

Az UAV pályatervezés szimulációja, új megoldás
létrehozása

Mihályi Géza

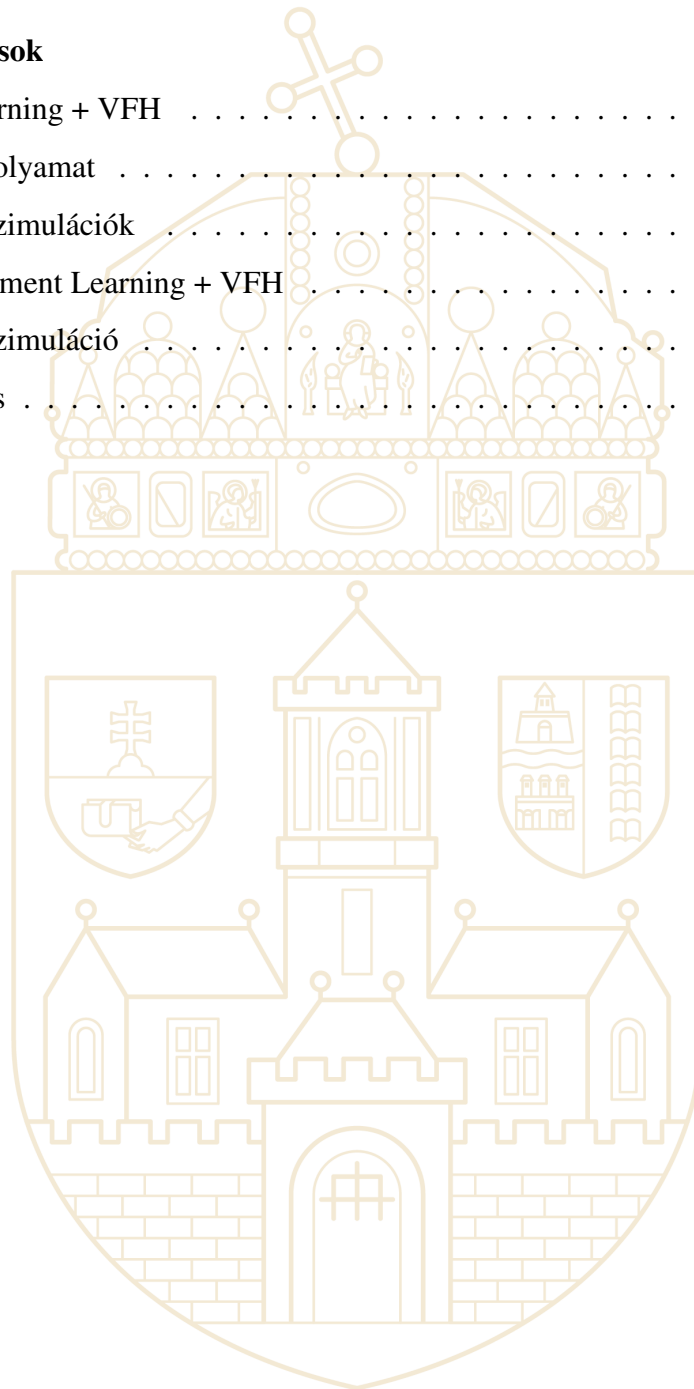
2024 Január

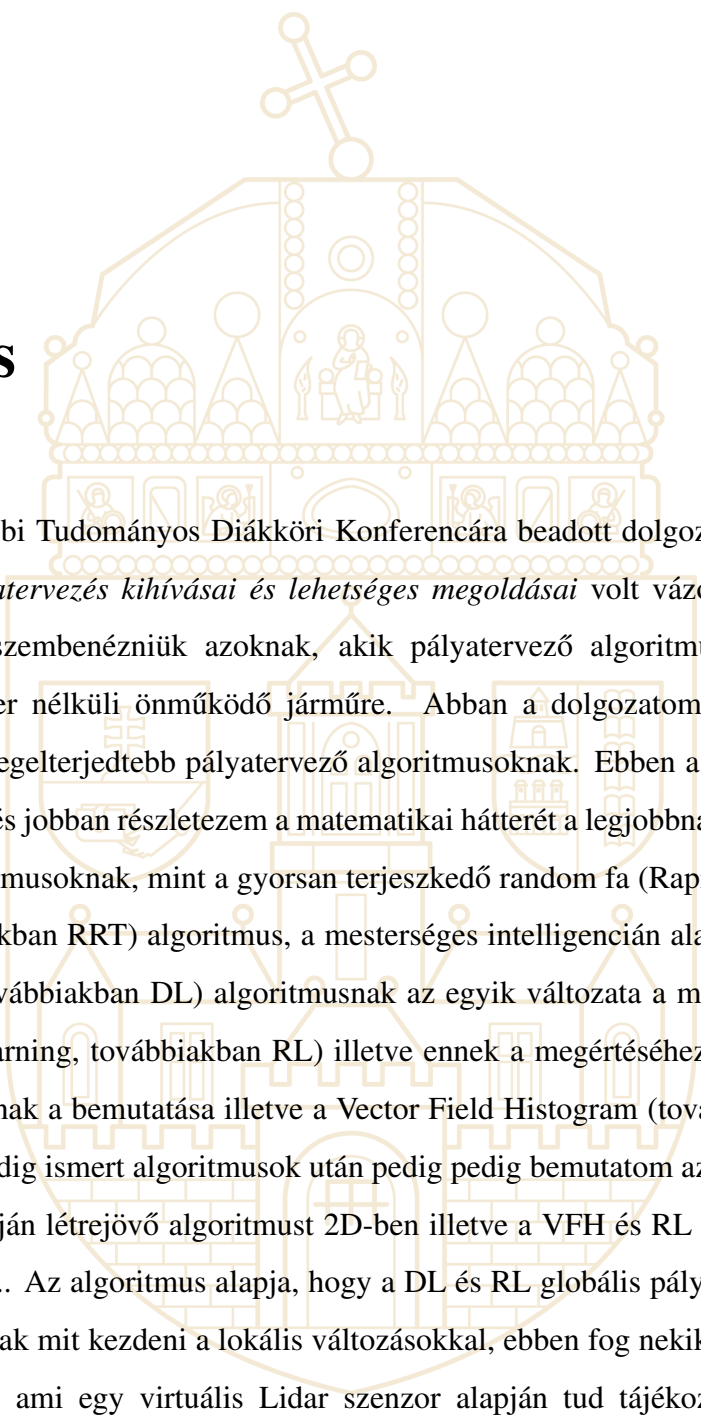
Tartalomjegyzék

1	Bevezetés	3
2	Az UAV 2D-s szimulációja	5
2.1	Gyorsan Terjeszkedő Random fák - RRT	6
2.1.1	Szimulációk	8
2.2	Deep Learning algoritmus	9
2.2.1	Tanítás	11
2.2.2	Szimulációk	12
2.3	VFH algoritmus	15
2.3.1	Szimuláció	17
2.4	Reinforcement Learning	18
2.4.1	Markovi döntési folyamatok	19
2.4.2	Tanítása	21
2.4.3	Szimulációk és eredmények	23
3	Az UAV 3D-s szimulációja	25
3.1	RRT	25
3.1.1	Szimulációk	26
3.2	VFH	26
3.2.1	Szimulációk	29
3.3	Reinforcement Learning	31
3.3.1	Programozása	32
3.3.2	Szimulációk	33



4	Saját Megoldások	36
4.1	Deep Learning + VFH	36
4.1.1	Folyamat	36
4.1.2	Szimulációk	39
4.2	Reinforcement Learning + VFH	41
4.2.1	Szimuláció	41
4.3	Összegzés	43
5	Konklúzió	44





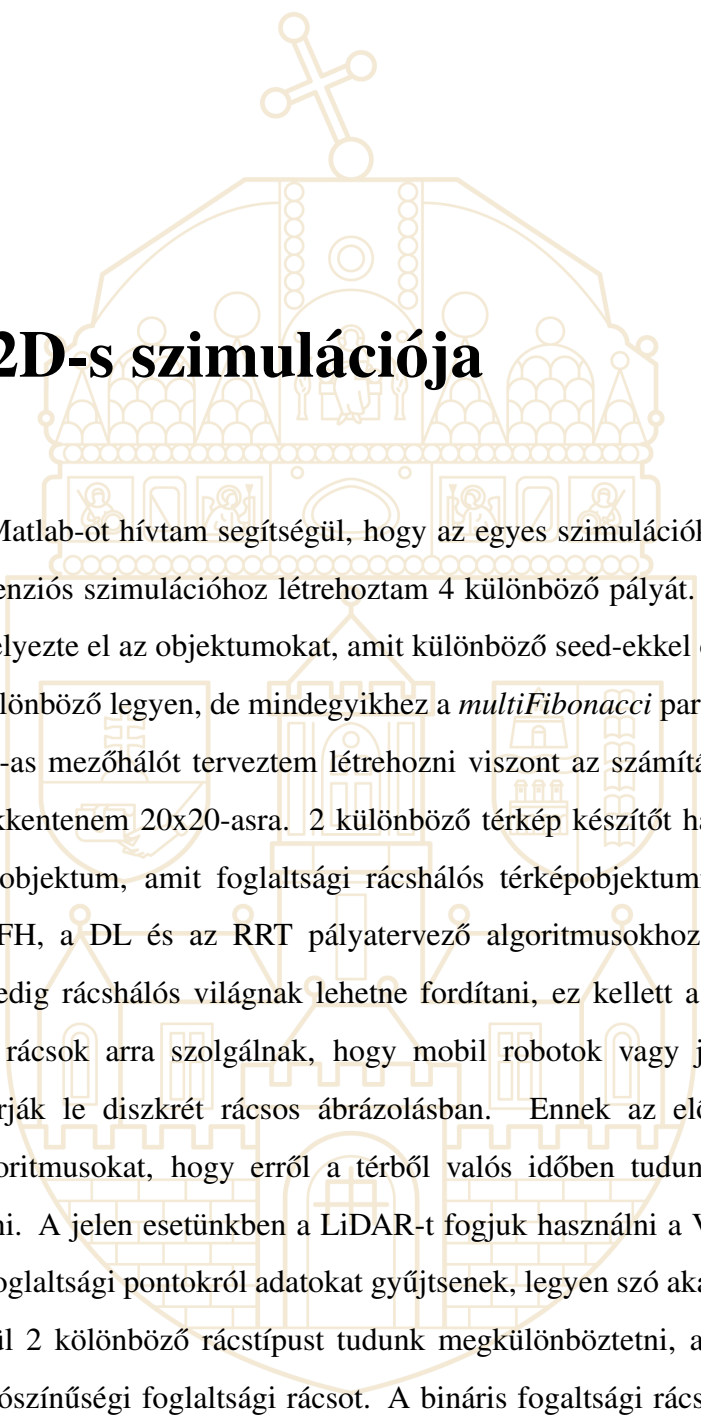
Bevezetés

Az egy évvel korábbi Tudományos Diákköri Konferenciára beadott dolgozatomban, aminek a címe *Az UAV pályatervezés kihívásai és lehetséges megoldásai* volt vázoltam, hogy milyen kihívásokkal kell szembenézniük azoknak, akik pályatervező algoritmus szeretnének írni drónra illetve ember nélküli önműködő járműre. Abban a dolgozatomban bemutattam az elméleti hátterét a legelterjedtebb pályatervező algoritmusoknak. Ebben a dolgozatban bemutatom, szimulálom és jobban részletezem a matematikai hátterét a legjobbnak tartott 2D-s, 3D-s pályatervező algoritmusoknak, mint a gyorsan terjeszkedő random fa (Rapidly Exploring Random Tree, továbbiakban RRT) algoritmus, a mesterséges intelligencián alapuló mélytanulások (Deep Learning, továbbiakban DL) algoritmusnak az egyik változata a megerősítéses tanulás (Reinforcement Learning, továbbiakban RL) illetve ennek a megértéséhez szükséges Markov döntési folyamatoknak a bemutatása illetve a Vector Field Histogram (továbbiakban VFH) algoritmus. A már eddig ismert algoritmusok után pedig bemutatom az általam létrehozott VFH és a DL fúzióján létrejövő algoritmust 2D-ben illetve a VFH és RL kapcsolatán alapuló algoritmust 3D-ben.. Az algoritmus alapja, hogy a DL és RL globális pályatervező algoritmusok, amik nem tudnak mit kezdeni a lokális változásokkal, ebben fog nekik segítséget nyújtani a VFH algoritmus, ami egy virtuális Lidar szenzor alapján tud tájékozódni és ha valami belekerül a térbe, akkor azt ki tudja majd kerülni. Ez a koncepció szállítmányozási drónoknál fog jól jönni, amikor is a boltból a vásárlóhoz kell vinni az árut, amikor is bármilyen előre nem látható akadály jön közbe vagy az elsősegély ellátásnál, amikor is mentőknél gyorsabb, a helikopternél meg költséghatékonyabb drón sürgősségi vérszállítást vagy újraélesztő készlet szállítását fogja lehetővé tenni. Akkor szükség lesz rá, hogy az útja megbízható legyen és az előre nem látható akadályokat is ki tudja kerülni. Egészségügyi drónokra már láthatunk példát a következő publikációban[1]. Miután elvégeztem a szimulációkat megállapítom, hogy melyik



a legjobb pályatervező algoritmus, illetve, hogy az általam létrehozott algoritmus mennyire effektív. A szimulációkat a MATLAB segítségével fogom elvégezni, azon belül is főleg 3 könyvtárat fogok használni, a Deep Learning könyvtárat a mesterséges intelligencia miatt, a Parallel Computing könyvtárat, mivel a mesterséges intelligenciát nem processzoron hanem videokártyán fogom tanítani mert sokkal gyorsabb, illetve a UAV könyvtárat. A dolgozat felépítése a következő módon épül fel:

- 2. fejezet - Az UAV 2D-s szimulációja
- 3. fejezet - Az UAV 3D-s szimulációja
- 4. fejezet - Saját megoldások
- 5. fejezet - Konklúzió



Az UAV 2D-s szimulációja

Dolgozatomban a Matlab-ot hívtam segítségül, hogy az egyes szimulációkat elvégezzem. Az 4 különböző 2 dimenziós szimulációhoz létrehoztam 4 különböző pályát. Ezeken a pályákon a Matlab random helyezte el az objektumokat, amit különböző seed-ekkel oldottam meg, hogy mindegyik pálya különböző legyen, de mindegyikhez a *multiFibonacci* paramétert használtam. Eredetileg 200x200-as mezőhálót terveztem létrehozni viszont az számítási kapacitás igénye miatt le kellett csökkentenem 20x20-asra. 2 különböző térkép készítőt használtam, az egyik az *occupancyMap* objektum, amit foglaltsági rácshálós térképobjektumnak lehet fordítani, ezek kellenek a VFH, a DL és az RRT pályatervező algoritmusokhoz, illetve a másik a *Grid World* amit pedig rácshálós világnak lehetne fordítani, ez kellett a RL algoritmushoz. Ezek a foglaltsági rácsok arra szolgálnak, hogy mobil robotok vagy jelen esetben UAV-k munkaterületét írják le diszkrét rácsos ábrázolásban. Ennek az előnye, hogy amikor szimuláljuk az algoritmusokat, hogy erről a térből valós időben tudunk szenzorokból információkat gyűjteni. A jelen esetünkben a LiDAR-t fogjuk használni a VFH algoritmusban, hogy a különböző foglaltsági pontokról adatokat gyűjtsenek, legyen szó akár 2D-ben vagy akár 3D-ben. Ezen belül 2 különböző rács típust tudunk megkülönböztetni, a Bináris foglaltsági rácsot, illetve a Valószínűségi foglaltsági rácsot. A bináris foglaltsági rácsban az értékek 0 és 1 értéket vehetnek fel, ha 1 értéket vesz fel az azt jelenti, hogy azon a ponton van akadály, ha nullát akkor nincs. Ebben a dolgozatban, habár memória szempontjából kedvezőbb lenne, nem ezt a módszert fogjuk alkalmazni meghagyva a későbbi fejleszthetőség lehetőségét. Ezzel szemben a valószínűségi rács a nevéből adódóan valószínűségi értékeket használ a részletesebb ábrázoláshoz. Itt a cellákhoz rendelt érték az adott cellák foglaltsági valószínűségét jelöli. 0 és 1 közötti értékeket vehet fel, ahol is minél közelebb van az egyhez annál nagyobb valószínűséggel tartalmaz akadályt és az ellentéte igaz a nullához közeli értékek nagyobb



eséllyel akadálymentesek. Ilyen típusú pályából lesz 4 különböző, amin az összehasonlító elemzések fognak folyni, illetve ebből fogunk 100000 különböző véletlenszerűen generált pályát létrehozni, hogy legyen min tanítani a Deep Learning alapú algoritmusunkat.

