető sebességek összességét. A sebességteret általában komponensekre bontva koordinátarendszerben ábrázolják. Ilyen felbontás lehet egy holonomikus, kör alakú robot esetében x és y tengely irányú sebesség szerinti felbontás.

Differenciál hajtású robotok esetén kétféle ábrázolás is elképzelhető. Az egyik megoldás az lenne, ha az egyik tengelyre a haladási sebességet vennénk fel, a másikra pedig a szögsebességet. A két kerék sebességéből ($v_r$ és $v_l$) az alábbi képletek alapján kaphatjuk a haladási- és szögsebességet (v és ω).

$$v = \frac{v_r + v_l}{2} \qquad 2\text{-}2$$

$$\omega = \frac{v_r - v_l}{W} \ (W: kerekek\ távolsága) \qquad 2\text{-}3$$

Látható, hogy a két változó nem független egymástól, így sokkal előnyösebb, ha a két általunk vezérelt változó szerint ($v_r$ és $v_l$) ábrázoljuk a sebességteret.

### 2.3.2 A dinamikus ablak megközelítés lépései

A ciklikus lefutású algoritmus lépései a következőek, differenciál hajtású robotra nézve.

- Az aktuális sebességből kiszámolja, hogy a következő ciklusig milyen sebességeket tud elérni a robot. Ezen sebességek a $v_r$ és $v_l$ szerint ábrázolt sebességtérben egy téglalapon belül találhatóak. Ez a téglalap a dinamikus ablak, innen az elnevezés.
- Ennek a téglalapnak a területét valamilyen módon mintavételezve megkapjuk a vizsgálandó sebességek listáját.

- Minden vizsgált sebességhez egy-egy körív tartozik, mely mentén a robot középpontja halad.
- Megnézzük minden ívre, hogy ezt követve a robot milyen hosszú út megtétele után ütközik először akadályba. Amennyiben ez a távolság hosszabb annál az útnál, amekkorát a robot az ívhez tartozó sebességgel egy ciklust haladva, majd még mindig az ívet követve nullára fékezve megtenne, a sebességet elfogadhatónak tekintjük. Másként kifejezve azokat az íveket kiszűrjük, mely mentén a robot a következő ciklust követően nem tudna lefékezni.
- Az elfogadható sebességeket az alábbi tényezők szerint rangsoroljuk:
    - Milyen messze halad az ív az akadályoktól?
    - Milyen szögben áll a robot a lefékezési pontban a célpont irányához képest?
    - Milyen sebességgel fog haladni a robot a következő ciklus elején?
- A tényezőket súlyozva az algoritmus megállapítja, hogy melyik a legmegfelelőbb ív, és a hozzá tartozó keréksebességeket kiadja a motorvezérlőknek.
- A következő ciklusban kezdődik az egész folyamat elölről.

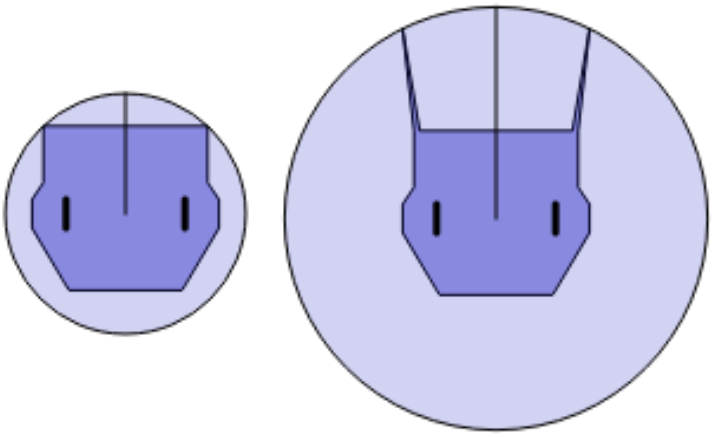
A megoldás fontos előnye az, hogy olyan sebességekre nem végzi el a számításokat, melyeket a robot a következő ciklusig nem tud elérni.

Önmagában a dinamikus ablak módszer érzékeny a lokális minimumokra. Az erre épülő akadályelkerülési algoritmusokban ezért szükséges valamilyen módot találni ennek kiküszöbölésére.

### 2.3.3 Az algoritmus számításigénye

Általánosságban elmondható, hogy a navigációs algoritmusok számításigényesek. Léteznek azonban olyan „trükkök”, melyek segítségével nagyban lehet ezeket gyorsítani. Az algoritmus azon fázisában, amikor eldöntjük, hogy adott ív mentén haladva lesz-e ütközés, nem mindegy, hogy milyen alakú a robotunk. Amennyiben van lehetőség arra, hogy a robotot kör alakúnak tekintsük, nagyban leegyszerűsödik a számítás, ugyanis csak azt kell figyelnünk, hogy a robotunk középpontja kerül-e a körének sugaránál kisebb távolságba az akadályoktól. Amennyiben nem, nincsen ütközés. Azonban ezt az egyszerűsítést sokszor nem

engedhetjük meg magunknak, ugyanis előfordulhat, hogy túl nagy körrel kell közelítenünk a robotunkat, és így az algoritmus gyakran mondaná egy olyan útvonalra azt, hogy nem járható, miközben valójában igen. Ez a helyzet az Eurobot versenyre készülő robottal is, ugyanis amikor a két karja nyitva van, nagyon nagy a befoglaló körének sugara.



**3. Ábra: A robot alakja bezárt, illetve nyitott karral**

# 3 Navigációs algoritmus megvalósítása

A lehetőségeket mérlegelve úgy döntöttem, hogy egy lokálisan működő, dinamikus ablak megközelítést alkalmazó navigációs algoritmust fejlesztek a robotra. Ebben a fejezetben először az általam használt adatstruktúrákra, azaz a robot és környezete számítógépes modelljére térek ki. Ezt követően ismertetem az ütközésdetektáló algoritmus működését mely megállapítja, hogy adott íven haladva mennyi utat lehet megtenni az első ütközésig. Ezután részletezem a sebességtér mintavételezésének különböző módszereit, majd az elfogadható sebességek rangsorolásának lehetséges módjait.

## 3.1 Alakzatok tárolása

Az útvonaltervezés legkomplexebb része az a szakasz, amikor kiszámítjuk, hogy adott íven haladva a robot ütközik-e akadállyal. Ennek elvégzéséhez valamilyen szoftveres modellt kell alkotnunk a környezetről, azaz le kell tárolnunk a robot, és a környezetében található akadályok alakját. Alapjában véve kétféle megoldás létezik erre a problémára.

Az első esetében a térképet bitmap-ként tároljuk, azaz egy rácsot illesztünk a pálya fölé, és minden rácspontról megjegyezzük, hogy található-e alatta akadály, vagy nem. A robot alakjáról szintén egy hasonló leképezést hozunk létre. Ennek az az előnye, hogy viszonylag egyszerű meghatározni, hogy a robot „belelóg-e” valamelyik akadályba. Egy másik előnye, hogy a rács felbontását állítva csökkenthetjük az algoritmus futási idejét, a pontosság rovására (persze itt sem szabad megengedni, hogy a robot ütközzön). Hátránya, hogy nagyobb memóriát igényel. Például abban az esetben, ha egy nagy üres pályán egy kicsi akadály található, az egész pályát az akadálynak megfelelő felbontásban kell eltárolni. Persze ezután a képet lehetséges különböző megoldásokkal tömöríteni (például négyágú faként [3] (quadtree) tárolni).

A második esetben a robotot és az akadályokat, mint geometriai formákat tároljuk [4]. Ebben az esetben az ütközéseket geometriai alakzatok metszésvizsgálatával állapítjuk meg. Ennek a megoldásnak az az előnye, hogy a felbontása sokkal nagyobb, mint a bitmap-es megoldásnak, tulajdonképpen a számításokban használt számábrázolás felbontásától függ (például lebegőpontos double, vagy egész számok). Itt a felhasznált memória mennyisége az alakzatok bonyolultságától függ, például sokszögek esetében a

csúcsok számától. Hátránya, hogy az ütközésvizsgálatot itt geometriai műveletek sorozatával lehet elvégezni, ahol elemi műveleteknél bonyolultabb számításokat is el kell végezni (például gyökvonás vagy szögfüggvények használata). Azonban mivel a jelenleg piaci forgalomban kapható nagyobb teljesítményű beágyazott számítógépekben már alapkövetelmény a matematikai koprocesszor, ezt nem tulajdonítottam túl nagy hátulütőnek.

A két lehetséges megoldás jó és rossz tulajdonságait mérlegelve végül úgy döntöttem, hogy a geometriai alakzatokat tároló megoldást valósítom meg. A választás során azt is figyelembe vettem, hogy amennyiben geometriai alakzatként tárolom a pályát, később abból könnyen le lehet generálni egy bitmap alapú leképezést.

Elkészítettem az alapvető geometriai fogalmaknak megfelelő osztályokat, majd megvalósítottam számos ezeken végezhető geometriai műveletet. Ezután elkészítettem azokat az osztályokat, amelyek az akadályokat, illetve a robot alakját tárolják. A továbbiakban ezekről az osztályokról következik egy részletesebb ismertetés.

### 3.1.1 A coord_pair osztály

A coord_pair osztály képes egy koordinátapár tárolására. Tartalmát meg lehet adni, illetve le lehet kérdezni euklideszi, illetve polárkoordinátás rendszerben is. Ezen kívül ez az osztály tartalmaz fontos segédfüggvényeket, mint például a szögek normálását végző függvényt, és a két irány közti szöget megállapító függvényt.

### 3.1.2 A point és a vector osztály

A point osztály a coord_pair leszármazottja. Annyiban terjeszti ki annak funkcióit, hogy megvalósítja az összeadás illetve kivonás operátort pontra, illetve vektorra. Ezen kívül lehetőség van XML-ből történő beolvasásra is.

A vector osztály szintén a coord_pair leszármazottja. Az osztály megvalósít különböző vektorműveleteket, illetve ezt a típust használjuk különböző alakzatok eltolására.

### 3.1.3 A line osztály

A line osztály egy egyenest ír le. Tartalmaz egy pontot, melyen az egyenes áthalad, illetve egy vektort, mely az egyenes irányába mutat.

### 3.1.4 A line_segment osztály

A line_segment osztály a line leszármazottja, és egy egyenes irányított szakaszt jelképez. Itt a line-ból örökölt pont a szakasz kezdőpontja, és a vector mutat a szakasz végpontjába. Az irányítottságnak az a jelentősége, hogy így értelmet nyer a jobbra illetve balra történő eltolás fogalma, melyet a későbbiekben felhasználunk.

Itt lehetőség van XML-fájlból történő beolvasásra, illetve az origótól mért távolság lekérdezésére. Ezen kívül tartalmaz egy mutatót, mely egy azonos típusú objektumra mutat. Ez utóbbit az akadályelkerülő algoritmus használja.

### 3.1.5 A circle osztály

A circle osztály egy kört leíró típus. A kört középpontjával és sugarával adhatjuk meg. Tartalmaz két darab line_segment-re mutató pointert, melyet az akadályelkerülő algoritmus használ. Meg tudja állapítani egy pontról, hogy az a körön belül található-e, illetve implementálásra került a metszéspontot kiszámító algoritmus egyenesre, szakaszra, illetve körre.

### 3.1.6 Az arc osztály

Az arc osztály egy körívet tartalmaz, és a circle és a line_segment közös leszármazottja. Ebből következik, hogy van kezdőpontja, végpontja, sugara, illetve középpontja. Ezek meghatározzák a „kör-szakasz” irányát is, melyet lekérdezhetünk (óramutatónak megfelelő, vagy azzal ellentétes irányú körív). Meg tudja állapítani egy pontról, hogy az rajta van-e az íven, illetve ki tudja számolni az egyenes szakasszal vett metszéspontok közül az ív kezdőpontjához közelebbit. Ezt a funkciót később az algoritmus intenzíven használja.

### 3.1.7 A bounding_rect osztály

A bounding_rect osztályt főleg az algoritmusok gyorsítására használjuk fel. Két vízszintes és két függőleges egyenest tartalmaz, ezért bármilyen geometriai alakzatra könnyű megállapítani, hogy benne van-e egy bounding_rect által leírt téglalapban, vagy van-e vele metszéspontja.

### 3.1.8 A polygon_shape osztály

Ezt az osztályt használjuk az akadályok, illetve a robot alakjának eltárolására. Egy line_segmentek-ből álló tömböt tartalmaz, itt óramutató járásával ellentétes

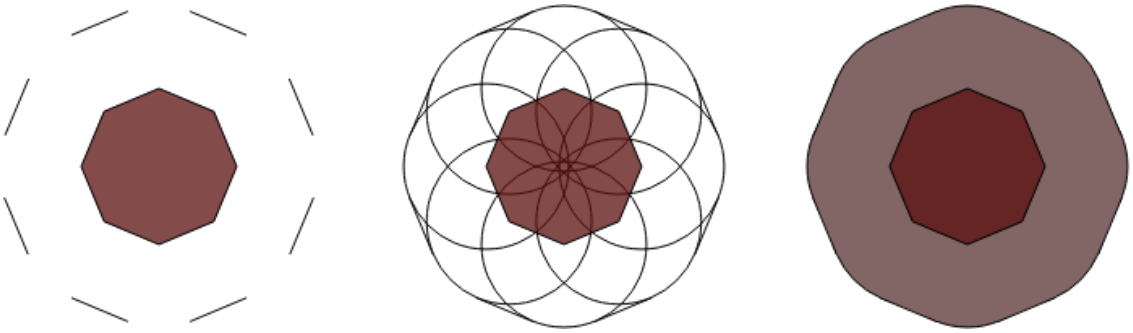
sorrendben kell megadni az alakzatot határoló szakaszokat. Amennyiben az alakzat nem szögletes, az azt befogó sokszöget kell megadni. Ezen objektumtípus képes XML-ből betölteni a tartalmát, illetve betöltéskor automatikusan kiszámolja a hozzá tartozó befoglaló téglalapot, és eltárolja egy bounding_rect osztályban. Az objektumon különböző transzformációkat lehet végezni, például a rotate_then_translate() függvény először elforgatja, majd egy vektorral eltolja a polygon_shape egy példányát. Az osztályhoz tartozik egy „color” nevű változó is, mely a megjelenítéskor használt színt tartalmazza.

