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

Kovács Boldizsár

Önálló laboratórium
Járókelők mozgásának szimulálása

Konzulens:
Goldschmidt Balázs

2022.05.22.

Tartalomjegyzék

1. Bevezetés

A hétköznapok során rengeteg alkalommal természetesnek veszünk cselekedeteket, eseményeket. Gyakran gondoljuk azt, hogy valami magától érthető. Például veszem a gyalogos közlekedés során az útvonalunk megválasztását. Gyaloglás során nem esik nehezünkre felmérni a többi embert, a gyalogos forgalmat. Ha látjuk, hogy megérkezett a busz a megállóba, akkor számítunk arra, hogy arról valaki leszáll. Ha egy sarkon befordulunk, akkor számíthatunk arra, hogy valaki szembe jön. Gyakran nézünk a lábunk elé, hogy ne botoljunk el semmiben, továbbá a talajt is felderítsük, hogy min is sétálhatunk. Értem ez alatt, hogy sétálhatunk az úttesten is, vagy a járdán is, de lehet szó az aszfalt és a füves rész különbségéről is. Felismerjük tudat alatt a lépcsőt, az aluljárót, a zebrát, a teljes látható környezetet, ami azt segíti, hogy tudjunk tájékozódni, és hogy a lehető legegyszerűbben, legrövidebben vagy leggyorsabban tudjuk eljutni egyik helyről a másikba.

Mind ezen triviálisnak hangzó állítások akkor vesztik el magától érthetőségüket, ha az informatikában akarunk ezzel foglalkozni. Ott semmit sem lehet venni természetesnek a valós világból. Ha tudat alatt felismertük a gyalogos átkelőhelyet, akkor azt az informatikában bonyolult módokon tudjuk csak szintén észre venni. Egy gyalogosnak állt robot esetében temérdek szenzor összehangolt és komplex adatfeldolgozás eredményeképpen lehet a természetes dolgokat legalább alap szinten megoldani. Szükséges lenne egy ilyen robotnak a szenzorokat használva például képfeldolgozás, a táblák és más objektumok felismerésére. Távolság mérésre is szüksége lenne, a forgalomi szituáció térbeli elrendezésének / modelljének felépítéséhez és az objektumok aktuális gyorsulásának és sebességének méréséhez. Továbbá hangfeldolgozás is, például autó dudálását felismerni, villamos záródó ajtaját kísérő hang felismerésére. Mindezek mellett egy bonyolult kiértékelő rendszerre, ami ezeket összeköti. A robot mozgáskoordinációjáért felelős rendszert nem is említve. Rettentően komplex feladat és nem is olyan jövőbeli [1], mint amennyire hangzik.

Azért választottam témának azt, hogy járókelőket szimuláljak, mert az informatikában különösen érdekes számomra az, hogy gyakran teljesen más gondolkodásmódot igényel egy egy probléma megoldása, mint a valóságban. Ilyen például a járókelők mozgása. Értjük miért mozognak úgy, de megalkotni az informatika keretein belül már teljesen más dolog.

Szerencsére nem csak számomra érdekes ez a téma, hanem ennek van több, a valóságra is kiható projektje. A sétáló robot még nem túl hétköznapi példa, de ha a tűzvédelmi próba riadót veszem, akkor ott nagyon is szükség van egy ilyen szimulációra. Ugyanis különböző épületeknek meg kell felelniük rengeteg előírásnak. Ilyen a kimenekítési terv is, melynek helyesnek és elegendőnek kell lennie. Tűzriadó próbán ember az életében legalább egyszer részesül. Viszont ami nem annyira közismert, hogy az emberekkel történő elpróbálás csak 500 – 1000 főig történik meg. Ezen intervallum fölötti létszámra tervezett épület esetében már szimulációt igényel a tűzriadó elpróbálása, még a tervezés fázisában

az alaprajzok alapján. Ennyi emberrel nehéz lenne elpróbálni és a későbbiekben az épületbe beköltözött cégeknél költséges is lenne a próba okán kiesett munkaidő.

