Examensarbete

HÖGSKOLAN I SKÖVDE

1977

JÄMFÖRELSE AV VÄGSÖKNINGSALGORITMER FÖR EN HASTIGHETSBASERAD AI

COMPARISON OF PATHFINDING ALGORITHMS FOR A VELOCITY BASED AI

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2025

Caleb Pilbrant
Elias Drake af Hagelsrum

Handledare: Martin Hagvall
Examinator: Sanny Syberfeldt

Sammanfattning

Vägsökningsalgoritmer inom spel fungerar ofta som rullband på det sätt att en agent flyttas längs en väg, där logik på agenten används för att hantera komplexa situationer samt beteenden för att göra deras rörelse realistisk. Studien undersöker vad som händer när en agent istället försöker följa en mer realistisk väg med hastighetsbaserad rörelse. Detta undersöktes genom skapandet av en artefakt som jämför effektiviteten av hur Unitys inbyggda metod, samt vägsökningsalgoritmerna A*, Theta* och Phi*, genererar vägar som använder positioner skapade av antingen Unity NavMesh eller ett quadtree. I slutändan var vägsökningsalgoritmer skapade för hastighetsbaserad rörelse mycket snabbare fast också mer riskabla än Unitys inbyggda. Hastighetsbaserad AI har många dygder och skapar rörelser som ser mycket realistiska ut men kräver mer finslipning och mer komplext beteende innan det kan användas i spel. För framtida arbeten är denna rapports förslag att testa flera kombinationer av algoritmer och metoder att skapa positioner.

Nyckelord: Pathfinding/Vägsöknin, Velocity/Hastighet, Artificial Intelligence/ Artificiell Intelligens, Phi*, Theta*, A*, Quadtree

Innehållsförteckning

1. Introduktion 1
2. Bakgrund 2
2.1. Vägplanering Utanför Spel 2
2.2. Unity Simulation 2
2.3. Dataset 3
2.3.1. Unity NavMesh 3
2.3.2. Quadtree 5
2.4. Vägsökningsalgoritmer 7
3. Problemformulering 8
3.1. Urval av algoritmer samt dataset 9
3.2. Projektmiljö 9
3.3. Agentutveckling 10
3.4. Metodbeskrivning 10
3.4.1. Metoddiskussion 11
4. Resultat 12
4.1. Genomförande 12
4.1.1. Bearbetning av väg 12
4.1.2. Agentens rörelse 13
4.1.3. Miljödesign 14
4.2. Presentation av resultat 15
4.3. Analys av resultat 17
4.4. Slutsats 18
5. Sammanfattning 19
5.1. Diskussion 19
5.1.1. Relation till tidigare forskning 20
5.1.1.1. Bézier-kurvor 20
5.1.1.2. Algoritmer och quadtree 21
5.2. Potentiella förbättringar 22
5.3. Samhälleliga och etiska aspekter 23
5.4. Framtida arbete 24
Referenser 25

1. Introduktion

Vägsökning är en viktig del av spel. Vägsökning bestämmer hur AI-kontrollerade entiteter (agenter) ska gå, oavsett om de agenterna ska hjälpa eller hindra spelaren. Denna sökning utförs av algoritmer som genom matematik och jämförelser hittar den kortaste vägen mellan en startpunkt och en slutpunkt. Problemet är dock just det, att de vanligaste algoritmerna söker efter den kortaste vägen. För hastighetsbaserade och realistiska agenter är den kortaste vägen inte alltid den snabbaste, detta kan ses bland annat på robottävlingen micromouse där den snabbaste vägen i en labyrint söks (Rathnayake, Wijesekara, & Samaranayake 2023).

För att lösa detta problem söktes litteratur om vägsökning utanför spel. Det introducerade algoritmer och sätt att samla data på som är passande för robotar eller andra realistiska och hastighetsbaserade agenter, bland annat att alla sätt att samla in data inte alltid är passande för både markbundna och flygande objekt. Även att det finns vägsökningsalgoritmer som är gjorda för att simulera objekt med hastighet.

Metoden som valdes för att få fram ett svar var att göra tester och samla kvantitativ data. Därefter analyserades och jämfördes datan för olika kombinationer av algoritmer och dataset. Insamlingen skedde genom att skapa två testmiljöer i spelmotorn Unity. För de miljöerna testades Unitys egna vägsökningsmetod mot tre andra algoritmer, där två olika dataset användes för att representera världen.

Efter experimentet genomfördes sammanställdes resultatet i grafer och tabeller för att lättare observeras.Efter experimentet genomfördes sammanställdes resultatet i grafer och tabeller för att lättare observeras. Genom att analysera graferna och tabellerna visade det att alla algoritmer har en chans att misslyckas, fast att alla oftast hittar en snabbare väg jämfört med basfallet när de lyckas.

2. Bakgrund

AI-rörelse inom spel handlar oftast om logiken som finns på agenten, att undvika hinder, att röra sig på ett realistiskt sätt, ett verkligt sätt, och att ta sig till målet på ett effektivt sätt. Detta hanteras nästan alltid inom AI-logiken och vägen mellan start och slut är oftast den kortaste möjliga vägen och agenten flyttas helt enkelt längs den. Detta är standard inom spel fast inom robotik och mer verkliga simulationer är det inte lika enkelt. I de fallen blir vägen någonting som agenten bör följa istället för att agenten flyttas längs vägen. När vägen då måste konstrueras med agentens rörelse i åtanke är det inte garanterat att den kortaste möjliga vägen mellan start och slut är den rimligaste eller snabbaste vägen att ta sig till målet.

2.1. Vägplanering Utanför Spel

I verkliga fall såsom micromouse-tävlingar är det viktigaste att åka en snabb och korrekt väg. Därför används vägsökningsalgoritmer där de har en tyngre vikt på långa raka sträckor där maxhastigheten av roboten kan användas. Även Bézier-kurvor används för att mer effektivt kunna ta skarpa svängar vilket tillåter en micromouse-robot att svänga utan att stanna vid hörn (Rathnayake, Wijesekara, & Samaranayake 2023).

Det finns gott om forskning som hanterar byggningen av simulationer för vägsökningsalgoritmer för robotar och drönare som hanterar generering av vägar genom verkliga miljöer. Till exempel visar Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022) en metod att generera detaljerad navigationsdata av komplexa utrymmen, såsom en högskola eller ett fotbollsstadion, som en robot kan använda för att effektivt röra sig igenom oberoende av lokomotionssystem.

Även AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) diskuterar en effektiv metod att generera korta och effektiva vägar som en drönare kan använda för att skapa användbara vägar optimerade för rörelse i hög hastighet. Deras metod är att använda algoritmer som kan navigera till punkter som inte är direkt bredvid andra punkter.

2.2. Unity Simulation

Det intressanta för detta arbete var att tillämpa kunskapen som används inom robotik för att skapa en vägsökningsalgoritm som är optimal för en hastighetsbaserad agent inom Unity 3D. Detta genom att undersöka olika vägsökningsalgoritmer och jämföra dem med standardvägen Unity bygger via NavMesh.

Hur Unity genererar navigationsdata och hur Unitys innbygda vägsökningsalgoritm fungerar är dolt, så exakt information är inte tillgänglig. Dock genereras navigationsdata generellt sett genom att rita ut punkter omkring hinder på en yta och de punkterna trianguleras sedan för att bygga en stor mesh av gåbar yta. Unitys inbyggda vägsökningsalgoritm använder punkterna NavMesh lägger ut för att förflytta sig.

2.3. Dataset

För att kunna bygga en rimlig väg mellan två punkter med komplexa objekt som hinder krävs det data som säger var en agent kan och inte kan gå. Dessa data är en samling av punkter som en algoritm har bestämt ska finnas, samt deras grannpunkter. Sammanfattningsvis blir det ett dataset av punkternas position i miljön, och vilka punkter som är grannar. Grannpunkter är de punkter en agent kan gå vidare till från punkten den befinner sig på.

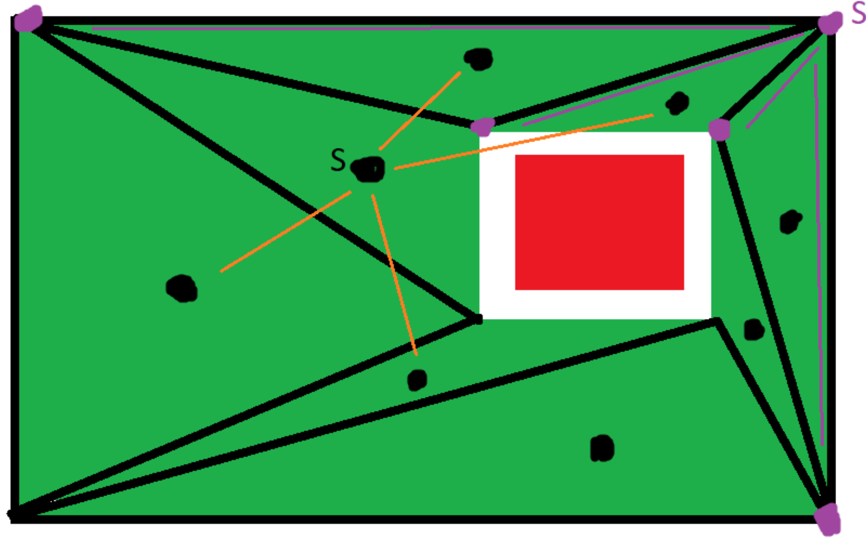
Hur dessa data skapas och vilken form den tar påverkar kvaliteten av vägen en algoritm kan skapa. Här definieras de metoder som användes i studien för att generera datan som använts av vägsökningsalgoritmerna.