### 3.1.9 Az expanded_shape osztály

Ez az osztály a polygon_shape leszármazottja, és egy olyan alakzatot tartalmaz, mely egy polygon_shape-től adott távolságra lévő pontokat határolja körül. Ez az alakzat szakaszokból és körökből áll.

Létrehozásakor meg kell neki adni egy polygon_shape példányra mutató referenciát, és a kiterjesztés sugarát. Az expanded_shape oldalai ennek az alakzatnak a kiterjesztési sugárral jobbra tolt oldalai lesznek. Mivel a polygon_shape az óramutató járásával ellentétes körüljárási irány szerint tartalmazza az oldalakat, és ezek mindig meghatározott irányultságúak a jobbra tolás mindig „kifelé” tolást fog eredményezni.

Az expanded_shape annyi kört fog tartalmazni, ahány pontja a kiindulási alakzatnak volt. Ezen körök középpontjai az eredeti sokszög pontjai, sugara pedig a kiterjesztési sugár.



**4. Ábra: A kiterjesztett alakzat kialakítása. Először az eredeti alakzat szakaszait jobbra toljuk, majd a csúcspontjaiba köröket veszük fel. Az ezek által határolt terület a kiterjesztett alakzat területe.**

### 3.1.10 Az obstacle osztály

Az obstacle osztály tartalmazza egy akadály leírását. Tartozik hozzá egy polygon_shape, mely az alakzat tényleges alakja, egy expanded_shape – melyet a polygon_shape-ből kapunk, illetve egy bounding_rect-et, mely az expanded_shape-hez tartozó határoló téglalap. Ezen kívül tartalmazza még az objektum nevét, és képes beolvasni magát XML-fájlból.

### 3.1.11 A map osztály

A map osztály tartalmazza a játéktér leírását. Egy bounding_rect-ben tartalmazza az pálya méreteit, illetve egy obstacle tömbben a pályán található akadályokat. Egy függvény meghívásával fel tudja bontani a konkáv akadályokat konvex akadályokká, így megkönnyítve a későbbi algoritmusok dolgát.

## 3.2 A robotot leíró adatstruktúrák

A robot jellemzőit tároló, illetve az algoritmust megvalósító osztályok a rob névtérben találhatóak.

### 3.2.1 A state_class osztály

A state_class osztály tartlamazza a robot egy lehetséges alakját. Ebből a később ismertetett robot osztály többet is tárolhat. Az alakzatot leíró polygon_shape-en kívül tartalmazza a robot állapotának nevét (például „karok nyitva” vagy „karok zárva”), és azt, hogy ez az alak-e az alapértelmezett. XML-fájból történő beolvasáskor automatikusan kiszámolja az alakzat befoglaló körét, melyre az ütközésdetektáló algoritmusnak van szüksége.

### 3.2.2 A wheels_class osztály

Ez az osztály a robot kerekeiről, és az azokat hajtó motorokról tartalmaz információkat, mégpedig a következőket:

- Kerekek távolsága,
- Kerekek átmérője és vastagsága (az ábrázoláshoz használjuk),
- Kerekek maximális sebessége (mm/s),
- Kerekek maximális gyorsulása ($mm/s^2$).

Az osztály képes magát beolvasni XML-fájlból.

### 3.2.3 A speed_point osztály

A sebességeket a coord_pair osztályhoz hasonló, speed_point nevű osztályban tároljuk. A két koordinátát a bal és a jobb kerék forgási sebessége adja. Az osztály létrehozásakor szükséges megadni a két kerék távolságát, így az objektum képes visszaadni a pillanatnyi haladási sebességet, és a szögsebességet.

### 3.2.4 A drive_class osztály

Ez az osztály tárolja a pillanatnyi sebességet és a szimulációs periódusidőt. Ezen kívül számos függvényt szolgáltat, melyet a többi osztály használ.

A drive_class tartalmazza a robot kerekeit leíró wheels_class példányt, illetve az aktuális sebességértéket tartalmazó speed_point objektumot. Ezeket felhasználva az alábbi számításokat tudja elvégezni:

- Fékút számítása adott sebességről (a haladási ívet tartva).
- Maximális sebesség kiszámítása, melyről adott távolságon belül a robot még meg tud állni.
- A következő ciklusra elérhető maximális és minimális sebesség számítása az aktuális sebesség, illetve a sebességkorlátok alapján.

Ez az osztály is képes tagváltozóinak alapértékét beolvasni XML-fájlból.

### 3.2.5 Curvature osztály

Ezen osztály feladata, hogy eldöntse egy adott ívről, hogy annak a mentén haladva a robot mikor ütközik először akadályba. Ehhez szüksége van egy, a pályára (map osztály) mutató pointerre, a robot pillanatnyi alakjára mutató pointerre, és az ellenőrizendő ívre. A műveletet a „max_dist” nevű függvény végzi el, mely a maximálisan megtehető úthosszt adja vissza. A számítás után lehetőség van egy külön függvény segítségével annak lekérdezésére, hogy a legutóbbi ív mentén volt-e ütközés.

### 3.2.6 A waypoint osztály

A „waypoint” osztály tárol egy elfogadható sebességet, és a hozzá tartozó haladási ívet.

### 3.2.7 A dyn_window_base osztály

Ez az osztály azoknak az osztályoknak az őse, melyek a sebességtér mintavételezését és a kapott ívek ellenőrzését végzik (a Curvature osztály segítségével).

Az elfogadható sebességeket egy „waypoint” listába gyűjti, majd elvégzi rangsorolásukat. Ezután a legmagasabb pontszámot kapó sebesség lekérdezhető. A „dyn_window_base” egy absztrakt osztály, azaz önmagában nem példányosítható. A későbbiekben két leszármaztatott osztály is ismertetésre kerül, a sebességtér mintavételezését leíró fejezetben.

### 3.2.8 A robot osztály

Ezen osztály fogja össze a korábban ismertetett osztályokat. Található benne egy mutató a „map” objektumra, a pályán elfoglalt aktuális pozíció, és sok egyéb tagváltozó.

Számos lekérdező függvényt biztosít, melyek segítségével a program megjelenítést végző része elkérheti a rajzoláshoz szükséges információkat.

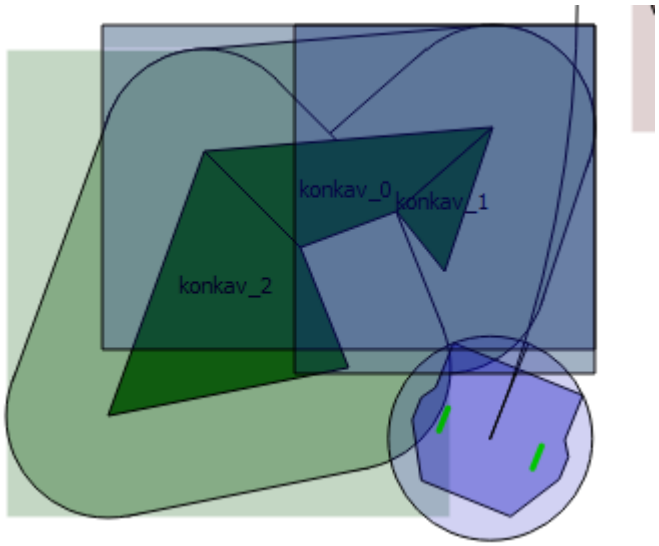
Itt található meg a „do_next_iteration” függvény, melyet a későbbiekben a valós roboton történő futtatás során ciklikusan fogunk hívogatni.

Lehetőség van egy pontsorozat megadására, melyeket sorban útba ejtve teljesít a robot.

## 3.3 Az ütközésdetektáló algoritmus működése

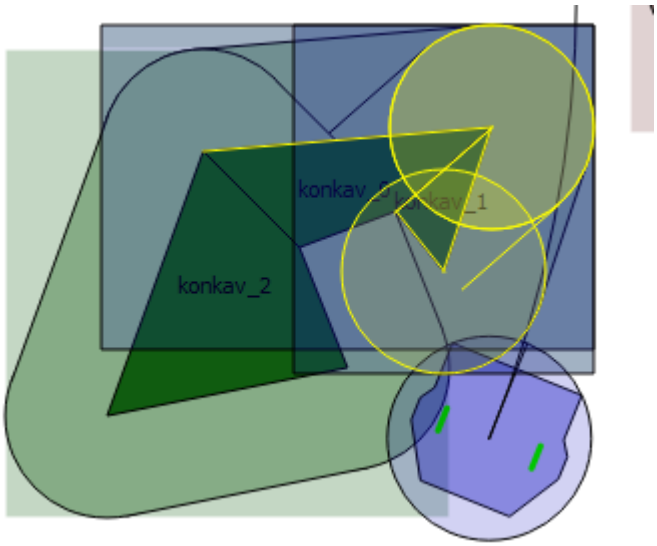
Az általam készített ütközésdetektáló algoritmus futása három lépésből áll. Ez a három lépés fut le minden egyes olyan haladási ívre, melyet a navigációs algoritmus vizsgál.

Az első lépésben a program kiválasztja azokat az akadályokat, amelyekkel szükséges ütközésvizsgálatot végezni. Ezt úgy állapítja meg, hogy a haladási ív kezdőpontját felveszi a robot forgástengelyébe, hosszának azt a távolságot állítja be, amit a robot egy ciklus alatt meg tud tenni megadott sebességgel, plusz amilyen hosszú úton le tud fékezni. Ezután megvizsgálja, hogy ennek az ívnek („arc” osztály példánya) mely akadályok (obstacle) befogó téglalapjával van közös pontja. Először megállapítja, hogy az ív kezdőpontja e téglalapon belül van-e. Amennyiben igen, akkor magától értetődő módon ez a pont mindkét alakzat közös pontja. Amennyiben nem, megvizsgálja, hogy az ívnek és a téglalap oldalainak van-e metszéspontja. Ha igen, akkor van közös pont, ellenkező esetben nincsen. Mivel ez a befogó téglalap a kiterjesztett alakzatot foglalja magában, amennyiben a robot haladási ívének nincs vele közös pontja, a robot biztosan nem ütközik az alakzattal ezt az utat követve. Ezzel a módszerrel egy alacsony költségű számítással sok magasabb költségű számítást spórolunk meg úgy, hogy a robot útjától távoli alakzatokat kiszűrjük.




**5. Ábra: A további vizsgálatra kiválasztott akadályok (kékkel bekeretezve)**

A második lépés már költségesebb, mint az előző. Itt ugyanerre az ívre azt vizsgáljuk, hogy az előző lépésben megkapott akadályok kiterjesztett alakzatjaiban mely köröket, illetve mely íveket metszi. Abban az esetben, ha egy metszéspontot találunk, a kiterjesztett alakzat elemének line_segmentre mutató mutatóját – mely már a tényleges akadály egyik oldalára mutat (lásd a circle illetve line_segment leírását) – betesszük egy listába. Az akadály valódi oldalai közül azok mutatóit is felvesszük a listára, melyekhez a haladási ív kezdőpontja a robot sugaránál közelebb van. Ez azért szükséges, mert abban az esetben, ha a robot az akadály közvetlen közelében rövid ív mentén halad, nem garantált, hogy útja metszi a kiterjesztett alakzat köreit vagy szakaszait. Az így előállított lista azokat a szakaszokat tartalmazza, melyekre a robot fizikai alakjának ütközési pontját kell kiszámolnunk. A szakasz végén ebből a listából eltávolítjuk a duplikált elemeket, hogy az algoritmus harmadik lépése ne végezzen felesleges számításokat.




**6. Ábra: A kiválasztott körök és szakaszok (sárgával kiemelve)**

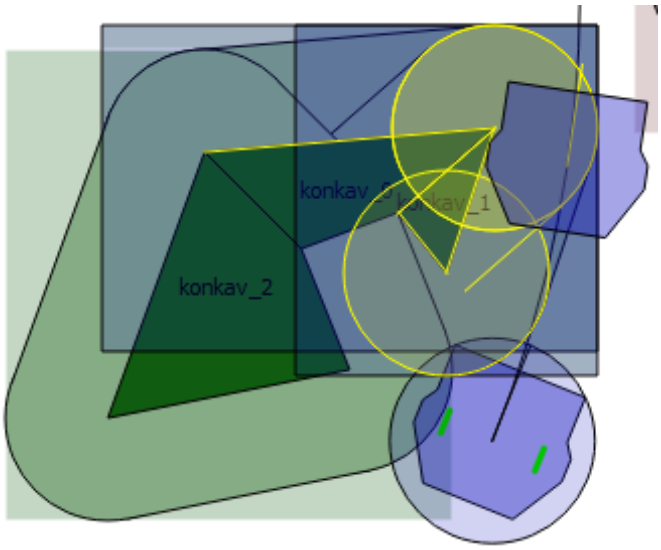
Az utolsó lépés, mely az algoritmus legnagyobb számításigényű szakasza, azt vizsgálja, hogy az előzőekben kiválasztott szakaszok és a robot aktuális alakzatjának szakaszai a haladási ív középpontja mentén elforgatva mekkora elforgatási szög esetén kerülnek először metszésbe. Ebből a szögből már kiszámolható, hogy milyen hosszú út megtétele után következik be az ütközés.

Tehát itt szakaszok ütközését, azaz érintkezését vizsgáljuk. Érintkezés alatt azt az állapotot értjük, amikor a két szakasznak van közös pontja, de valamelyik szakaszra illesztett egyenesnek csak az egyik oldalán találhatóak a másik szakasz pontjai. Egyszerűen belátható, hogy két szakasz nem érintkezhet anélkül, hogy valamelyikük egyik végpontja ne szerepeljen az érintkező pontok között.

Ezek alapján a harmadik lépés az alábbi műveletekből áll:

- Megnézzük a robot szakaszainak végpontjaira, hogy milyen szögelfordulás után érintkeznek először a kiválasztott szakaszokkal.
- Aztán megnézzük, hogy a kiválasztott szakaszok végpontjai milyen ellentétes irányú elforgatás után érintik először a robot oldalait.
- Kiválasztjuk a két érték közül a kisebbet, majd kiszámoljuk a hozzá tartozó megtett utat. Az így kiszámolt érték az ív mentén megtehető maximális út.