A másik térképtípus amit a Reinforcement Learninghez fogunk használni az is hasonló, ott is meg kell adni, hogy melyik mezőben vannak az akadályok illetve melyik mezők a szabad mezők. Ennek viszont annyi limitációja van, hogy itt nem lehetnek benne tizedestört értékek csak egész számok. Míg a foglaltsági rácsnál tudunk jelölni pontot az $x = 0.5$, $y = 1.3$ koordinátájú pontban, addig ezt nem tudjuk megtenni, a jelen típusnál. Ebben a környezetben megtudjuk adni, hogy az ügynökünk, mivel a Reinforcement Learningnél ügynökök vannak, erről a későbbiekben lesz részletesebben szó, melyik irányokba tudjon tovább haladni. Az alap beállítás szerint csak a 4 négy égtáj felé, azonban a *Kings* beállítással megtudjuk neki mondani, hogy átlósan is mozoghat.

Minden térképen az útvonaltervezés egységesen az 5;8 pontból fog indulni és a 17;17-es pontba kell megérkeznie. Ez az egyetlen közös paraméter az algoritmusok között. A különböző paraméterezéseket a szimulációs részüknél fogom részletezni.

2.1 Gyorsan Terjeszkedő Random fák - RRT

A gyorsan terjeszkedő random fákkal (Rapidly-Exploring Random Trees - RRT) kutatás szinten először 1999-ben foglalkoztak [2]. A metrikus térben amit X -el jelölnek történik a pályakeresés. Az X -et kétféle képpen lehet meghatározni vagy $X = C$, ami egy egy merev test konfigurációs terét jelenti 2D-s vagy 3D-s környezetben vagy pedig $X = T(C)$, amikor már belekódoljuk a konfigurációs teret illetve a sebességet is. Feltételezik, hogy $X_{akadaly} \subset X$, ahol is ezeket az $X_{akadaly}$ -okat el kell kerülnie. Az RRT-t úgy hozták létre, hogy minden csúcsa egy állapot az X_{szabad} térben, illetve minden éle is ugyanebben a halmazban foglaljon helyet. Ahhoz, hogy ezt megtudjuk valósítani kellett egy állapot átmeneti egyenlet amivel biztosítják, hogy $\dot{x} = f(x, u)$ amiben meghatározzák a nem holonomikus kényszereket. Nem holonomikusról akkor beszélünk amikor az UAV effektív szabadságfoka nagyobb, mint az irányítható szabadságfoka. Ebben az u vektor az U bemenetekből lévő érték, a vektor $\dot{x}$ pedig az adott állapot időszerinti deriváltját fejezi ki. Azt találták, hogyha egy adott időintervallumon (Δt) integráljuk f -et akkor a következő állapotot x_{uj} megtudjuk kapni a kiinduló állapot $x_{kiindulo}$ és a bemenet $u \in U$ is-

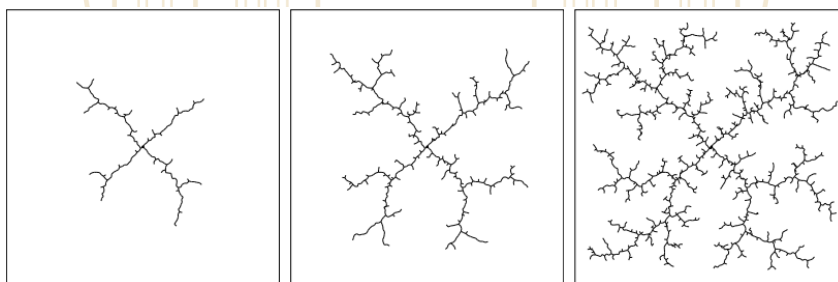
meretében. Az Euler metódust használva a következőképpen jön ki az egyenlet:

$$x_{uj} \approx x + f(x, u)\Delta t$$

Megkülönböztetik a holonomikus tervezést, ami azt jelenti, hogy bármilyen sebesség elérhető, ahol is miután integráljuk $f \Delta t$ -n, megkapunk egy új állapotot, ami a rendszert bármilyen irányba eltudja mozdtítani x -hez képest. A nem holonomikus tervezésnél a következő állapot az f kényszerezettségétől függ. A publikációban több jó tulajdonságát is felsorolták az RRT algoritmusnak:

- A terjeszkedése az algoritmusnak a számára ismeretlen terület felé húz
- A csúcsok eloszlása egy RRT-ben megközelíti a mintavételi eloszlást, ami konzisztens viselkedéshez vezet
- Egy relatíve egyszerű algoritmusról van szó
- Az algoritmus mindig kapcsolódik egymáshoz

Kutatásukban bemutatják egy példán keresztül, hogy egy 100x100-as síkon hogyan terjeszkedik az algoritmus, amikor a kiinduló helyzete az 50,50-es koordináta. Ahogy az általuk



Ábra 2.1: RRT terjedése [2]

szimulált ábrán (Ábra 2.1) is, illetve az általam szimulált ábrákon (Ábra 2.2) is lehet látni, hogy az algoritmus gyorsan terjeszkedik a tér mind a 4 sarka felé. Míg a klasszikusnak nevezett véletlen séta (*Random walk*) algoritmusoknak az a hibájuk, hogy hajlanak az olyan helyek felé keresni amerre már jártak egyszer, addig ez az algoritmus az ellentétét csinálja és az olyan irányokba terjeszkedik inkább ahol ezelőtt még nem volt. Ez az algoritmus leírható úgy is mintha iteratíván bontaná fel a nagyobb Voronoi régiókat kisebbekre. Elmondható, hogyha az



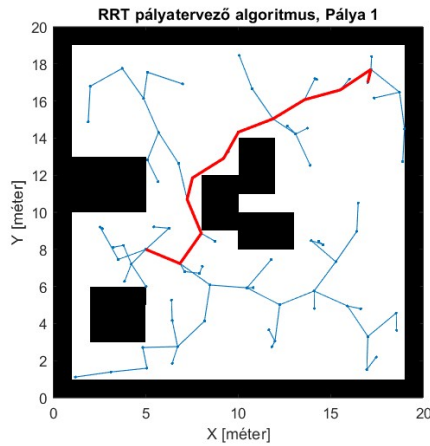
x_{random} pontokat egy valószínűségi mintafüggvényből vételezzük $p(x)$ vagyis az élei az RRT-nek a $p(x)$ szerint fognak eloszlani. A kutatásban megmutatták, hogy a legnagyobb nehézsége az algoritmusnak az az, hogy megfelelően effektív közeli szomszéd algoritmust (*nearest-neighbor*) kell alkalmazni és találni. Azonban kimutatták azt is, hogy ez a megoldás kifejezetten alkalmas lehet pályatervező algoritmusnak is kiegészítve a potenciális mező algoritmussal (*Potential Field Method*) vagy a későbbiekben szó eső VFH algoritmussal is.

2.1.1 Szimulációk

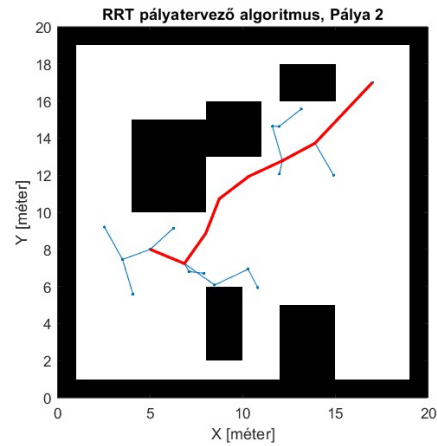
Szimulációmhoz segítségül hívtam a *Navigation Toolbox*-ot, aminek segítségével tudtam szimulálni az RRT algoritmust az általam véletlenszerűen elkészített 4 térképre. Működése a következőképpen történik[3] :

1. Az algoritmus elkezd mintavételezni véletlenszerű állapotokat (x_{random}) a állapot térben.
2. Az algoritmus talál egy állapotot $x_{mellette}$ ami benne van a keresési fában és a legközelebbi a x_{random} -hoz
3. Az algoritmus elkezd terjeszkedni $x_{mellette}$ -ből az x_{random} -ig amíg el új állapot x_{uj} nem kerül elérésre
4. Az új állapot x_{uj} hozzáadódik a keresési fához

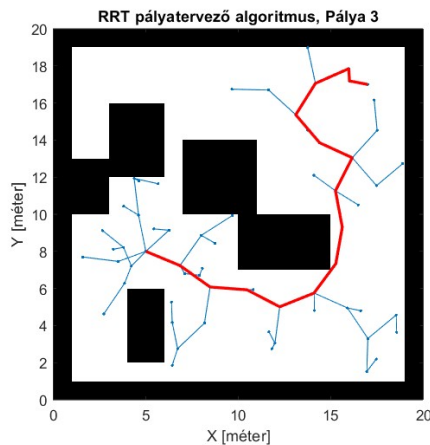
A szimulációkban is jól látszik, amit a kutatásban is leírtak, hogy az algoritmus közelít a tér sarkaihoz, kifejezetten jól megfigyelhető ez a 2.2 (a) ábráján, ahol is a kereső fa a legjobban kiterjedt a 4 térkép közül. Az algoritmus hatékonyságát és optimalizálhatóságát a *MaxConnectionDistance* paraméterrel lehet beállítani, amiben megtudjuk határozni, hogy milyen messzire nyúljon ki maximálisan 1-1 ága a fának. A szimulációimban úgy találtam, hogy ez az érték nálam a 2. Azaz 1-1 ága maximálisan 2 méterre tudott elérni az adott szimulációs környezetben. Későbbiekben ezt az algoritmust hasonlítottam össze a DL algoritmussal, hogy megvizsgáljam melyikkel érdemes tovább folytatni a lokális és globális keresőalgoritmus összekapcsolását.



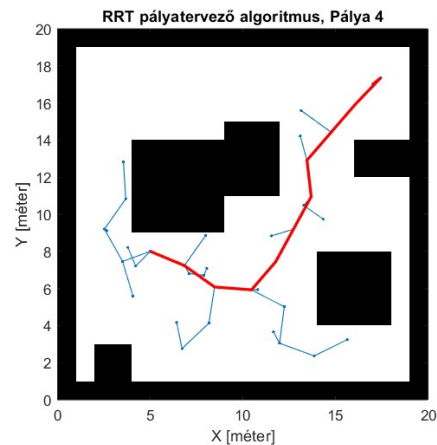
(a) Az 1. térkép megoldása RRT algoritmussal



(b) A 2. térkép megoldása RRT algoritmussal



(c) A 3. térkép megoldása RRT algoritmussal



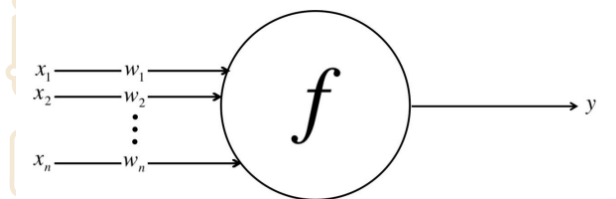
(d) A 4. térkép megoldása RRT algoritmussal

Ábra 2.2: A 4 térkép megoldása RRT algoritmussal

2.2 Deep Learning algoritmus

A Deep Learning, azaz mély tanulás a mesterséges intelligencián belül a gépi tanulás egy fajtája, mégpedig az, amikor több neurális hálót kapcsolunk össze. A Gépi tanulást úgy lehetne leírni, mint egy olyan számítási módszert, ami múltbeli tapasztalatot használ annak érdekében, hogy a teljesítményt javítsa. A gépi tanulást fel lehet bontani 3 különböző csoportba, ilyen a felügyelt, nem felügyelt, illetve a megerősítéses tanulás. A felügyeltnél a tanuláshoz szükséges információk és a hozzájuk tartozó megoldások ismertek míg a nem felügyelt tanulásnál nem áll rendelkezésre az megoldáshoz szükséges információ. Ebben a dolgozatban az előbbit fogjuk alkalmazni. Ez megfelelő arra, hogy számára ismeretlen példákön is felismerje azokat a formajegyeket, amiket a tanult példán felismert. Emiatt kifejezetten alkalmas például a képfelismerésre.

Ezek a felügyelt rendszerek neurális hálózatokból épülnek fel, ezeknek mindig van egy tanulási algoritmusuk, amik a mintákat megtanulják vagyis ami alapján feldolgozzák az információt, illetve van nekik egy előhívási módjuk is, amiben a már megtanult mintákat tudják előhívni és ráhúzni az eddig nem látott adatokra. Egy ilyen neurális hálózatnak a központi eleme a neuron, ami a következő képpen működik (2.3).



Ábra 2.3: Neuron felépítése [4]

Ezeknek a neuronoknak a számítása úgy működik, hogy adottak a bemenetek ($X = \{x_1, x_2, \dots, x_n\}$), majd ezeket valamilyen súly értékkel $W = \{w_1, w_2, \dots, w_n\}$ megszorozzuk, majd ezekhez hozzáadunk egy úgynevezett eltolás *bias* értéket és ezekkel állítjuk elő a neuron kimenetét Z :

$$Z = \sum_{i=0}^n X_i W_i + b$$

Ezeknek az neuronoknak valahogyan aktiválódniuk is kell, amiatt van egy úgynevezett nem-lineáris aktivációs függvény, ami a következőképpen néz ki:

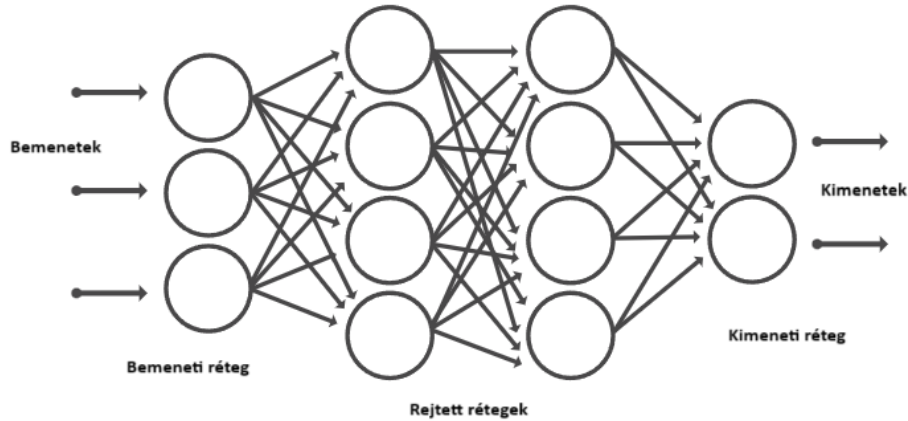
$$f(Z) = Y$$

A 3 leggyakrabban használt nem lineáris függvény a szigmoid (σ), a tangens hiperbolikus (tanh) és a ReLU (Rectified Linear Unit). Az utóbbit leggyakrabban a belső rétegeket használják mert gyorsabban konvergál. [4]

$$f(x) = \max(0, x)$$

Ezeknek a neuronoknak a kimenete lehet vagy a végső kimenet vagy pedig hozzacsatlakozhat egy másik neuronhoz is. A második esetben beszélünk Deep Learning algoritmusról. Gyakorlatban azokat a hálózatokat nevezzük mély neurális hálózatoknak amikben 2 vagy több elrejtett

szint (*hidden layer*) található. Erre a következő ábrán láthatunk példát (2.4)[5].



Ábra 2.4: Az ábrán látható hogyan is épül fel egy Deep Learning hálózat [6]

2.2.1 Tanítás

Neuron hálók tanításán egy többváltozós optimalizálási eljárást értünk aminek van egy előre meghatározott kritériumfüggvénye $E(\xi)$. A legismertebb ilyen optimalizálási eljárások a gradiens alapú stratégiák, ahol is a gradiens a függvény deriválásán alapul. Itt kereshetünk vagy maximumot vagy minimumot, majd aszerint kell az $E(\xi)$ -nek csökkennie vagy növekednie. Ezt úgy kell elképzelni, hogy egy adott k pontban a $\xi(k)$ ponthoz egy érintőt húzunk, ami a függvény deriváltja. Ha a derivált pozitív akkor növekvő ξ értékekre a költségfüggvény is növekszik, ha a derivált negatív akkor a növekvő ξ értékekre a költségfüggvény csökken. Vagyis ha a minimumot szeretnénk számolni akkor ellentétes előjellel kell változtatni a ξ paraméter értékét. A következő módon keressük a minimumot:

$$\xi[k+1] = \xi[k] - \mu \frac{\partial E(\xi)}{\partial \xi}$$

Ebben a képletben a μ vagy máshol α a tanulási sebességet (*learning rate*) jelenti és az fogja meghatározni, hogy az algoritmus milyen gyorsan konvergál. Ezeket az értékeket általában tapasztalatok alapján állítjuk be, abban az esetben ha túl nagy ez az érték akkor nem fog megtörténni a tanulás és ugrálni fog a hibafelületen viszont ezzel ellentétben ha túl kicsi ez az érték akkor a tanulás lassabb lesz és elképzelhető, hogy megakad a lokális optimumban.