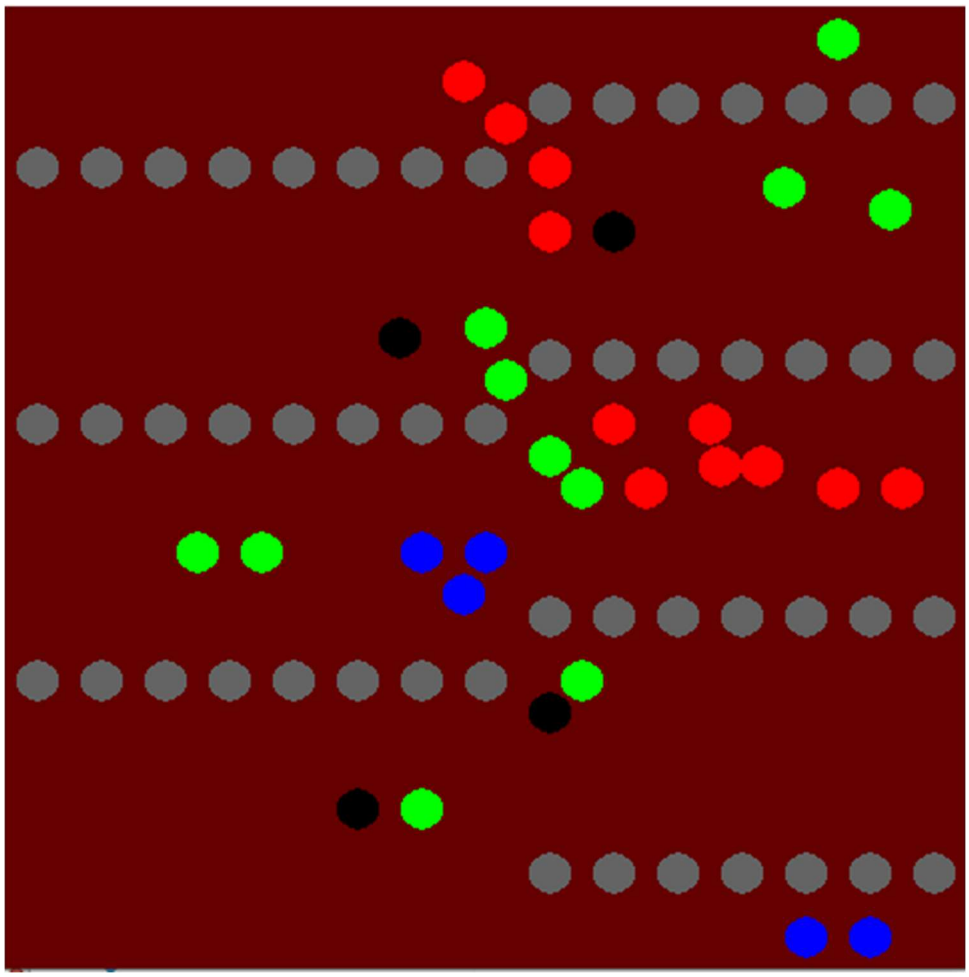
Egészen változatos területeken is foglalkoznak az ember mozgásával. Mint például az önvezető autóknál, hogy az útkereszteződésekben, zebráknál helyt tudjon állni az autó. Különböző (főképp zárt térben történő) kétkezi munkát igénylő munkahelyek kialakításánál is fontos szerepet játszhat, hogy az emberek alkalmasan el tudjanak férni, mint például egy konyhán, vagy éppen a helység legjobb átfolyása lenne a cél az átutazó forgalom segítése érdekében, például egy aluljáróban.

2. Téma körülírása

A járókelők számára különböző modellekben útvonal tervező algoritmus készítése, továbbá az útvonalon történő végighaladás megvalósítása. A környezet felépítéséhez különböző objektumoklétrehozása és alkalmazása, mint a fal, az asztal és az ajtó például. Lehetőleg legyen minél optimálisabb, adott kereteken belül.

3. Féléves munkám

3.1 Első program



1. ábra, egy kép a programról a működése közben

3.1.1 Általánosan:

A félév elején elkészítettem egy modellt, amiben meg is valósítottam az útvonal keresést. Egyfajta mozgó falat is kipróbáltam abban a környezetben, és a programom helyt állt.

Ezen környezet egy sakktáblának tudható be. Adott egy X-szer Y méretű négyzetháló, amelyben a négyzetek a mezők, és azokon haladhatnak a járókelők. Természetesen egymás útját keresztezhetik a járókelők, de nem haladhatnak át egymáson, hanem az egyik a másik előtt vagy után haladhat csak tovább azon a mezőn. Időpillanatokra osztott megoldás ez.

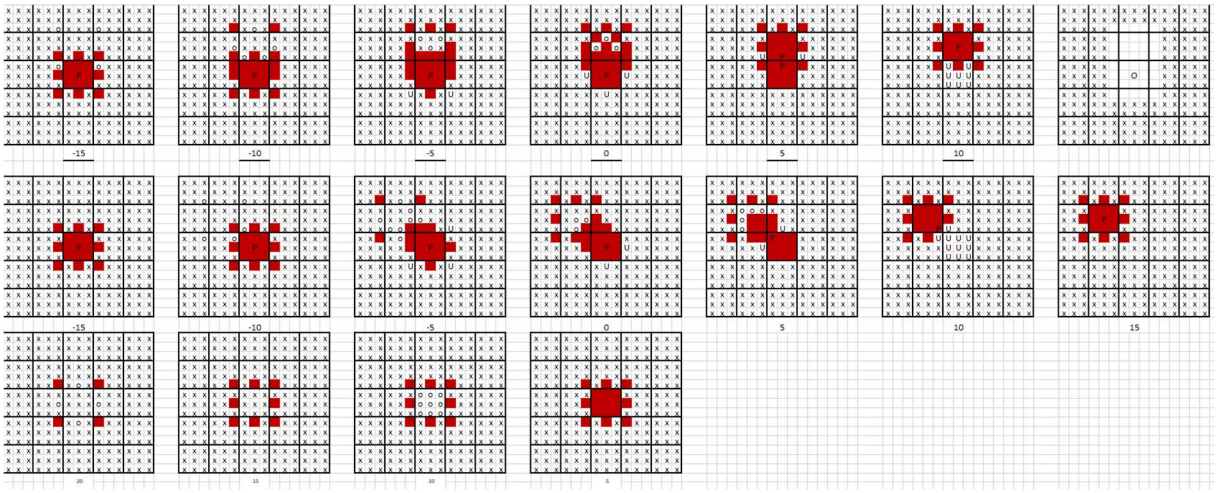
3.1.2 Részletezve

Minden időpillanatban 9 akciót hajthat végre a gyalogos, ha nem foglalt éppen és ha más nem blokkolja valamelyiket a 9 közül. A 9 az