Jól látható, hogy ez a harmadik lépés sokkal több dolgot vizsgál, mint az előző kettő, ezért megéri az első két lépéssel lecsökkenteni a harmadik lépés által elvégzendő munka mennyiségét.




**7. Ábra: A harmadik lépés már meghatározza az ütközési pontot**

### 3.3.1 Szélső eset: Egyenes haladás

A fent említett megoldás abból indul ki, hogy a robot mozgása közben egy körív mentén halad. A körívet annak sugara, és középponti szöge határozza meg. Egyenes haladás esetén ez előbbi végtelen, utóbbi nulla lesz. Így a harmadik lépésben, amikor azt ellenőrizzük, hogy a robot pontjai mekkora szögelfordulás után metszik először az akadályok szakaszait, mindig nullát kapunk. Ez az eredmény ugyan helyes, de nem használható. Ezért ebben az esetben szükséges egy külön számítási módszer alkalmazása.

Az első két lépés nagyrészt megegyezik az eredeti megoldás első két lépésével, a különbség csak annyi, hogy a robot középpontjába nem a haladási ívet („arc” orsztály példánya) vesszük fel, hanem azt a szakaszt („line_segment” osztály példánya), ami mentén a robot közlekedik. Ennek hosszát ugyanúgy határozzuk meg, ahogy azt az eredeti megoldás teszi. Kiválasztjuk, hogy ennek a szakasznak mely akadályok befoglaló téglalapjával van közös pontja. Ezután megnézzük, hogy az ezekhez tartozó kiterjesztett alakzatok mely köreivel, illetve szakaszaival van metszéspontja, illetve a valós alakzat mely oldalaihoz van a kezdőpontja a robot sugaránál közelebb. Így elkészül a lista, mely az akadályok azon oldalait tartalmazza, melyre már a robot szakaszaira lebontott ütközésvizsgálatot kell elvégeznünk.

A harmadik lépés során először a robot sokszögének pontjaiba veszünk fel egy-egy olyan szakaszt, melynek iránya és hossza megegyezik az előzőleg vizsgált szakaszéval. Ezt követően mindre megnézzük, hogy mekkora út után metszi először a

listában szereplő szakaszokat. (Itt látható a különbség a két megoldás között, ugyanis itt utat számítunk, ott pedig szögelfordulást néztünk.) Ezután a lista szakaszainak végpontjaiból veszünk fel olyan szakaszokat, melyek iránya ellentétes az eddigi szakaszok irányával, majd ezeken vizsgáljuk, hogy mekkora út után metszik először a robot szakaszait. Végül, a legkisebb így kiszámolt út lesz az a távolság, melyet a robot ütközés nélkül meg tud tenni.

Az egyenes haladásból következik egy egyszerűsítési lehetőség. Amennyiben a robot sebessége pozitív (azaz egyenesen előre megy), nem szükséges az ütközésvizsgálatot elvégezni azon szakaszaira, melyek szemből nézve a robot többi szakasza által takarásban vannak. Ugyanez érvényes tolatásra is, ekkor azokat a szakaszokat nem kell néznünk, melyek hátulról nézve vannak takarásban. Az, hogy milyen irányú haladáskor mely szakaszokat kell vizsgálnunk, nem változik menet közben, tehát elég ezeket egyszer, a robot alakjának beolvasásakor eldönteni.

Vegyünk egy olyan esetet, amikor a robot egyenesen előre halad, és – a példa kedvéért – öt akadályt határoló szakasszal vizsgáljuk a harmadik lépésben a robot alakjának ütközését. Egy másik esetben a robot ugyanabból a pontból és orientációból indulva, egy nagyon nagy sugarú ív mentén halad. Ebben az esetben is ugyanarra az öt szakaszra számolunk ütközésvizsgálatot. Mivel szakaszok metszéspontjának kiszámítása kisebb számításigényű, mint körívek és szakaszok metszéspontjának meghatározása, az első megoldás lényegesen gyorsabban fogja meghatározni az ütközés nélkül megtehető maximális út hosszát. Ha a két módszer által számított haladási útvonal között elhanyagolható a különbség, akkor érdemes azt választani, ami kisebb számítási költségű. Ez az egyszerűsítés természetesen apró hibát visz a számításba, mely azonban nem akkumulálódik, ugyanis minden ciklus elején a robot mért pozíciójából indulunk ki.

Egy másik érv is szól amellett, hogy a nagyon nagy sugarú ívek menti haladást kezeljük egyenes haladásként. Amennyiben a két kerék sebessége már közel azonos, kijöhet olyan haladási sugár, melyet a számítógép által használt lebegőpontos ábrázolás nem tud eltárolni, helyette „inf”, azaz végtelen lesz az értéke. Ez azt jelenti, hogy itt már a számítógép sem képes különbséget tenni az ilyen ívű és az egyenes haladás között.

### 3.3.2 Szélső eset: Helyben forgás

Létezik egy másik eset is, amikor az eredeti módszer nem működik. Amikor a robot egy helyben, a középpontja körül forog, a haladási ív sugara nulla lesz. Ekkor a

robot által megtett út, azaz az ív hossza is nulla. Ebből az következik, hogy amikor létrehozzuk az „arc” osztály egy példányát, az ív központi szögének kiszámolásakor $\frac{0}{0}$ alakhoz jutunk, mely nem értelmezhető.

Ezért, ha olyan sebességre szeretnénk ütközésvizsgálatot végezni, mely helyben forgáshoz tartozik, egy új módszert kell bevezetnünk.

Az eredeti algoritmus első lépésének megfelelő művelet jelen esetben nagyon egyszerű, ugyanis csak azokat az akadályokat kell a továbbiakban vizsgálni, melyek befoglaló téglalapjában benne található a robot középpontja.

A második lépés ezen akadályok valódi oldalai közül választja ki azokat, melyek közelebb vannak a robot középpontjához, mint annak sugara.

A harmadik lépés nagyon hasonló az eredeti megoldás harmadik lépéséhez. A robot középpontja körül forgatjuk el a robot alakzatának pontjait, és nézzük, mekkora szögelfordulás után metszik először a kiválasztott akadályoldalakat. Ezután az ellenkező irányba forgatjuk ezen oldalak pontjait és nézzük, mekkora szögelfordulás után metszik először a robot oldalait. A legkisebb kapott szögelforduláshoz tartozik a legnagyobb ütközés nélküli forgási sebesség.

## 3.4 A sebességtér mintavételezése

A dinamikus ablak módszer megvalósítása során el kell döntenünk, hogy a dinamikus ablakban – azaz a sebességtér következő ciklusig elérhető tartományában – lévő sebességek közül melyeket vizsgáljuk meg. Ezt a lépést nevezzük a sebességtér mintavételezésének.

A mintavételezés módjának tervezésekor több szempontot is figyelembe kell venni.

Egyrészt kérdés, hogy hány pontot lehet biztonságosan kiszámolni a rendelkezésre álló ciklusidő alatt, ugyanis a következő ciklusba nem szabad belelógni. Sajnos az általam fejlesztett algoritmus hátránya, hogy az ütközésvizsgálat lefutása egy adott sebességre nem konstans idejű, függ a pályán található akadályok számától, alakjától, elhelyezkedésétől. Tehát szükséges egy worst-case becslés, mely meghatározza a minimálisan kiszámolható pontok számát. Természetesen amennyiben ennyi pont kiszámolása után még van bőven idő, lehetséges további pontok felvétele is.

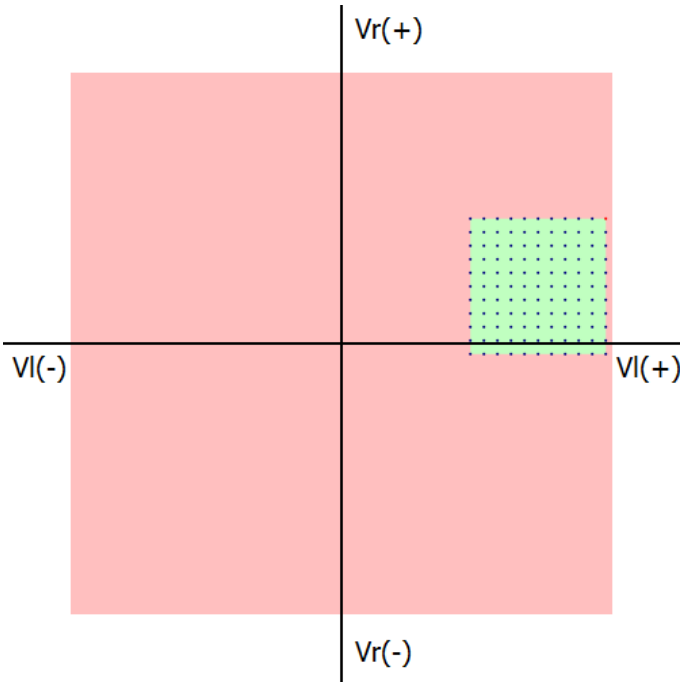
Másrészt kérdés, hogy a dinamikus ablakot milyen elv szerint mintavételezzük. A kiválasztott pontok legyenek azonos távolságra szomszédaiktól, vagy

mintavételezzük-e sűrűbben bizonyos részét. Például előnyös lehet, ha az alacsonyabb sebességekhez tartozó tartományt sűrűbben mintavételezzük, ugyanis ebben az esetben a robot kis sebességgel precízebben fog tudni haladni.

A továbbiakban az általam megvalósított megoldások kerülnek ismertetésre.

### 3.4.1 Az egyszerű mintavételezés

A legkönnyebb elképzelhető megoldás a sebességtér négyzetrácsos mintavételezése.




**8. Ábra: Négyzetesen mintavételezett sebességtér**

Az ábrán a vízszintes tengely tartozik a bal oldali kerék sebességéhez, a függőleges pedig a jobb oldali sebességéhez. A piros téglalapban találhatóak a robot által elérhető sebességek. A következő ciklusig a robot azonban csak a zöld területben található sebességeket veheti fel, ez a dinamikus ablak.

A mintavételezés során két egymásba ágyazott for ciklus segítségével végigmegyünk a dinamikus ablak területén soronként, két pont között „step” távolságot megtéve.

Minden egyes pontra lefuttatjuk az ütközésdetektáló algoritmust. Itt nem szükséges kiszámolni az ütközésig megtehető utat, tehát van lehetőség az ütközésdetektálás gyorsítására úgy, hogy már egy ütköző pont esetén visszatérünk.

A megoldás hátránya az, hogy ha sorról sorra haladva mintavételezzük a területet és a számítási idő elfogy, azaz kifutunk a ciklusidőből, akkor a mintavételezést félbehagyva egy nagy egybefüggő tartomány marad ki a vizsgálatból.

### 3.4.2 A sugárirányú mintavételezés

Vegyük a robot kerekeinek sebessége és a haladási sebesség illetve szögsebesség közötti összefüggéseket:

$$v = \frac{v_r + v_l}{2} \qquad 3\text{-}1$$

$$\omega = \frac{v_r - v_l}{W}, \qquad 3\text{-}2$$

ahol $W$ a két kerék távolsága. Amennyiben a sebességre vonatkozó egyenletet elosztjuk a szögsebességre vonatkozóval, megkapjuk a pályaív sugarára vonatkozó képletet:

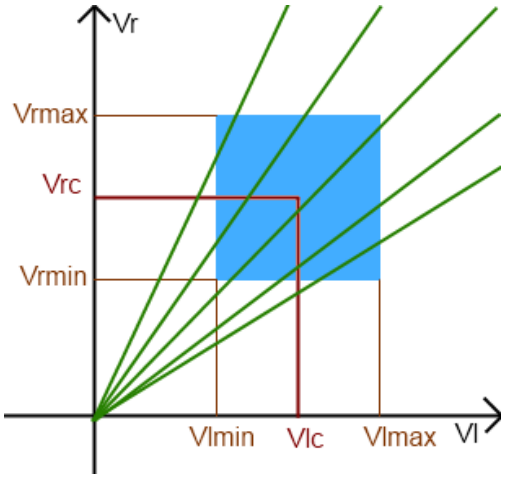
$$r = \frac{v}{\omega} = \frac{W}{2} \cdot \frac{v_r + v_l}{v_r - v_l} \qquad 3\text{-}3$$

Amennyiben a két kerék sebesség arányát fixnek választjuk, az alábbi képletet kapjuk:

$$v_r = a \cdot v_l \qquad 3\text{-}4$$

$$r = \frac{W}{2} \cdot \frac{a + 1}{a - 1} \qquad 3\text{-}5$$

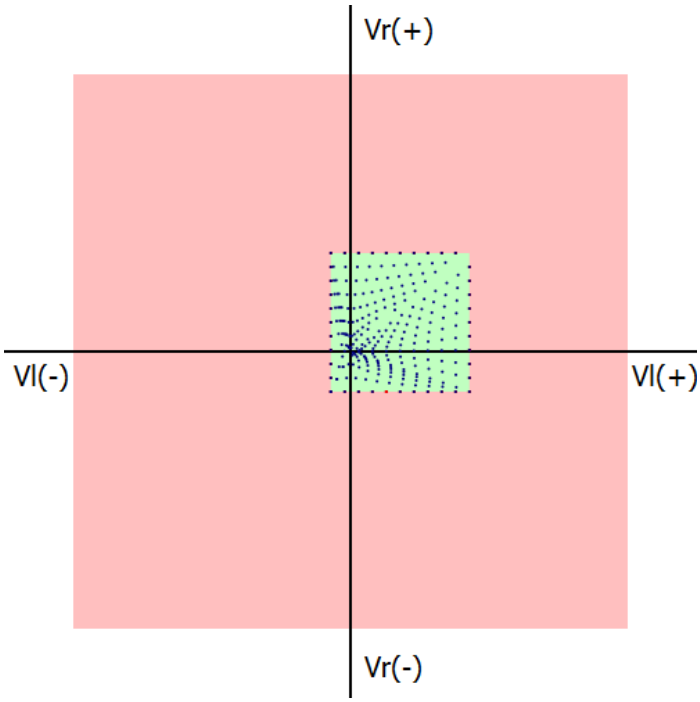
Jól látható, hogy a haladási ív sugara csak a két kerék sebességének arányától függ, közvetlenül a sebességektől nem. Tehát az általunk használt sebességtérben azok a pontok, melyeknél a két koordináta aránya megegyezik, azonos sugarú ívhez tartoznak. Koordinátarendszerünkben az azonos ívhez tartozó sebességek egy-egy origón átmenő egyenest alkotnak, melyen a gyorsabb abszolút értékű sebességek az origótól messzebb találhatóak, a lassabbak pedig közelebb.