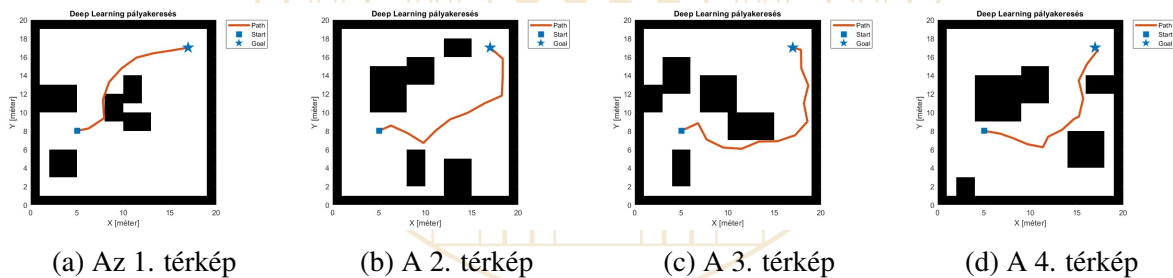
Szükségünk van egy hibafüggvényre is, aminek a feltételei, hogy deriválható legyen és, hogy számszerűsíteni tudjuk vele a hálózat hibáját. Ebből az egyik legismertebb hiba az átlagos négyzetes eltérés (*mean squared error*) (2.1). [5]

$$C_{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad (2.1)$$

Majd ez a hiba alapján fogjuk frissíteni a neuronhálóknak a súlyait, amivel pedig végbemegy a tanulás.

2.2.2 Szimulációk

Szimulációmhoz segítségül hívtam a *Deep Learning Toolbox*-ot, amiben már bele vannak implementálva a tanítási algoritmusok, így már csak az maradt hátra, hogy meghatározzuk a paramétereket. A tanuláshoz, hogy a Deep Learning algoritmusnak legyen miből tanulnia létrehoztam 100000 különböző elrendezésű és kezdő illetve véghelyzetű 20x20-as térképet, amiket meg is oldottam előre az A* algoritmus egyik speciális fajtájával. Majd ezt az adathalmazt szétbontottam tanulási illetve validációs adathalmazokra 80:20-as arányban. Majd a tanítás végeztével a mély neurális hálózatot elmentettem és elkezdtem letesztelni, hogyan működik a környezetünkben. Az eredményt a alábbiakban láthatjuk (2.5).

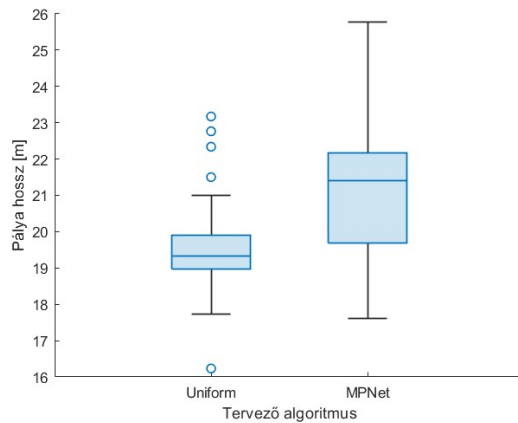


Ábra 2.5: A 4 térkép megoldása DL algoritmussal

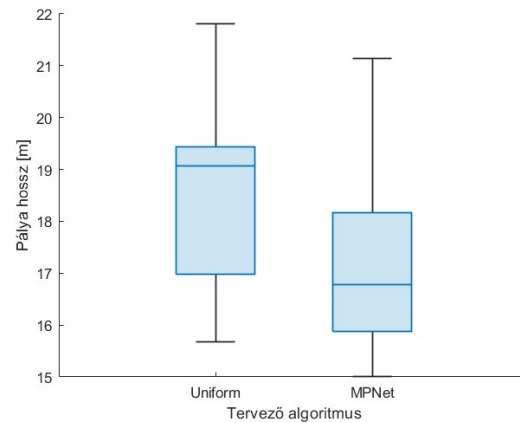
Az pályatervezésen jól látszik, hogy sikerült mindegyik térképen eljutnia a kiinduló helyzetből a cél helyzetben. A 2.5 ábra (a) és (c) térképén jól látható, hogy ott is nagyon megközelíti a falat, amit lehetséges, hogy egy lokális pályatervező algoritmus ki tud korrigálni és emiatt nem áll fent a veszélye az ütközésnek.

Majd ezután összehasonlítottam az előzőekben bemutatott RRT algoritmussal, hogy megvizsgáljam, hogy mennyivel jobb vagy esetleg rosszabb az algoritmus. A tesztben a 30

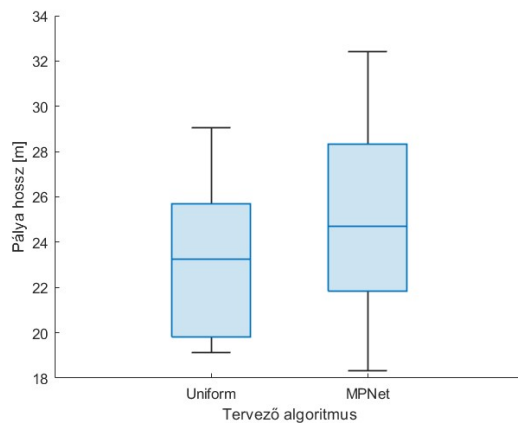
futást hajtottam végre amiben a pálya hosszát (2.6 Ábra) illetve a pálya egyenletességét (Ábra 2.7 Ábra) vizsgáltam. Ez a vizsgálat bele van építve a MATLABba így azzal hajtottam végre a teszteket.



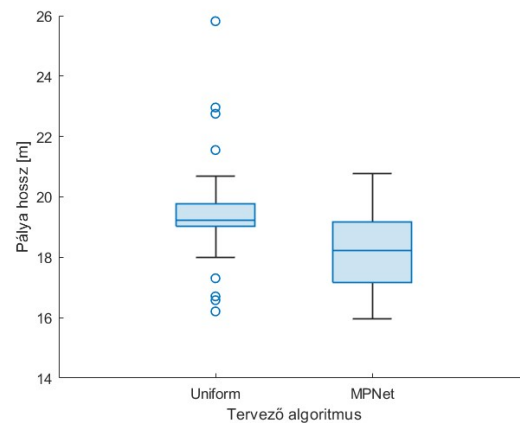
(a) Az 1. térkép összehasonlítása



(b) A 2. térkép összehasonlítása



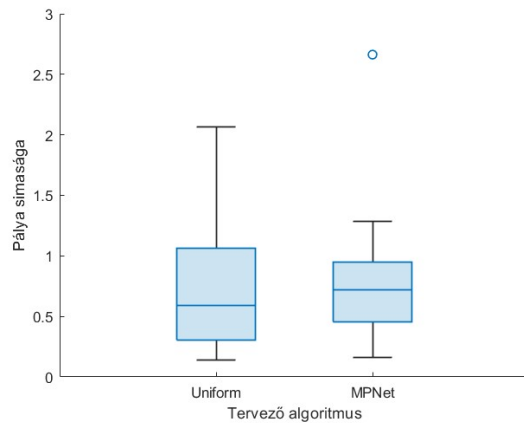
(c) A 3. térkép összehasonlítása



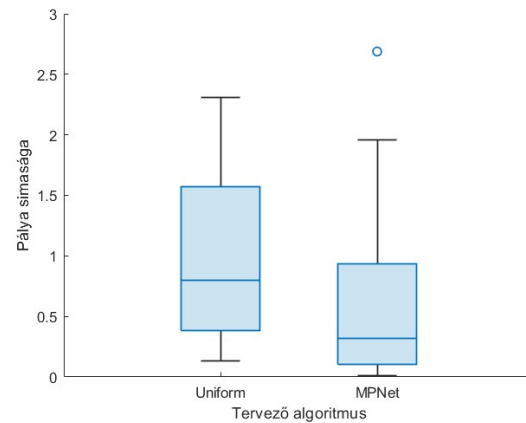
(d) A 4. térkép összehasonlítás

Ábra 2.6: A 4 térkép Összehasonlítás az RRT algoritmussal pálya hossza szerint

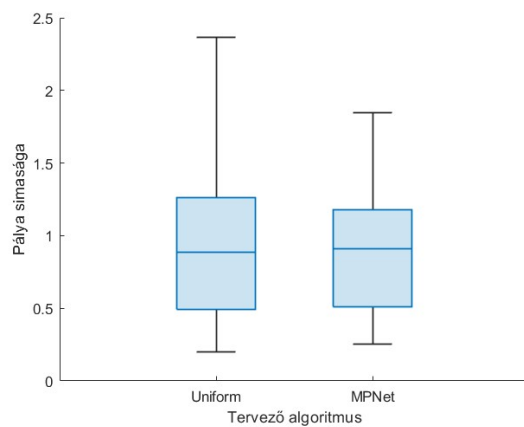
A tesztekben megállapítható, hogy 2 térképen jobban teljesített a DL algoritmus, 1 térképen hasonlóan illetve 1 térképen jobban teljesített az RRT algoritmus. Jól látható, hogy az első térképen 2-3 méterrel rövidebb utat sikerült találnia az RRT algoritmusnak, viszont pontosan az ellentéte a helyzet a 2. térképen illetve a 4. térképen. A harmadik térképen pedig habár átlagosan 1 méter körül lehet az eltérés, addig az RRT algoritmus alsó kvartilis értéke jóval kisebb mint a DL társáé, viszont legrövidebb utat, itt is mint mindegyik tesztben a DL algoritmus hajtotta végre.



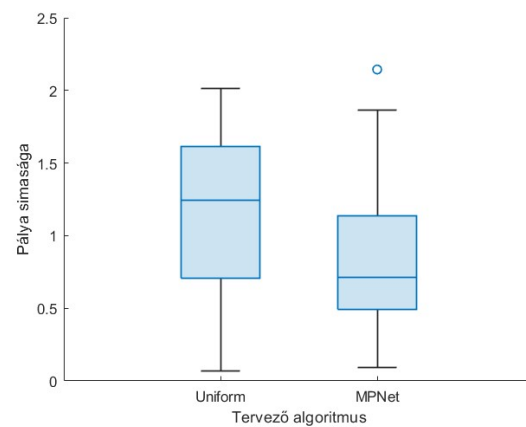
(a) Az 1. térkép összehasonlítása



(b) A 2. térkép összehasonlítása



(c) A 3. térkép összehasonlítása



(d) A 4. térkép összehasonlítása

Ábra 2.7: A 4 térkép Összehasonlítás az RRT algoritmussal pálya egyenletessége szerint

Másik teszt ami lefuttatásra került, az pedig a pálya egyenletességére volt viszonyítás. Ahogy láthatjuk egyenletességben, hasonló a helyzet mint a pálya hosszánál. Az első térképen a medián érték inkább az RRT algoritmus felé húz, de nem olyan jelentősen mint a pálya hosszánál. A harmadik térképnél pedig szinte teljesen megegyező értéket kapunk, a 2. kvartilis érték közel ugyanott helyezkedik el. Viszont a második és negyedik pályán hasonlóan nagy az eltérés a DL javára mint az első pálya hosszára vonatkozó elemzésen.

Elemzésem szerint kimondható, hogy a vizsgált mintákon a DL algoritmus felülmúlta az RRT algoritmust.



2.3 VFH algoritmus

A Vektortér hisztogram módszer (Vector Field Histogram, továbbiakban VFH) a többiekétől eltérően egy lokális pályatervező algoritmus, ami a környezetét érzékelő szenzorok alapján mutatja meg, hogy melyik irányba kell kikerülni az akadályt. Nem ismeri a teljes pályát, csak a cél illetve a saját koordinátáját tudja. Ezt a módszert Johann Borenstein és Yoram Koren dolgozták ki 1991-ben [7]. Az a módszer nagyban építkezik az szerzőpáros előző objektum kikerülő algoritmusára a Virtuális erőter algoritmusra (Virtual Force Field - VFF) csak annak a hibáinak a kijavításával. Arra jöttek rá, hogy az VFF-nél túlzottan is le lett egyszerűsítve az eljárás és a helyi akadályelosztásról szóló részletes információk elvesztek. Annak érdekében, hogy ezt a hibát kiküszöböljék 3 adat reprezentációs szint lett létrehozva:

- A legfelső szint tartalmazza a részletes információt a robot környezetéről. Ebben a környezetben egy 2 dimenziós C koordináta rendszerben lévő adatokat folyamatosan frissítjük a valós idejű szenzorok által.
- A középső szinten van egy 1 dimenziós polár hisztogram H , amit a robot pillanatnyi helyzete köré képez. Ez a H n darab α szélességű szektorból áll. Egy transzformációval az aktív régió C^* -ot beletranszformáljuk a H -ba, amiben is minden k egy h_k értéket tartalmaz. Ez az érték a *poláris akadály sűrűséget* jelenti a k szektor irányába.
- A legalsóbb adat reprezentációs szint pedig a kimenete az algoritmusnak. Ez adja meg a referencia értéket a fordulási értéket az eszköznek.

Minden aktív cellának a tartalma a hisztogram rácsban egy akadály vektornak felel meg és az iránya a robot középpontjától β (2.2) és a nagyságrendje m (2.3).

$$\beta_{i,j} = \tan^{-1} \frac{y_i - y_0}{x_i - x_0} \quad (2.2)$$

$$m_{i,j} = (c_{i,j}^*)^2 (a - b d_{i,j}) \quad (2.3)$$

ahol is

a, b pozitív konstansok

$C_{i,j}^*$ Valószínűségi érték, hogy mennyire valószínű, hogy az aktív cellában (i,j) van érték

$d_{i,j}$ Az aktív cella (i,j) és a UAV középpontja közötti távolság

$m_{i,j}$ Az akadályvektor nagysága a cellában (i,j)

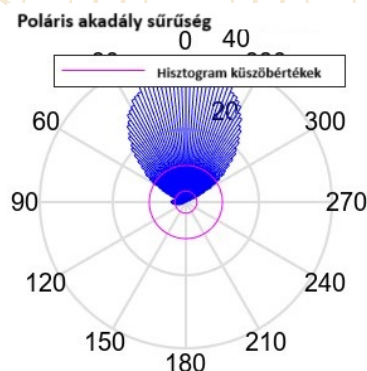
x_0, y_0 Az UAV koordinátái

x_i, y_i Az aktív cella koordinátái (i, j)

$\beta_{i,j}$ Az iránya az aktív cellának az UAV-től

A $C_{i,j}^*$ négyzetre van emelve, amiatt, hogy ott ténylegesen egy akadályt ábrázoljon ne pedig valamilyen zajt. Ezen kívül $m_{i,j}$ proporcionális $-d$ -re. Vagyis azok az cellák amik a robot közvetlen közelségében vannak nagyobb akadályvektort fognak jelenteni, míg azok amíg távolabb vannak kisebbeket.

Majd a második szinten ki lesz számolva a kanyarodási szög θ . Ahogy lehet látni a 2.8 ábrán, tartalmaz kiemelkedő pontokat és völgyeket ahol nem feltétlenül van érték. Minden olyan völgy ami egy küszöbérték alatt van az egy potenciális lehetőség. A VFH algoritmus azt a völgyet választja ki ami a legközelebbi egyezést mutatja a k_{cel} irányával. Majd miután ez megvan utána egy megfelelő szektort kell találni azon a völgyön keresztül. Majd ezután 2 különböző völgyet különböztetünk meg széleset illetve szűket és a számolása mindkettőnél különböző. Az, hogy mi alapján dől el, hogy szűk vagy széles völgyről beszélünk attól függ, hogy mennyi egymást követő szektor van a poláris akadály sűrűségben (Polar Obstacle Density).



Ábra 2.8: A 2.9 ábra (c) ábrájának a hisztogramja



A völgy szélesnek számít, ha több mint s_{max} egymást követő szektor esik a küszöbérték alá (Kutatásukban az s_{max} 18 volt). Ilyenkor az a szektor ami a legközelebb van a k_{cel} -hez és a küszöbérték alá esik k_n lesz ez a közelebbi határ. A távolabbi határ k_f -el van jelölve és a következő képlettel kerül kiszámításra: $k_f = k_n + s_{max}$. Ezek után a kívánt fordulási irány

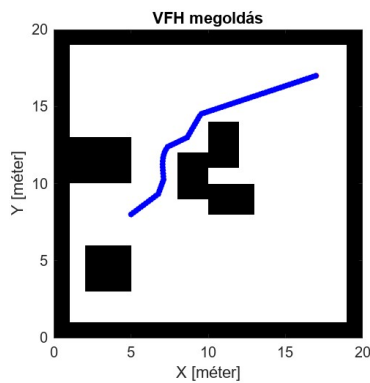
$$\theta = \frac{(k_n + k_f)}{2} \quad (2.4)$$

Ebben az esetben, ha a UAV túl közel kerül az akadályhoz akkor a θ tőle elfele mutat, ellenben ha a UAV távol van az akadálytól akkor a θ közelebb engedi. Azt, hogy milyen messze kerül az UAV az akadálytól azt az s_{max} határozza meg. Ha ez az érték nő, akkor az UAV-t nem engedi közelebb, ellenben minél kisebb a szám annál közelebb tud kerülni az UAV az akadályhoz. Szűk állapot akkor áll fent amikor az UAV egy szűk helyen akar átmenni. Ebben az esetben a távoli határnak k_f a közelebbi határtól k_n való távolsága kisebb mint az s_{max} . De a kanyarodási irány így is ugyanaz mint a széles esetben (2.4). Ezzel az UAV egyenlő távolságot tud tartani a 2 akadálytól [7].