alábbi: maradhat egyhelyben, átléphet egy élszomszédos mezőre, ebből van 4, vagy átléphet csak egy közös sarokkal rendelkező szomszédra, ebből is van 4. Ezeknek akciók bizonyos időbe kerülnek. Ha élszomszédos mezőre lép a gyalogos, akkor azt 10 időegység alatt teszi meg. Ha átlósan lép, akkor elvileg $\sqrt{2} * 10$ lenne a költsége ennek, de az algoritmus gyorsabb működéséhez ezt az értéket 15-nek választottam, és a helyben maradás időköltségét meg 5-nek, hogy lényegében 1, 2 és 3 legyen a költségük. Ezzel egy könnyen léptethető rendszert alkottam meg.
Ezt felhasználva minden járókelőről tudtam ebben a bontásban, hogy adott pillanatban hol van. Melyik mezőn áll, vagy éppen, hogy áll a mozgásával (fél úton van, vagy harmadokon).

A mezők végtére egy kötött gráfot alkotnak, amelyeken az A*-ot [2] futtatva könnyen eredményre lehet jutni, viszont az első témával kapcsolatos nehézség itt jött elő, hogy az A* nem elég. Egyetemi féléveim során például Algoritmus elméletből is foglalkoztunk az A* algoritmussal, és a hatásfokával nem is kívántam a foglalkozni a projekt keretein belül. Viszont, amivel kellett, és foglalkoztam is, az az, hogy miként kell „dinamikus” környezetben alkalmazni. Ezt én itt időpillanatonként eltárolt pillanatképekkel oldottam meg. Minden időpillanatban minden mezőnek a róla végrehajtható akciókat eltárolom, hogy mit lehet közölük, és mit nem.

Ezt a tárolót minden egyes új útvonal létrehozásánál frissíteni kell. Egy úgy gyalogos amikor belekerül a rendszerben, legenerálódik neki egy útvonal. Ezen útvonal ebben a megoldásban már etalonnak fog számítani a következő gyalogosnak. Ugyanis ezen útvonalon különböző mozgást végez a gyalogos, és minden egyes mozgáshoz tartozik néhány blokkoló mátrix. Az alábbiak:



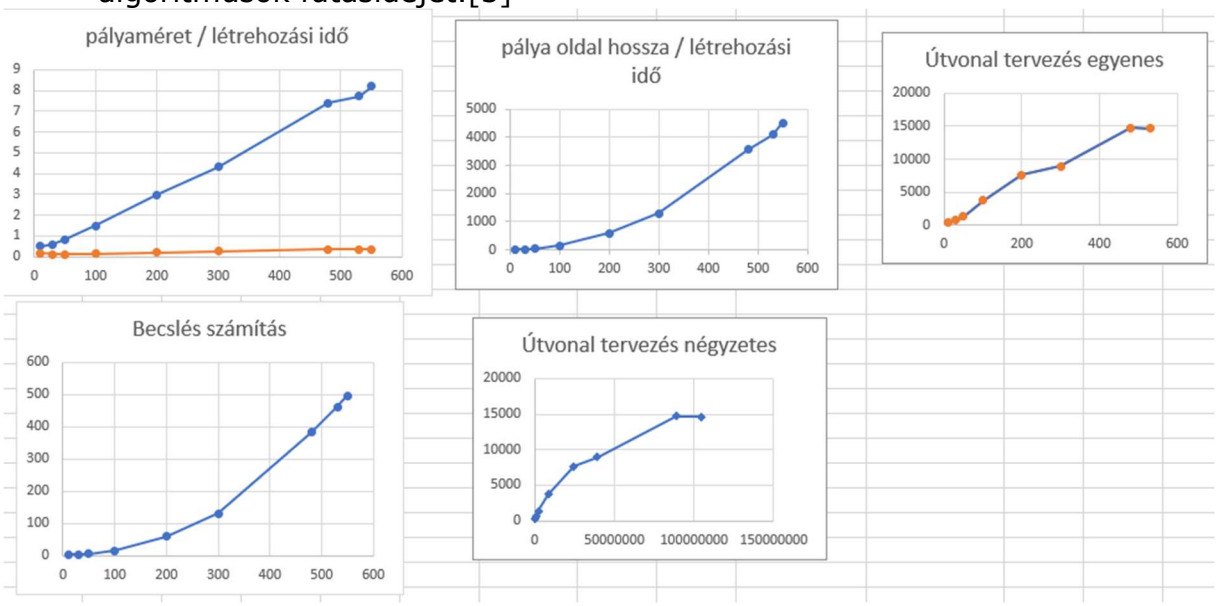
2. ábra, blokkolási mátrixok

Az ábrának első sorában az élszomszédos mozgás során fellépő blokkolást mutatja, a következő sora az átlós lépését, és a harmadik meg az egyhelyben maradást (vagy a pályán előforduló falak blokkolása, az az alsó sor vége).

Ez ahhoz szükséges, hogy az útvonalkeresés során ne generáljon olyan útvonalat, amelyt követve egy másik emberbe belemenne. 5x5-ös területen lehet blokkolással érintett mező, így a mezőnkként 9 akcióval több 15x15-ös mátrixot hoztam létre. Figyelve arra, hogy folyékony legyen a járókelők mozgása. Ne hagyjanak fölösleges hézagokat maguk között.