2.3.1. Unity NavMesh

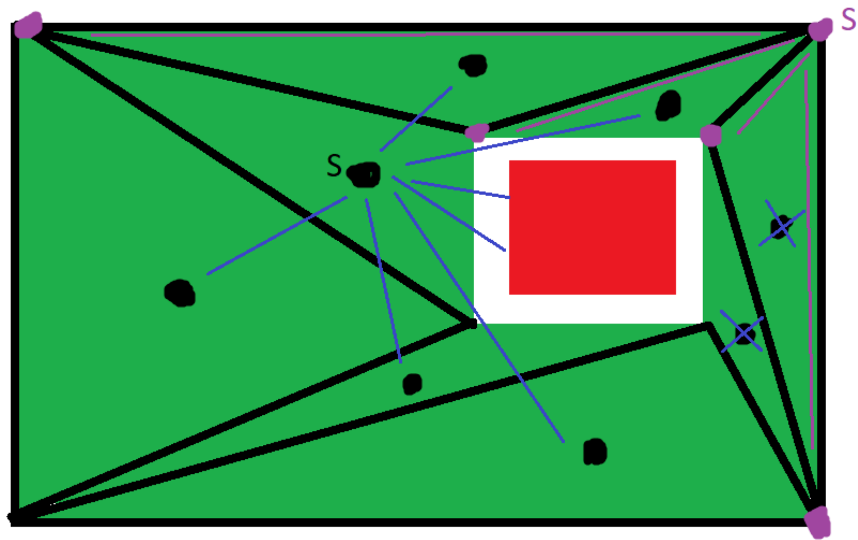
Unitys inbyggda navigationssystem delar upp miljön i trianglar, baserat på inställningar som säger var agenten kan gå. Huvudsakligen är det agentens bredd, som bestämmer både hur brett något ska vara för att agenten ska kunna gå på det, samt hur långt ifrån andra kollisions-objekt agenten kan vara. De trianglar den skapar är ytorna agenten kan gå (Unity 2025).

I denna studie ingår start- och målpunkterna i datasetet för alla algoritmer. Resterande punkter i datasetet är, för NavMesh (vägsökningsalgoritmen), trianglarnas hörn. För de andra algoritmerna är det i denna studie trianglarnas centrumpunkter. Även punkternas grannpunkter är viktigt att lagra. För NavMesh är grannpunkterna alla andra hörn som ett hörn är kopplat till. För A* är grannpunkterna i denna studie centerpunkterna för trianglarna som delar hörnpunkter. Grannpunkterna för Theta* är alla punkter som det kan ritas en rak linje mot. Phi* fungerar som Theta*, fast att alla punkter utanför en synvinkel bortträknas.

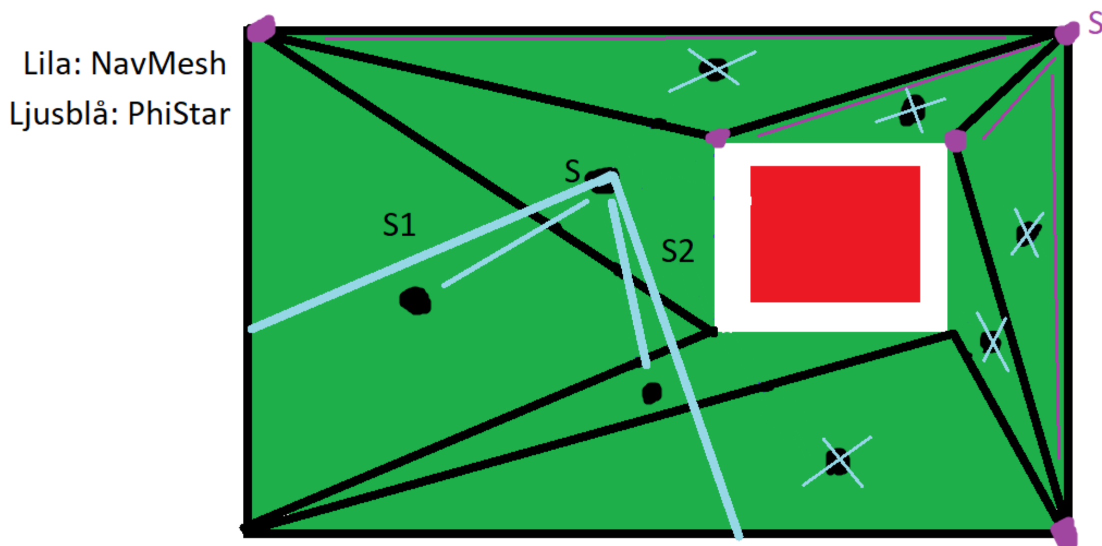
Bilderna nedan, figur 1, 2 och 3, är representationer som visar alla algoritmer och hur de bestämmer grannar i denna studie. Observera att bilderna är representativa, utifrån hur NavMesh bygger miljön. Punkten med ett S bredvid är punkten som det räknas från, och färgerna visar vilken algoritm det är, lila för NavMesh, orange för A*, mörkblå för Theta*, och ljusblå för Phi*. Bilderna visar också ett hinder (den röda rektangeln), samt var en agent kan gå (den gröna ytan indelad i trianglar är vart agenten kan gå, den vita ytan är den distans från ett hinder som agenten inte kan gå på). Bilderna visar även en uppdelning av miljön i trianglar baserat på hindret. Figur 3 visar synvinkeln för Phi*, de tjockare blåa linjerna (S1 och S2), och punkterna med ett X representerar hur den ursprungligen fungerar som Theta* fast bortser punkterna då de är utanför synvinkeln.



Figur 1: Representation av miljö som visar grannar för A och NavMesh. (Gjord i Microsoft Paint).*



Figur 2: Representation av miljö som visar grannar för Theta och NavMesh. (Gjord i Microsoft Paint).*



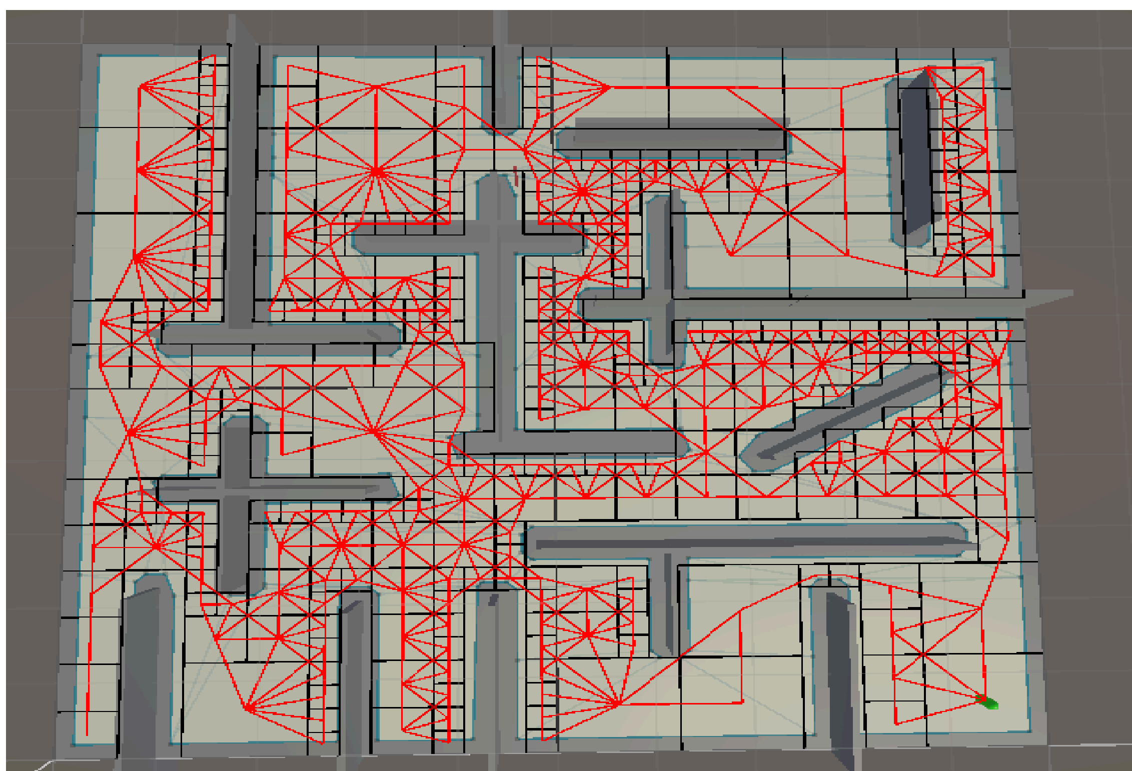
Figur 3: Representation av miljö som visar grannar för Phi och NavMesh. (Gjord i Microsoft Paint).*

2.3.2. Quadtree

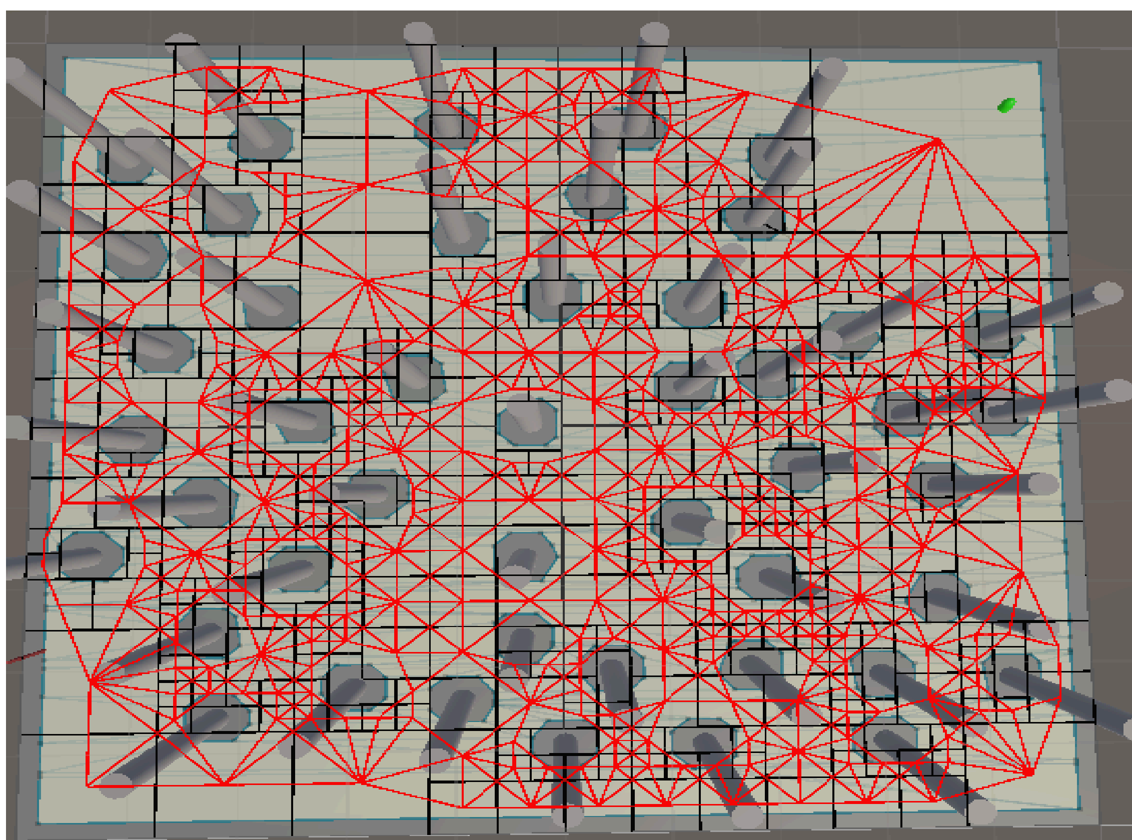
Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022) skriver om vägsökning i tre olika fall, där agenten går, rullar eller flyger, och påpekar att de flesta studier hanterar antingen de markbundna eller flygande fallen. Deras arbete hanterar alla tre transportsätt, och skriver vidare att det vanligaste sättet att bygga miljön på för flygande fall negativt påverkar datan för gående och rullande, på grund av den resulterande detaljnivån. De väljer istället att använda ett octree, som har en dynamisk detaljnivå baserat på vad som behövs.