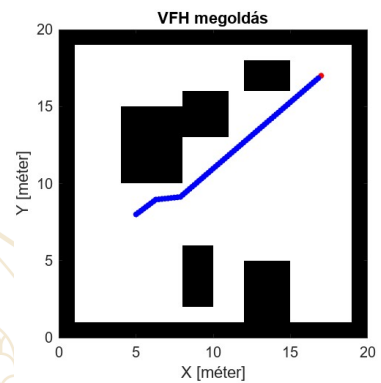
2.3.1 Szimuláció

Mivel ez egy lokális pályatervezési eljárás ezért ennél a mindig a leképezett háromszög átlója mentén haladt volna az UAV és amikor valamilyen akadály került elé akkor korrigálta magát és irányt változtatott.

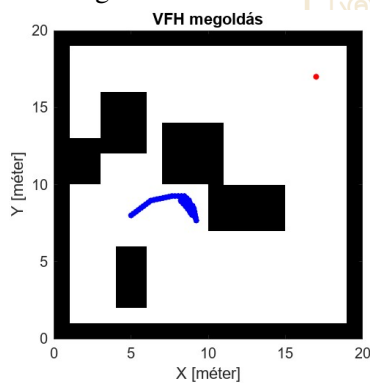
A 2.9 (c) ábráján jól látható, hogy miért nem elegendő csak ez a megoldás. A VFH algoritmus nem tudott kijönni a 2 épület szorításából. Ez volt eddig az egyetlen olyan algoritmus ami nem tudott minden pályán eljutni a kiinduló helyzetből a cél helyzetben. Természetesen ennek nem is az lenne a célja, hanem, hogy az akadályokat kikerülje, emiatt is fogom ezt a metódust illetve az egyik legsikeresebb másik metódust, a Deep Learning algoritmust vegyíteni, hogy létrehozzák egy minden igény kielégítő pályatervező algoritmus párost. A Deep Learning algoritmus lesz majd a felelős azért, hogy a nagy és statikus dolgokat betájolja és ne menjen a közelébe míg a VFH lesz a felelős, hogy a dinamikus dolgokat pedig kikerülje.



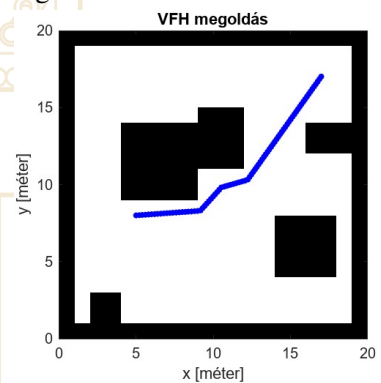
(a) Az 1. térkép megoldása VFH algoritmussal



(b) A 2. térkép megoldása VFH algoritmussal



(c) A 3. térkép megoldása VFH algoritmussal



(d) A 4. térkép megoldása VFH algoritmussal

Ábra 2.9: Az 4 térkép megoldása VFH algoritmussal

2.4 Reinforcement Learning

A Reinforcement Learning, azaz Megerősítéses tanulás a Deep Learningen belül egy speciális tanulási forma. Több téren is eltér a hagyományosnak nevezhető felügyelt (Supervised) vagy felügyeletlen (Unsupervised) gépi tanulási módszertantól. Egy hagyományos tanulási eljárásnál a legtöbb esetben egy döntést/cselekvést kell meghoznia, ilyen például a klasszikus példa, hogy a képen egy macskát vagy egy kutyát látunk. Az RL-nél ezzel ellentétben több döntést, döntések sorozatát kell meghoznia. A tanításánál ellentétben nem minden lépésnél kap egy visszajelzést, hanem amikor egy szimulációt lefuttatott, akkor a szimuláció teljese alapján kap valamilyen pontszámot, amit *reward*-nak nevezünk vagy magyarul jutalomnak. Ebből következik, hogy a fő célja egy olyan stratégia megtanulása, amivel az egy szimuláció alatt elért pontszámait maximalizálni tudja.

Talán a legismertebb formája a Q-Learning vagy Q-tanulás. Onnan kapta a nevét, hogy egy



Q-függvényt tanul meg, ami az adott helyzetben az adott cselekvés hasznosságát becsüli meg, vagyis az állapot+cselekvés párhoz rendel hasznosságot.

Mielőtt elkezdődik a tanulás két opció áll fent, az ügynökünk vagy ismeri már a teljes környezetet vagy még azt is meg kell tanulnia, majd tanítás közben 2 fajta tanuló lehet, aktív és passzív. A passzív tanuló figyel a világ alakulását és tanul, ez egy előre rögzített eljárásmod, itt az ügynökünk nem mérlegel csak előre kódoltan cselekszik. Míg az aktív tanuló a megtanult információ birtokában az eljárásmodot is alakítja és ennek tudatában optimálisan fog cselekedni, ezt nevezik még felfedezőnek is.[8]

2.4.1 Markovi döntési folyamatok

A megerősítéses tanulásnak az egyik alappillére a Markovi döntési folyamatok (Markovian Decision Process) amit Richard Bellman publikált 1957-ben [9]. Ezek a folyamatok kifejezetten alkalmasak arra, hogy szekvenciális döntéseket írjunk le velük. Alapvetései, hogy adottnak tekintünk egy állapotteret (Ezt a MATLABban *statespace*-nek nevezik) amivel lekódoljuk azt a teret amit használni szeretnénk. Ez lehet vagy a valóságnak a lekódolása vagy véletlenszerű környezeteké amiket ebben dolgozatban is létrehoztam. Ezeknek az állapottereknek minden eleméhez tartozik valamilyen jutalom vagy költség, ez írja le, hogy mennyire is kedvező az az állapot. Így felmerül a kérdés, hogy hogyan kell a legoptimálisabb döntéseket meghozni. Ahhoz, hogy ezt megtudjuk vizsgálni, fontos azzal a feltételezéssel élni, hogy a döntési folyamat *Markov tulajdonságú*[10]. Akkor nevezünk egy *sztochasztikus* folyamatot Markov tulajdonságúnak ha minden $B \in \mathcal{B}$ halmazra és minden olyan esetben ahol t nagyobb mint s fennáll a

$$\mathbb{P}(X_t \in B | \mathcal{F}_s) = \mathbb{P}(X_t \in B | X_s)$$

amiben is a következőket jelentik a betűk:

$\mathbb{P}$ Valószínűségi mező

X_t, X_s Adott sztochasztikus folyamat valószínűségi változók összessége

B mérhető tér

$\mathcal{F}_s$ s időpontbeli σ -algebra



Ennek egy speciális esete amikor az I állapotter megszámítható mert ebben az esetben Markov-lánccról beszélünk [11]. Az $\{X_t\}_{t \in \mathbb{N}}$ valószínűségi változó sorozatot nevezzük Markov-lánccnak ahol

- X_t értékei az $1, 2, \dots, M$ halmazból kerülnek ki minden $t \in \mathbb{N}$ esetén
- $\{X_t\}_{t \in \mathbb{N}}$ teljesíti a Markov tulajdonságot, azaz $\forall t \in \mathbb{N}$ és $\forall i \in [\mathcal{M}]$

$$\mathbb{P}(X_{t+1} = i | X_0, X_1, X_2) = \mathbb{P}(X_{t+1} = i | X_t)$$

Az egyszerűség kedvéért csak a homogén Markov láncokat használunk, ahol az átmenetvalószínűségek változatlanok azaz $\mathbb{P}(X_{t+1} = j | X_t = i)$ valószínűségek csak az i és j értékétől függenek, t -től nem. Ezután érdemes az egylépéses átmenetvalószínűségeket egy $\mathcal{P} \in \mathbb{R}^{M \times M}$ átmeneti mátrixba rendezzük:

$$\mathcal{P}_{ij} := \mathbb{P}(X_{t+1} = j | X_t = i)$$

Vagyis egy i állapotból elérhető a j állapot ($i \rightarrow j$), ha $\exists n$, hogy $\mathbb{P}(X_n = j | X_0 = i) > 0$. Az i állapot rekurrens amennyiben ($i \rightarrow j$) maga után vonja, hogy ($j \rightarrow i$). Azokat az állapotokat, amik nem rekurrens, tranzienseknek nevezzük. Ezeknek a Markov-láncoknak a folyamat i állapotához egy $r_i \in \mathbb{R}$ jutalom vagy költség van hozzárendelve.

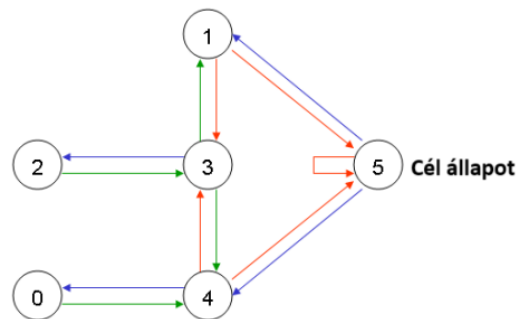
Majd ezek után bevezetésre kerülnek a modellbe a döntéshozás lehetősége ami azt jelenti, hogy a következőkben a döntéshozónak választania kell a különböző jutalmak is hozzájuk tartozó átmenetvalószínűségek között. Egy adott i állapotból hozható döntések száma legyen K_i . Ezután a minden egyes fejlődési lépés előtt a döntéshozó a következő halmazból választ egy jutalmat és hozzá tartozó egylépéses átmenetvalószínűséget.

$$(r_i^{(k)}, \{\mathcal{P}_{ij}^{(k)} : j \in [\mathcal{M}]\}); k \in [\mathcal{K}_i]$$

Majd megszületik az i állapotban a $k \in [\mathcal{K}_i]$ döntés, a folyamat következő állapotába való lépés a következő $\{\mathcal{P}_{ij}^{(k)} : j \in [\mathcal{M}]\}$ eloszlás szerint történik, illetve az $r_i^{(k)}$ jutalommal adatik. [10]

2.4.2 Tanítása

A Q-tanulás egyszerű szemléltetésére egy a Szegedi Tudományegyetem által készített példát fogok használni. [12] Tegyük fel, hogy van 6 állapotunk, mindegyik állapotnak van egy száma 0-tól 5-ig. Ebben az 5-ös a cél állomásunk. Ezt feltudjuk vázolni egy gráfként, amit a következő ábrán szemléltetek 2.10. Minden cselekvéshez tartozni fog egy jutalom, ezek közül egy kiemelkedő jutalom lesz, a 5-ös mezőben, ott kezdetben 100-as jutalommal dolgozunk. A többi akcióhoz, amit itt az élek testesítenek meg 0-át adunk, mivel kezdetben nem tudjuk, hogy az adott cselekvés mennyire lesz hasznos. A hiányos értékeket -1-el kódoljuk. Ez azért kell, hogy ezekhez biztos ne menjünk, mert ott valószínűleg nem lehet eljutni az egyik állapottól a másikba. Például a 0-as állapottól az 1-es állapotba -1 lesz az érték, mivel ott nincsen él amivel el tudnánk oda jutni. Ezen kívül kell létrehoznunk egy Q táblát is, amiben a becsült hasznosságot fogjuk tárolni. Kezdetben minden Q értéke 0 majd ez folyamatosan frissülni fog. Ebben az példában mindig véletlenszerű pozícióból indulunk, ez a saját példánkban nem így lesz, ott mindig egy meghatározott kezdőpontban fog indulni a tanítás. Az egész algoritmust



Ábra 2.10: Gráf ábrázolása az 5 állapotnak, amiben az 5-ös a cél állapot [12]

megfogja határozni az frissítési szabály, ami a következő:

$$Q(\text{allapot}, \text{akcio}) = R(\text{allapot}, \text{akcio}) + \alpha * \max[Q(\text{kovetkezoallapot}, \text{mindenlehetoseg})] \quad (2.5)$$

Az α a tanulási sebességet jelenti (learning rate), aminek a hatása különböző ha nullához közeli az érték illetve ha egyhez közeli az érték. Abban az esetben ha 0-hoz közelít, az azt jelenti, hogy az aktuális jutalom fontosabb mint a hosszú távon várható jutalom. Ennek az



ellentéte igaz, ha az érték 1 közeli, akkor az aktuális jutalom helyett a hosszú távú haszonra fókuszál. Ezt az értéket a tanítás során nekünk kell megválasztani. Miután elvégeztük a betanulást ugyanezt az algoritmust használjuk, csak akkor már nem történik meg maga a tanítás, amit frissítésnek is nevezhetünk. Tételizzük fel, hogy az $\alpha = 0.8$ illetve azt, hogy az 1-es pontból indulunk. Innen tudunk továbbmenni a 3. és az 5. szobába. A véletlenszámgenerátor ebben az esetben az 5-öt hozta ki győztesnek így az ötös szobába megyünk tovább. Ebben az esetben a következőképpen épül fel a Q mátrixot meghatározó egyenlet:

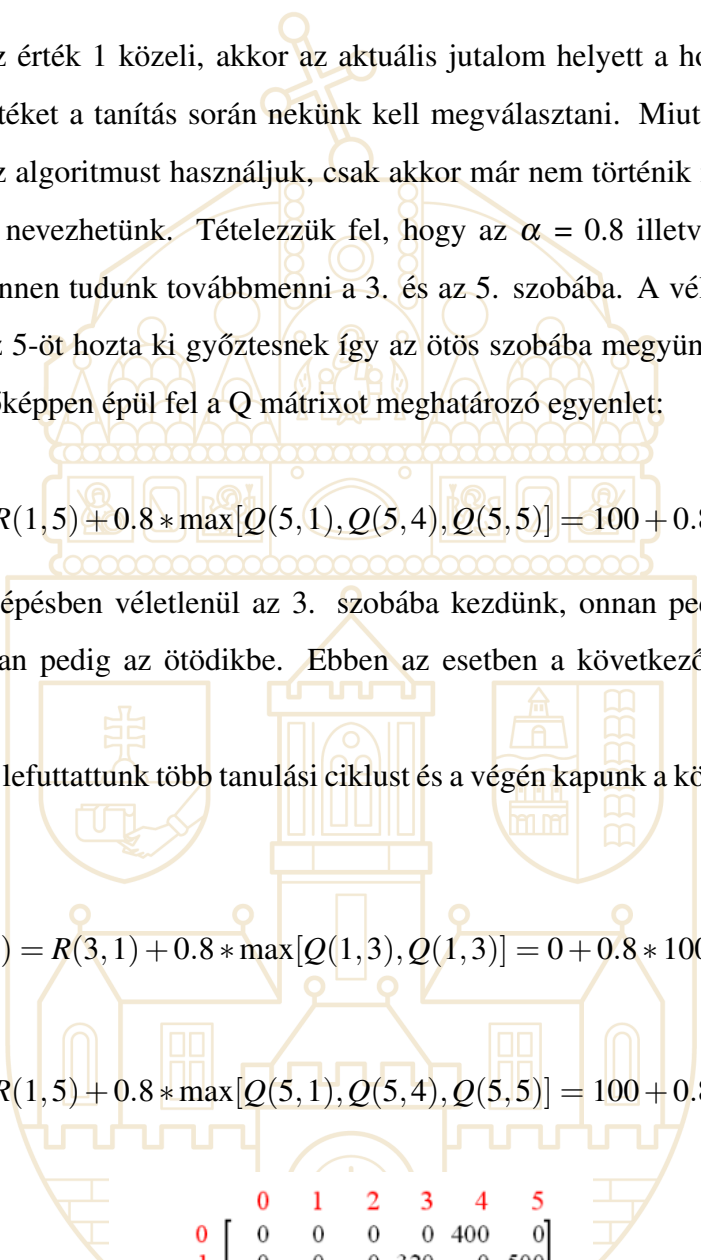
$$Q(1,5) = R(1,5) + 0.8 * \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 * 0 = 100$$

Majd második lépésben véletlenül az 3. szobába kezdünk, onnan pedig átfogunk menni az 1. szobába onnan pedig az ötödikbe. Ebben az esetben a következőképpen néz ki a Q függvényünk:

Majd hasonlóan lefuttattunk több tanulási ciklust és a végén kapunk a következőhöz hasonló Q táblát 2.11.

$$Q(3,1) = R(3,1) + 0.8 * \max[Q(1,3), Q(1,3)] = 0 + 0.8 * 100 = 80$$

$$Q(1,5) = R(1,5) + 0.8 * \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 * 0 = 100$$



	0	1	2	3	4	5
0	0	0	0	0	400	0
1	0	0	0	320	0	500
2	0	0	0	320	0	0
3	0	400	256	0	400	0
4	320	0	0	320	0	500
5	0	400	0	0	400	500

Ábra 2.11: A végleges Q tábla [12]

Ez lett a betanult Q táblája a RL algoritmusunknak. A gyakorlatban ez úgy fog működni, hogy véletlenszerűen kiválasztunk egy kezdő állapotot, legyen az a 0-ás. Itt az algoritmusunk megnézi, hogy milyen lépéseket tud megtenni. Ami most csak a négyes állapot. Ha megvizsgáljuk a Q táblát, látjuk, hogy 0-ás állapotból a 4-es be 400 pontot szerez ha odamegy. Átmegy a négyes állapotba, onnan 3 felé mehet tovább, vissza nullásba, át a hármasba vagy