3.1.3 Problémák:

A megoldás helyesnek bizonyult. Viszont főképp a mátrixok létrehozásánál rontottam el egy két akciót le nem blokkolva. C++ nyelvi elemei ritkán megleptek feledékenységem okán, például a set nem szeret két ugyan olyan elemet tárolni, ehhez kell egy megfelelő <operátor definiálás. Vizsgáltam a program futás idejét különböző szituációkban, hogy tudjam, mennyire sikerült jóra megalkotnom. Ehhez írtam egy teszt esetet, hogy abban több paramétert változtatva vizsgáljam az algoritmusok futásidejét.[3]



3. ábra, első program teljesítménye

A pályaméret az oldalhosszal négyzetesen nő, így nem meglepő, hogy négyzetesen növekszik annak az időköltsége. Az útvonal tervezéshez szükséges idő, az első sor utolsó ábráján látható, hogy nem négyzetes, hanem legfeljebb lineáris. Ez annak is köszönhető, hogy a pálya felépítésénél sok mindent kiszámolok előre, ami sokban segíti az algoritmust. Ilyen a becslés számítása is. A statikus falakat veszi csak figyelembe, és minden mezőre megmondja, hogy onnan pontosan milyen messze van az elvi legrövidebb úton a cél. Ezt minden célra kiszámolja.

A pálya felépítésénél történő számításokkal megtizedeltem az útvonalkeresés időköltségét. Nem csak az algoritmust optimalizáltam, hanem a használt tárolókat is. A célnak megfelelőt terveztem választani ahogy csak tudtam.

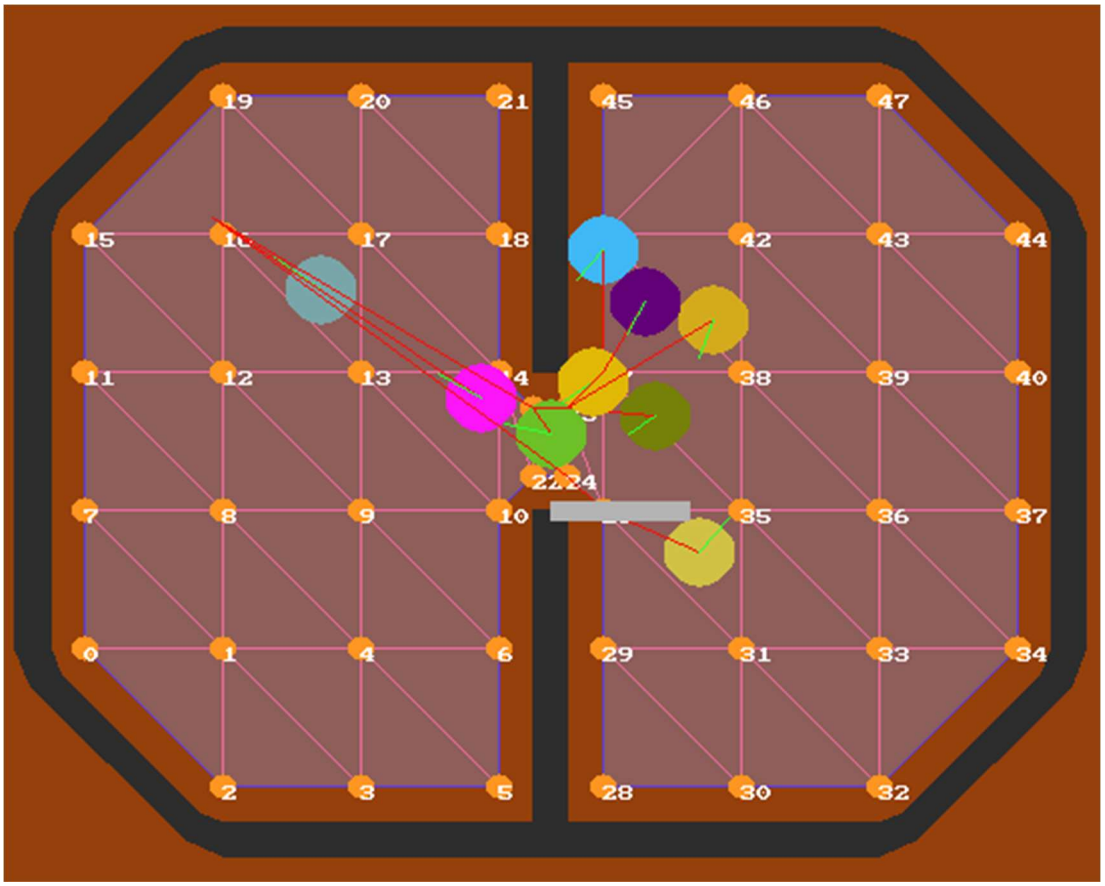
Probléma még az is, hogy az útvonalkeresés során egy korlátot szabok, hogy hány akcióval próbálkozhat összesen az algoritmus. Viszont az egyhelyben maradás is egy akció, és a csiki-csukizás is, ezért nagyon pazarló alkalmanként, ha csak sokára tud elérni a céljához az embertömeg okán. Ha nincs „dugó”, akkor a pálya méretének néhányszorosával megegyező akciószám elegendő a célhoz eljutni. De ha dugó van, akkor akár száz vagy ezerszerese is szükséges.