Denna studie använder istället ett quadtree, som octree är en tredimensionell variation av. Ett quadtree delar upp miljön i kvadrater, baserat på inställningar. I denna studies fall delas en kvadrat upp i fyra mindre kvadrater om det finns hinder i kvadraten, tills det inte finns hinder eller att kvadraten nått den minsta tillåtna storleken.

När uppdelningen, som beskrivs i stycket ovan, är klar omvandlar denna studies implementation av quadtree kvadraternas centerpunkter till punkter som algoritmerna använder för att räkna ut vägar. Detta görs så att algoritmerna kan implementeras mer generellt. Punkternas grannar räknas ut på nästan samma sätt som de gör när NavMesh används som dataset, förutom att NavMesh (algoritmen) inte använder quadtree-datasetet alls, då den är dold av Unity. När denna studie använder ett quadtree måste A* räkna ut grannpunkter på ett annat sätt än när NavMesh används. Detta beror på att kvadrater inte alltid är samma storlek, därför blir istället alla kvadrater bredvid en kvadrat dess grannar. Funktionellt sätt räknas grannarna ut på samma sätt, fast koden skrivs annorlunda. Se figur 4 och 5, som visar quadtree:et samt alla kvadraters grannar (röda linjer mellan kvadrater), för vardera karta. Vidare förklaras kartorna i 3.4.4 på sida 14.



Figur 4: Labyrintkartan, uppdelad i kvadrater med linjer mellan grannar. (Skärmbilden på arbete i Unity)



Figur 5: Pelarekartan, uppdelad i kvadrater med linjer mellan grannar. (Skärmbilden på arbete i Unity)

2.4. Vägsökningsalgoritmer

Basfallet som de andra algoritmerna jämförs med är Unitys egna metod, NavMesh, som används för att den är inbyggd i Unity och mer generaliserat än de andra. För att jämföra Unitys generella lösning med en algoritm specifikt utvecklad för hastighetsbaserad AI-rörelse använder denna studie huvudsakligen Phi* algoritmen (uttalat Phi-Star), en modifierad A*. Vidare används A* och Theta*.

Unity NavMesh har en inbyggd funktion för vägsökning, CalculatePath. CalculatePath är en intern funktion och det är inte allmän information om hur den fungerar. Testmiljöns tidigaste implementation av A* fick ett liknande nog resultat, när skillnaderna var inräknade, att det är rimligt att anta att CalculatePath använder någon variation av A* för att hitta den kortaste vägen.

A* (uttalat A-Star) är en av de mest använda vägsökningsalgoritmerna och är den vanligaste vägsökningsalgoritmen som används inom dataspel (Lawande, Jasmine, Anbarasi, & Izhar 2022; Hu, gen Wan & Yu 2012). Exempel på hur den används och fungerar beskrivs av Yang & Cai (2023) och Zengzhen, Hongjian, & Chonghua (2023). A* är i sin grund en variation av Dijkstras algoritm fast att den är riktad enligt en heuristisk metod. Att A* använder en heuristisk metod innebär att algoritmen estimerar kostnaden mellan en punkt och målet. Yang & Cai (2023) skriver också att A* är den mest effektiva sökalgoritmen för att hitta den kortaste vägen i en miljö med hinder som inte förflyttas. A* söker efter den billigaste vägen enligt vikter, då vikterna oftast representerar distans blir det den kortaste vägen. Den fungerar genom att alla punkter har ett värde som beskriver kostnaden att gå till den punkten. Kostnaden är summan av distansen mellan punkten och startpunkten + estimerad distans mellan punkten och slutpunkten. Den senaste punkten som läggs till i vägen kollar då alla punkter som är grannar till den punkten och jämför deras kostnad. Grannen med minst kostnad bestäms vara nästa punkt agenten går till, sedan blir den grannen punkten som kollar dess grannar (Zengzhen, Hongjian, & Chonghua 2023).

Eftersom A* är snabb och garanterad att hitta den kortaste vägen genom ett nod-system är det oftast den första metoden som används. Kvaliteten av en väg som genereras av A* är baserad på den data den använder. De flesta navigationssystem använder data genom ett grannsystem som inte nödvändigtvis garanterar den kortaste vägen (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021).

A* är grunden till Theta*. Theta* fungerar huvudsakligen på samma sätt, skillnaden är att istället för att endast kolla bland grann-punkter som A* gör, kollar alla punkter inom “line of sight”. En punkt är inom “line of sight” om det går att dra en rak linje mellan punkten som kollar och punkten som kollas (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021). Theta* är grunden till Phi*. Phi* börjar som Theta*, istället för att kolla alla grannar, kollar en punkt alla punkter som kan ses från den. Skillnaden är att den kollar punkter endast inom en vinkel (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021).

3. Problemformulering

I många spel används artificiell intelligens (AI) för att kontrollera entiteter som inte kontrolleras av spelaren. Dessa AI-kontrollerande entiteter (agenter) kan uppfylla olika krav beroende på spel och situation. Oftast används agenter för att slåss mot spelaren (fiender), för att hjälpa spelaren slåss (kamrater) eller för att populera spelvärlden på andra sätt, till exempel genom att sälja varor (NPC). Oberoende av en agents roll, finns det många spel där de ej har statiska positioner.

Det finns olika sätt att uppnå rörelse för dessa agenter beroende på vad som behövs, exempelvis kan deras position interpoleras, förflyttas med jämn hastighet, mellan förutbestämda positioner. Hur komplicerat det är att hantera förflyttningen beror på spelarfrihet/rörelsefrihet samt hur adaptiva agenterna ska vara. Ifall spelaren kan putta objekt framför en agent, behöver det vara bestämt om agenten ska gå in i objektet eller undvika det. I situationen som objektet ska undvikas behövs en algoritm, exempelvis en vägsökningsalgoritm. Vägsökningsalgoritmer söker efter vilken väg en agent ska ta, beroende på olika krav såsom, till exempel, att hålla en specifik distans mellan agenten och hinder. Andra exempel på ett krav, i ett fall utanför spel, är att undvika vägar med trafikljus, eller att bara använda högersvängar.

Det är vanligast i spel att dessa algoritmer söker efter den *kortaste* vägen (Lawande, Jasmine, Anbarasi, & Izhar 2022; Hu, gen Wan & Yu 2012). Utanför spel, inom bland annat robotik samt verkliga scenarion, söker dessa algoritmer ofta istället efter den *snabbaste* vägen (Rathnayake, Wijesekara, & Samaranayake 2023). Det kan vara vägar med unika kvaliteter, exempelvis vägar som tillåter högre hastigheter, eller vägar med mindre trafik eller stoppljus, oberoende av längd eller tid. En kort väg med mycket trafik tar längre tid att åka än en längre väg utan trafik. Vägen som kan hittas beror därför på miljön agenten befinner sig i, fast även på hur datan för miljön samlas in, alltså vilket dataset som används.

Om dessa algoritmer kan användas inom spel är värt att undersöka, för att göra AI mer realistiskt. Rogers, Karaosmanoglu, Altmeyer, Suarez & E. Nacke (2022) skriver att realism är en grundläggande och fundamental del av spel, och därför är mer realistiska vägsökningsalgoritmer rimligt att forska om.

Detta leder då till en frågeställning: *Vilken kombination av en vägsökningsalgoritm - utav A*, Theta*, Phi* och Unitys inbyggda - samt ett dataset - utav NavMesh och Quadtree - hittar den snabbaste vägen mellan två punkter*.

För att besvara detta testas fyra algoritmer, två som är optimerade för att söka vägar för hastighetsbaserade agenter, ett basfall, och en av de vanligaste vägsökningsalgoritmerna. Även datan för miljön samlas in på två olika sätt. Då testen är kvantitativa är utdatan också det, vilket gör att komparation kan utföras för att se vilken algoritm som hittar den snabbaste vägen, då metoden komparation handlar om jämförelser (Ejvegård 2009, s. 44). Ejvegård (2009) skriver även att komparation behöver rimligt jämförbara enheter och att jämförelser skall ske på generaliserade saker. Då testen samt utdatan är kvantitativ, kan komparation ske utan problem (Ejvegård 2009, s. 38).

3.1. Urval av algoritmer samt dataset

Studiens algoritmer valdes för att ge fyra olika inblickar på pathfinding. Unitys inbyggda är en generell och snabb lösning som ger en effektiv väg om den kortaste sökes, fast ej angående den snabbaste vägen. Phi* är vald för att den är optimerad för att hitta en icke-skarp väg för att undvika hinder, och för att den används mycket inom robotik. (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021).

A* valdes att vara med för att det är en av de vanligaste vägsökningsalgoritmerna, för att den valdes göra i början av arbetet för att uppfriska minnet om hur vägsökningsalgoritmer programmeras, samt för att Phi* är en variation av den. Theta* valdes huvudsakligen för att det är en enkel variation av Phi* kodmässigt, och kunde därför programmeras in snabbt när Phi* var klar.