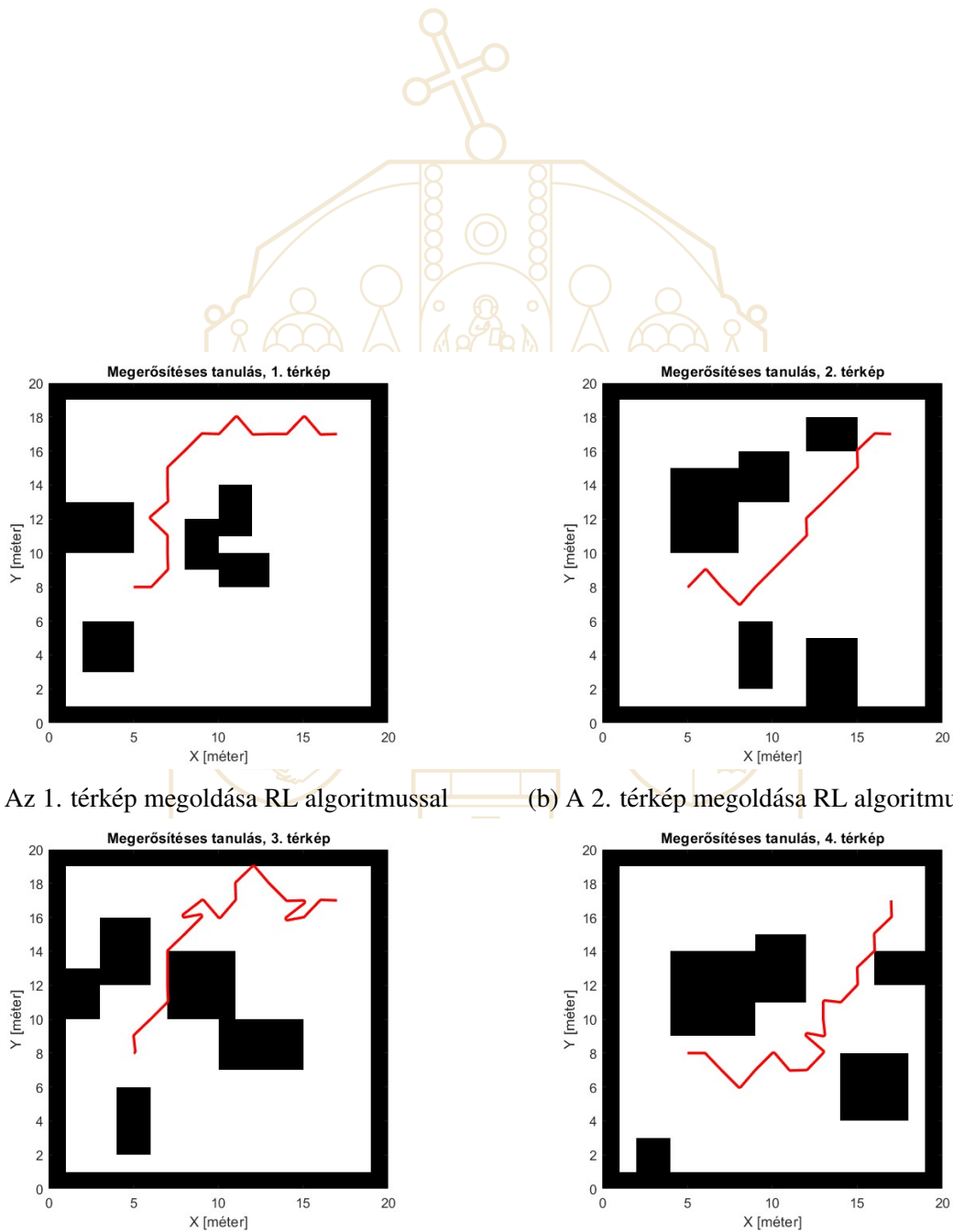


az ötösbe. Az algoritmus megint megvizsgálja, hogy melyiknek a legtöbb az értéke. A 4-es állapotból ha a 0-ába akarnak menni akkor 320 pontot kapna, ha a 3-as állapotba menne tovább akkor is 320 pontot kapna és ha a 4-esből az ötösbe menne akkor 500 pontot kapna. Vagyis az 5-ös (cél) állapotot választja és ezzel összesen 900 pontot szerez. Vagyis az útvonala úgy néz ki, hogy 0-4-5. Ennek számos továbbfejlesztett változata is ismert mint a Mély Q tanulás, a Dupla Q tanulás (Double Q-learning) illetve a párbajozó Q tanulás (Dueling Q-learning).

2.4.3 Szimulációk és eredmények

Szimulációmhoz segítségül hívtam a *Reinforcement Learning Toolbox*-ot, amiben már bele vannak implementálva az tanítási algoritmusok, így már csak az maradt hátra, hogy meghatározzuk a paramétereket. Első körben készítettem egy alap modellt, hogy a fejlesztett RL algoritmusoknak legyen egy pontjuk, amikhez tudjuk hasonlítani. Ez a legalapabb beállítást kapta, megadtuk neki, hogy hol vannak akadályok és, hogy mennyi büntető pontot kap ha azokba belemegy, hogy milyen irányokba tud mozogni, illetve megadtuk neki, hogy a cél állapot 1000 pontot fog érni. A többi értéket alapbeállításon hagytam. Az eredményt az alábbiakban szemléltetem:

Mint látható az algoritmus megtalálja a célját, el fog tudni jutni a kiinduló helyről a cél helyére. Azonban láthatjuk, hogy a 2.12 ábrának a (b), (c), (d) térképén az útvonal nagyon közel kerül az akadályhoz, ami egy UAV esetén nem szerencsés. Ennek korrigálására 2 lehetséges opciónk van. Az egyik, hogy egy lokális algoritmussal ki kell korrigálni (ami majd a saját megoldásnak a témája lesz), a másik pedig, hogy a *reward* értékeket úgy alakítjuk, hogy ne tudjon az akadályokhoz 1 méternél közelebb menni, mert akkor büntető pontot fog szerezni. Ezek közül az 2.12 ábrának (a) útvonaltervezését lehet a legsikeresebbnek nevezni, de abban is vannak tüskék, amiket a fejlesztésekkel a későbbiekben ki lehet majd javítani.



(a) Az 1. térkép megoldása RL algoritmussal

(b) A 2. térkép megoldása RL algoritmussal

(c) A 3. térkép megoldása RL algoritmussal

(d) A 4. térkép megoldása RL algoritmussal

Ábra 2.12: Az 4 térkép megoldása RL *base* algoritmussal

Az UAV 3D-s szimulációja

A 2D-s sikeres tesztek után bemutatásra kerülnek a 3D-s szimulációk is. 3D-s szimulációknál egy nagyobb térképet fogok használni ami 50 méterszer 50 méter alapterületű, illetve az épületek a térben 5 méter magasságig nyúlhatnak fel maximum. Hasonlóan a 2D-s megoldásnál itt is 2 térkép készített használtam, a *occupancyMap3D*-t illetve az *UAV scene* készített. Az *UAV scene* egy UAV szimulációkhoz tartozó környezetet hoz létre, ami segítségével könnyebben tudunk szenzorokat programozni a virtuális UAV-nkre. Itt is 4 különböző véletlenszerűen legenerált térképen fog történni a pályatervezés. Ennél a szimuláció sorozatnál annyi lesz még a különbség a 2D-hez képest, hogy itt a kiinduló koordináta, illetve a cél koordináta minden térképen eltérő lesz. Az alábbiakban láthatók a koordináták.

- 1. térkép kiinduló hely: $x = 47, y = 27, z = 2$ cél: $x = 10, y = 5, z = 2$
- 2. térkép kiinduló hely: $x = 35, y = 10, z = 5$ cél: $x = 5, y = 45, z = 2$
- 3. térkép kiinduló hely: $x = 12, y = 6, z = 3$ cél: $x = 27, y = 35, z = 3$
- 4. térkép kiinduló hely: $x = 3, y = 35, z = 3$ cél: $x = 35, y = 10, z = 4$

3.1 RRT

Az RRT-nek a szimulálásához a *UAV Toolboxot* hívtam segítségül, majd ahogyan 2D-ben is, itt is a *plannerRRT* objektum volt használva. A különbség az 2D-s tervezőhöz képest az volt, hogy itt *StateSpace*-t 3D-ben kellett megadni, amiben meg kellett adni az x, y és z irányokat illetve az a szöveget, hogy merre megy a UAV. Az x, y, z az UAV pozícióját adják meg, szöveget pedig radiánban adjuk meg. Kellott még egy határ mezőt is meghatározni, ami egy 4x2-es mátrix volt



amiben meg kellett határozni, hogy az x , y , z irányokban mennyi lehet ameddig elmehet, illetve a szöget, hogy mekkora fordulatokat tehet meg, szintén radiánban. Amikor a *plannerRRT*-t állítottuk be, itt is megtudtuk adni, hogy mekkora lehet az a maximális kapcsolódási érték ami még megengedett, a mi esetünkben 10 méterre volt állítva. Majd itt is beállítottunk egy olyan funkciót ami azt figyeli, hogy elegendő távolságon belül van-e az UAV-nk, ami itt 1 méterre volt megszabva.

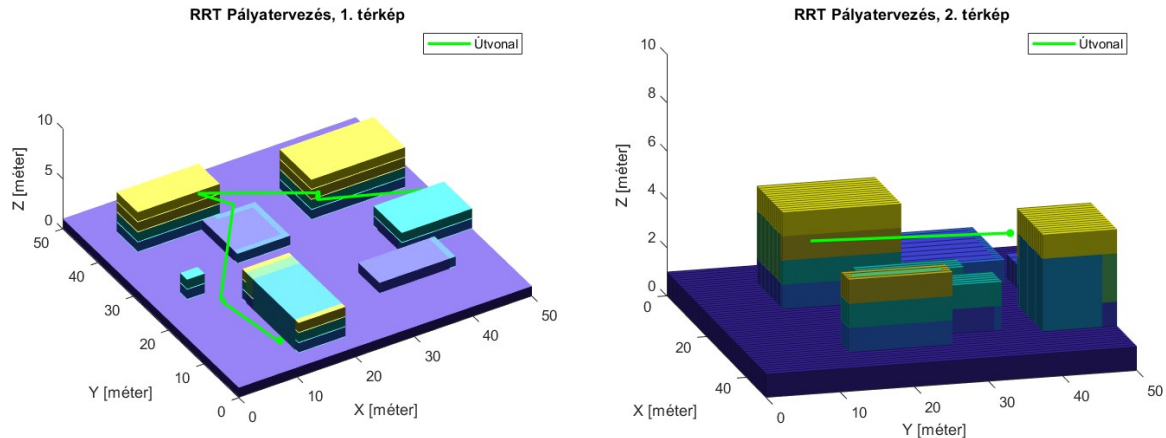
3.1.1 Szimulációk

A 3.1 ábra szimulációi alapján látható, hogy az algoritmus talált minden esetben a kiinduló és a véghelyzet között utat azonban ez egyedül a (b) térkép esetén lett a legrövidebb, a többinél az algoritmus nagyon hosszú, kitérő utakat tett. Összehasonlításképpen a VFH algoritmus pályatervezésével (3.2 és 3.3 ábra) illetve a RL pályatervezésével (3.4 ábra) látható, hogy sokkal hosszabb utat talált mindegyik esetben, emiatt fogom az utolsó fejezetben az RL algoritmust a VFH algoritmussal összekapcsolni, hogy 3D-ben is megfelelően effektív pályatervező algoritmusokat tudjunk létrehozni.

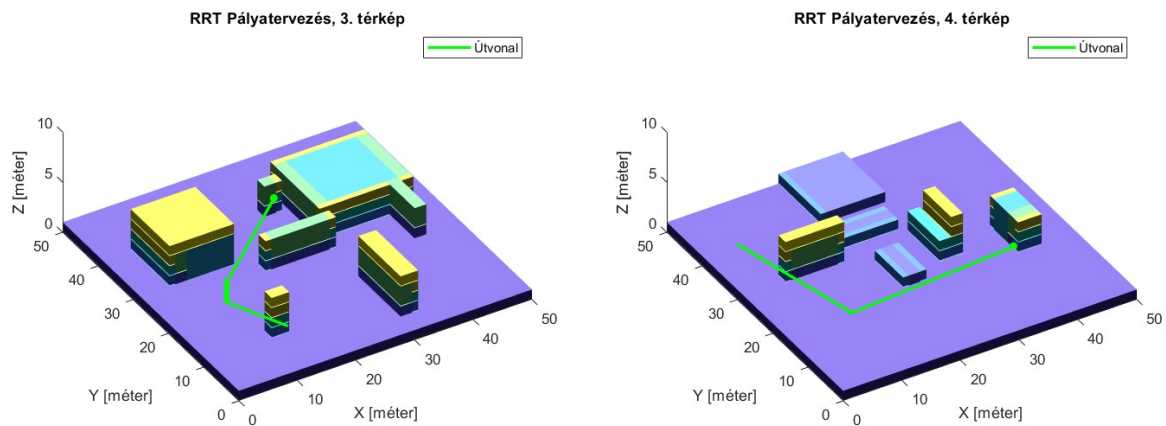
3.2 VFH

A VFH algoritmus 3 dimenziós térre való átültetésével Simon Vanneste és csapata foglalkozott mélyebben az alábbi kutatásban [13]. Ezt *Octomap* típusú adat struktúrán mutatták be, ami egy effektív módja annak, hogy a 3D-s környezetet leképezzük. Kutatásukban 5 különböző lépést prezentáltak, arra, hogy hogyan lehet a VFH-ot átültetni 3D-be.

- *Octomap* felfedezése
- 2D-s elsődleges polár hisztogram leképezése
- Fizikai karakterisztikák
- 2D-s bináris polár hisztogram elkészítése
- Út kiválasztás



(a) Az 1. térkép megoldása a 3D RRT* algoritmus-sal (b) Az 2. térkép megoldása a 3D RRT* algoritmus-sal



(c) Az 3. térkép megoldása a 3D RRT* algoritmus-sal (d) Az 4. térkép megoldása a 3D RRT* algoritmus-sal

Ábra 3.1: Az 4 térkép megoldása a 3D RRT* algoritmussal

Hasonlóan a 2D-s VFH algoritmushoz, itt is az UAV-nk köré egy $w_s * w_s * w_s$ nagyságú dobozt készítünk, illetve megadjuk a robotnak a középpontját (*Vehicle Center Point*, továbbiakban VCP). Ezután a teljes UAV-nkat ez a VCP fogja reprezentálni. Ha a tér az adott dobozunkon belül helyezkedik el, akkor az egy aktív cellája lesz a terünknek amit a következő módon fogunk ábrázolni: $C_{i,j,k}$ ahol az i, j, k a koordinátákat fogja jelölni a C akív régióban. Majd ezek alapján megtudjuk határozni, hogy az adott pontok milyen messze vannak a VCP-től majd ezután eldől a közelség szerint, hogy melyikkel érdemes foglalkozni az algoritmusnak és melyikkel nem. Miután egy ilyen objektumot megtalált a felfedező rész, következik, hogy összeállítódik a 2D-s elsődleges polár hisztogram. A megtalált objektumokat 2 szög segítségével fogjuk betájolni az UAV-nkhez képest. Ez a két szög az Azimuth β_z és magasság



β_e , ahol az Azimuth az x tengelyt jelenti a magasság pedig az y tengelyt. A β_z a következő egyenlettel 3.1 tudjuk kiszámolni amit használtak a 2D-s VFH algoritmusnál is. Amiben is a floor funkció a számot lefelé egész számra kerekíti.

$$\beta_z = \text{floor}\left(\frac{1}{\alpha} \arctan \frac{x_i - x_0}{y_i - y_0}\right) \quad (3.1)$$

Az emelkedést a következő egyenlet alapján tudjuk kiszámolni 3.3, amiben egy kicsit eltér az előzőtől mivel ennek függetlennek kell lennie az Azimuthtól. Ezt úgy tudták megtenni, hogy létrehoztak egy p -t ami az x -től és y -től illetve a VCP-től függött 3.2.

$$p = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} \quad (3.2)$$

$$\beta_e = \text{floor}\left(\frac{1}{\alpha} \arctan \frac{z_i - z_0}{p}\right) \quad (3.3)$$

Miután az értékek megvannak és elkészült a 2D polár hisztogram utána következik, hogy a fizikai karakterisztikákat is hozzárendelik az UAV-hoz. Ez abban nyilvánul meg, hogy nem tud egyből irányt változtatni, emiatt kell, hogy valahogyan kanyarodjon. Emiatt az UAV fordulási pályája az UAV előre irányú sebességétől, a fordulási sebességétől illetve a mászási sebességétől (*climbing speed*) fog függeni. Először is a fordulókör 2 középpontját kell meghatározni, amit a következő két egyenlettel tudunk kiszámolni (3.4 és 3.5) ahol a θ az irányszög.