3.1.4 Összegzés:

Összességben elégedett voltam a megoldással. **Előnye** az volt, hogy a szimulációban a járókelők mozgása folyamatos volt, és a legjobb útvonalat végigjárva, folyékonyan mozogtak a járókelők. Minden járókelő mindenről tudott, ami számára fontos lehet, a legoptimálisabb útvonalon haladt végig és külső szemmel nézve valóságosnak tűnhet, mivel a járókelők remekül kerülték ki egymást és oldották meg az átjutást a kezdőmezőről a célmezőre.

Hátránya az az, hogy csak egyes mezőkre tudtam lépni, nem lehet tetszőleges kis távolságot megtenni. Diszkrét értékeket tudok csak felvenni. Továbbá azon járókelők, amelyek korábban rendelkeznek útvonallal, azok a többi járókelőről szinte tudomást sem vesznek. Akinek az útvonala korábban lett legenerálva, az lesz onnantól a mérvadó a többinek. Mintha mozgó fal lenne a következő járókelőknek a meglévő járókelő.

3.2 Második program



4. ábra, a második program a működése közben (doorTest pálya)

3.2.1 Az elsőhöz képest a különbségek

Próbáltam az előzőtől egy merőben eltérő, és előrehaladásnak tekintethő megoldást létrehozni. Továbbá lehetőleg az előző megoldás hátrányait próbálja feloldani, és az előnyeit ne maga ellen fordítani.

Kezdetben ez már **folyamatos**, azaz nem mezőkből áll a pálya, hanem háromszögekből. Ezen háromszögeken lehet mozogni. Ha kettő szomszédos akár csúccsal is, akkor azon az egy csúcson / ponton lehet átjárni csak a kettő között.

Nincsek eleve elrendeltetett útvonalak, így belekerülhet véletlen szituáció is, mint például, hogy leszakad a plafon, vagy egy teljesen dinamikusan mozgó járókelő is, amit például a felhasználó irányít.

Nincsen a szimuláció **előre legenerálva** és **időpillanatokra osztva**. Ezzel rengeteg memóriát spórolunk, némi futásidejű számításért feláldozva. Nem létezik blokkolásmátrix, vagyis ebben a környezetben teljesen más megvalósítást igényelt a szimuláció.

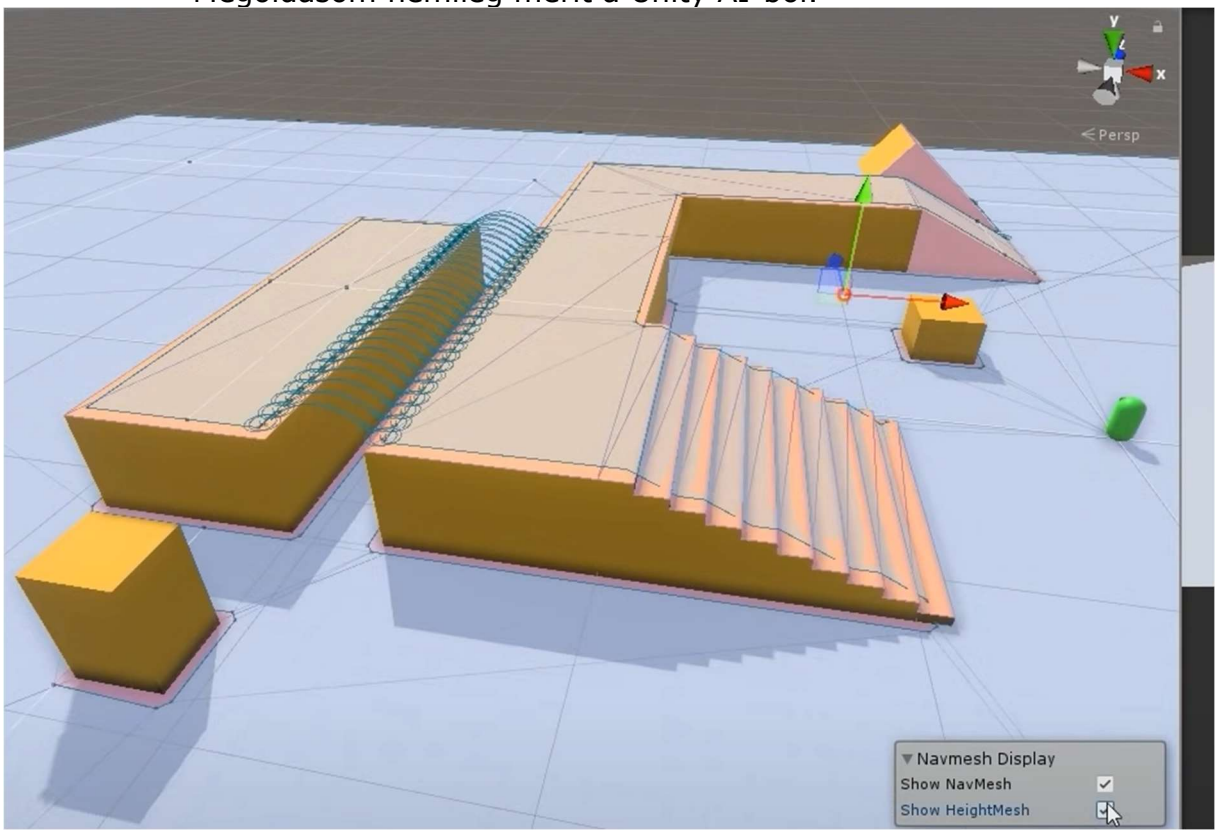
3.2.2 Általánosan:

Háromszögekből épül fel a pálya a háromszögek maguk a bejárható járófelület. Csúcsaiknál szomszédosak lehetnek, ha kettőnél is, akkor élszomszédosak lesznek. Szomszédos háromszögek között lehet átjárni csak. A járókelők körök. A középpontjuk sosem kerülhet háromszögön kívülre. Kék vonallal látható az ábrán a pálya tényleges falai, amin áthaladni nem szabad a járókelőknek. A rendszer erő vezérelt, a

létrehozott útvonal következő megállójának irányába hat a járókelőre alapvetőleg erő, és másodpercenként több alkalommal alkalmazom rá a fizikát. Ha falba ütközik, akkor kiszámolja, hogy feltehetően mennyit „csúszik” rajta.