Först valdes Dijkstras bort för att A* är en variation av den som oftast är snabbare och hittar samma väg (Yang & Cai 2023, Zengzhen, Hongjian, & Chonghua 2023). Suurballe, & Tarjan (1984) skriver att det bästa fallet för Suurballes algoritm är två iterationer av Dijkstras. Det är varför det valdes att inte ha med den eftersom det redan hade valts att ha med A*. Bellman-Ford algoritmen är en variation av Dijkstra som kan hantera punkter med negativ vikt (Jin Y. Yen, 1970). Eftersom negativa vikter inte är relevanta för denna studie, och den annars är en variation av Dijkstra, valdes det att inte ha med den.

Bhattacharya, P., & L. Gavrilova, M. (2007) skriver om att använda Voronoi diagram för vägsökning. I deras arbete skriver de om hur de använder Voronoi för att dela upp miljön och skapa ett dataset. De beskriver att deras metod, och metoder som tidigare har försökt använda dessa diagram, inte blir optimala för att hitta den kortaste vägen. Deras lösning till det var att iterera över vägen ett flertal gånger och skära bort mer och mer, fast att den fortfarande inte är passande efter det. Vid det laget skapar de andra punkter för att göra skarpa hörn till kurvade. Det beslutades att all bearbetning de utför skulle antingen ta för lång tid i relation till det övriga i arbetet, eller ändå resultera i ett sämre resultat än de andra algoritmerna som valdes att användas.

Antalet algoritmer begränsas på grund av tidsbegränsningar för arbetet. I fallet där det finns mycket extra tid, är en rimlig utveckling av arbetet att implementera och testa flera algoritmer.

3.2. Projektmiljö

Testmiljön är skapad i Unity 3D och är skapad för att testa de olika algoritmerna. För att generera användbar data byggs miljön för att testa algoritmernas kapacitet att hitta målet, undvika hinder och skapa en väg som tillåter agenten att röra sig med maximal hastighet. Algoritmerna ska vara komplexa nog att hantera olika miljöer utan att krascha in i hinder. Detta testas genom att använda två olika kartor där målpositionen randomiseras 100 gånger.

3.3. Agentutveckling

Agenten för experimentet är hastighetsbaserad och rör sig med samma logik oberoende på

vilken algoritm som använts för att generera vägen den följer. Vägsökningsalgoritmerna som valts och bearbetningen av vägarna som genereras är utformade för att skapa optimala vägar för en hastighetsbaserad agent.

Den optimala vägen för en hastighetsbaserad agent är den vägen som tillåter högst hastighet kopplat med den kortaste vägen där den kan hålla den hastigheten (Rathnayake, Wijesekara, & Samaranayake 2023). Det blir då ett antal variabler som vägsökningsalgoritmerna behöver ta hänsyn till. Den största variabeln är tröghet eller agentens motstånd att ändra sin nuvarande hastighet. Detta blir ett problem vid svängar där stora riktningsändringar leder till att agenten överskjuter svängen. För att minska risken för att det händer byggs svängarna på ett sätt som tillåter en gradvis ändring av riktning genom att skapa bredare kurvor som mer efterliknar agentens rörelse. Då behöver agenten inte heller bromsa in lika mycket (Rathnayake, Wijesekara, & Samaranayake 2023).

3.4. Metodbeskrivning

Målet med studien var att testa olika algoritmer för att ta reda på vilken algoritm som kan hitta den snabbaste vägen mellan två punkter. För att uppnå detta mål skapades två simulerade miljöer i Unity3D där agenten hittade olika vägar med hjälp av sju olika implementationer av vägsökningsalgoritmer. Algoritmerna var byggda med ett fokus på behållning av hastighet och släta kurvor. För varje väg som skapades mättes tiden det tog för agenten att nå målet.

Alla algoritmer testades 100 gånger i samma miljö, med olika målpunkter. Efter att all data samlades in, sammanställdes den i grafer och tabeller för analys. När agenten kom i mål sparades tiden det tog den att navigera vägen algoritmen skapade. Vid de fall där agenten inte nådde målet returnerades ett möjligt resultat. Resultaten användes för att skapa en tabell som beskriver chansen för alla algoritmer att skapa en väg som inte misslyckas. Slutligen beräknades chansen att en väg är bättre än basfallet (NavMesh), detta representeras genom ett genomsnittligt värde. Allt detta för att ta reda på vilken kombination som oftast är snabbast, på ett trovärdigt sätt.

För vilka vägsökningsalgoritmerna och dataset som är implementerade se 3.2.1 Urval på sida 10-11. Mätvärdena är tiden det tar för agenten att navigera till olika mål från startpunkten i vägarna som algoritmerna skapade. Dessa värden, av olika kombinationer, jämförs. Denna form av experiment är populärt inom robotik där vägens hastighet bestäms bland annat av skarpa svängar, antal längre sträckor av acceleration samt längd av väg (Rathnayake, Wijesekara, & Samaranayake 2023).

För att studiens resultat ska vara användbart krävs det att agenten har en rimlig hastighetsbaserad rörelse. Samt att agenten kan röra sig längs en väg baserat på riktningar istället för att gå från punkt till punkt. Detta är även hur liknande studier har studerat effektiviteten av vägsökning för simulationer av robotar som ska kunna utforska och röra sig genom verklig terräng (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021).

3.4.1. Metoddiskussion

När det kom till urval fanns det ett antal algoritmer och dataset som valdes bort antingen i mån av tid eller rimlighet, se 3.1 Urval, sida 10-11, för mer detaljer. Ett annat stort val var att

inte räkna med höjdskillnader, som AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) och Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022) hade i sina artiklar.

Det hade varit intressant med fler varianter av agentrörelse, insamling av navigationsdata, och vägsökningsalgoritmer. Dock, givet studiens urval, är det ändå säkert att det som valts kommer att kunna besvara frågeställningen till den grad att resultatet är användbart, pålitligt och att andra tillgängliga vägsökningsalgoritmer som vi har hittat inte skapar en snabbare väg än de som testas.

4. Resultat

Detta delkapitel är uppdelat i 4.1 Genomförande som diskuterar genomförandet och uppbyggnaden av testen, 4.2 som presenterar resultatet i grafer, samt 4.3 som analyserar datan som graferna visar. Slutligen presenteras slutsatsen i 4.4.

4.1. Genomförande

Experimentet genomfördes genom att använda en fröad slumpgenerator för att få fram 100 slumpmässiga fast fortfarande rimliga positioner på NavMesh-utrymmet. Detta gör att alla algoritmer använder samma positioner att gå emellan. Sedan simuleras alla 100 stycken vägar genom att tillåta agenten att gå mellan startpunkten och den slumpmässiga slutpunkten med en ökad tidsskala, så att flera sekunder i experimentet händer under en verklig sekund. En ökad tidsskala har ingen påverkan på resultatet.

Om agenten tog över 100 sekunder på sig att gå en väg antogs det att agenten fastnade, då avslutades den vägen, och tiden som sparades för den vägen blev noll. När alla 100 vägar var klara började experimentet om med nästa algoritm. Denna procedur gjordes i båda miljöerna. All data skrevs ut till textdokument som sedan bearbetades i Google Sheets.

4.1.1. Bearbetning av väg

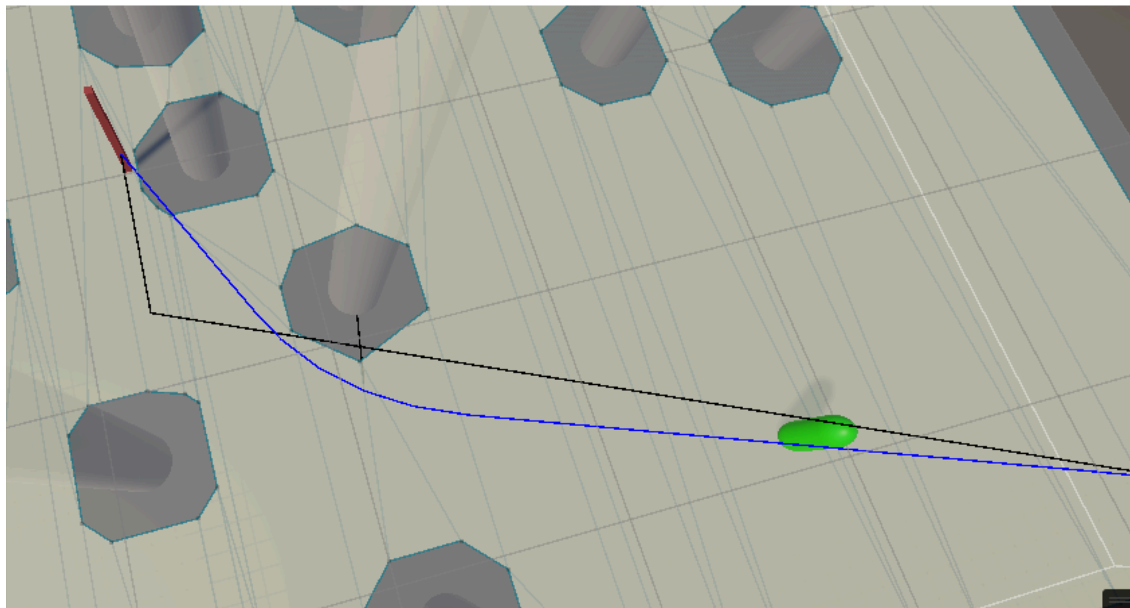
Då det finns en mängd olika sätt att generera en väg mellan start och slut är det inte nödvändigtvis sant att den vägen är bra för en hastighetsbaserad agent att följa. Därför segmenteras vägen så att den enklare kan bearbetas, för att se till att agenten följer den. Detta skapar dock enbart många raka linjer mellan vägsökningspunkter vilket inte är optimalt för svängar, särskilt skarpa svängar. För att mer korrekt skapa en väg som agenten kan följa genereras Bézier-kurvor mellan punkter. Notera att bearbetning av vägen ej används för Unitys inbyggda vägsökningsalgoritm. Detta görs för att den ska vara ett basfall att jämföra de andra mot.

Metoden för att generera Bézier-kurvor är baserat på “Bézier Curves-Based Optimal Trajectory Design for Multirotor UAVs with Any-Angle Pathfinding Algorithms” (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021). Där den bästa metoden enligt deras tester var att skapa mittpunkter mellan vägsökningspunkter där vägen är helt rak och sedan Bézier-kurvor mellan mittpunkter när vägen går till nästa punktpar. Detta är dock otillräckligt för denna studies tester då punktpositioner är mindre fria än i den källan och en rimlig distans från hinder inte alltid är givet.