**9. Ábra: Azonos sugarú ívek a sebességtérben (zölddel jelölve)**

A sugárirányú mintavételezés alapötlete tehát az, hogy ha azonos sugarú ívek mentén mintavételezzük a sebességteret, sugaranként elég csak egy ütközésvizsgálatot végezni. A módszer az alábbi lépésekből áll:

- Vegyünk egy origón átmenő egyesest a sebességtérben, mely áthalad a dinamikus ablakon.
- Vegyük ezen az egyenesen a legnagyobb abszolút értékű sebességet, mely még az ablakon belül van.
- Erre a sebességre végezzük el az ütközésvizsgálatot, ebből megkapjuk az ív mentén megtehető maximális utat.
- Ebből vissza tudjuk számolni, hogy mekkora az a maximális sebesség, mellyel a robot egy ciklust haladva, majd az ívet tartva nullára fékezve meg tud állni.
- Amennyiben ez a sebesség a dinamikus ablakon kívül található, dobjuk el, ezen ív mentén nem lehetséges a haladás.
- Ellenkező esetben vegyük fel a kapott sebességet a rangsorolandó sebességek listájára.
- A kiindulási egyenesből az origó felé haladva „step” lépésközzel lépkedjünk egészen addig, amíg el nem érjük az origót, vagy ki nem megyünk a dinamikus ablakból.
- Az így kapott pontokat ütközésvizsgálat nélkül felvehetjük a listára, ugyanis garantáltan van egy gyorsabb sebesség, amit ezen a haladási íven már elfogadtunk.




**10. ábra Sugárirányú mintavételezés képe az általam készített szimulátorban**

Ennek a mintavételezési algoritmusnak a legnagyobb előnye, hogy spórol a számításigényes ütközésdetektáló algoritmus használatával. Az egyszerű mintavételezés során az ütközésvizsgálatok száma megegyezik a vizsgált sebességek számával, itt ez az érték a vizsgált sebességek gyökével arányos.

Másik előnye, hogy a lassú sebességeket sűrűbben mintavételezi, ezért a robot akadályok közelében – ahol egyébként is lassabban halad – precízebben tud mozogni.

Ezeknek az előnyöknek ára is van, egyrészt az ütközésvizsgálatnál végig kell számolni, hogy mekkora utat lehet megtenni ütközésig (nem elég eldönteni, hogy az ív mentén van-e ütközés). Másrészt maga a mintavételező algoritmus is jóval bonyolultabb, mint az egyszerű megoldás esetében.

Mindazonáltal véleményem szerint összességében megéri ezt a megoldást alkalmazni.

### 3.4.3 Sztochasztikus jellegű mintavételezés

Az előzőleg ismertetett két megoldást használva úgy lehet garantálni, hogy az algoritmus az előre meghatározott időszelet alatt lefut, hogy egy worst-case becslés segítségével meghatározzuk, hány pontot lehetséges garantáltan mintavételezni ennyi idő alatt. Ezt követően úgy állítjuk be a mintavételezés paramétereit, hogy ennyi pontot számoljon ki minden ciklusban. Ennek a megoldásnak nagy hátránya, hogy az esetek

döntő többségében a worst-case esetnél jóval gyorsabban végez az algoritmus, így a rendelkezésre álló idő nem elhanyagolható részében a program tétlenül várakozik. Pedig ezalatt ki lehetne számolni további pontokat, így sűrűbb mintavételezést kapnánk mely az ideálishoz közebbi haladási útvonalat eredményezne.

Tehát érdemes egy olyan mintavételezési eljárást kifejleszteni, mely a rendelkezésre álló időt teljesen kihasználja. Fontos kikötés, hogy a mintavételezés során nem szabad nagy, összefüggő területeket kihagyni. Az eddigi megoldások ennek a kikötésnek megfeleltek, azonban jelen helyzetben a probléma annyival bonyolultabb, hogy a kiszámolt pontok száma előre nem meghatározott. Amennyiben az egyszerű megoldást – mely sorról sorra halad a mintavételezéssel – futásának felénél megszakítanánk, a dinamikus ablak alsó felében egyetlen mintavételezett pontot sem találnánk. Tehát célunk, hogy a mintavételezést bármikor megszakítva a mintavételezett pontok eloszlása körülbelül egyenletes legyen.

Amennyiben elegendő pont mintavételezésére van számítási kapacitás, lehetséges egy sztochasztikus jellegű megoldás használata. Egyenletes eloszlás szerint vegyünk fel pontokat a dinamikus ablakon belül. Belátható, hogy amennyiben csak annyi pontot mintavételezünk, amennyit az „egyszerű” esetben mintavételeznénk, kevésbé egyenletes mintavételezést kapnánk. Azonban, mivel az esetek döntő többségében ennél lényegesen több pont mintavételezésére van elegendő idő, átlagosan ez a megoldás jobban teljesít, mint az „egyszerű” megoldás.

Felmerülhet a kérdés, hogy ezt az elvet követve lehetséges-e a sugár irányú mintavételezési algoritmust úgy módosítani, hogy a fent említett megoldáshoz hasonlóan, bármikor megszakítva használható eredményt adjon. A válasz igen. Sugárirányú mintavételezéskor a dinamikus ablak „oldalai” mentén haladunk, minden esetben 2, 3, vagy 4 oldal mentén veszünk fel pontokat, és azokra végezzük el az ütközésdetektálást. Osszuk a rendelkezésre álló időt annyi részre, ahány oldal mentén vizsgálódunk, majd minden ilyen oldalon válasszunk ki és vizsgáljunk meg egyenletes eloszlású véletlen pontokat, amíg az adott időhányad le nem telik. Ez a megoldás az esetek nagy részében sűrűbben fogja mintavételezni a dinamikus ablakot, mint az eredeti, sugárirányú mintavételezés.

### 3.4.4 A biztonságos lefékezés lehetősége

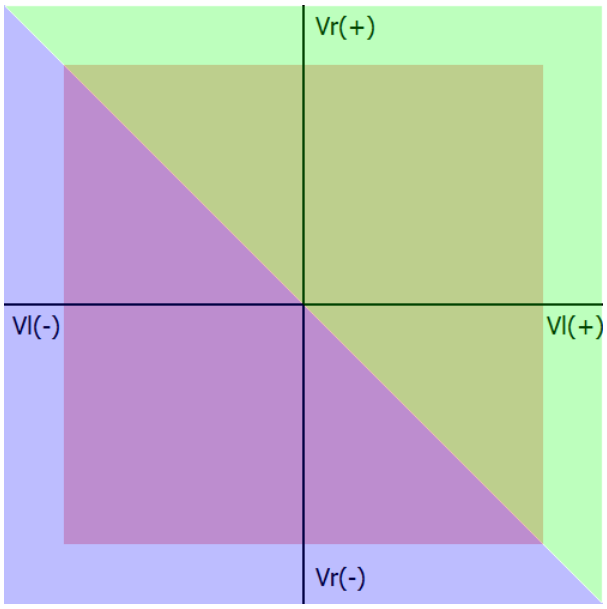
A sebességtér azon pontjait tekintjük elfogadhatónak, amelyekhez tartozó sebességeket felvéve a robot a következő ciklusig nem ütközik, majd a haladási ívet tartva ütközés nélkül meg tud állni. Biztosítanunk kell tehát, hogy a robot minden ciklusban dönthessen úgy, hogy az előző haladási ívet tartva lefékez. Ez abból áll, hogy a jelenlegi ciklusban azt a sebességet választjuk, amely az elérhető leglassabb sebesség az adott íven.

Ezen pont felvételét a fent bemutatott lehetséges mintavételezési stratégiák nem garantálják. Tehát bármelyiket választjuk, szükséges a biztonságos fékezéshez tartozó pont felvétele az elfogadható sebességek listájára. Ilyenkor nem kell ütközésvizsgálatot végezni, ugyanis e sebesség ütközésmentességét az előző ciklusbeli számítás garantálja.

### 3.4.5 A haladási irány meghatározása

A robot haladási irányát eddig nem határoztuk meg. Így a robot minden esetben úgy indult el a cél felé, ahogy az számára „egyszerűbb” volt. Azonban előfordulhat olyan eset, hogy külön ki szeretnénk kötni, milyen irányban haladva jusson el a célba a robot. Ilyen lehet például, ha a robot elején valamiféle kotró eszköz található, és út közben azzal szeretnénk valamilyen tárgyakat kotorni.

Az általunk használt sebességtér ábrázolásban az előre haladáshoz tartozó sebességek az alábbi ábrán zölddel, a tolatáshoz tartozó sebességek kékkel vannak jelölve.




**11. ábra Az előre és hátra irányú sebességek elhelyezkedése a sebességtérben**

Amennyiben azt szeretnénk, hogy a robot csak az egyik irányú mozgást választhassa, a mintavételezés elején egyszerűen csak el kell dobni azokat a sebességeket, amelyek a tiltott haladási irányhoz tartoznak.

## 3.5 A vizsgált sebességek pontozása

A sebességtér mintavételezése után előáll egy lista az elfogadható sebességekről. Ezek közül a bármelyiket felvehetjük, ugyanis a hozzájuk tartozó íven haladva a robot ütközés nélkül tud közlekedni, azaz a következő ciklusban az algoritmus dönthet úgy, hogy az ívet követve álló helyzetre fékezi a robotot.

Felmerül a kérdés, hogy ezek közül melyiket válasszuk, milyen sebesség és milyen haladási ív viszi a robotot közelebb céljához. Ebben a fejezetben két lehetséges megoldás kerül bemutatásra.

### 3.5.1 A célfüggvény

Ezt a megoldást az első dinamikus ablak módszerről szóló publikációban mutatták be [1]. A függvény minden egyes sebességhez pontszámot rendel, mely három különböző súllyal figyelembe vett tag összegeként áll elő.

$$G(v) = \alpha \cdot irány + \beta \cdot szabad\ úthossz + \gamma \cdot sebesség \qquad 3\text{-}6$$

#### 3.5.1.1 Az irányfüggvény

Az első tag célja az, hogy azokat a sebességeket díjazza, melyek a célpont felé viszik a robotot. Vegyük azt a pontot, amelybe a robot érkezne abban az esetben, ha egy ciklusidőn át az adott sebességet tartaná, majd az ívet követve nullára fékezne. Nevezzük el ezt a pontot megállási pontnak. Irányfüggvényünk bemeneti változója legyen az a szög ($\varphi$), ami a megállási pontban a haladási irány és a célpont iránya között található. Legyen irányfüggvényünk az alábbi:

$$irány(\varphi) = \frac{\cos(\varphi) + 1}{2} \qquad 3\text{-}7$$

Jól látható, hogy amennyiben a megállási pontban a robot a célpont felé néz, a függvény értéke 1. Amennyiben pont az ellentétes irányban áll a robot, a függvény értéke 0.

Felmerülhet a kérdés, hogy miért pont a megállási pontban számítjuk az irányfüggvényt. Nem lenne-e jobb abban a pontban nézni, ahová a robot a következő ciklus elején kerül? A válasz nem triviális.

Amennyiben a célpontban meg szeretnénk állni, érdemes a megállási pontban nézni az irányfüggvényt. Ugyanis itt azokat a sebességeket nem díjazzuk, melyeket felvéve a robot nem tud megállni a célpontig.

Ha azonban aktuális célpontunk csak egy hosszabb útvonal egyik köztes pontja, nem feltétlenül érdemes abban a pontban lefékezni. Ebben az esetben választhatjuk azt, hogy a robot következő ciklusbeli pozícióját vesszük alapul. Itt arra kell figyelni, hogy robotunk nem fog pontosan áthaladni a célponton. Amikor attól egy bizonyos távolságnál közelebb kerül, érdemes új célpontot adni neki.

#### 3.5.1.2 A szabad úthossz

A célfüggvény második tagja azokat a sebességeket részesíti előnyben, melyeket felvéve a robot hosszabb utat tehet meg ütközés nélkül. Amennyiben az ehhez tartozó súlyozó tényező nagy, a robot olyan utakat fog választani, melyek messze viszik az akadályoktól. Ellenkező esetben a robot az akadályok széléhez közel merészkedik.

Az általam készített ütközésdetektáló algoritmus csak a megállási pontig fut végig a haladási íven, ennél tovább nem is lenne gazdaságos futtatni, ugyanis az ennél messzebb lévő akadályok nem jelentenek veszélyt a robotra. Azonban így nem tudjuk, hogy a megállási pont után még mekkora szabad úthossz van előttünk. Amennyiben a megállási pontig tartó utat vennénk szabad úthossznak, azt mondanánk, hogy annál a pontnál akadály van, és ez nem helytálló.

Megoldásként az a javaslat született, hogy amennyiben nem ütközés miatt van vége a haladási ívnek egy adott pontban, a szabad úthossz legyen egy előre meghatározott fix érték (például 2 méter).

Egy másik felmerülő probléma az az eset, amikor a vizsgált haladási ív egy teljes kör. Ez abban az esetben fordulhatna elő, ha a robot nagy sebességgel, kis sugarú íven haladna. Noha ezeken az íveken az ütközés nélkül megtehető út végtelen, mégis ésszerű lenne az ilyen állapotokat elkerülni. Itt válasszuk tehát szabad úthossznak a kör kerületét.

Tehát a szabad úthosszt visszaadó függvény legyen az alábbi:

$$szabad\ úthossz(v) = \begin{cases} a\ kör\ kerülete, & ha\ az\ ív\ teljes\ kör \\ az\ ív\ hossza, & ha\ az\ ív\ vége\ akadály \\ nagy\ konstans, & a\ többi\ esetben \end{cases} \qquad 3\text{-}8$$

#### 3.5.1.3 A sebességfüggvény

A harmadik tag a célfüggvény legegyszerűbb része, melynek célja az, hogy a gyorsabb utakat előnyben részesítse a lassúakkal szemben. Tehát ennek a tagnak az értéke a haladási sebesség abszolút értéke. Amennyiben az ehhez tartozó szorzótényezőt kicsinek vesszük, a robot túlságosan lassan fog haladni a cél felé. Abban az esetben azonban, amikor ezt a tényezőt túlságosan magasnak vesszük, a robot céltalanul fog „rohangálni” a pályán.