Lényegében ez egyfajta fizikai szimulációnak tudható be. Ebben a környezetben léteznek ajtók is. Ezeket lehet egy vagy két irányból nyitni. Egy irány esetén lehet tolni, vagy kilinccsel kinyitni és szabadon „húzni”. Az ajtók automatikusan bezáródnak 3 másodperc után. Ha járókelő egy másikkal találkozik, akkor megpróbálják eltaszítani egymástól magukat, mint valami taszító mágnesek.

Megoldásom némileg merít a Unity AI-ból.



5. ábra, Unity NavMesh-e azok a háromszögek a járófelületeken [4]

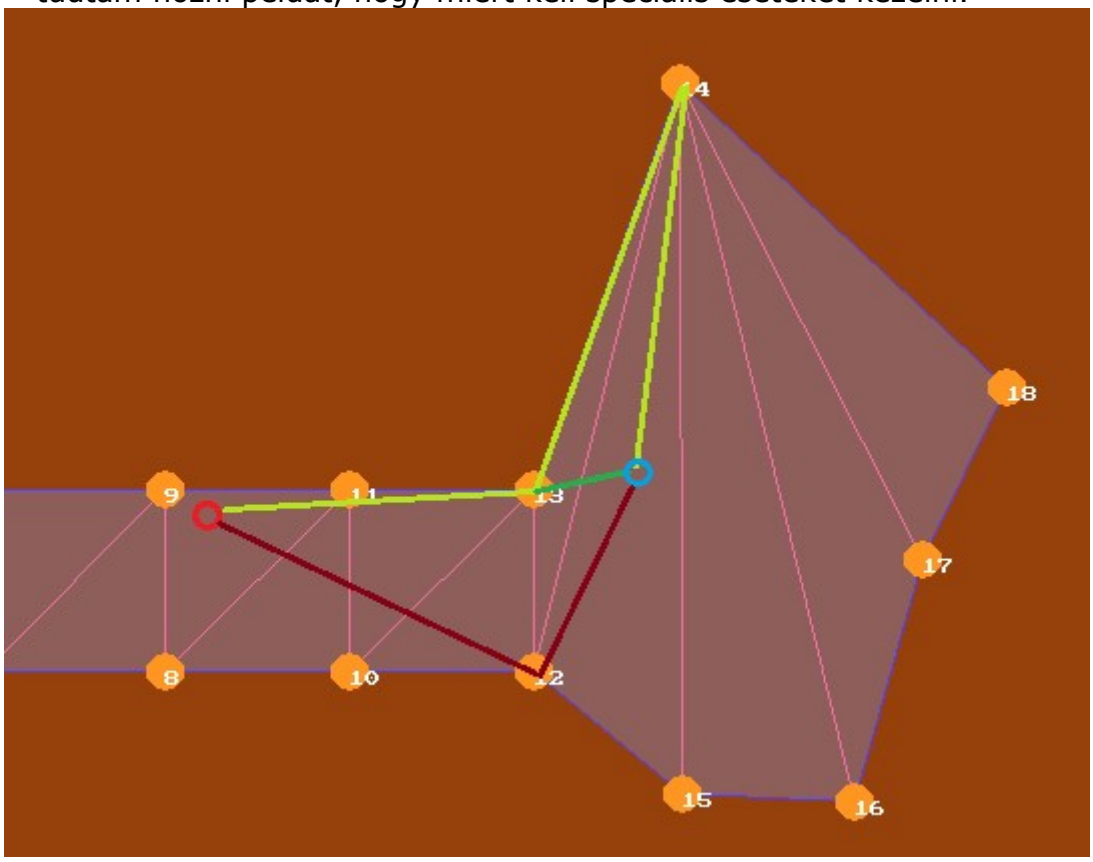
3.2.3 Részletesen, a megoldáshoz vezető út:

Kezdetben csak az útvonalkeresés megvalósítása is problémát okozott, mert nem volt elég a háromszög csúcsait a gráf csúcsainak tekinteni, az éleit meg a meg az éleinek, ugyanis ekkor törtvonalak lettek a járókelők útvonalai. Szóval fejleszteni kellett valahogy az A*-ot.

Ahhoz, hogy egyenes vonaltat is kaphassunk két pont között, melyeknek van egymásra rálátásuk, de nem élszomszédosak, ahhoz az útvonalat vizsgálni kell. Minden egyes töréspontot meg kell vizsgálni, hogy onnan hova van rálátásunk. Ekkor kaphatjuk meg, hogy mi lenne az adott útvonalból a kihozható legrövidebb.