För att bemöta dessa problem testas alla vägsegment genom att se om segmentet åker igenom en vägg, då försöker algoritmen placera ut påhittade punkter (fantompunkter) som hanteras på samma sätt som de originella punkterna. Fantompunkterna skapas endast ifall de ej finns hinder mellan den fantompunkten och startpunkten av segmentet som åkte igenom en vägg.

Om fantompunkten hittar ett nytt sätt att ta sig till punkten efter slutpunkten i segmentet som åkte igenom en vägg, byter algoritmen ut den slutpunkten till fantompunkten, då den slutpunkten låg för nära en vägg för att användas. Om fantompunkten istället enbart ser

segmentets slutpunkt läggs den till på vägen mellan segment-punkterna, så att agenten tar en bredare kurva längs den originella vägen och undviker hinder mer effektivt. Se figur 6 för resultatet av bearbetningsprocessen. För en mer detaljerad förklaring se också figur 9 på sida 24.



Figur 6: Bilden visar hur vägen byts ut genom Bézier kurvor. Den originella vägen med den skarpa kurvan är den svarta linjen. Den blåa linjen är vägen efter bearbetning. Starten är utanför bilden till höger, den röda pelaren är målet och den gröna kapseln är agenten (Skärmbilden på arbete i Unity)

4.1.2. Agentens rörelse

I denna studies experiment rör sig agenten med hjälp av Unitys Rigidbody system som är ett enkelt system för att tillåta hastighetsbaserad rörelse. I sin grund rör sig agenten genom att applicera kraft ((Newton*tid)/kg) i en riktning (Unity Manual 2025). Riktningen bestäms i denna studie genom att jämföra agentens nuvarande riktning och hastighet med vägens nästa punkt och räkna ut hur agenten bör applicera kraft för att få nuvarande riktning att ändras till det önskvärda målet. Det önskvärda målet är inte direkt självklart, då att följa vägen enbart tills nästa steg antingen aldrig tillåter hög hastighet eller kastar agenten av vägen vid skarpa svängar.

För att förbättra detta följer agenten i denna studie istället efter medelvärdet av flera punkters position längs vägen samt rör sig mot nästa punkt, vilket gav ett bättre resultat under funktionstester under artefaktskapandet. Hur många punkter som tittas på beror på vinkelskillnaden mellan segment av vägen vilket gör att vägdelar med större kurvor använder flera punkter i medelvärdet. Detta gör att vid längre sträckor och mjuka kurvor tillåter algoritmen högre hastigheter och vid skarpa svängar används färre punkter vilket gör att agenten saktar in för att den inte ska överskrida vägen den ska följa.

Hur agenten bör röra sig är oftast beroende på agentens omgivning. I denna studie, för att öka den maximala hastigheten och öka noggrannheten kring hörn och smala korridorer, justeras automatiskt punkten som agenten rör sig mot. Detta sker dynamiskt mellan nästa

punkt på vägen och en medelpunkt mellan flera av de nästa punkterna på vägen. Detta ändras huvudsakligen beroende på agentens distans till den närmaste väggen, och gör att agenten inte krockar med en vägg.

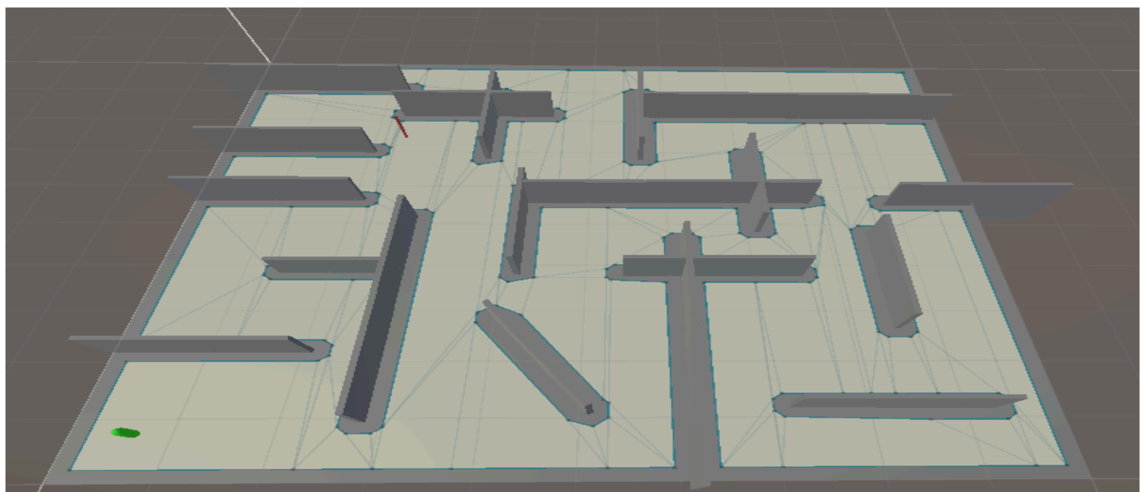
I jämförelse med Unitys inbyggda agentrörelse är den som används i detta test enklare, då Unitys rörelsesystem ska hantera mer komplexa och dynamiska miljöer såsom till exempel interaktion med spelare eller hinder-undvikande. Då detta experiment testar för konstruktion av vägar för optimal hastighetsbaserad rörelse leder det till mer komplex vägsökning istället för agentrörelse.

4.1.3. Miljödesign

Experimentet i denna studie går ut på att alla algoritmer ska sättas i en miljö där den optimala vägen är oklar och som en hastighetsbaserad agent kan ha problem att ta sig igenom. NavMesh trianglar ritas i denna studie med en agentbredd av 1.5 Unity-enheter (1 Unity-enhet är 1 meter) så att ingen navigationspunkt är för nära en vägg.

För att göra resultatet mer trovärdigt genomfördes testet på två olika kartor. En byggd som en labyrint, den andra ett plan med flera pelare utplacerade, som agerade som hinder. Inspiration för labyrintkartan kommer från micromouse-tävlingar (Rathnayake, Wijesekara, & Samaranayake 2023). Inspiration för pelarekartan kommer ifrån "Bézier Curves-Based Optimal Trajectory Design for Multirotor UAVs with Any-Angle Pathfinding Algorithms" (AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu 2021).

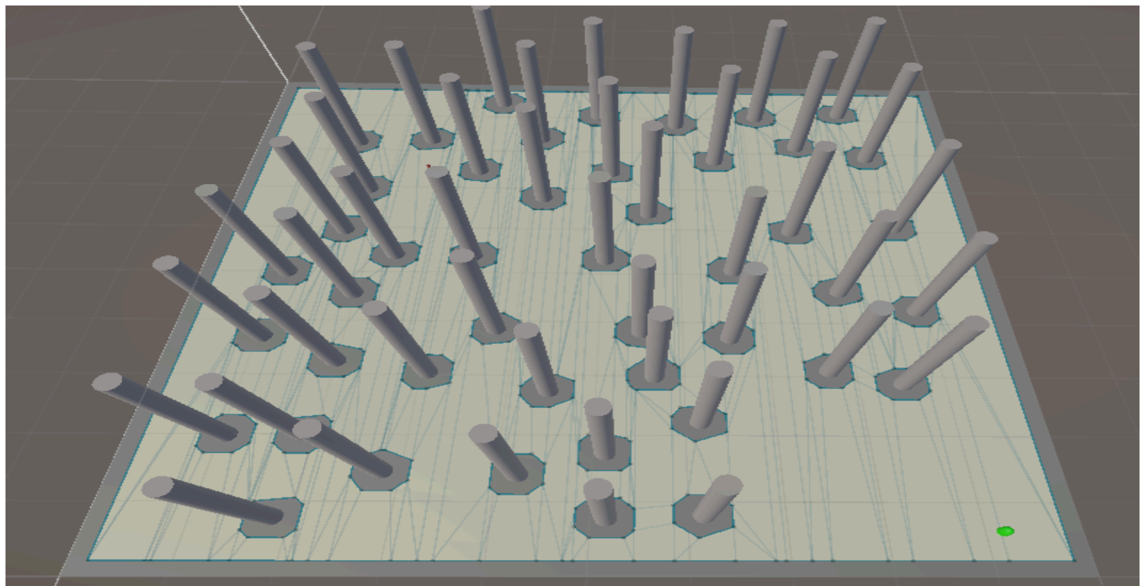
Labyrintkartan ska likna och representera en klassisk labyrint, många långa korridorer med raka väggar som hindrar agenten att åka rakt i många fall. Öppningarna är breda nog för att låta agenten svänga igenom. Ett realistiskt fall för en agent i spel eller verklighet, om väggarna tänks som byggnader. Denna karta testar hur agenten och vägsökningen hanterar en segmenterad miljö där vägarna är utplacerade med tanke bakom utplaceringen.



Figur 7: Labyrintkartan. (Skärmbilden på arbete i Unity)

Pelarekartan ska vara mer slumpmässig med många pelare spontant utspridda på ett bräde. Miljön ska testa hur vägsökningen och agenten hanterar en mer öppen miljö med högre risk att krascha och fler mindre hinder jämfört med ett fåtal stora hinder. En sådan här miljö kan

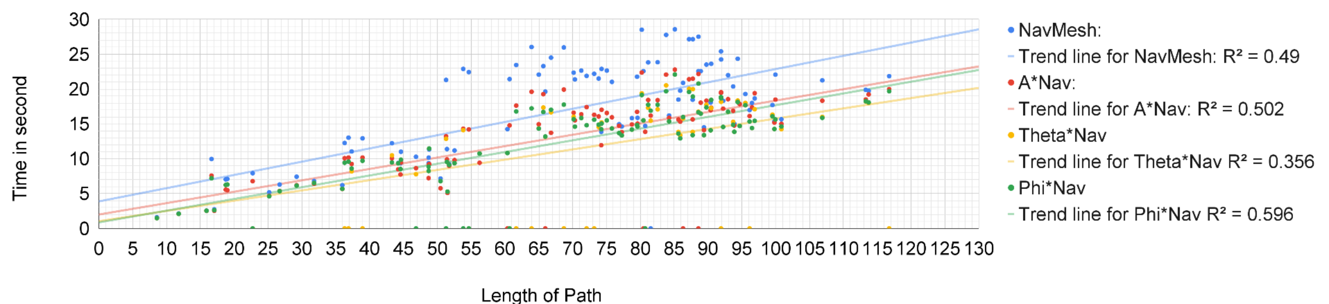
också komma upp i spel och verklighet, exempelvis i en skog där pelarna är träd.