### 3.5.2 Rács alapú navigációs függvény

Amennyiben minden egyes elfogadható sebességhez tartozó haladási ív megállási pontjáról tudnánk, hogy onnan milyen hosszú a célba vezető legrövidebb út, a választás könnyű lenne. Egyszerűen vehetnénk azt a sebességet, ami a célhoz a legközelebb viszi a robotot.

Azonban ezen távolságinformáció megszerzéséhez a környezet globális ismereteire van szükség. Meghatározásához a robot pályáján fel kell vennünk egy láthatósági gráfot, a gráfhoz hozzávenni a célpontot és a sebességhez tartozó megállási pontot, majd egy legrövidebb út-kereső algoritmussal kiszámítani a kívánt értéket.

A láthatósági gráfot elegendő csak egyszer, a program indulásakor kiszámolni, azonban ehhez a célpontot minden egyes új célpont megadása esetén, a megállási pontot minden haladási ív esetén hozzá kellene adni, majd eltávolítani. Egy pont hozzáadásakor azonban végig kell néznünk, hogy adott pontból mely többi pont látható, mely számításigényes művelet. Belátható, hogy ezt a módszert alkalmazva nagyon megnőne algoritmusunk futásideje.

Azonban a legjobb sebesség kiválasztásához nem szükséges a célponttól mért távolságok pontos ismerete, elegendő, ha teljesül az, hogy a közelebb lévő pontokhoz kisebb értékeket kapunk, mint a távolabbiakhoz.

Az itt bemutatott módszer Kiss Domokos és Tevesz Gábor „A Receding Horizon Control Approach to Navigation in Virtual Corridors” című cikkében jelent meg [5].

#### 3.5.2.1 A pálya felosztása

Osszuk fel a pálya területét adott oldalhosszúságú négyzetekre. Minden egyes négyzetről állapítsuk meg, hogy van-e közös pontja bármelyik kiterjesztett akadállyal.

Azokat a négyzeteket, melyek teljes terjedelmükben szabad területen találhatóak, tároljuk el.

Az általam megvalósított megoldásban egy ilyen négyzet tárolásáról a „nav_grid_element” osztály egy-egy példánya gondoskodik. Minden „nav_grid_element” tartalmazza a négyzet bal alsó sarkának koordinátáit és oldalhosszúságát, illetve 8 darab mutatót a szomszédos négyzetekre. Amennyiben egy irányban a négyzetnek nincsen szomszédja, a mutató értéke NULL. Ezen kívül tárol még egy távolság értéket, melyet a későbbiek folyamán használunk.

Ezeket a példányokat a C++ Standard Template Library által biztosított „map” nevű tárolóban raktározzuk el. Ennek előnye, hogy elemeit egy meghatározott kulcs szerint rendezve tárolja, és ezen kulcs alapján gyorsan – bináris kereséssel – vissza tudja adni a tárolt elemet. Ennek nagy hasznát vesszük, ugyanis ha az elem sor- és oszlopindexéből képezzük a kulcsot, minden egyes pontra könnyedén ki tudjuk keresni, hogy melyik négyzetben található.

Miután feltöltöttük a tárolót, fussunk végig rajta, és állítsuk be a szomszédossági mutatókat. Így egy olyan gráfot kapunk, melyben minden pont fokszáma maximum 8. A gráfban kétféle él található. Az egyik az adott négyzet közvetlen oldalszomszédjára mutat, a másik a négyzet közvetlen átlós szomszédjára. Ezen utóbbi élek súlya legyen $\sqrt{2}$, a többi élé legyen 1.

#### 3.5.2.2 A távolságok meghatározása

Amikor a robotnak egy új célpontot adunk meg, keressük ki azt a négyzetet, amelyben ez a pont található. Ennek a négyzetnek a távolságát állítsuk 0-ra, az összes többiét pedig a lehető legnagyobb értékre. Ezután Dijkstra algoritmusát futtassuk egészen addig, amíg el nem jutunk a kiindulási négyzethez. Tovább nem érdemes a távolságokat kiszámítani, ugyanis ekkor az összes ki nem számolt négyzet távolabb lesz a célponttól, mint a robot.

Létezik Dijkstra algoritmusának egy továbbfejlesztése, mely hamarabb jut el a célponthoz, mint az eredeti változat. Ezt A* módszernek hívják. Itt a kiindulási ponttól számított távolsághoz minden négyzet esetében hozzávesszük a célponttól becsült távolságot, és mindig abból a négyzetből indulunk ki, melynél ez az összeg a legkisebb. Ez a megoldás az esetek döntő hányadában lényegesen hamarabb eljut a célig. Hátránya azonban az, hogy itt már nem igaz az az állítás, hogy az összes ki nem számolt négyzet távolabb van a céltól, mint a robot. Sőt, előfordulhat, hogy csak egy egészen szűk

folyosó kerül kiszámításra. A probléma akkor jelentkezik, ha ezt a kis folyosót az ellenfél robot elállja. Ilyenkor lehetséges, hogy ha ki lenne számolva a többi négyzet, akkor a robot találna utat a célba. Ebben az esetben vagy újra kell számolnunk a távolságokat – mely viszonylag számításigényes művelet –, vagy úgy kell módosítanunk az algoritmust, hogy ne keletkezhessenek ilyen szűk folyosók.

#### 3.5.2.3 Egy adott pont távolságának meghatározása

Ezek után már viszonylag egyszerű meghatározni egy-egy haladási ívhez tartozó megállási pontról, hogy az nagyjából milyen távolságra van a célponttól. Egyszerűen kikereshetjük azt a négyzetet, mely a pontot tartalmazza, és megnézhetjük a hozzá rendelt távolságértéket.

Azonban felmerül a kérdés, hogy mi legyen abban az esetben, ha több vizsgált pontunk kerül ugyanabba a négyzetbe, vagy egy másik ugyanilyen távolságú négyzetbe? Ezen pontok között is fel kell tudnunk állítani valamilyen sorrendet.

Erre a kérdésre a megoldás a négyzeteken belüli bilineáris interpoláció. Ehhez négy olyan pontra van szükség, melyek távolságértékét ismerjük. Amennyiben ezek az ismert pontok egy egység oldalú négyzet csúcspontjaiba esnek, az interpolációs képlet az alábbi:

$$f(x,y) = \begin{bmatrix} 1-x & x \end{bmatrix} \begin{bmatrix} f(A,B) & f(A,B+1) \\ f(A+1,B) & f(A+1,B+1) \end{bmatrix} \begin{bmatrix} 1-y \\ y \end{bmatrix}, \qquad 3\text{-}9$$

ahol $(A,B)$ a négyzet bal alsó csúcsának koordinátái, $f(x,y)$ pedig a távolság értéke az $(x,y)$ pontban.

Vegyük úgy, hogy a négyzethez rendelt távolságérték annak közepére vonatkozik. Amennyiben egy pont távolságát akarjuk kiszámítani, az azt tartalmazó négyzet távolságán kívül még három négyzet távolságadatára van szükség. Válasszuk a négyzet pontunkhoz legközelebbi két oldalszomszédját, és a legközelebbi átlós szomszédját. Így pontosan olyan, egység oldalú négyzetet kapunk, melyre szükségünk van az interpolációhoz.

Előfordulhat azonban, hogy az egyik általunk kiválasztott szomszéd esetében a távolságot nem számoltuk ki. Ebben az esetben ennek távolságát vegyük egy nagyon nagy számnak, így a robot biztosan nem fog a kiszámolatlan szomszéd felé haladni.

Ezt a megoldást alkalmazva minden egyes pontra egyedi távolságértéket tudunk számolni, és ezekre az is igaz lesz, hogy a célhoz közelebbi pont kisebb távolságértéket

kap. Továbbá a négyzetek távolságát elegendő új célpont megadása esetén kiszámolni, a megállási pontokhoz tartozó távolságok kevés, egyszerű számítás segítségével meghatározhatóak.

# 4 Lokális minimum kiküszöbölése globális információk felhasználásával

A lokális navigációs algoritmusok csak a robot közvetlen környezetének ismeretét használják fel a robot irányításához. Lokális minimumnak azt az állapotot nevezzük, amikor a robot még nem érte el a célpontot, de jelenlegi helyzetében bármilyen elmozdulás esetén – a navigációs algoritmus szerint – az aktuális pozíciónál rosszabb pozícióba kerülne.

Ezen probléma egy lehetséges megoldása az, ha észrevesszük, hogy robotunk egy lokális minimumban tartózkodik. Ezt például onnan tudhatjuk, hogy robotunk az elmúlt valahány ciklus során egy pont adott, kis sugarú körén belül tartózkodott. Ilyenkor utasíthatjuk a robotot, hogy távolodjon el a lokális minimum pontjától, majd próbálkozzon újra. Megjegyezhetjük, hogy hol volt lokális minimum, és azokból a pontokból egy virtuális „taszító erőt” fejthetünk ki a robotra, hogy ne kerüljön ismét abba az állapotba. Belátható, hogy ez a megoldás szélsőséges esetben a teljes pálya megjegyzését eredményezi, így algoritmusunk elveszti lokális mivoltát.

Egy másik lehetséges megoldás, ha a céltalan bolyongás elvét alkalmazzuk. Véletlenszerű mozgás esetén is nagyobb esélyünk van a célba jutni, mintha beakadnánk egy lokális minimumba. Azonban ez a megoldás természetéből adódóan egyáltalán nem célravezető.

Belátható, hogy pusztán lokális jellegű információk alapján nem lehetséges garantálni, hogy a robot korlátos időn belül célba talál. Mivel egy navigációs algoritmus nem tekinthető működőnek abban az esetben, ha nem képes a robotot a célpontba irányítani, valamilyen megoldást kellett találni erre a problémára.

Egy lehetséges módja a lokális minimumok kiszűrésének, ha a robotnak nem pusztán csak a célpont koordinátáit adjuk meg, hanem egy olyan pontsorozatot, melyeket sorban útba ejtve a robot végül eljut a célpontba úgy, hogy közben nem kerül lokális minimumba.

A következő pár alfejezetben célfüggvény alapú navigáció használatát feltételezem. Erre azért van szükség, mert a rács alapú megoldás már tartalmaz globális jellegű információkat. Az ezzel kapcsolatban felmerülő kérdések külön alfejezetben kerülnek tárgyalásra.

## 4.1 Láthatósági gráf

Láthatósági gráfot sokszögek esetében lehetséges felvenni. A láthatósági gráf pontjai ezek csúcsai, egy pontból akkor vezet egy másikba él, amennyiben azok egymásból láthatóak, azaz az őket összekötő szakasz nem metsz akadályt. A sokszögek oldalai nem számítanak akadályt metsző élnek, tehát ezek mindig részei a gráfnak. Ha a robot kiindulási pozícióját és a célpontot, majd az ezekből látható pontokba vezető éleket is hozzáadjuk a gráfhoz, könnyedén meg lehet határozni a célba vezető utat. Ez az út nulla vagy több köztes pontból, és a célpontból áll. Ezután robotot sorban a köztes pontokra, majd a célpontba küldjük. A köztes pontok esetében nem szükséges, hogy a robot pontosan elérje azt, elegendő, ha bizonyos távolságra megközelíti. (Nevezzük ezt a távolságot megközelítési távolságnak.) Ettől a távolságtól függően fog a robot lelassítani a köztes pontoknál. Ha a távolságot kicsire vesszük, szinte teljesen le fog fékezni. Amennyiben nagyra vesszük, gyorsabb átlagsebességet érünk el, azonban egy bizonyos távolság fölött nem lehetséges garantálni a lokális minimumok elkerülését.

Összességében elmondható, hogy ha a köztes pontoknál eléggé lefékezünk, és a célfüggvény paramétereit úgy állítjuk be, hogy képes legyen a robotot egy, az aktuális pozícióból látható pontba irányítani, akkor garantáltan lokális minimum nélkül fog működni navigációs algoritmusunk.

### 4.1.1 Kiterjesztett sokszögek kiszámítása

A láthatósági gráf alapjául az akadályok fizikai alakzatai nem használhatóak, mert a köztes célpontok mind az akadályok csúcspontjai lennének, és célpont alatt azt a pontot értjük, ahová a robot középpontját szeretnénk vinni. Belátható, hogy nem célszerű olyan célpontot adni a robotnak, ahová képtelen eljutni. Ez a probléma kiküszöbölhető úgy, hogy a köztes pontok megközelítési távolságát megnöveljük legalább akkorára, mint amekkora a robot sugara. Azonban ekkor jelentősen megnő a lokális minimumba kerülés kockázata.

Az ütközésdetektáló algoritmus által használt kiterjesztett alakzat szintén nem használható láthatósági gráf készítéséhez, ugyanis ez egy szakaszok és körök által határolt alakzat, a láthatósági gráf készítéséhez pedig poligon alakú akadályok szükségesek.

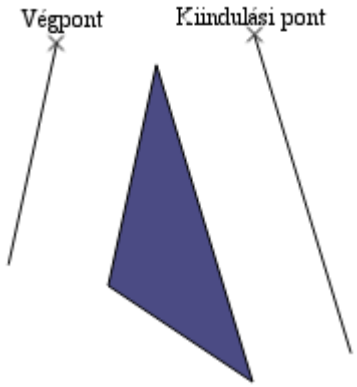
A megoldás a fizikai alakzat olyan jellegű kiterjesztése, mely csak egyenes szakaszokból áll. Ez nagyon hasonlít a kiterjesztett alakzatokhoz, a különbség annyi, hogy itt a körök helyén azokat érintő szakasz-sorozatok szerepelnek.

Az ütközésdetektáláshoz a beolvasott fizikai alakzatokat konvex alakzatokká bontottuk fel, ugyanis ez jelentősen lecsökkentette a számítás bonyolultságát. A láthatósági gráf készítésekor pont fordított a helyzet. Itt egymással érintkező akadályok jelentenének problémát. Tehát induljunk ki az eredetileg beolvasott alakzatokból.