Két probléma volt a ténylegesen legrövidebb útvonal okán. Hogy minden lehetőséget bejártam, és hogy a bejárt lehetőségekből tényleg a

legrövidebbet válasszam ki. Kis gondolkodás után mind a két szituációra tudtam hozni példát, hogy miért kell speciális eseteket kezelni.



6. ábra, útvonalkeresési probléma

A világos zöld mutatja a jó esetet, a vörös a rosszat. Ha nem derítem fel mindegyik csúcsot kellőképpen, akkor lehet, hogy a vörös lesz a létrehozott útvonal, és nem a világos+sötét zöld. Itt az a baj, hogy a túl messze van a 14-es csúcs. A 12-es csúcs meg sokkal közelebb van hozzá, így azt mikor felfedezem, akkor megérkezem ahhoz a háromszöghöz, amiben a cél van. Ekkor végzett a dolgával és nem ismerjük meg a legrövidebb útvonalat. Ezért mind a három csúcsát kifejtem a célháromszögnek, mielőtt ítélkezem. Ez sem mindig elég, Egy spéci eset, amikor a háromszög egésze a rossz megközelítés miatt takarásban van, viszont szinte egyenes lenne az út odáig, egy kis törést leszámítva, és ehelyett a takarást, a lyukat körbe sétálja. De erre találtam ki megoldást. Ha a pálya felépítése, azaz a háromszögek, jól vannak létrehozva, akkor nem történhet ilyen fennakadás. Szóval megoldva nem lett, csak figyelek rá, hogy ne forduljon elő.

A másik probléma meg az, hogy ha megtalálom a legrövidebb törtvonalas megoldást, akkor annak a kisimítása nem biztos, hogy a legrövidebb tényleges útvonalat adja vissza. Ahhoz minden egyes újonnan felfedezett csúcsra meg kell nézni, hogy oda milyen hosszú úton lehet eljutni. Azaz minden csúcsra kell egy vágást végezni. Ez sajnos költséges.

Kezdetben más tervezői döntést hoztam, mint a jelenlegi. A vágások során, valahogy el kellett dönteni, hogy van rálátás a két pont között, vagy sem. Először a háromszögeken zongoráztam végig, hogy az útvonal egyes lehetséges vágásai milyen háromszögeken keresztül jut el a vágás

végéhez. Ha háromszögek oldalait átlépegetve, akkor jó, ha viszont úgy lép ki, hogy nincs ott háromszög, akkor az azt jelenti, hogy nincs rálátás onnan. Ezt a megoldást felváltottam azzal, hogy inkább végig nézem az összes falat, hogy hátha elmetszi az egyik a vágást. Ha mégsem, akkor a vágás helyt áll. Ezzel volt egy olyan probléma, hogy két fal találkozásánál az útvonal vágást végzett és kimutatott az útvonal a járófelületen kívülre, de azt megoldottam egy alkalmas ellenőrzéssel.

Probléma volt azzal, hogy néha beragadt két járókelő, mert egy csúcsba akartak menni, de két irányból. Ekkor patthelyzet alakult ki. Azért jöttek létre ezek a helyzetek, mert az egyik gyalogos már túlhaladt a megállón, de nem sikerült érintenie a megállót, így addig próbálkozik oda jutni, amíg meg nem érinti. Erre az lett a megoldásom, hogy ellenőrzöm a haladást. Ha a célmegálló távolsága növekszik, de az azt követő megálló távolsága csökken, akkor átváltok arra és nem tervezem érinteni az aktuális megállót. Egész jól működik és az esetek nagy részében működik is.

A tényleges falak csak díszek. Az asztalok nem tennének semmit sem a járókelőkkel. Csak a kontextus kedvéért vannak ott. A járókelők beragadnak éles, egy csúcsban megoldandó „kanyaroknál”. Ezek viszont nem fordulnak elő a való életben, tényleges falak nem tudnak indokolni olyan helyzetet. Mindig lenne egy kis visszakanyar a rendszerben.

A járókelők között kreáltam csoportokat. Ezen csoportokat azért hoztam létre, hogy a prioritásokat tudjam kezelni. Így, ha egy teremben sokan vannak, és ki akarnak jönni onnan, de be is akarnak menni, akkor elsőbbséget adok a bent lévőknek. Azaz, ha ők ki akarnak jönni, akkor ki tudnak jönni. Nagyon erőt gyakorolnak másokra, mint saját szintjükre.

Kisebb problémák merültek fel az erő vezérelt működéssel kapcsolatban. Például kilőttek, kirobbantak egyes helyzetekből. Ezt egy maximális sebességgel és gyorsulással tudtam kezelni. Ha valaki valamiért mégis beleragad egy falba, akkor azt detektálom, és keresek neki egy útvonalat.

3.2.4 Ajtók:

Létrehoztam ebben a megoldásban ajtókat. Ezek félig falak, félig nem. Falak, mert nem szabad beléjük menni, de nem falak, mert el lehet mozdítani őket. Az ajtók lehet, hogy csak egyik irányból nyílnak, beállítástól függően. Ekkor el kell menni az ajtó kilincséhez (végéhez) és onnantól lehet húzni, tolni az ajtót. (toláshoz nem szükséges a kilincs)

Az ajtó nyitása automatikusabb, viszont az ajtó kikerülése érdekes. Az ajtó közelében mindig az ajtó vége fele szeretnénk menni, hogy meg tudjuk kerülni, de ha túljutottunk rajta, akkor már nem akarunk a vége fele menni. Hogy „túljutottunk”, azt kellet detektálni. Sikerült.