$$\Delta x_r = r_r * \sin \theta, \Delta y_r = r_r * \cos \theta \quad (3.4)$$

$$\Delta x_l = -r_l * \sin \theta, \Delta y_l = -r_l * \cos \theta \quad (3.5)$$

Majd ezután az algoritmusnak meg kell vizsgálni minden aktív cellát, hogy beleesik-e a fordulási sugárba. Ezt a következő egyenletekkel (3.6, 3.7 és 3.8) tudjuk megtenni, amiben is kiszámoljuk a fordulási kör középpontjának a távolságát ($d_{l,r}$) az akadályunktól. Majd ezután az algoritmus összehasonlítja, hogy a biztonságos távolságba beleesik-e. Az alábbi egyenletben (3.6 és 3.7) a $\Delta x(i)$ kiszámolja az x távolságot az akadály és a VCP között.

$$d_r = \sqrt{(\Delta x_r - \Delta x(i))^2 + (\Delta y_r - \Delta y(j))^2} \quad (3.6)$$



$$d_l = \sqrt{(\Delta x_l - \Delta x(i))^2 + (\Delta y_l - \Delta y(j))^2} \quad (3.7)$$

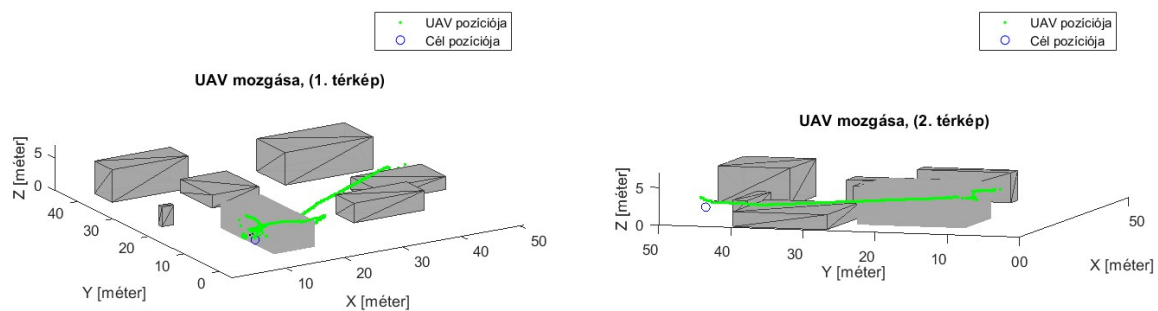
$$d_r < (r_r + r_{r+s+v}), d_l < (r_l + r_{r+s+v}) \quad (3.8)$$

amiben r_{r+s+v} a biztonságos távolság és $r_{l,r}$ a fordulókör átmérője. Ezután a negyedik lépésben elkészítésre kerül a 2D-s bináris polár hisztogram. Ez úgy fog történni, hogy a létrehozott 2D-s elsődleges hisztogramot tovább redukálják olyan formában, hogy meghatároznak egy T_{magas} és egy $T_{alacsony}$ küszögértéket. Ha az érték a T_{magas} fölé esik akkor az érték 1-et fog felvenni, ha az érték a $T_{alacsony}$ alá esik akkor 0-ás értéket vesz fel. Abban az esetben ha az érték a két küszöbérték közé esik, akkor a szomszédos mezők fogják meghatározni az értékét. Az utolsó fázisban pedig megtörténik az út kiválasztása, ami pedig úgy fog történni, hogy a legkisebb súlyú utat fogja kiválasztani az algoritmus. Ahhoz, hogy megvizsgálja, hogy melyik utak a választhatók, az algoritmus nyitásokat fog keresni a hisztogramon és akkor fogja átkelhetőnek megjelölni, ha minden eleme 0 lesz az adott nyitásnak. Ennek köszönhetően csak megfelelő nagyságú nyitásokat fog átkelhetőnek megjelölni az algoritmus [13].

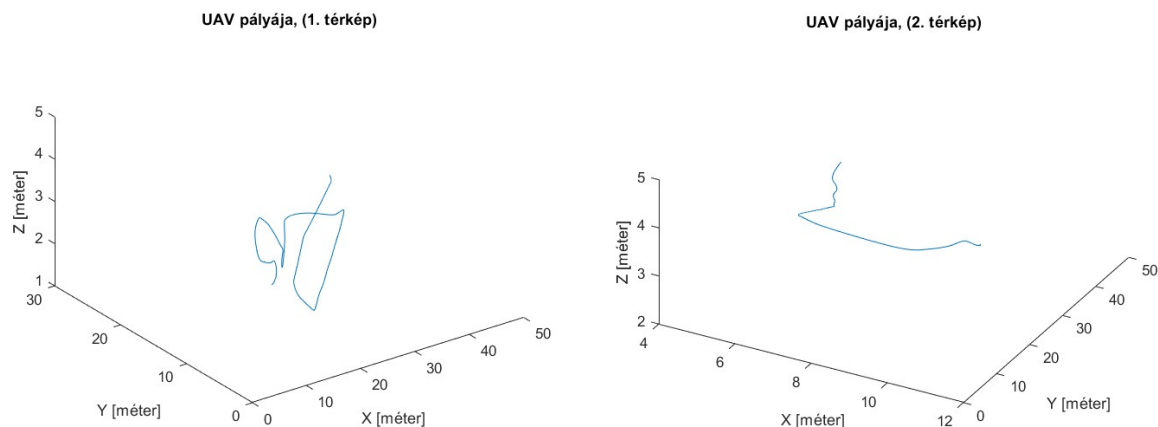
3.2.1 Szimulációk

Szimulációmhoz a MATLAB *UAV Toolbox*-át hívtam segítségül. Ebben a Toolboxban az UAV szimulációkhoz szükséges funkciók találhatók. Ahhoz, hogy ezt könnyen és egyszerűen tudjuk implementálni, létre kellett hozni egy *uavScenariot*. Ez igazából nem más mint egy környezet, amiben könnyedén tudunk akadályokat elhelyezni, illetve a drónt tudjuk több rotorosként vagy merevszárnyúként is szimulálni. A MATLAB nem igazán szereti azt, hogy az egyik környezetből a másikba átkonvertáljuk a környezetünket, emiatt a már létrehozott gridworld környezetünket meg kellett csinálni ebben az *uavScenario*-ban is. Mivel egy egyedül csak sima téglatestek voltak az épületeink, így viszonylag egyszerű volt a feladatunk. A legnagyobb indok, hogy ezt a környezetet célszerű használni, hogy ebben vannak UAV szenzorok amiket az *uavSensor*-al tudunk megcsinálni. Számunkra a legfontosabb szenzor a LiDAR szenzor lesz. Ehhez először egy *uavLidarPointCloudGenerator* objektumot kell létrehoznunk ami tudja érzékelni a környezetben lévő akadályokat. Majd ezt az objektumot tudjuk ráhelyezni az UAV-re. Az *uavSensor* segítségével még azt is megtudjuk neki adni, hogy hány fokban és milyen

pozícióba helyezzük el az adott szenzorunkat. Ezek kellene ahhoz, hogy a *controllerVFH3D* objektumunk tudja számolni, hogy merre van akadálymentes út a célig. Ez az objektum az előzőekben bemutatott 3DVFH+ algoritmust használja [13]. Ahhoz, hogy ezt az objektumot működésre hozzuk, meg kell neki adnunk a 4 bementi elemet, az UAV pillanatnyi pozícióját, az UAV orientációját, a Lidar által generált $N \times 3$ -as mátrixot, illetve a célnak a koordinátáit, majd ez visszaad nekünk egy irányt, egy fordulást illetve egy státuszt, hogy sikerült-e akadálymentes utat találni a vagy sem. Az alábbi két ábrában (3.2 és 3.3) látható, hogy ez az algoritmus hogyan talált magának utat az általunk meghatározott 4 térképen. Az első két-két ábrákon az



(a) Az 1. térkép megoldása a 3D VFH algoritmussal (b) Az 2. térkép megoldása a 3D VFH algoritmussal



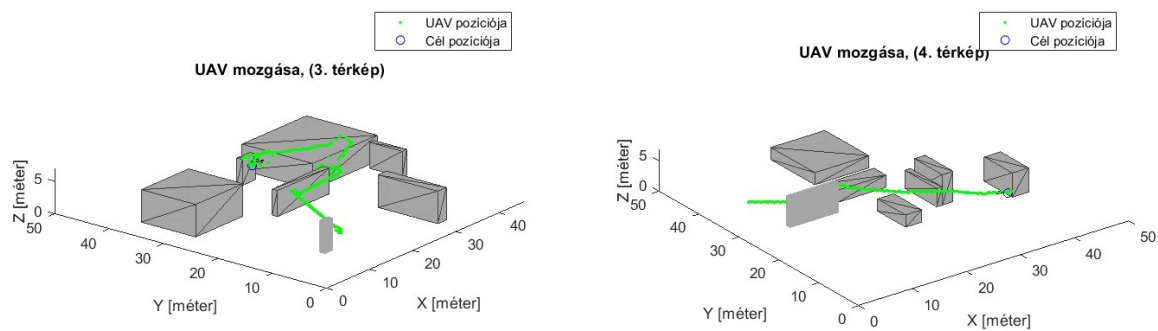
(c) Az 1. térkép útja

(d) Az 2. térkép útja

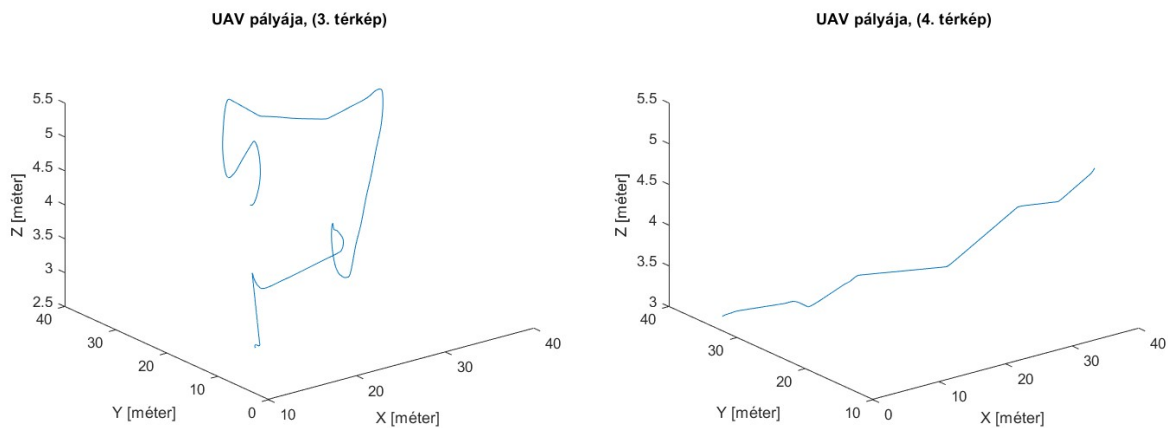
Ábra 3.2: Az első 2 térkép megoldása 3D VFH algoritmussal

látható, hogy az akadályokkal teli környezetben hogyan néz ki az UAV útja, míg a második két-két képen pedig az UAV útja látható az objektumok nélkül. Ebben az esetben az UAV-nak nem kellett feltétlenül pontosan a cél koordinátákra megérkeznie, elég volt, ha 1 méterre

megközelíti minden irányból mert utána már például egy leszálló algoritmus betudja fejezni a műveletet. Ennek az algoritmusnak sikerült megtalálnia mind a 4 esetben a célhoz vezető utat ahogy azonban az 3.2 ábra (a) térképen látható nem a legrövidebb utat tette meg az UAV-nk, így megállapítom szerint itt is szükséges egy globális tervező algoritmus, hogy az meghatározzon egy ideálisabb pályát és ennek az algoritmusnak csak korrigálnia kelljen, ha valami az útjába kerül.



(a) Az 3. térkép megoldása a 3D VFH algoritmussal (b) Az 4. térkép megoldása a 3D VFH algoritmussal



(c) Az 3. térkép útja

(d) Az 3. térkép útja

Ábra 3.3: Az második 2 térkép megoldása 3D VFH algoritmussal

3.3 Reinforcement Learning

A MATLAB-nak nincsen beépített megoldása arra, ha RL segítségével szeretnénk pályatervező algoritmust írni így ezt nulláról kellett felépíteni. Arra, hogy ezt felépítsem a githubon található



egyik megoldást alkalmaztam és írtam át [14]. Ebben a szerző Pythonban építi fel a Q tanulás algoritmust egy 2D-s, 9x9-es környezetben. Emiatt úgy kellett módosítanom, hogy az algoritmus 3D-ben működjön és 50x50x5-ös méretű legyen. Összehasonlításképpen, az általam felvázolt környezetben több mint 154-szer annyi állapot szerepel. A kezdeti modell ebben az esetben az volt, hogy azok a cellák, amik nem tartalmaznak akadályt vagy cél állapotot, azoknak a kezdő értéke a Q táblában 0 volt. Viszont mivel ezek a térkép viszonylag nagy 50x50 méter hosszú így az kezdeti megoldás nagyon számításigényes lenne. Emiatt bevezettem, hogy a kezdő értékek a Q táblában nem 0, hanem az euklideszi távolsága az adott cellának a cél cellától.

3.3.1 Programozása

Mivel a térképkészítéshez már létre lett hozva egy 50x50x5-ös mátrix, emiatt azt a mátrixot fogom tudni használni Q táblának is egy kis módosítással. Mivel az elkészített térkép mátrixban csak 1-esek és 0-ák találhatók így ezeket az értékeket módosítanom kellett, hogy nekem megfelelőek legyen. Elsőnek beállítottam, hogy a cél mezőjének az értéke 100-at érjen, majd beállítottam, hogy minden egyes akadály pedig -100 pontot érjen. Létrehoztam egy funkciót, ami minden mezőre kiszámolja az euklideszi távolság reciprok értékét. Mivel az euklideszi távolságnál, minél távolabb vagy a céltól annál nagyobb az érték emiatt kellett a reciprokát vennem, hogy azok az értékek legyenek a nagyobbak amik közelebb vannak a célhoz, mivel a Q tanulás alapvetően egy maximum értékre törekvő algoritmus. Az RL környezetnek létrehoztam egy osztályt, aminek a lényege, hogy a környezetet legenerálja, illetve frissítse minden lefutott iteráció után. Ebben az osztályban 2 lehetséges funkció van környezet létrehozásra, az egyik, hogy egy már meglévő *Grid World*-ot teszünk bele és az alapján hozza létre a környezetünket, a másik pedig, hogy létrehozunk egy valamilyen méretű *Grid World* környezetet és az akadályokat külön adjuk hozzá, megadva a akadályok x, y és z tengelyen lévő kezdő és végpontjait. Az ügynöknek egy külön osztályt hoztam létre amiben a koordinátái vannak meghatározva illetve létrehoztam egy osztály az RL különböző metódusainak mint a lépés és a tanulás. Miután létrehoztuk a kötelező 3 értékünket elkezdődhet a tanulás folyamata. Elsőként a lépés utána pedig a tanulás funkció fog életbe lépni. A lépés fázisban elsőként eldöntődik, hogy véletlenszerűen fog lépni vagy a legmagasabb érték irányába mozdul el. Mindkét esetben egy úgynevezett *MatrixExtraction* funkció fog létrejönni, aminek a lényege, hogy kiszedje azt a 3x3-at mátrixot ami körülveszi az ügynökünket. Erre



azért van szükség mert az ügynökünk léphet előre, hátra, jobbra, balra, fel és le illetve átlósan is. Miután ez megvan ebben a mátrixban megkeresi, hogy melyik a legnagyobb érték és arra indul tovább vagy véletlenszerűen kiválaszt egy irányt, hogy merre indul tovább. Miután ez megvan megnézi, mire lépett rá. Itt 3 lehetőség van:

1. A célra lépett rá, ebben az esetben sikeres volt a pályatervezése és ezután véget fog érni az iteráció - Ilyenkor 100 pontot kap
2. Egy akadályra lépett rá, ebben az esetben is véget ér az iteráció, de ilyenkor nem volt sikere - Ilyenkor -100 pontot kap
3. Olyan helyre lép, ahol sem akadály, sem cél nincs, ebben az esetben megy tovább az iteráció - Ilyenkor a -0.2 értéket fog kapni