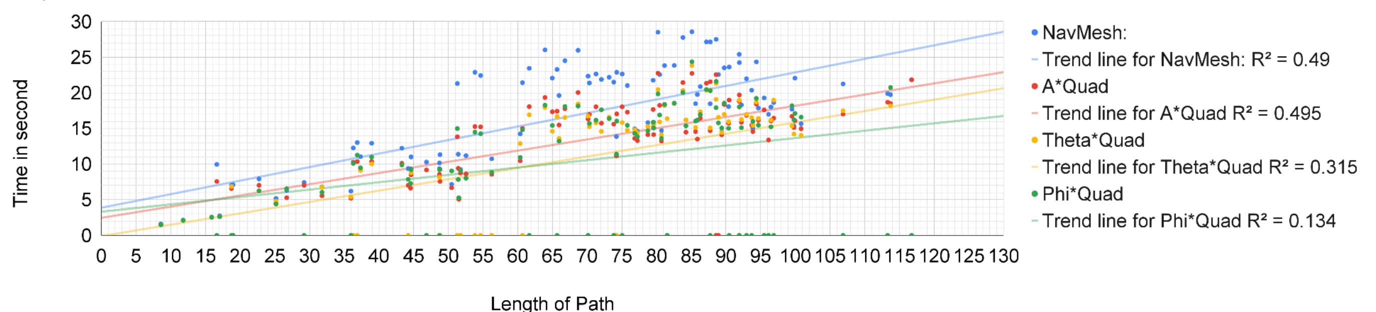


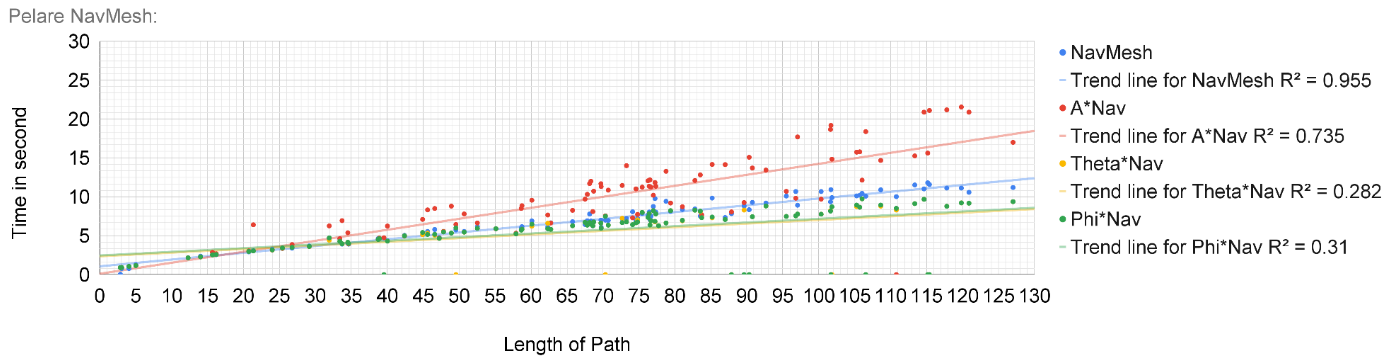
Figur 8: Pelarekartan. (Skärmbilden på arbete i Unity)

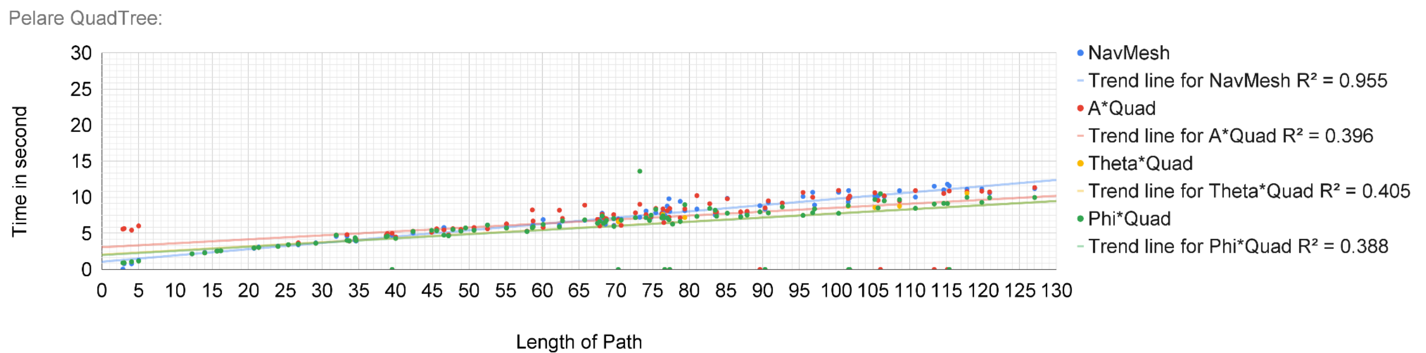
4.2. Presentation av resultat

För att visualisera resultatet används ett spridningsdiagram som visar tiden det tar för agenten att nå målet beroende på hur långt bort målet är från spelaren. Vidare visas trendlinjer och R^2 värdet. All data kan ses i appendix A.









Graferna Labyrint NavMesh, Labyrint QuadTree, Pelare NavMesh, Pelare QuadTree: Totala resultatet av alla experiment. X-axeln är längden mellan startposition och målposition via fågelvägen. Y-axeln är tid i sekunder för agenten att ta sig i mål, där tiden är noll betyder det att agenten inte tog sig till målet. Graferna är uppdelade i de olika kartorna, och i dataset som användes.

4.3. Analys av resultat

Studiens frågeställning var: *Vilken kombination av en vägsökningsalgoritm - utav A*, Theta*, Phi* och Unitys inbyggda - samt ett dataset - utav NavMesh och Quadtree - hittar den snabbaste vägen mellan två punkter*. För att svara på den jämförs de nya algoritmernas resultat med Unitys inbyggda som ger en säker och kort väg till ett mål. För att hjälpa den jämförelsen omvandlas datan som graferna visar till tabeller som visar den procentuella chansen för kombinationer att misslyckas, förbättras och den genomsnittliga förbättringen.

Labyrint	NavMesh	A*Nav	A*Quad	Theta*Nav	Theta*Quad	Phi*Nav	Phi*Quad
Chans att misslyckas	1.00%	5.00%	4.00%	15.00%	23.00%	7.00%	24.00%
Chans att förbättra	0.00%	93.00%	96.00%	81.00%	71.00%	89.00%	66.00%
Genomsnittlig förbättring	0%	21.24%	22.48%	26.00%	24.68%	26.66%	25.79%

Pelare	NavMesh	A*Nav	A*Quad	Theta*Nav	Theta*Quad	Phi*Nav	Phi*Quad

Chans att misslyckas	0.00%	1.00%	4.00%	10.00%	7.00%	8.00%	8.00%
Chans att förbättra	0.00%	14.00%	30.00%	67.00%	64.00%	68.00%	62.00%
Genomsnittlig förbättring	0%	3.89%	7.87%	11.44%	10.48%	11.35%	10.22%

Tabell Labyrint och Pelare: Labyrint/Pelare statistik. För alla experiment visas chansen att agenten misslyckas, chansen att en algoritm är bättre än basfallet (NavMesh) och procentuell förbättring i tid jämfört med basfallet, där dess tid förbättras.

Graferna visar att distansen mellan spelare och mål har en svag korrelation till genomsnittlig väglängd då antal hinder mellan start och slut inte representeras. Korrelationen blir svag eftersom distansen som räknas är fågeldistansen mellan start och slut, och eftersom hinder endast kan gås runt, blir vägarna algoritmerna skapar oftast längre än fågeldistansen.

Vidare visar graferna generellt att ju längre vägen är, desto mer tid vinner hastighetsbaserade algoritmer. Vägar på pelarekartan är kortare och mer direkta än på labyrintkartan, vilket leder till mindre skillnad i genomsnittlig tid mellan NavMesh (algoritm) och andra algoritmer. Anledningen till att A*Nav har en högre tid jämfört med andra algoritmer i pelarekartan beror på hur Unity bygger NavMesh-trianglarna.

Tabellerna visar att Unitys inbyggda NavMesh-algoritm har störst chans att skapa en väg som tillåter agenten att ta sig till mål men att alla andra algoritmer har ofta en bra chans att förbättra tiden på NavMesh. Detta är mer uppenbart vid labyrintexperimentet då svängar är mer extrema och den rimligaste rutten är längre. Statistiken säger generellt att de nya metoderna har större chans att misslyckas men i en majoritet av de fall där de lyckades har de en bättre tid än basfallet.

I labyrintkartan är den sammanlagda chansen att misslyckas mindre när NavMesh används som dataset. I pelarekartan är chansen att misslyckas densamma, oberoende av dataset. Därmed är det rimligt att anta, med tanke på hur quads:en genereras just nu, att NavMesh är bättre. Fortsatt från det, har Phi* sammanlagt den bästa chansen att förbättras, och har ändå en rimligt låg chans att misslyckas. Vidare har Phi* den sammanlagt högsta genomsnittliga förbättringen.

Resultatet visar vidare att NavMesh är den mest pålitliga, med en 1% och 0% chans att misslyckas, beroende på karta. För labyrintkartan är NavMesh det bästa datasetet, då den totala chansen att misslyckas är mindre. För pelarekartan är chansen att misslyckas densamma för båda dataseten. Dock, för karta ett är A*Quad den enda som alltid är bättre än NavMesh när den inte misslyckas. Phi*Nav har den totalt största chansen att vara bättre, och har en relativt låg chans att misslyckas.

4.4. Slutsats

Resultatet visar att för agenter med hastighetsbaserad rörelse finns det nya och bättre algoritmer än Unitys inbyggda vägsökningsalgoritm. Algoritmerna är för tillfället inte tillräckligt pålitliga för realistisk applikation i spel då de har en för hög chans att misslyckas.

Dock visar resultatet klart att med mer finslipning och agentlogik kan agentens väg starkt förbättras. Att agenten var hastighetsbaserad gjorde att rörelsen den gjorde var mer flytande och realistisk, vilket visar att om denna metod används i ett spel kan spelkänslan förbättras.