Az ajtók nyitásakor, zárásakor járókelőket vissza kell löki, mert nyílik az ajtó. Lekezeltem, hogy senki se kössön ki az ajtóba ékelődve.

3.2.5 Alkalmazott technikák:

Az útvonalkeresésen sokat gyorsított az, hogy van némi információnk a pályáról. Minden csúcs tartozik egy vagy több háromszöghöz, ezen háromszögekhez tartozik mindig három csúcs. Egymásra mutatnak, hogy lehessen összetetten gondolkodni. Továbbá használok egy Grid rendszert. Lényege az az, hogy nem kell minden objektumot végigvizsgálnom, hanem azon a grid-en / négyzeten belül lévő pont esetén, cask néhány objektum játszhasson szerepet.

Például egy háromszöget úgy kapok meg, hogy a létrehozásakor megvizsgálom, hogy mely láthatatlan négyzetekbe nyúl bele, és azon négyzetekhez felveszem, mint potenciális háromszöget. Amikor kattintok, és érdekel, hogy a kattintás helyén melyik háromszög van, akkor elég lekérdezni az ahhoz a láthatatlan kattintott négyzethez tartozó háromszögeket. Azokból nincsen sok, míg a pályán lehet akár több száz is. Rengeteg időt spóroltam meg ezzel. Ezt a módszert rengeteg helyen használják, általában "Chunk"-nak hívják, bár annak a működés / használata elég tág.

Az erők számolásánál a fizikai valóságot próbáltam megvalósítani. A matekosabb részeknél próbáltam minél optimálisabb megoldással előállni. Vágásoknál a grafikában egy már megoldott és kioptimlizált problémákkal sikerült megoldanom néhány dolgot. Főképp a szakasz metszésénél. Háromszögben lét ellenőrzésénél. Minimális matekot igényel a program megértése.

4. Projektből tanultak, jövőbeli tervek

Megbecsülöm azt, akinek sikerül egy szélesebb körben elismert, auditált megoldást létrehoznia egy adott problémával kapcsolatban. Ugyanis egy ilyen kis szegletében az informatikában is annyi kérdés merült fel, hogy arra hogy széles körben mindenkinek egységes legyen a válasza valamivel komplexebbel, általánosabbal kapcsolatban, azt nehezen lehet elérni, megoldani.

Mesterséges intelligenciát és Deep Learning-et tanultan látom, hogy lehet egészen jól futó programokat létrehozni "könnyen". Viszont nehezen lehet egy biztosan működő és jó / helyes programot létrehozni. Egy biztosan működő program általában sokkal determinisztikusabb - pontosabban – nem tud olyt nyújtani, amit mi nem írtunk le. Viszont tökéletesen lehet látni azt, hogy hol rossz az ami, mert minden a "saját" munkánk.

Nagyon fontos az útvonlkersés során, hogy legyenek előre legenerált útvonlak két pon között. Például a szobák ajtjai között. Az nagyon sokat tud gyrosítani az algorimuson. Érdemes használni szkatulyázó / Grid / Chunk rendszert, hogy ne kelljen mindennel foglalkozni, cask az adott közeggel. Ne kelljen minden háromszöget

végignézni. Ne kelljen minden játékossal ütközni, csak amik tényleg közel vannak. Érdemes lehet szobákat és kisebb egységeket létrehozni, hogy kis közegekben keressünk csak algoritmust, és ne kelljen az egész világot körbepásztázni.

5. Források

[1] https://www.youtube.com/watch?v=vjSohj-Iclc – futásra és ugrásra képes „emberszabású” robot

[2] https://www.youtube.com/watch?v=-L-WgKMFuhE&t=1s – A*-ról és az útvonalkeresésről szóló videósorozat eleje

[3] FutásidőV1.ods Excel tábla

[4] https://www.youtube.com/watch?v=atCOd4o7tG4 Unity megoldása az útvonalkereséshez, NavMesh

Egyéb, a dokumentáció során nem megemlített források:

http://users.itk.ppke.hu/~tihanyia/HelyUltra/Bacskai_Gergely_Mor_Szakdolgozat.pdf
Egy másik egyetem hallgatójának szakdolgozata. Hasonló algoritmusokat használ, továbbá az I épület bizonyos szintjét szintén a programjába beleépítette. Ő egy lánccal próbálta megoldani az útvonalkeresést. Az számomra is hasznos egy megoldás, viszont nem akartam ilyet generálni, hogy tudjam, mire képes az algoritmus önmagában.

https://adoc.pub/queue/specialis-matematikai-modellt-alkalmazo-szoftver-optimalis-u.html
Nem teljesen van fedésben az én témámmal, de rávilágított, hogy milyen sok féle útvonallal kapcsolatos célunk lehet. Érdemes hogyan lehet kiszámolni a saját útvonalam keresése közben azt, hogy a közösségnek is a lehető legjobb legyen. Továbbá, hogy temérdeknyi elemből, állítható paraméterből állhat egy szimuláció.

http://hdl.handle.net/10890/13342
Egy kitűnő munka szerintem. A hangyás megoldás szintén csak színesítette a lehetőségek palettáját. Gondolkodtam azon, hogy a véletlennel jó esélyem lenne megtalálna az útvonalat, de konkrét számolásokba bonyolódva nem jöttek kis számomra kedvezőnek mondható értékek.