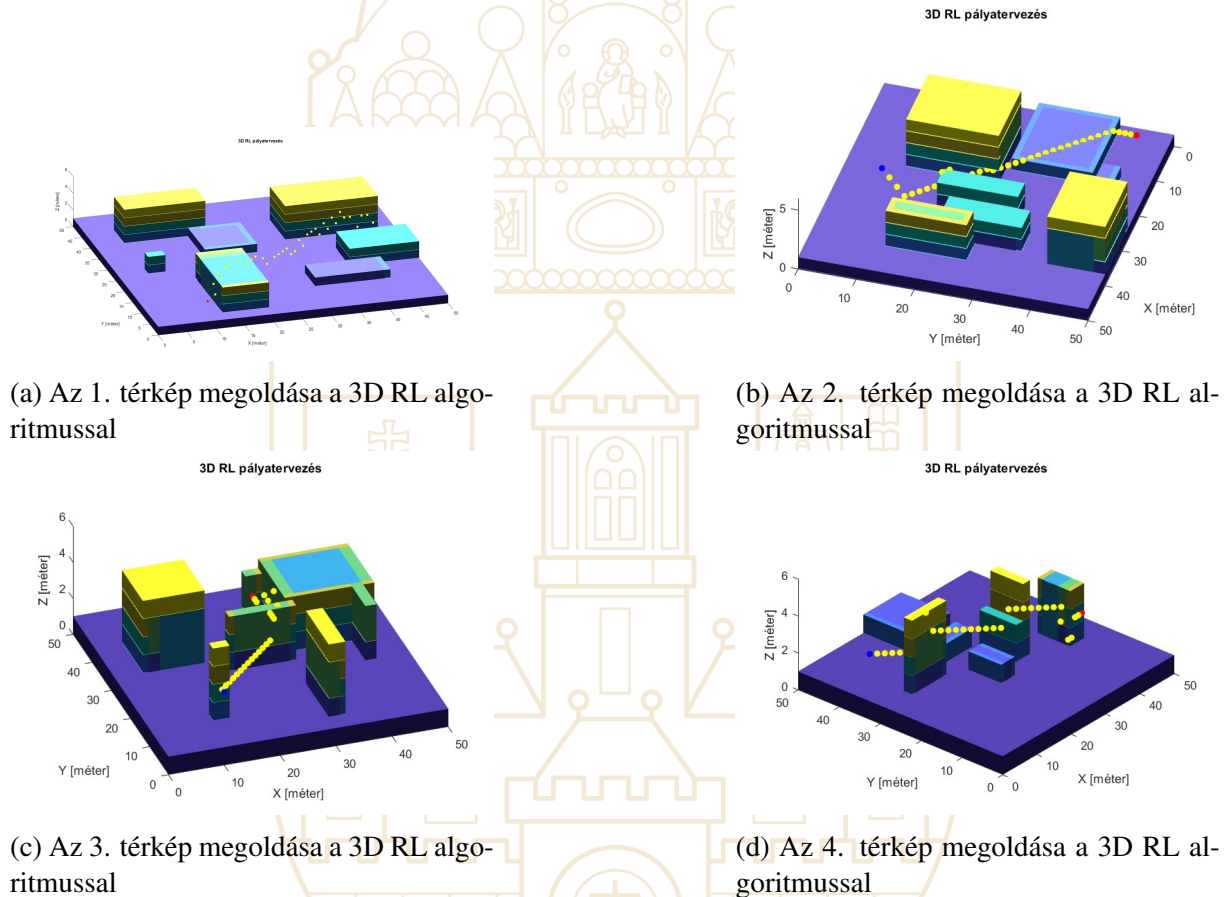
Az utolsó pont alól kivétel volt az első térkép mivel ő bekerült egy olyan lokális szélsőértékbe ahonnan nem tudott kijutni, emiatt finomítottam az értéket -2-re. Ezután történik meg a 2.5 egyenlet alapján a tanulás lépés, ahol is az adott mezőnek a pontszáma újraszámolódik attól függően, hogy hogyan értékeli a program. Miután a cellák új értékeinek a kiszámítása megtörtént, az ügynökünknek frissítjük a pozícióját, ha elérte a célt, akkor az célhoz vezető koordinátákat elmenti és ha már voltak előtte sikeres iterációk, akkor összehasonlítja annak a hosszával és a rövidebbet megtartja. Ezek után újra indul az iteráció, az ügynökünk visszakerül az kiinduló pozícióba, viszont itt már van frissített érték a Q táblában.

3.3.2 Szimulációk

A térképek tanítása külön külön körülbelül 20 -30 percet vettek igénybe. Voltak nehézségek is amikor az első térkép beszorult egy lokális maximumba és onnan nem tudott kijutni emiatt kellett a jutalom értékeket illetve a tanulási sebességet módosítanom. Jelen pillanatban az euklideszi távolságok alapján lett meghatározva a kezdő Q tábla, ami a szimulációinkhoz és tanításhoz tökéletesen megfelelt. Azonban ahogy láthatjuk az alábbi ábrán (3.4) nem lettek olyanok az utak amiket mi gondolnánk vagy tudunk, hogy a legrövidebbek lennének. Például a (d) ábrán látszik, hogy az algoritmus 2 lépés után lemegy egyet és ott marad egyet ami 2 felesleges lépés mivel utána visszamegy a kiinduló szintre. Ezt a jövőben lehet korrigálni olyan módon, hogy tovább futtatjuk a tanítást, nem állunk meg 50.000 iterációnál hanem elmegyünk

akár 100.000-ig is és megnézzük, hogy akkor hogyan változna az út. Ami másik lehetőség pedig, hogy más algoritmus alapján adunk kezdőértékeket a Q táblának.

Az alábbi térképeket látható, hogy hogyan oldotta meg a pályatervezést a 3D-s környezetben az RL algoritmus (Ábra 3.4).

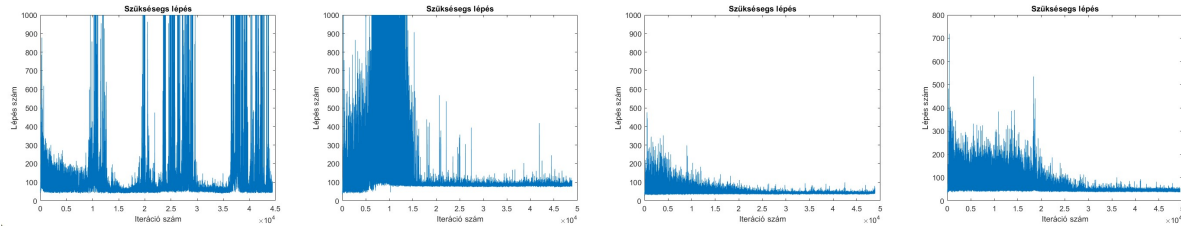


Ábra 3.4: Az 4 térkép megoldása a 3D RL algoritmussal

Mivel a engedélyeztük az átló mozgást így a térképeken előfordulhat az, hogy ha összekötnénk a pontokat akkor belemenni az épületben. Ez azért lehet mivel az RL csak egész számokban tud tervezni, emiatt lesz szükség hozzá egy másik algoritmushoz, ami jelen esetben a VFH lesz, amivel kitudja kerülni ezeket az akadályokat.

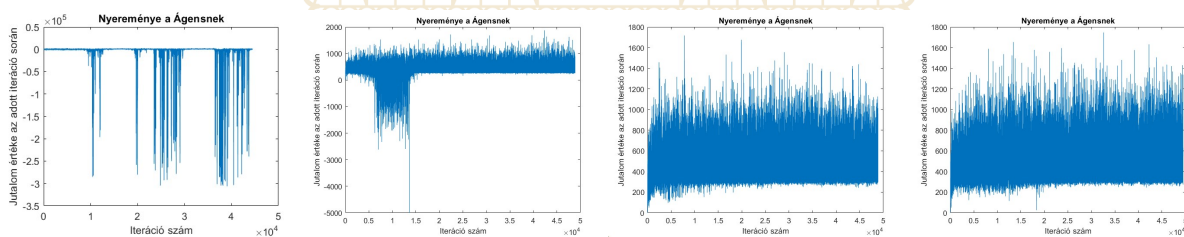
A következő ábrákon a célt elérő lépésszám látható az éppen adott iteráció közben.

Az előzőekben bemutatott 2 ábrán (Ábra 3.5 és Ábra 3.6) az alábbiakat érdemes célszerű megnézni. Jól lehet látni, ahogy haladunk előre az időben és nő az iterációs szám úgy csökken egyre jobban a kiinduló helyzetből a célhelyzetbe való eljutáshoz való szükséges lépéseknek a száma. A másik dolog ami megfigyelhető, hogy a megszerzett jutalom minimális értéke



(a) Az 1. iteránciónkénti lépésszáma (b) Az 2. iteránciónkénti lépésszáma (c) Az 3. iteránciónkénti lépésszáma (d) Az 4. iteránciónkénti lépésszáma

Ábra 3.5: Az 4 térkép lépéseinek száma iterációnként



(a) Az 1. iteránciónkénti jutalma (b) Az 2. iteránciónkénti jutalma (c) Az 3. iteránciónkénti jutalma (d) Az 4. iteránciónkénti jutalma

Ábra 3.6: Az 4 térkép ügynök jutalmainak értéke iterációnként

iterációnként nő, feltehetően, ha tovább futtattuk volna őket, mint 50000 iteráció, akkor ez még látványosabb lenne, de ha megvizsgáljuk, akkor észrevehető, hogy a kezdeti pontcsökkenés után elkezd stabilan növekedni az értéke. A harmadik dolog amit észre lehet venni az a megszerzett pontok és a lépések számának a kapcsolata, jól látható, főleg a 3.5 és a 3.6 (b) ábráján, hogy amikor a lépésszám kiugró az 5000. és 15000. iteráció között akkor az iterációként szerzett jutalma az ágensnek is drasztikusan csökken. Ez a jelenség észrevehető az (a) térkép tanítása közben is. Az (a) térkép olyan érdekességeket is mutatott, hogy amikor már azt hinnénk, hogy elkezd csökkenni a lépésszám olyan 30000 és 35000 iteráció között, akkor valami történik és olyan 37000-től a végéig megint kileng. Ezt valószínűleg ha tovább lett volna tanítva a modell kiszéplült volt. Viszont mivel nem az utolsó ismert iterációt használja az algoritmus, hanem létre lett hozva egy *legrövidebb Út* változó. Ez a változó a tartalmát minden sikeres iteráció után összeveti az iterációban lévő lépések számával. Ha az új lépések száma kevesebb mint ami a változóban van, akkor megváltoztatja a változó értékét azokra amik az iterációban vannak. Abban az esetben ha nem kevesebb akkor a változóban tárolt érték nem változik.

Saját Megoldások

4.1 Deep Learning + VFH

Amint eddig láthattunk, hogy léteznek globális, mint az RRT vagy Deep Learning algoritmusok, és vannak lokális algoritmusok is mint a VFH algoritmus. Ez a kettő működhet külön-külön is, azonban egy szállító drónhoz vagy egy alacsonyan repülő drónhoz a kettő együttes használatára van szükség. A globális algoritmusok megfelelőek lennének önmagukban akkor, ha a térben nem lennének más tárgyak is. Ha a szállító drón példájából akarunk kiindulni akkor egy Deep Learning algoritmussal megtudjuk neki taníttatni a városnak a térképét, ami alapján egyszerűen tudna eljutni A-ból B-be, azonban ha valamilyen akadály bekerül elé ami eddig nem volt ott akkor az nem feltétlenül fogja kikerülni. Az ilyen helyzetekben tudnak segíteni a lokális algoritmusok, amik nem ismerik a teljes utat, csak körülöttük lévő területet, és ezek alapján tudnak támpontot adni, hogy az eddig ismeretlen akadályokat, hogyan kéne kikerülni.

4.1.1 Folyamat

Ezt az algoritmus párost is ugyanazon a 4 térképen teszteltem mint külön külön őket, azonban ebben az esetben hozzáadtunk 1-1 elemet mindegyik térképhez, hogy legyen benne valamilyen ismeretlen elem, amit előre nem tudott az algoritmus. Első lépésként az eredeti térképekre lefuttattuk a Deep Learning algoritmusunkat. Ez az algoritmus visszaadott egy n sorból és 3 oszlopból álló mátrixot. Ebben a mátrixban az oszlopok az x, y, z koordinátákat jelölték, azonban mivel 2D-ben dolgozunk, így a 3. oszlopot elhanyagoltam. Ennek a mátrixnak a pontjai lettek az útpontjai (waypoint). Ahhoz, hogy a UAV tudja követni ezeket a pontokat segítségül hívtam a MATLAB beépített *uavWaypointFollower* funkcióját. Ez a funkció segítségével tud

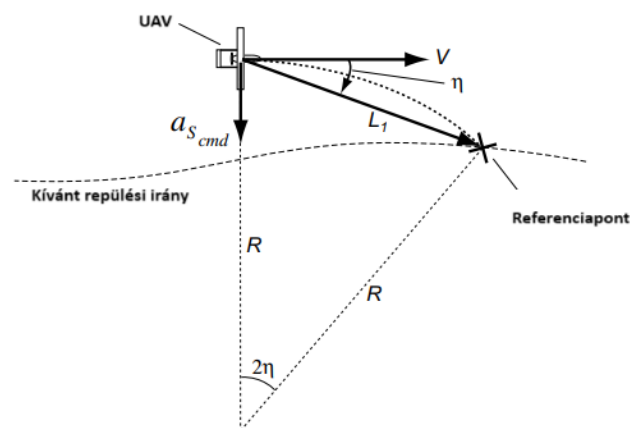
mozogni az UAV a megadott pontok között de ehhez szükséges neki egy előrelátási távolság is (*lookahead distance*) ami alapján tudja kiszámítani a következő koordinátát *lookahead point*.

Ennek a funkciónak az alapját készítő kutatást Sanghyuk Park és csapata végezte [15] amiben is meghatároznak egy referencia pontot a kívánt útvonalon és generálnak egy oldalirányú gyorsulási parancsot a referencia pontot felhasználva. A referenciapont L_1 távolságra van az UAV-tól. Illetve az oldalirányú gyorsulási parancs a következőképpen van meghatározva, amit a következő ábrán vizuálisan is lehet látni, hogy mit jelent (4.1):

$$a_{s_{cmd}} = 2 \frac{V^2}{L_1} \sin \eta \quad (4.1)$$

amiben is a jelölések a következők:

- V az UAV sebessége
- L_1 A referenciapontot és az UAV-t összekötő vonal
- η A V és az L_1 közötti szög (órajárásnak megfelelő a pozitív)
- $a_{s_{cmd}}$ Az oldalirányú gyorsulás
- R A kör sugara



Ábra 4.1: Az irányítási logika diagramja [15]

Ebben a kutatásban 3 különböző esetet is megvizsgáltak:

- Első eset: Egyenes pályakövetés

- Második eset: Nem egyenes pályakövetés
- Harmadik eset: Körpályakövetés

Mivel szimulációm során útpontból útpontra egyenes utak vittek így az általuk bemutatott első esetet fogom csak részletezni. A következő ábrán (4.2) bemutatja az egyes jelölések mire vonatkoznak. L_1 az UAV távolsága a referenciaponttól, a d az a pálya hiba (*cross-track error*), a V az UAV sebessége, illetve még feltételezzük azt is, hogy a η kis mennyiségű a következő két egyenletet kapjuk (4.2, 4.3)

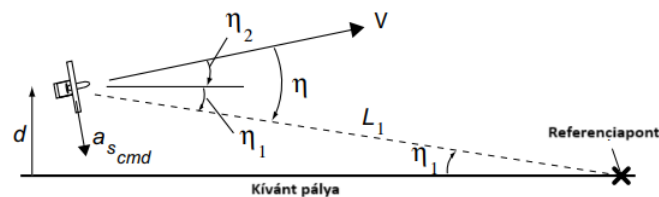
$$\sin \eta \approx \eta = \eta_1 + \eta_2 \quad (4.2)$$

$$\eta_1 \approx \frac{d}{L_1}, \eta_2 \approx \frac{\dot{d}}{V} \quad (4.3)$$

Vagyis ha összekapcsoljuk a 4.1, a 4.2 és a 4.3 egyenleteket, a következő formulához jutunk:

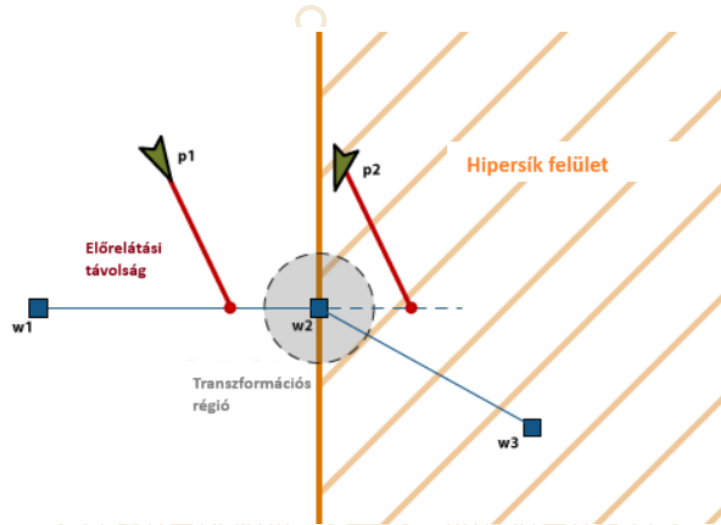
$$a_{s_{cmd}} = 2 \frac{V^2}{L_1} \sin \eta \approx 2 \frac{V}{L_1} \left(\dot{d} + \frac{V}{L_1} d \right)$$

Ha a pálya hibájára egy PD szabályozót kapunk amikor linearizáljuk a nem lineáris irányítási logikát.