Utöver det kommer mycket av den realistiska känslan från att agenten rör sig med kraftvektorer mot vägen och bygger hastighet baserat på raka sträckor. Att följa vägen direkt minskar känslan av flytande rörelse, och minskar därmed realismen.

Utifrån studiens frågeställning: *Vilken kombination av en vägsökningsalgoritm - utav A*, Theta*, Phi* och Unitys inbyggda - samt ett dataset - utav NavMesh och Quadtree - hittar den snabbaste vägen mellan två punkter.*, var kombinationen som oftast hittade den snabbaste vägen Phi* som använde NavMesh som dataset. Dock är det inte alltid bäst att använda den, eftersom den i nuvarande skick har en chans att misslyckas.

5. Sammanfattning

Denna studie skapar en miljö för att testa kvaliteten av olika metoder att generera navigationsdata och att använda den datan för att hitta den snabbaste möjliga vägen för en hastighetsbaserad agent. För att uppnå ett rimligt och trovärdigt resultat gjordes ett noggrant urval av vilka algoritmer som testas och hur navigationsdata genereras.

Algoritmerna och dataset som framgick av urvalet presenteras i bakgrunden. Vidare beskrivs hur källorna använder en annan teknik för att göra vägar mer passande för hastighetsbaserade agenter, Bézier-kurvor. Dessa kurvor används för att undvika svängar som var för skarpa, som saktar ner agenten.

För att jämföra algoritmerna användes spelmotorn Unity då den har en effektiv och rimlig fysiksimulering. Vidare har den en inbyggd metod för navigationsdata och vägsökning som ger ett bra basfall som andra algoritmer kan jämföras med. Hur Unity skapar navigationsdata och vilken algoritm som används för vägsökning är ej tillgängligt fast resultatet som den ger är ändå användbart.

Algoritmerna som implementerades var A*, Theta* och Phi*. En alternativ metod för att generera navigationsdata implementerades också: quadtree. Detta ledde då till frågeställningen: *Vilken kombination av en vägsökningsalgoritm - utav A*, Theta*, Phi* och Unitys inbyggda - samt ett dataset - utav NavMesh och Quadtree - hittar den snabbaste vägen mellan två punkter.*

För att testa vilken kombination som var snabbast skapades två testmiljöer med olika typer av hinder och en hastighetsbaserad agent som rörde sig längs vägen så gott den kunde. För att ge ett användbart resultat hade agenten ingen komplex logik utan rörde sig endast enligt vägen. Hastigheten agenten rörde sig i bestämdes av skarphet av svängar och distans från hinder, vilket ledde till att den kunde följa vägen mer pålitligt.

Resultatet av experimenten visade att de implementerade algoritmerna hade en ökad risk att misslyckas då den genererade vägen maximerar hastigheten. Dock i de fall där de inte misslyckades var en majoritet av vägarna snabbare än basfallet. I slutändan var Phi* mycket snabbare än Unitys inbyggda vägsökningsmetod och måttligt snabbare än de andra algoritmerna som testades. Den bästa kombinationen var Phi* med NavMesh som dataset.

5.1. Diskussion

Att skapa rimliga vägar för agenter som inte nödvändigtvis följer sin väg och som är byggda för att hantera höga hastigheter är oftast ett problem inom robotik. I praktiken har detta arbete hanterat att skapa en förenklad simulator av hur en robot skulle röra sig genom en komplex miljö och vad den optimala vägsökningsalgoritmen är för en höghastighetsrobot.

Denna studie visar att Phi* hittar en väg som efter bearbetning har sammanlagt högst chans att vara snabb och sammanlagt högst tidsparning. Detta kan användas för vidare studier som ska optimera rörelse av AI antingen inom realistiska simulationer (bilar, flygplan, drönare) och potentiellt inom robotik där navigationsdata är känt. I slutsatsen nämndes det dock att Phi* inte alltid är den bästa att använda, för att agenten inte alltid kunde nå målet. Vi spekulerar att ifall agenten hade bättre och/eller egen kod för att undvika hinder, skulle Phi* vara bättre nästintill alltid.

Även, teoretiskt sett, kan vägsökningsalgoritmerna i denna studie användas med en icke hastighetsbaserad agent där den enbart följer vägen. Detta skulle dock vara ett misstag då de hastighetsbaserade vägarna alla är längre än Unitys inbyggda vägsökningsalgoritm och därmed långsammare.

5.1.1. Relation till tidigare forskning

Detta avsnitt handlar om hur studien relaterar till forskningen som artikeln använde och refererar till. 5.1.1.1 handlar om Bézier-kurvor. 5.1.1.2 handlar om algoritmerna och datasetet quadtree.

5.1.1.1. Bézier-kurvor

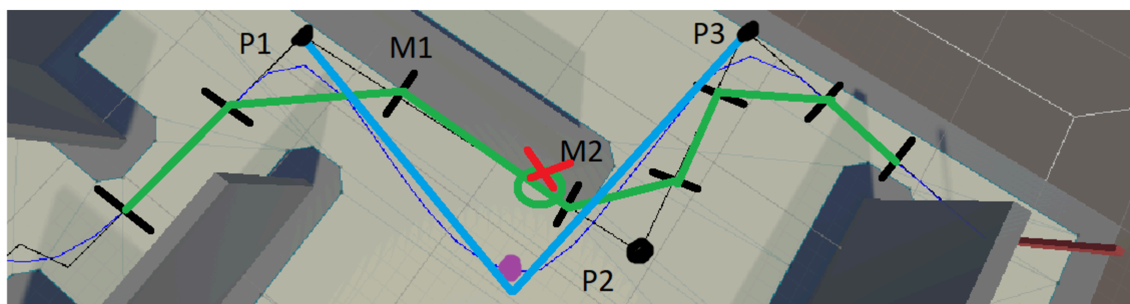
Både Rathnayake, Wijesekara, & Samaranayake (2023) och AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) använder Bézier-kurvor i sina studier, och det är dem som denna studie tog inspirationen från. Rathnayake, Wijesekara, & Samaranayake (2023) använder dem för att förbättra 90 och 180 graders svängar, där roboten annars tar extra lång tid att navigera. AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) testade fyra olika sätt att

skapa Bézier-kurvor, och deras resultat visade att det var bäst att dela upp varje vägsegment i tre delar, genom att sätta ut flera punkter på 25% och 75% av segmentet. Sedan gjorde de Bézier-kurvor mellan punken 75% in i ett segment och punkten 25% in i nästa segment.

Det beslutades att göra på ett liknande sätt, men att antalet mittpunkter som sätts ut beror på längden av vägen med ett minimum av 2, och alltid jämnt fördelat. Denna modifikation skapades efter testning, då det framgick att vid längre segment behövdes flera mittpunkter för att svängarna inte skulle vara för skarpa. Vidare hjälper mittpunkterna agenten att undvika väggar. I bilden nedan kan detta observeras. Bézier-kurvor är bra för att de förkortar vägen mellan punkter, som kan ses på bilden nedan mellan M1 och det svarta strecket innan.

Originellt testades det att implementera Bézier-kurvor på samma sätt som AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) gjorde för att få deras bästa resultat. Dock genom icke-formell testning under skapandet av denna artefakt märktes det att deras implementation hade problem när det var för stora skillnader i längd på vägsegment, vilket ledde till utvecklingen av implementationen som används nu. Huvudsakligen fanns problemet när agenten skulle gå runt en vägg, vilket inte förekom i studien av AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021).

Figur 9 visar hur algoritmen bygger vägen. Efter att den originella vägen är skapad (tunn svart linje mellan kontrollpunkter), delas alla segment upp i mittpunkter (svarta tvärsnittslinjer). Mellan de mittpunkterna "kastas" en "boll" (grön linje mellan tvärsnitt, samt cirkel med kross) för att se om en av mittpunkterna är för nära väggen. Det röda krosset ska visa att "bollen" träffar väggen, mellan M1 och M2, därmed placeras en fantompunkt ut (lila punkt mellan blåa linjer mellan P1 och P2). Direkt kollas det om P1 till fantompunkten är en giltig väg (blå linje mellan P1 och fantompunkten), då den är det läggs fantompunkten till på vägen. Därmed kollas det om en väg kan skapas mellan fantompunkten och P3, ifall det är möjligt tas P2 bort från vägen, och ett segment skapas mellan fantompunkten och P3. Hade det ej varit giltigt hade ett segment skapats mellan fantompunkten och P2. Därefter skapas Bézier-kurvor (tunn blå linje).



Figur 9: Skärmbilden på arbete i Unity, bearbetat i Microsoft Paint, beskrivning ovan

5.1.1.2. Algoritmer och quadtree

Studiens huvudkälla för Theta* och Phi* är AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021). Dock förklarar de inte hur de använder eller skapar algoritmerna, endast hur de fungerar. På grund av detta kan en jämförelse inte enkelt göras, istället görs en uppskattning. Den mest sannolika skillnaden är i hur algoritmerna är uppbyggda, vi kom på två sätt att göra det på.

För Phi* är det ena sättet att en “line of sight check” händer mellan alla punkter inom synvinkeln. Det andra sättet, vilket också är sättet denna studie använder, är att köra checken på alla punkter, och först när nästa punkt ska väljas kollas det ifall den punkten är inom synvinkeln av punkten som kollar. Den skillnaden tillät algoritmen att ändra synvinkeln ifall den inte hittade en väg till målet, så att den alltid skulle hitta en väg till målet. Vid det laget, eftersom en “line of sight check” hade gjorts mellan alla punkter, kunde Theta* enkelt implementeras med en bool.

Oavsett hur AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) än byggde algoritmen, fick dem samma svar som detta test, att Phi* gav det snabbaste resultatet. Dock, deras resultat visar att deras implementation av Phi* var snabbast tillsammans med en annan, en variant av Theta* som heter lazy-Theta*. Vidare har deras version av Phi* inte en chans att misslyckas.

Zengzhen, Hongjian, & Chonghua (2023) och Yang & Cai (2023) skriver båda om A*. Dock beskriver de båda A* genom pseudokod. Därför användes en blogg: Swift (2017) som gick igenom A* i mer detalj, dock i ett annat språk, denna källa var lämplig nog då den beskrev A* likt nog de vetenskapliga artiklarna. A* var också den första algoritmen som skrevs, vilket är anledningen till att en tydligare guide för den användes, vilket sedan inte behövdes för de andra algoritmerna. Den huvudsakliga skillnaden mellan A* från den bloggen och algoritmen som användes var att den skrevs i C# medan bloggen hade A* i Python.