Az általam készített kiterjesztő algoritmusnak három bemenete van. Az első a kiindulási alakzat, a második a kiterjesztési sugár (a robot sugara), a harmadik pedig körök helyére beillesztendő sokszög középponti szöge.

Az algoritmus óramutató járásával ellentétes irányban haladva az eredeti alakzat minden pontjára az alábbi lépéseket végzi el:

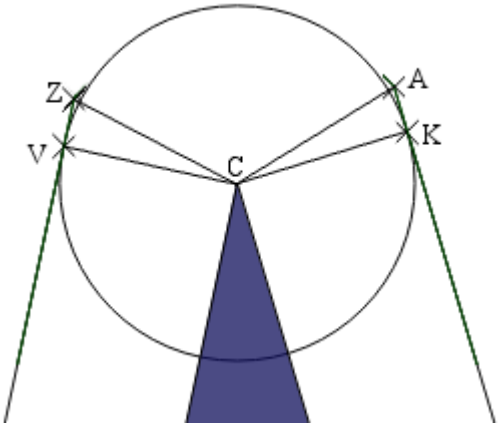
- Eltolja a pontot megelőző oldalt a kiterjesztési sugárral jobbra. Ennek második pontját nevezzük kiindulási pontnak.
- Eltolja a pontot követő oldalt a kiterjesztési sugárral jobbra (azaz „kifelé”), majd megnézi, hol található annak kezdeti pontja. Ez a végpont.




**12. Ábra: A kiindulási pont és a végpont meghatározása**

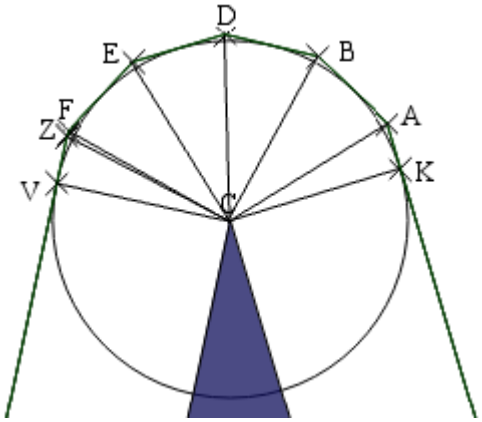
- Ezután felveszi a következő ábrán látható A és Z pontot. Az A pont a K pontban végződő szakasz meghosszabbításán található, és a KCA szög a paraméterként megadott központi szög felével egyezik meg. A Z pont a V pontból induló szakasz meghosszabbításán található, és a VCZ szög szintén a középponti szög fele. Ha a KCA háromszöget KC szakaszra tengelyesen tükrözzük, egy olyan egyenlő szárú háromszöget kapunk,

melynek belső szöge a beadott középponti szög, alapja pedig érinti a C középpontú kört.




**13. Ábra: A kiterjesztett sokszög számítása - második lépés**

- A következő lépés az, hogy az előbb említett egyenlő szárú háromszöget annyiszor felvesszük egymás után az ACZ szög által határolt tartományba úgy, hogy csúcsa a C pontban van, ahányszor elfér. Így keletkezik az ACB, BCD, DCE, ECF háromszög.

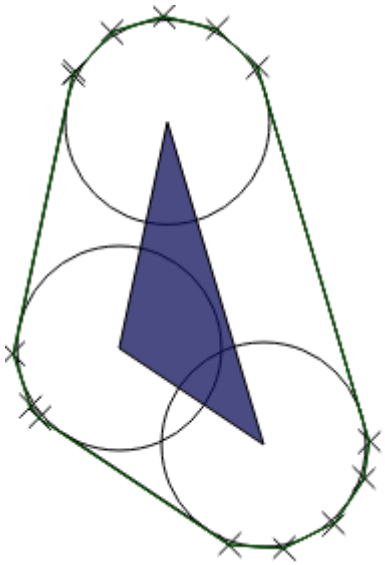



**14. Ábra: A kiterjesztett sokszög számítása - harmadik lépés**

- A fenti képen is látható, hogy amennyiben az ACZ szög nem egyezik meg a beadott központi szög valamely többszörösével, létrejön egy FCZ (vagy hasonló) háromszög, mely egyenlő szárú, de központi szöge kisebb, mint a beadott szög. FZ szakasz azonban felvehető a körülírt sokszögbe, ugyanis soha sem metszi a kört.

- Az ábrán látható pontok közül a kiterjesztett sokszög csúcsai az A, B, D, E, F és Z pontok lesznek. K és V pont elhagyató, ugyanis egy egyenes szakasz mentén találhatóak.

Miután ezeket a pontokat a kiindulási alakzat minden egyes csúcsára kiszámítottuk, az óramutató járásának ellentétes irányban haladva a szomszédos pontokat összekötve megkapjuk a kiterjesztett sokszöget. Az így keletkezett alakzat a következő ábrán látható.



**15. Ábra: Kész kiterjesztett alakzat.**

Problémát jelent, hogy az így számított kiterjesztett sokszög oldalai konkáv kiindulási alakzat esetén hurkot képezhetnek, azaz az keletkezi néhány olyan csúcspont, melyek az alakzat belsejében találhatóak. Ezeket a pontokat viszonylag egyszerű megtalálni, majd eltávolítani az alakzatból. Ezen lépés után már rendelkezésünkre áll egy olyan kiterjesztett sokszög, mely oldalait felvehetjük a láthatósági gráfba.

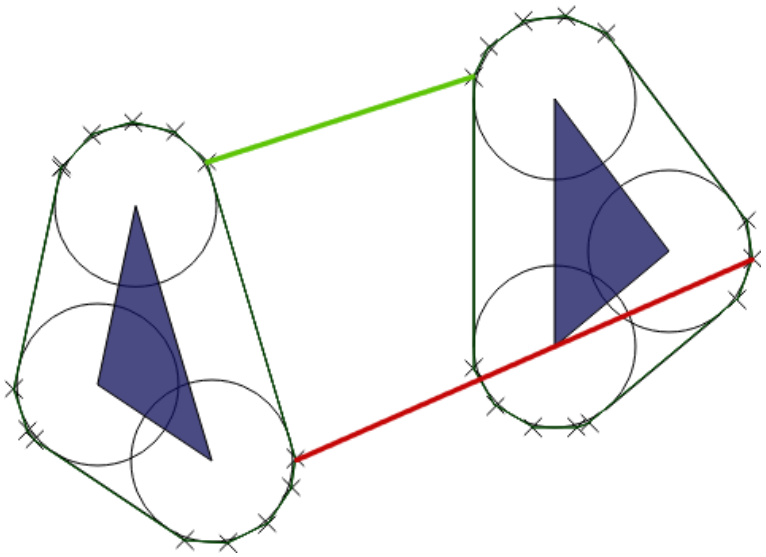
### 4.1.2 Az akadályok közti élek meghatározása

Láthatósági gráfunk csúcspontjai az előzőleg kiszámolt kiterjesztett sokszögek csúcsai. A következő feladat azon éleknek meghatározása, melyek ezen sokszögek között vezetnek.

Minden lehetséges akadálypár-kombináción végigmenve az egyik akadály összes pontjára megnézzük, hogy abból a másik akadály mely pontjai láthatóak. Mivel a láthatóság kölcsönös, a másik akadályból nem kell ellenőriznünk az első pontjainak láthatóságát.

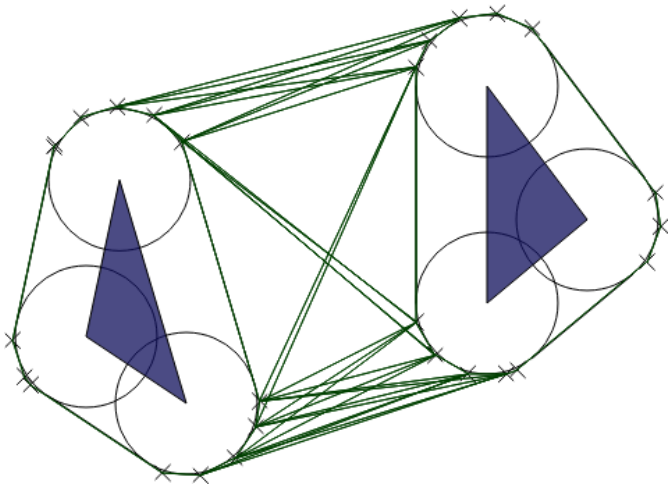
Két pont kölcsönös láthatóságát az alábbi módon döntjük el:

- Összekötjük a két pontot egy szakasszal („line_segment” példánya).
- A szakasznak a triviális metszéspontokon kívül nem lehet más metszéspontja a két általa összekötött alakzattal. Triviális metszéspont akadályonként kettő van, ezek azokon a szakaszokon találhatóak, melyek a vizsgált szakasz valamely végpontjában kezdődnek vagy végződnek. Amennyiben találunk nem triviális metszéspontot, a vizsgált szakaszt eldobjuk.



**16. Ábra: A piros szakaszt eldobjuk nem triviális metszéspont miatt.**

- Ezután meg kell nézni, hogy a szakasz nem metszi-e bármely másik akadály szakaszait. Amennyiben nem találunk több metszéspontot, a szakaszt felvehetjük a láthatósági gráfba.



**17. Ábra: A teljes láthatósági gráf**

Miután megvizsgáltuk az összes lehetséges akadályok közti élt, megkapjuk a láthatósági gráfot. A példaként vizsgált két háromszög esetében a láthatósági gráf a fenti képen látható.

### 4.1.3 Becslés az élek számára

Az itt bemutatott példában két háromszög alakú objektumra készítettük el a láthatósági gráfot, tehát összesen 6 akadály-oldalból indultunk ki.

Az akadályokhoz elkészítettük a kiterjesztett sokszögeket, melyeket a láthatósági gráf felépítéséhez használunk. Kiterjesztés közben újabb csúcspontokat hozunk létre, amikor a körívek mentén érintő szakaszokat veszünk fel. Az, hogy mennyi ilyen szakaszt veszünk fel, attól függ, hogy mekkora kívánt középponti szöget adunk meg a kiterjesztő algoritmusnak. Jelen esetben ez a szög 30° volt, a kiterjesztett alakzatok oldalainak száma 27 lett.

Az akadály menti élek számára az alábbi megközelítő becslést adhatjuk:

$$e_{akadály\ menti} \approx \sum_{i=1}^{N} n_i + \left( N \cdot \frac{360°}{\alpha} - \sum_{i=1}^{N} n_i \right), \qquad 4\text{-}1$$

ahol N az akadályok száma, $n_i$ az $i$-ik akadály oldalainak száma, $\alpha$ középponti szög. A képlet szándékosan nem került egyszerűsítésre, ugyanis az egymást kioltó két összegzés származása nem triviális. Az első tag az eredeti alakzatok oldalainak összege. A második tag első tagja abból adódik, hogy várhatóan ennyi kis érintő szakaszt kell a körök mentén beszúrnunk. Ez akadályonként 360°-nyi körív körberajzolását jelenti. A tényleges beszúrt szakaszok száma ennél várhatóan több, ugyanis az áthidalandó körívek középponti szöge általában nem osztható a beadott középponti szöggel. A harmadik tag azt tartalmazza, hogy a kezdő és végpont eltávolítása (lásd 4.1.1-es fejezet) miatt csúcsonként várhatóan egy $\alpha$-val kevesebb szöget kell áthidalni.

A képletet egyszerűsítve egy nagyon könnyű összefüggést kapunk az akadály menti élek számára.

$$e \approx N \cdot \frac{360°}{\alpha} \qquad 4\text{-}2$$

Természetesen ez csak egy közelítő értéket az az élek valós számára. Jelen esetben a képlet szerint 24 akadály menti él várható, a valóságban 27 keletkezett.

Az akadályok közötti élek számára nem lehet ilyen könnyen becslést adni, ugyanis az nagymértékben függ az akadályok egymáshoz viszonyított

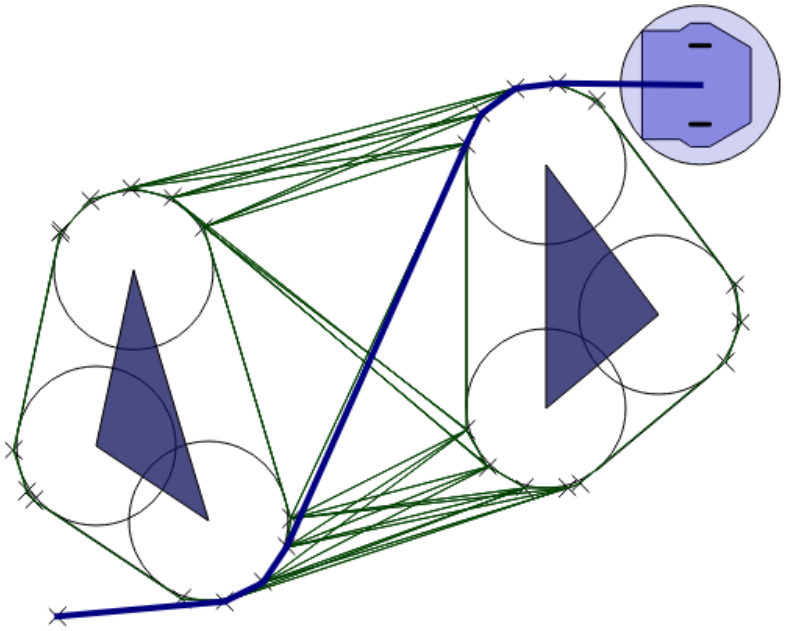
elhelyezkedésétől. Az itt bemutatott egyszerű példa esetében 29 akadály közti élt lehetett behúzni.

Összesen tehát 56 élből áll a láthatósági gráf. Ez ahhoz képest, hogy két egyszerű háromszög alakú akadályból indultunk ki, meglehetősen sok.

## 4.2 Érintő gráf

A láthatósági gráf készítésekor azt az elvet követtük, hogy minden olyan pontpár közé felvettünk egy-egy élt, melyek kölcsönösen láthatóak voltak. Ezáltal garantáltuk, hogy ha két pont között létezik olyan út, amely csak egymásból látható pontokon át vezet, akkor azt meg tudjuk találni. Sőt, az összes ilyen utat meg tudjuk találni két pont között. Természetesen több lehetséges útvonal közül azt választjuk, amelyik rövidebb. Felmerülhet tehát az a kérdés, hogy vajon az összes élre szükségünk van? Van-e olyan él, amely soha sem fog szerepelni legrövidebb útban?