Ábra 4.2: Lineáris modell egyenes pályakövetés esetére [15]

A következő módon van ez implementálva MATLABban. A követő megvizsgálja, hogy az UAV közel van-e a következő transzformációs régióhoz. Ezt vizsgálja az előrelátási távolsággal, ha benne van akkor transzformálódik a következő pont irányába, viszont kívül esik ezen a területen, akkor egy hípersík van húzva a következő ponthoz és ha az UAV ezen a síkon belül van akkor az útvonalkövető a következő pontot veszi már irányának [16]. A következő ábra (4.3) demonstrálja, hogyan is néz ki ez



Ábra 4.3: Pályakövetés megvalósulása [16]

2 eset merül fel ezek után, vagy a transzformációs régióban vált követendő ponton vagy pedig a hipersíkon. A 4.4 bemutatja, hogy melyik egyenlet teljesülésekor fog a transzformációs régióban követendő pontot váltani, illetve a 4.5 egyenlet megmutatja, hogy mikor fog a hipersíkon követendő pontot váltani.

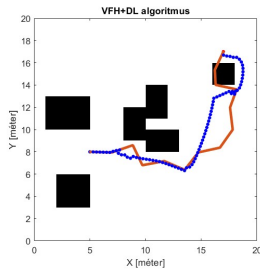
$$(p1 - w2)^T (w2 - w1) < 0 \quad (4.4)$$

$$(p2 - w2)^T (w2 - w1) \geq 0 \quad (4.5)$$

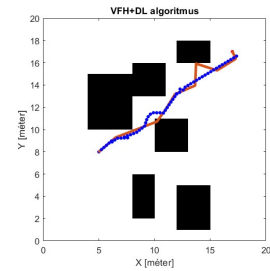
ahol a $p1$ és $p2$ az UAV különböző helyzetei és a $w1$ és $w2$ pedig az egymást követő pályapontok. Majd miután következő pont kialakul, ez bekerül a VFH algoritmus és az meghatározza, hogy melyik irányba kell fordulnia a UAV-nek, ami 2 dimenziós környezetben egy pontként van bemutatva. A következő részben bemutatásra kerül, hogyan alakultak a tesztek a különböző pályákon.

4.1.2 Szimulációk

Mint látható a 4.4 ábrán, mindegyik térképre elhelyezésre került egy-egy új eddig a környezet számára ismeretlen akadály. Az elhelyezése manuálisan történt figyelve arra, hogy a pálya valamilyen szinten belelőgjon a DL pályatervező algoritmus útjába, hogy demonstrálni lehessen, hogyan is oldja meg a kikerülést az algoritmus.



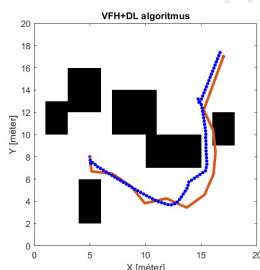
(a) Az 1. térkép megoldása
DL+VFH algoritmussal



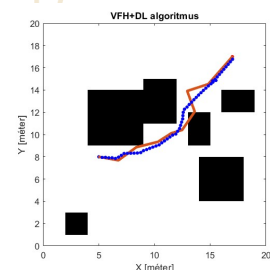
(b) A 2. térkép megoldása
DL+VFH algoritmussal

Ábra 4.4: Az 1. és 2. térkép megoldása DL+VFH algoritmussal

A 4.4 és a 4.5 ábrán a narancssárga út volt az amit előre eltervezett a DL algoritmus a kék pedig amin végül az UAV haladt. Látható, hogy a 4.4 ábra (b), illetve 4.5 ábra (a), (b) térképen az UAV szinte teljesen követi az előre megírt útvonalat, míg az (a) térképnél tett egy elég nagy levágást, ezzel jelentősen lerövidítette az útját. Az, hogy 4.4 ábrán az (a) térképen az UAV miért jobbról kerüli az újonnan betett akadályt azért van mert a DL által meghatározott állomás közvetlenül a jobb alsó sarkában helyezkedik el, így amikor a VFH kalkulálta közelebbinek találta, arról az oldalról kerülnie. Ha a másik oldalra esett volna az állomás akkor a bal oldalról kerülte volna. Mindegyik útvonalnál elmondható, hogy azokat a tüskéket amiket a DL algoritmus beletett az útvonalába azokat kiegyenesítette a waypointFollower így ennek köszönhetően egy sokkal simább utat tudott követni az UAV.



(a) Az 3. térkép megoldása
DL+VFH algoritmussal



(b) A 4. térkép megoldása
DL+VFH algoritmussal

Ábra 4.5: Az 3. és 4. térkép megoldása DL+VFH algoritmussal

Összességében sikeres volt a művelet, ugyanis az UAV el tud jutni a kiinduló helyzetből a véghelyzetbe úgy is, hogy olyan akadály/akadályok kerülnek be a térképre amik az eredeti tervezésben nem voltak ott.



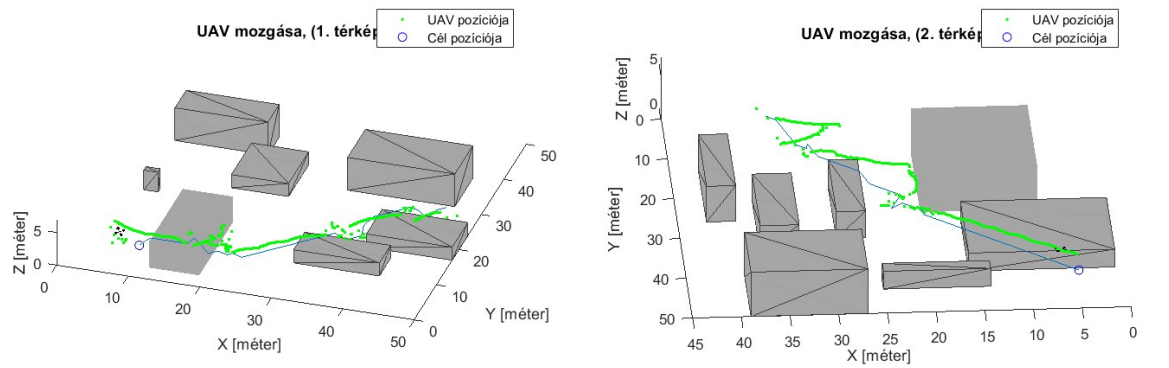
4.2 Reinforcement Learning + VFH

Az előzőtől eltérően nem egy 2D-s megvalósításról, hanem egy 3D-s megvalósításról van szó, szóval kijelenthetjük, hogy ez a megoldás fogja legjobban tükrözni a való életet. Ebben az esetben az előző fejezetekben (3.3) bemutatott RL algoritmust fogom használni illetve lokális hibák kiküszöbölése miatt hozzákapcsolom a VFH algoritmust is. Jelen esetben újra fogom tanítani az algoritmust, viszont az előzőektől különbözően csak 35000-szor fogjuk tanítani, mert ahogy az ábrán (3.5) is látszik, ez az a tartomány ahonnan már nem sokat fog változni a lépésszám. A koncepció hasonló mint az előbbi részben 4.1 történetknél. A különbség annyi, hogy itt nem adunk hozzá plusz egy épületet, mivel látható, hogy azokon a helyeken ahol átlósan lépett az algoritmus, de akadály van előtte ott szüksége van a VFH algoritmusra, hogy ne menjen neki. Ezen kívül a teljes procedúra ugyanaz lesz mint a DL és VFH párosításánál. Meghatározzuk a DL algoritmussal a pályapontokat amiket a drónnak meg kell tennie majd az *uavWaypointFollower*-be megadjuk neki ezeket a pontokat, hogy kövesse és amikor az akadály kerül elé, akkor a VFH algoritmus akcióba lép, kikerüli az akadályt és visszatér a kijelölt útvonalra.

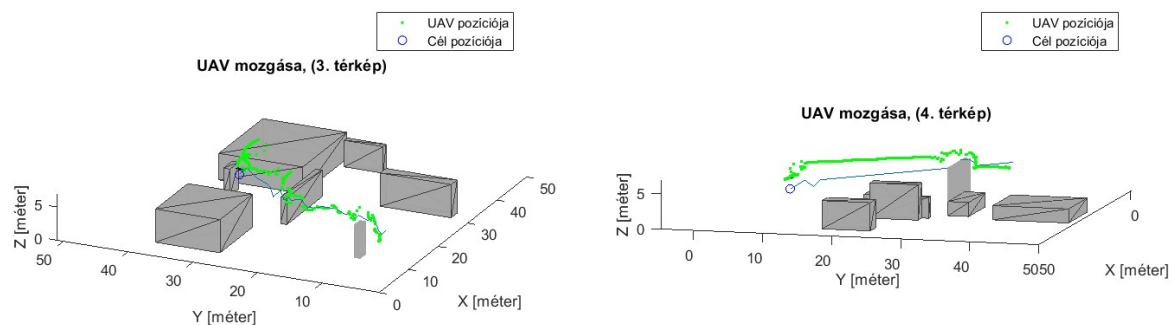
4.2.1 Szimuláció

A szimulációkban kiterjesztettem a képzeletbeli landoló algoritmusnak a mértékét 2.5 méterre így amikor az UAV 2.5 méteren belülre ért akkor sikernek értékelte az algoritmus a műveletet. Látható, hogy a 4.6 ábrának az (a) térképe követi a legjobban és legpontosabban az utat és ott akcióba is lépett a VFH algoritmus mert az adott akadályon nem tudott volna felmenni mert túl közel került. A (d) alábrán is látható, hogy az UAV jól követi az utat, csak mivel ott egyből egy akadály került elé így végül az adott úthoz képest kicsivel feljebb tette meg az utat és onnan süllyedt rá az akadályra. A (b) ábrán az UAV nagyon kitért, ami valószínűleg az *uavWaypointFollower* nem megfelelően való konfigurálása okozott. Ez volt az egyetlen olyan térkép is amelyben az RRT algoritmus rövidebb utat tett meg mint a RL+VFH kombinációja. A (c) ábrán az UAV jól követi pályát egészen addig amíg az RL algoritmus nem tesz egy hirtelen süllyedést az akadály után. Ezt a UAV nem tudja megtenni mivel a fizikája nem engedi azt így ugyanazon a magasságon repül tovább mint előtte és amikor a célhoz közeledik akkor kezdene el süllyedni, csak a célhoz közel vannak az akadályok, emiatt látható, hogy nem egyből kezd

el süllyedni, hanem először felmegy az épület tetejére és onnan indul el körkörös lefelé. De ezen a pályán is a kombinált algoritmus rövidebb utat tett meg mintha az RRT-t használta volna, vagy ha csak a VFH-ra támaszkodott volna.



(a) Az 1. térkép megoldása a 3D RLVFH algorit-mussal (b) Az 2. térkép megoldása a 3D RLVFH algorit-mussal



(c) Az 3. térkép megoldása a 3D RLVFH algorit-mussal (d) Az 4. térkép megoldása a 3D RLVFH algorit-mussal

Ábra 4.6: Az 4 térkép megoldása a 3D RL algorit-mussal

Mindegyik szimuláció sikeres volt, a VFH jól működik az RL algorit-mussal együtt. Az RL-nek sikerült megtalálni a legrövidebb utakat, és valószínűleg több iterációs-számmal vagy más Q táblás kezdőértékekkel minden esetben megtalálta volna őket. Ezen felül a VFH kitudta korrigálni a RL hibáit, amikor átlóban akart menni egy akadály sarkánál és minden esetben elkerülte az ütközést.



Pálya megtételéhez szükséges út				
Algoritmus Neve	RRT	RL	VFH	RL+VFH
1. pálya	79.66 m	52.48 m	66 m	58.66 m
2. pálya	46.22 m	49.58 m	45.31 m	59.77 m
3. pálya	59.47 m	43.81 m	68.95 m	56.69 m
4. pálya	60.39 m	42.89 m	41.68 m	45.62 m
Átlag	61.46 m	47.19 m	55.49 m	55.19 m

Táblázat 4.1: Pálya megtételéhez szükséges út

4.3 Összegzés

A 3D-s szimulációknál már egy valósnak nevezhető UAV modellt használtam így elmondható, hogy az a pálya amit megtett a valóságban is hasonlóan tenné meg. Emiatt megvizsgáltam és összehasonlítottam, hogy az egyes pályákon mennyi lenne az általuk megtett út, ezt az alábbi táblázatban (4.1) összegeztem. Megállapítható tehát, hogy az RL+VFH algoritmus megfelelően hatékony összevetve a vetélytársaival. Jól kivehető az is, hogy az RL algoritmusnak volt szüksége a legrövidebb útra, hogy elérjen a kiinduló állapotból a cél állapotba, átlagosan 10.18 méterrel kevesebb utat tett meg az adott szimulációnk alatt. Azonban nem lehetne csak az RL algoritmust használni a korábban említett hibái miatt, hogy az átlósan való mozgása miatt, beleütközne az akadályokba. Ennek a kiküszöbölésére került segítségül hívásra a VFH algoritmus. A táblázatból jól látható, hogy a jelenlegi verziójában is 0.3 méterrel kevesebb távolságot tesz meg átlagosan. Ha megnézzük a 4.6 ábrának a (b) térképét, jól látható, hogy jelen pillanatban az algoritmus még nagyon kileng egyes helyeknél, amiatt történt, hogy ennyivel hosszabb utat kellett megtennie a második térképen a RL+VFH algoritmusnak. Ez kiküszöbölhető egy finomhangolással, ennek tudatában feltételezem, hogy a különbség egy továbbfejlesztett változatban még nagyobb lenne. Az RRT algoritmussal összehasonlításban jól látható, hogy mind a 3 verzió jobb értékeket produkált így az eredetileg felállított tézisem, hogy az RL algoritmus jobban működik mint az RRT bebizonyításra került. Továbbá kijelenthető az is, hogy az RL+VFH kombinációja megfelelő lehet a valós életű felhasználásra is.

Konklúzió

Dolgozatomban megvizsgáltam és bemutattam részletesebben a jelen pillanatban elérhető legjobb pályatervező algoritmusokat, akár globális, akár lokális szinten, mint a RRT vagy a VFH algoritmus. Vizsgálatot tettem arra, hogy a már eddig is elérhető algoritmusok közül melyek a legjobbak. Megállapítottam a globális pályatervező algoritmusok közül a DL-el készített az algoritmus a legsikeresebb. Bemutattam ezeknek az algoritmusoknak a 3D-s változatát, amikhez már az UAV kinematikája is számolva volt. Majd bemutattam a saját pályatervező algoritmusomat ami a DL-t illetve a VFH-t ötvözi. Ezt egy sikeresen pályatervező algoritmusnak tartom mivel jobb úton sikerült eljutnia a célállomásból a végállomásba mint az előzőleg tesztelt algoritmusok, illetve megtudott küzdeni olyan akadályokkal is amiket eredetileg a DL algoritmus nem ismerhetett föl mert nem voltak rajta a térképen. Elkészítettem az RL+VFH algoritmus párost és összevettem a már meglévő algoritmusokkal külön külön és az összehasonlítás eredménye az lett, hogy az adott pályákon az RL+VFH jobb eredményeket ért el mint az ellenfelei. Kutatásomat azzal szeretném tovább fejleszteni, hogy elkészítsek egy UAV-t ami Lidar szenzorral van felszerelve és a való életben is megcsinálom azt amit most szimulációban megcsináltam.

Bibliography

- [1] Timothy Amukele. Using drones to deliver blood products in rwanda. *The Lancet Global Health*, 10(4):e463–e464, 2022.
- [2] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.
- [3] MathWorks. planner. Megnyitva: 2024-03-31.
- [4] Dankó Bence. Szakdolgozat. Accessed: 2024-03-22.
- [5] Sándor Tihamér Brassai. Neurális hálózatok és fuzzy logika. pages 63–69, 2019.
- [6] MathWorks. What is deep learning. Accessed: 2024-03-22.
- [7] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.
- [8] Pataki Béla. Gépi tanulás (szekvenciális döntési probléma (megerősítéses tanulás)). Accessed: 2024-03-21.
- [9] RICHARD BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.
- [10] Jung Ádám. Markov döntési folyamatok. Accessed: 2024-03-21.
- [11] Dr. Csiszár Villő. Diszkrét és folytonos idejű markov-láncok (elektronikus jegyzet). Accessed: 2024-03-21.
- [12] Szegedi Tudományegyetem. Megerősítéses tanulás, mély q-hálók. Accessed: 2024-03-20.



- [13] Simon Vanneste, Ben Bellekens, and Maarten Weyn. 3dvfh+: Real-time three-dimensional obstacle avoidance using an octomap. volume 1319, 07 2014.
- [14] Valentyn N Sichkar. sichkar-valentyn/Reinforcement_Learning_in_Python: Reinforcement Learning in Python, October 2018.
- [15] Sanghyuk Park, John Deyst, and Jonathan How. *A New Nonlinear Guidance Logic for Trajectory Tracking*.
- [16] MathWorks. uavwaypointfollower. Accessed: 2024-03-28.