Egyszerűen belátható, hogy az akadály menti élek mindegyikére szükségünk van. Egy egyetlen háromszöget tartalmazó pályán könnyű három olyan példát mutatni, melyek mindegyike tartalmazza a háromszög valamely oldalához tartozó élt.



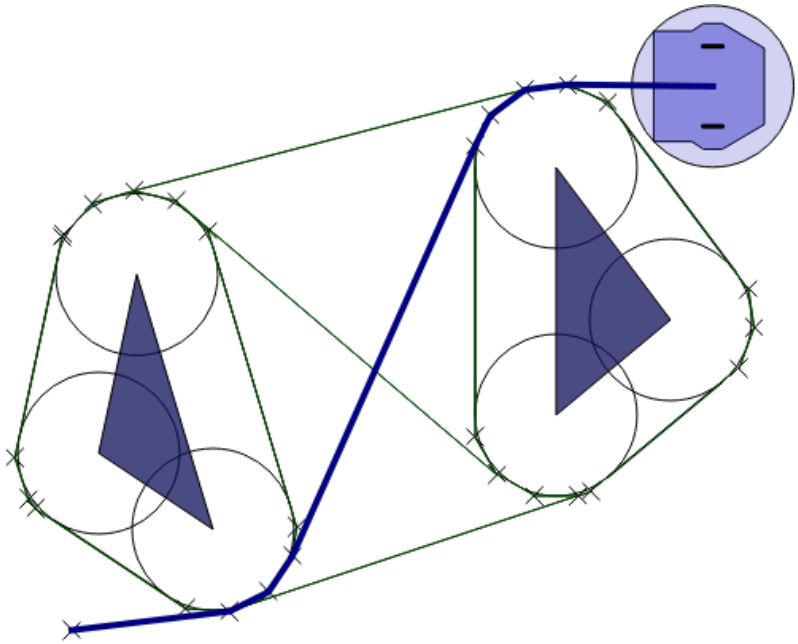
**18. Ábra: Útvonal a láthatósági gráfban**

Az akadályközti élekről azonban ez nem mondható el. A fenti ábrán jól látható, hogy a két akadály közötti átvezető él helyére bármelyik másik kettőt felvéve – a háromszög-szabály miatt – hosszabb utat kapnánk. Amennyiben a célpont a két akadály között lenne, akkor nem használnánk az akadályok között vezető éleket, ugyanis új

célpont megadásakor azt is felvesszük a gráfba, és behúzzuk az éleket az abból látható pontokba. Így a jobb oldali akadály és a célpont között egy ilyen, újonnan behúzott élen közlekedne a robot.

Tehát kérdés, hogy melyek azok az akadályközti élek, amelyeket meg kell tartanunk. Ezek azok az élek, melyek meghosszabbított egyenese csak érinti, és nem metszi az akadályt. Azt a gráfot, amely az az alakzatokat határoló éleken kívül csak olyan éleket tartalmaz, melyek kölcsönösen látható pontok között vezetnek, és meghosszabbított egyenesük a két alakzatot csak érinti, érintő gráfnak nevezzük.

Szerencsére az általam készített szoftveres környezetben könnyű a láthatósági gráfot felépítő algoritmust úgy módosítani, hogy érintő gráfot készítsen. Amikor a triviális metszéspontokat keressük, akkor a vizsgált „line_segment” példányt elegendő típuskonverzió segítségével „line” példánnyá alakítani, ugyanis ez utóbbi az előbbi ősosztálya. Így egyenesre vizsgáljuk a metszéspontot, és tulajdonképpen ez az, amire szükségünk van.



**19. Ábra: Útvonal az érintő gráfban**

Ennek a megoldásnak két nagy előnye van. Egyrészt szemmel láthatóan, lényegesen csökkent az akadályok közötti élek száma. Ez meggyorsítja az útkereső algoritmus futását, és csökkenti a program memóriahasználatát.

A másik nagy előny az, hogy most már kézzel fogható maximális becslést tudunk adni az akadályok közti élek számára. Két akadály között maximum négy él

vezethet. Tehát az akadályok között vezető élek maximális száma az alábbi képlet szerint alakul:

$$e_{akadály\ közti} \leq 4 \cdot \frac{N(N-1)}{2} = 2 \cdot N \cdot (N-1) \qquad 4\text{-}3$$

Az összefüggés alapján az élek száma négyzetesen függ az akadályok számától. Azonban ez csak egy worst case-becslés. Nagy akadályszám esetén magas valószínűséggel csak az egymáshoz közeli akadályok között vezet él, mert a szomszédos akadályok eltakarják a távolikat.

Érdekes észrevétel, hogy ebben az esetben az élek száma egyáltalán nem függ az akadályok oldalszámától, azaz alakjuk bonyolultságától.

## 4.3 Az útkeresés számításigénye

Az általam elkészített megoldás alapvetően az érintő gráfot használja a kiindulási- és a célpont közötti legrövidebb út megkeresésére. Ezen gráf pontjait és éleit a pályán található összes akadály, és azok elhelyezkedése határozza meg. Magát a gráfot csak egyszer, a program betöltésekor szükséges kiszámolni, tehát a gráfot felépítő algoritmus futása nem sebesség-kritikus.

Azonban az útkereső algoritmust minden esetben lefuttatjuk, amikor a robot új célpontot kap. Ekkor még az sem garantált, hogy a robot álló helyzetben van. Tehát nagyon fontos, hogy ismerjük – illetve számításba vegyük – az útkeresés számításigényét.

Az útkereséshez használt $A^*$ algoritmus komplexitása legrosszabb esetben megegyezik a Dijkstra-algoritmus számításigényével, ami:

$$O(|V| * \log(|V|) + |E|), \qquad 4\text{-}4$$

ahol $|V|$ a pontok száma, $|E|$ az élek száma. A mi esetünkben a pontok száma megegyezik az akadály menti élek számával (egy sokszögnek annyi pontja van, ahány oldala). A fenti becsléseket behelyettesítve a képletbe az alábbit kapjuk:

$$O\left(2 \cdot N \cdot (N-1) + N \cdot \frac{360°}{\alpha} + N \cdot \frac{360°}{\alpha} \cdot log\left(N \cdot \frac{360°}{\alpha}\right)\right) \qquad 4\text{-}5$$

Látható, hogy az algoritmus számításigénye az akadályok számától négyzetesen függ. Tehát amennyiben nagyon sok akadályt veszünk fel a pályán, nagyon megnőhet az útvonalkeresés futási ideje. Azonban ez nem meglepő, ugyanis ez a viselkedés jellemző a globális jellegű információkat feldolgozó algoritmusokra.

## 4.4 Speciális esetek

Felmerülhet a kérdés, hogy előfordulhat-e olyan eset, amikor a fent bemutatott módszer nem talál utat két pont között, holott a robot valójában el tudna jutni a kiindulási pontból a végpontba. Ilyen állapot két féle módon alakulhat ki.

### 4.4.1 Kiindulási- és célpont helye

Amennyiben a robot jelenleg a kiterjesztett sokszögen belül tartózkodik, vagy a megadott célpont egy kiterjesztett sokszög belsejében található, az útkereső algoritmus nem fog utat találni. Ennek oka az, hogy ebből a két pontból az érintő gráf egyik pontjába sem tudunk felvenni élt, ugyanis minden lehetséges él az akadályon keresztül vezet.

Erre a problémára három lehetséges megoldást találtam:

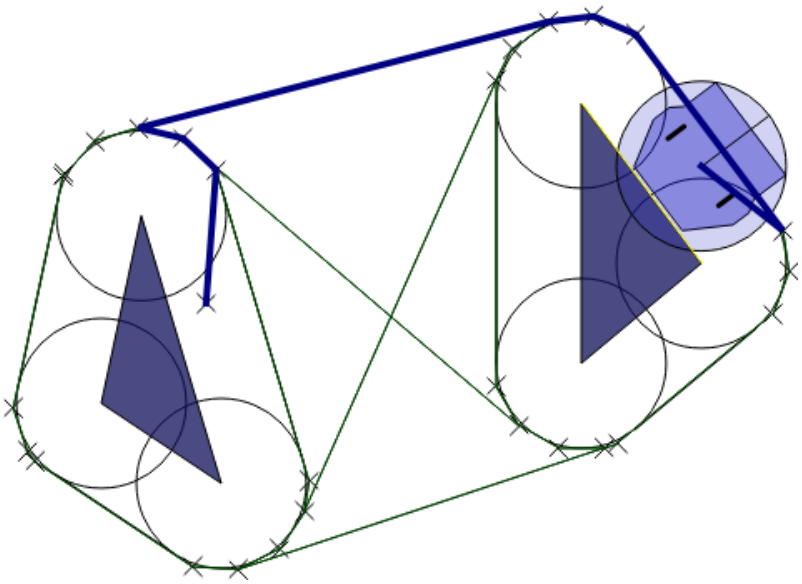
a) Nézzük meg, hogy a problémás pontból a hozzá tartozó akadály mely pontjai láthatóak a valóságban, azaz mely élek nem metszik a valódi akadály alakzatját, majd ezeket vegyük fel a gráfba és hajtsuk végre az útkeresést.

b) Vegyük fel a problémás pont és a hozzá legközelebb található navigációs pont közötti élt a gráfba, majd hajtsuk végre a keresést. Könnyen belátható, hogy ez az él soha sem metszi a valódi akadályt.

c) Keressük meg a problémás ponthoz legközelebbi olyan pontot, mely a kiterjesztett sokszögen kívül van, majd vegyük fel és kössük be a gráfba, majd kössük össze a problémás ponttal.

Az „a” megoldás hátránya, hogy olyan éleket engedélyez, melyek nagyon közel vihetik a robotot a valódi akadályhoz, így kockáztatva, hogy az esetleg lokális minimumba kerül. Jobb lenne egy olyan megoldást alkalmazni, mely hamar eltávolítja a robotot az akadálytól.

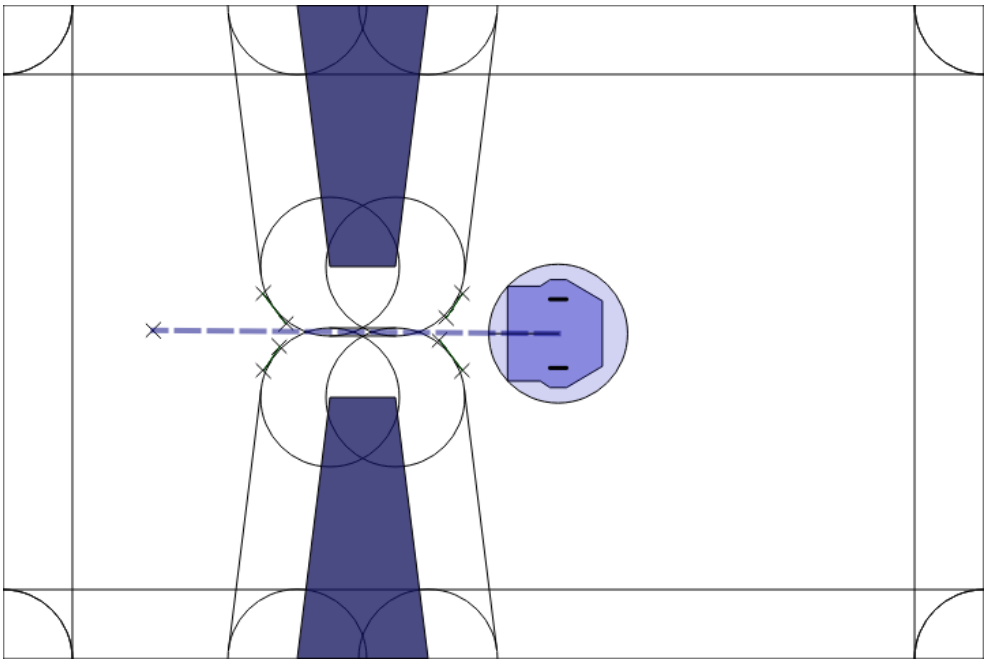
A „c” megoldás pontosan ezt teszi, azonban ez viszonylag számításigényes. Úgy döntöttem, hogy a „b” lehetőséget választom, mert ez nem ad sokkal rosszabb eredményt a „c” opciónál, ellenben nagyon egyszerűen megvalósítható.



**20. Ábra: Kiinduálsi- és célpont a kiterjesztett sokszög belsejében**

### 4.4.2 Nem összefüggő érintő gráf

A következő ábrán egy olyan eset látható, amikor az érintő gráf nem összefüggő. Ezt az okozza, hogy a két trapéz alakú akadály olyan közel van egymáshoz, hogy azok kiterjesztett sokszögei egymásba lógnak. Azonban, mivel a kiterjesztett sokszög készítésekor a kiterjesztési sugár a robot befoglaló körének sugarával egyezik meg, a valóságban a robot elférne a két akadály között.



**21. Ábra: Nem összefüggő érintő gráf esete**

A képen úgy látszik, hogy a problémára megfelelő megoldást jelentene az, hogy ha ilyen esetben nem vennénk fel köztes pontokat, hanem közvetlenül a célpontot adnánk meg a robotnak. Azonban ez nem jó megoldás, mert így megszűnne a lokális minimumba kerülés elleni védelem.

Egy másik megoldást jelentene az, ha összefüggővé tennénk a gráfot úgy, hogy valamilyen módszerrel behúznánk olyan éleket, melyek nem metszik a valós akadályok alakzatját. Azonban nagyon helyzetfüggő, hogy mikor mely éleket kell felvenni a gráfba, ezért nagyon nehéz lenne egy ilyen módszer kifejlesztése, illetve helyességének igazolása.

Valójában azonban nem szükséges a gráfot automatikusan összefüggővé tenni. Jelen esetben ugyanis a robot munkaterülete előre ismert, az akadályok pedig nem mozognak. Tehát, ha lehetővé tesszük, hogy a pálya betáplálásakor, emberi beavatkozással fel lehessen venni éleket a gráfba, akkor könnyedén áthidalhatjuk a problémát.

## 4.5 Virtuális folyosó

Korábban, az elfogadható sebességek ragsorolását ismertető fejezetben tárgyalásra került egy megoldás, mely globális információk felhasználásával döntötte el, hogy mely lehetséges sebesség viszi a következő ciklusban a legközelebb a robotot a célpontba.

A célponttól való távolságot úgy számítottuk, hogy a pályát kis négyzetekre osztottuk, majd a célpontot tartalmazó négyzetből elkezdtünk egy távolságfüggvényt terjeszteni. Ez a függvény természetéből adódóan lokális minimum-mentes volt, ezért a robot haladás közben nem kerülhetett ilyen állapotba.

A navigációs függvény kiszámítására két lehetséges algoritmus vetődött fel. A Dijkstra-algoritmus minden olyan négyzet távolságát kiszámolja, ami a robotnál közelebb van a célponthoz. Tehát a kiszámolt négyzetek számát felülbecsülhetjük az alábbi képlettel:

$$n = \left(\frac{s}{d}\right)^2 \cdot \pi, \qquad 4\text{-}6$$

ahol $n$ a kiszámolt négyzetek száma, $d$ a négyzet oldalhosszúsága, $s$ pedig a robot és a célpont közötti, érintő gráfbeli távolság.

Látható, hogy az összefüggés négyzetes, azaz nagy pálya és nagy célpont-távolság esetén a navigációs függvény számítása sokáig eltarthat.

A másik korábban ismertetett megoldás az *A** módszer, mely a Dijkstra algoritmus heurisztikus módszerrel gyorsított változata. Legrosszabb esetben a kiszámolt négyzetek száma megegyezik a Dijkstra algoritmuséval. Azonban az is előfordulhat, hogy csak egy nagyon szűk útvonal kerül kiszámításra, és így a robot mozgástere leszűkülne.

Lehetséges azonban egy olyan megoldás is, melyet használva a kiszámított négyzetek száma a távolsággal lineárisan arányos. Ugyanis rendelkezésre áll az érintő gráf, melyben viszonylag egyszerűen meg lehet találni az útvonalat a robottól a célba. Miután megvan az útvonal, elegendő csak az attól egy bizonyos távolságnál közelebb lévő négyzetek távolságának meghatározása. Az így kialakuló kiszámított sávot nevezzük virtuális folyosónak [5]. A kiszámolt távolságú négyzetek száma az alábbi szerint alakul:

$$n = \frac{s}{d} \cdot w, \qquad 4\text{-}7$$

ahol *w* az a szám, amennyi négyzet széles a folyosó. Látható, hogy ez az összefüggés már lineáris.

Azonban ennek a módszernek ára is van, egyrészt végre kell hajtani az érintő gráfban az útkeresést, másrészt pedig minden egyes négyzet vizsgálatakor el kell döntenünk az adott négyzetről, hogy az nincsen-e túl távol a folyosótól. Ehhez ki kell számítani az útvonal összes szakaszára, hogy azok milyen messze vannak a négyzettől.

Kicsi pálya esetén előfordulhat, hogy pont ezen távolságszámítások miatt ez a megoldás lassabb, mint a legegyszerűbb Dijkstra-módszert használó. Nagy pálya esetén azonban szinte bizonyos, hogy megéri a virtuális folyosó-módszert használni.

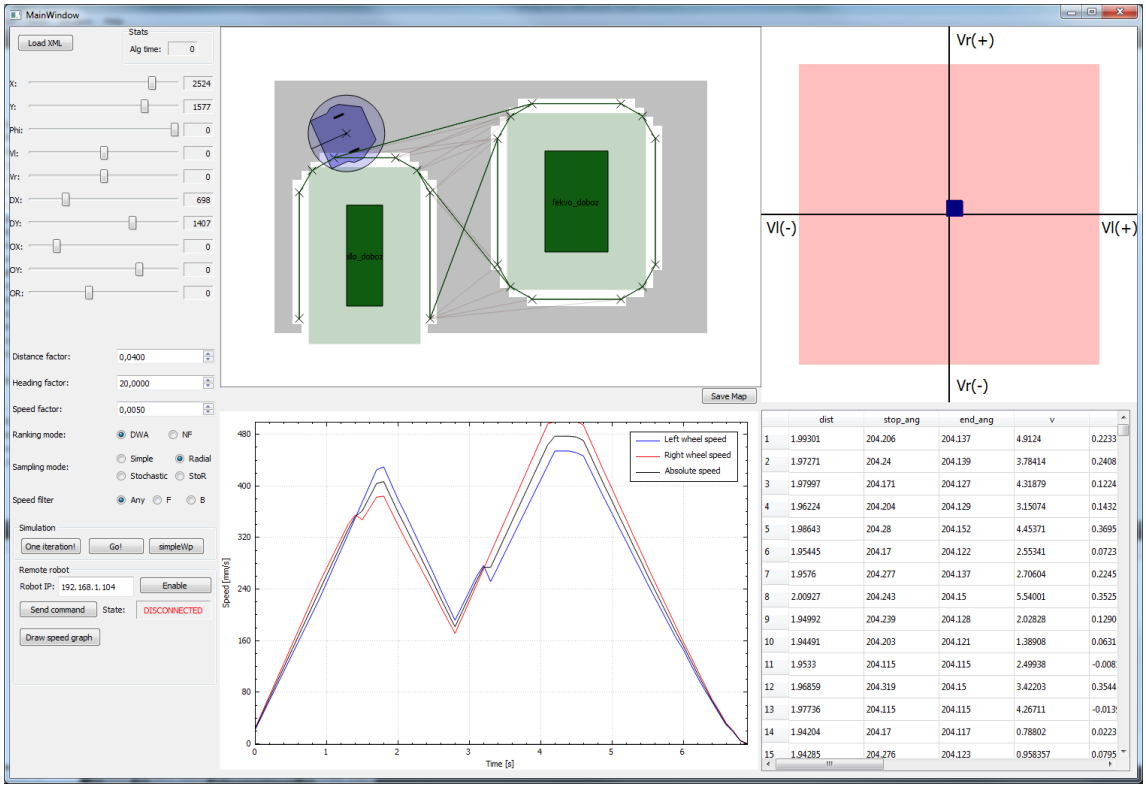
# 5 Szimuláció és valós működés

A harmadik és negyedik fejezet arról szólt, hogy milyen módon valósítottam meg a robotot navigáló algoritmust. Sorra vettük a megoldandó problémákat, és azok lehetséges – és általam implementált – megoldásait. Azonban a fejlesztés során szükségem volt egy hatékony eszközre, mely segítségével ellenőrizni lehetett egy-egy programrész helyes működését.

Természetesen ez az eszköz nem lehetett maga a robot, hiszen hibás működés esetén kárt tehetett volna benne a program, illetve a hiba okának felderítése is nagyon bonyolult lett volna.

## 5.1 A szimulátor program

Ezért már a tervezési szakasz elején úgy döntöttem, hogy az algoritmussal párhuzamosan fejlesztek egy szimulátort, melynek segítségével lehetséges az általam készített vezérlőprogram monitorozása, illetve bemeneteinek megadása.

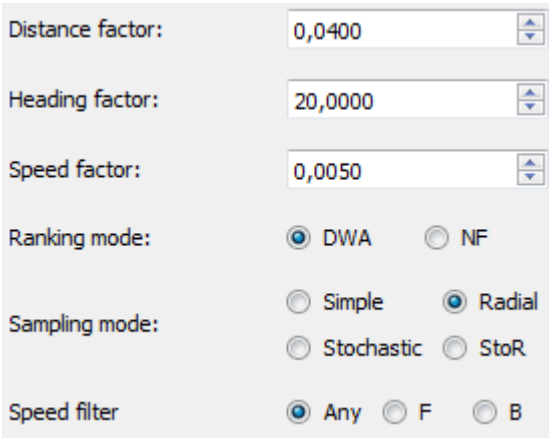



**22. Ábra: A szimulátor program**

A szimulátor legfontosabb eleme a robot aktuális helyzetét és környezetét megjelenítő komponens, mely középen, fent található. Itt ábrázoljuk az akadályokat, azok kiterjesztett változatait és az érintő gráfot. Miközben mozog a robot, megjeleníti az ütközésdetektáló algoritmus által vizsgált haladási íveket, az ütközésvizsgálatra kijelölt alakzatokat és akadály oldalakat. A fejlesztési folyamat során ennek nagy hasznát vettem, ugyanis többször előfordult olyan hiba, melynek hatására a robot belement az akadályokba. Ekkor újra futtatva és lépésről lépésre ellenőrizve az algoritmus futását, könnyű volt megtalálni a hiba okát.

A szimulátor jobb felső sarkában található a sebességtér ábrázolása. Ebben a program az algoritmus minden iterációja után megjelenítni az elérhető sebességtartományt, és az abban mintavételezett sebességeket. Ez a komponens a sebességteret mintavételező megoldások tesztelésekor tett jó szolgálatot, ugyanis könnyen észre lehetett venni, ha a valamilyen hiba folytán a program például az elérhető tartományon kívüli pontot mintavételezett.

Bal fölül található több csúszka, melyek segítségével a robot helyét és orientációját, illetve a célpont helyét lehet állítani. Ezen kívül megadható a kerekek sebessége is.




**23. Ábra: Az algoritmus paramétereinek állítására szolgáló komponens**

A csúszkák alatt találhatóak azok a vezérlők, melyek segítségével az algoritmus paramétereit lehet állítani. Itt lehetséges a dinamikus ablak módszer súlyozó paramétereinek megadása, itt lehet kiválasztani, hogy a vizsgált sebességek rangsorolására a célfüggvényt, vagy a rács alapú navigációs függvényt használja. Ezen kívül itt lehet megadni, hogy a sebességteret melyik korábban ismertetett megoldást

használva mintavételezze. Lehetőség van arra is, hogy csak hátramenetet vagy csak előre haladást engedélyezzünk a robotnak.

A robot helyzetét ábrázoló widget alatt található egy grafikon, mely az algoritmus futása során kiadott sebességparancsokat ábrázolja. Kék színnel a bal kerék sebessége, piros színnel a jobb keréké, zölddel pedig a robot haladási sebessége van árbrázolva. A program a futási adatokat nem csak ezen a felületen keresztül tudja megjeleníteni, a legfontosabb információkat folyamatosan feljegyzi egy vesszőkkel elválasztott adatokat tartalmazó fájlba, melyet a MATLAB segítségével be lehet olvasni, majd feldolgozni.

A jobb alsó sarokban található táblázat a vizsgált sebességekhez tartozó „waypoint” osztály példányait tartalmazza. Ennek segítségével lehetséges a dinamikus ablak módszer súlyozó paramétereinek finomhangolása.

## 5.2 A roboton futó program

A szimulátor programot használva idővel elkészült az algoritmus egy olyan változata, mely megbízhatóan működött ezen szimulált környezeten belül. Ekkor már érdemes volt elkészíteni a program robotra szánt változatát.

A szimulátort és magát az algoritmust is C++ nyelven, Windows-t használó számítógépen fejlesztettem, azonban az algoritmusban csak standard C++ függvényeket és tárolókat használtam. Mivel követtem a document-view architektúra előírásait, az algoritmus könnyen elválasztható volt a megjelenítést végző komponensektől. Így tehát viszonylag egyszerűen tudtam készíteni egy konzolos programot az algoritmusból, melyet azután a Linux-ot futtató robotra le lehetett fordítani.

Szükséges volt az általam készített program összekötése a robot szoftverével. Ezt „pipe”-ok segítségével valósítottam meg. A robot egy ilyen csővezetéken keresztül továbbítja az algoritmus felé a saját, mért pozícióját, és az algoritmus egy másik „pipe”-on keresztül továbbítja a beállítandó keréksebesség értékeket.

Ezután már csak azt kellet megoldani, hogy a roboton futó program tudjon kommunikálni a szimulátor programmal. Erre a célra UDP alapú kommunikációt használok, melynek részletes ismertetésére nem térek ki.

A robotot a szimulátoron keresztül lehet vezérelni, a program bal alsó részén található vezérlőgombok segítségével.

## 5.3 Tapasztalatok

Munkám jelen fázisában a robot tesztelése folyik valós körülmények között. Konkrét mérési eredmények még nem állnak rendelkezésre, azonban a robot már képes megbízhatóan, *0,4 m/s* sebességgel közlekedni a pályán, akadályok között.

A tesztek során számos olyan probléma merült fel, mely a szimulátoros tesztelés során nem jelentkezett. Az algoritmus nem volt felkészítve olyan jellegű problémák megoldására, melyek a robot pozíció mérésének pontatlanságából adódnak. Például, amikor robotunk egy akadály közelében halad, kaphatunk olyan pozíció információt, mely szerint a robot belelóg az akadályba. Ebben az esetben a robot hibásan működött, a hibajelenség pedig az volt, hogy nem volt képes elhagyni az akadály területét.

Ennek a problémának a megoldása jelentős időbe telt, részletes leírását dolgozatom nem tartalmazza.

Folyamatban van az ellenfél robotok kezelésének megvalósítása. Ez a szimulátorban már működik, a valóságban azonban még javításra szorul.

Összességében elmondható, hogy két féléves munkám során sikerült kifejleszteni egy olyan navigációs algoritmust, mely egy valós roboton, valós körülmények között működik, és megelégszik a roboton található számítógép korlátos számítási kapacitásával.

A továbbiakban folytatom a teszteket és méréseket végzek, melyek eredményeinek részletes ismertetése a diplomamunkámban lesz olvasható.

# Ábrajegyzék

# Irodalomjegyzék

[1] Dieter Fox, Wolfram Burgard, Sebastian Thrun: The dynamic window approach to collision avoidance, 1997.

[2] Oliver Brock, Oussaba Khatib: High-Speed Navigation Using the Global Dynamic Window Approach, 1999.

[3] Raphael Finkel, J.L. Bentley: Quad Trees: A Data Structure for Retrieval on Composite Keys, 1974.

[4] Kai O. Arras, Jan Persson, Nicola Tomatis, Roland Siegwart: Real-Time Obstacle Avoidance For Polygonal Robots With A Reduced Dynamic Window, 2002.

[5] Domokos Kiss, Gábor Tevesz: A Receding Horizon Control Approach to Navigation in Virtual Corridors, 2012.