Beroende på hur det räknas användes inte en källa för quadtree, dock testade Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022) effektiviteten av att använda ett octree i deras arbete. Då de hade ett användbart resultat valdes det att också använda ett octree, dock eftersom studiens testmiljö var i 2D valdes det att istället använda den tvådimensionella variationen av ett octree, ett quadtree.

Det är svårt att jämföra resultatet av quadtree med det octreet av Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022), eftersom deras studie testade om ett octree skulle fungera för flera typer av fordon. Å andra sidan, i både deras studie och denna rapport visas det att quadtree fungerar för att samla in navigationsdata. Då resultatet av Zhao, Xu, Zlatanova, Liu, Ye, & Feng (2022) är mer korrekt, skapar flera korrekta vägar, så är det rimligt att anta att de hade en bättre implementation av tekniken eller hade bättre övrig logik på agenten.

5.2. Potentiella förbättringar

Det finns ett antal delar av arbetet som kunde ha förbättrats och implementationer som hade gett ett mer trovärdigt och användbart resultat som inte infördes antingen i mån av tid eller att en ändring skulle innebära en total refakturering av en stor mängd kod.

Att alla vägsökningsalgoritmer kunde använda NavMesh-navigationsdata gjorde projektet mer komplicerat då navigationsdatan enbart är byggd för att fungera för NavMesh. AL Satai, Abdul Zahra, Rasool, Abd-Ali, & Pruncu (2021) behövde inte oroa sig för att deras vägar överskred hinder på samma sätt då deras navigationsdata var pixelbaserad och rimliga distanser från hinder kunde sättas mer direkt in i vägsökningsalgoritmen.

För att minska problemen som uppkom när NavMesh-datan användes genererades triangel-centerpunkter istället för att använda triangel-hörnpunkter. Detta ledde till mer rimliga vägar för Theta* och Phi* fast A* som använder grannlogik hade problem i de fall att

trianglarna är stretchade och små (vilket var fallet i pelarekartan). Utöver det har NavMesh en kortare och kantigare vägar på grund av att punkterna den följer oftast ligger nära hinder.

Studiens andra navigationsdata-system quadtree skulle troligtvis fungera bättre med mer finslipning, den saknar en hinderradius som tar bort rutor nära en vägg istället för enbart dom som överlappar en vägg. Skillnaden skulle bli att algoritmerna har mindre chans att skapa vägar som är för nära väggar, vilket kan antas skulle göra att chansen att misslyckas skulle gå ner.

Fler källor som diskuterade de implementerade algoritmerna kunde ha använts. Källan som användes var godtycklig och beskrev algoritmerna i tillräcklig detalj. Utöver det nämnde den även att det inte fanns mycket forskning om dem, därför valdes det att endast använda den. Dock hade det varit klokt kolla källorna den refererar till när den beskriver algoritmerna för att försäkra att dem förstods rätt.

En “line of sight check” kunde ha gjorts mellan alla punkter och alla fantompunkter i slutet för att se om någon punkt kunde ha skippats då. Om den bearbetningstekniken hade fungerat, skulle studiens algoritmer ha kunnat skapa en kortare väg, som antagligen fortfarande skulle vara optimerad för en hastighetsbaserad agent.

En källa kunde ha använts under konstrueringen av studiens quadtree. Det tycktes att konceptet av ett quadtree var tydligt nog att vi kunde skapa det utan källa eller problem. Dock kan det finnas optimeringar av processen av att antingen skapa quadtree:et eller i processen för quads att hitta sina grannar.

5.3. Samhälleliga och etiska aspekter

Eftersom studien handlar om realistisk rörelse av agenter inom spel finns det en chans att den kan användas för simulationer. Vidare, en annan etisk aspekt att ta hänsyn till är att vägsökningsalgoritmer potentiellt kan användas som logik för drönare och robotar, som kan användas för oetiska saker till exempel attackrobotar.

Från samma perspektiv kan vägsökningsalgoritmer för drönare och robotar eller bilar vara till nytta för allmänheten då självkörande bilar och automatiserade leveransdrönare har blivit allt mer populärt. Dock borde inte algoritmer som har en för stor chans att misslyckas användas i fordon, då ett misslyckande kan orsaka allvarliga skador eller leda till dödsfall. Just därför bör mer forskning inom vägsökning genomföras.

Vägsökningsalgoritmer kan användas för att optimera logistiska processer som samhällen behöver effektivisera. Ett exempel på det är i Skövde där mjukvara används för att hitta den bästa möjliga vägen för sopbilar att hämta alla sopor, med en mängd mätvärden i åtanke (Högskolan i Skövde 2011). Den använder troligtvis en mer traditionell nodbaserad vägsökning fast på samma perspektiv kan algoritmerna som testas tillämpas för att spara tid utan att uppoffra pålitlighet av automatiskt förflyttande fordon inom logistik.

Othman (2023) undersöker hur samhälleliga åsikten ändras beroende på mängd kunskap kring självkörande bilar. I sin studie kom han fram till att tilliten till datorstyrda bilar minskas drastiskt när den som fyller i enkäten får lära sig att även med moderna självstyrande bilar händer krascher som kan vara brutala. Även om personen också informeras att samma självstyrande bilar har en extremt mindre chans att misslyckas minskas tilliten och intresset extremt och oron ökar. Så länge som ett datorstyrt fordon kan

misslyckas kommer det samhälleliga perspektivet att vara negativt då det mänskliga perspektivet är att de inte skulle begå de misstagen och enbart dåliga förare kraschar även om statistiken motbevisar detta.

5.4. Framtida arbete

Detta arbete är en intressant startpunkt för många potentiella arbeten i framtiden. En av de stora implementationerna att testa är hur man optimerar agentens väg när man lägger till en tredje dimension. Hur vägsökningen ska ändras för att bygga och behålla hastigheten när den hanterar berg och dalar.

Det skulle även vara intressant att se hur algoritmerna kan ändras för att hantera dynamiskt uppdaterande terräng eller ett mål som flyttar längs en kurva vilket skulle göra det nödvändigt att justera vägen agenten tar medan den flyttar sig längs sin rutt. Ett sätt att göra det på skulle vara att ändra implementationen av Phi* till att vara mer lik varianten incremental Phi*.

Utöver det vore det intressant att gå mer mot det realistiska hållet. Hur nära verkligheten kan simulationen göras och hur mycket av logiken kan användas för att bygga en väg som en verklig robot kan följa, eller som en förare skulle kunna följa. Det skulle vara intressant att vidareutveckla vår lösning mot ett sådant håll. I det fallet skulle det nog vara relevant att testa flera algoritmer som är mer likt hur Dijkstras fungerar, då Dijkstras kan kräva att en väg inkluderar en punkt.

Ett annat sätt att expandera arbetet är att testa flera algoritmer och dataset. Exempelvis hade det varit intressant att testa Voronoi diagram som dataset. Även hade sökningsmetoden bidirectional-search varit intressant att testa. Teoretisk sett gör den vägsökandet snabbare genoma att sökningen händer från start till mål och mål till start, tills de sökningarna möts vid en punkt. Även hade andra metoder att bearbeta vägen varit intressant att testa.

Det skulle även vara intressant att skapa flera kartor, som också är baserade på verkliga ställen, såsom städer. För att ta reda på hur allt som finns nu skulle prestera i en verklig situation, och vad resultatet skulle bli.

Ett test som borde göras i framtiden är att implementera alla algoritmer som redan nu finns fast utan att bearbetavägen de skapar med Bézier-kurvor, för att studera deras påverkan på vägsökning.

Referenser

AL Satai, H., Abdul Zahra, M.M., Rasool, Z.I., Abd-Ali, R.S., & Pruncu, C.I . (2021). Bézier Curves-Based Optimal Trajectory Design for Multirotor UAVs with Any-Angle Pathfinding Algorithms. *Sensors*, 21(7), s. 2460-2482. doi.org/10.3390/s21072460

Bhattacharya, P., & L. Gavrilova, M. (2007). Voronoi diagram in optimal path planning. I *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007).* Pontypridd, Wales 09-11 Juli 2007, s. 38-47. doi.org/10.1109/ISVD.2007.43

Ejvegård, R. (2009). *Vetenskaplig metod*. 4:1 uppl. Lund Studentlitteratur.

Hu, J., gen Wan, W. and Yu, X. (2012). A pathfinding algorithm in real-time strategy game based on Unity3D. I *2012 International Conference on Audio, Language and Image Processing, Audio, Language and Image Processing (ICALIP).* Shanghai, China 16-18 July 2012, s. 1159–1162. doi.org/10.1109/ICALIP.2012.6376792

Högskolan i Skövde (2011). *Sveriges smartaste sopbilshytt invigd* [pressmeddelande], 22 december.
https://www.mynewsdesk.com/se/his/pressreleases/sveriges-smartaste-sopbilshytt-invigd-718251

Jin Y. Yen, (1970). An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Journal: Quart. Appl. Math.* 27, s. 526-530 doi.org/10.1090/qam/253822

Lawande, S. R., Jasmine, G., Anbarasi, J., & Izhar, L. I. (2022). A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games. A*pplied Sciences*, 12(11), s. 5499-5529. doi.org/10.3390/app12115499

Othman, K. (2023) 'Investigating the Influence of Self-Driving Cars Accidents on the The Public Attitude: Evidence from Different Countries in Different Continents', *2023 5th International Conference on Smart Systems and Inventive Technology (ICSSIT), Smart Systems and Inventive Technology (ICSSIT), 2023 5th International Conference on*,Tirunelveli, January India pp. 1579–1584. doi.org/10.1109/ICSSIT55814.2023.10061032

Rathnayake, O., Wijesekara, G., & Samaranayake, L. (2023). A Path Planning Technique to Reduce the Travel Time of a Micromouse. I *2023 IEEE 17th International Conference on Industrial and Information Systems (ICIIS).* Peradeniya, Sri Lanka 25-26 August 2023, s. 519-523. doi.org/10.1109/ICIIS58898.2023.10253498

Rogers, K., Karaosmanoglu, S., Altmeyer, M., Suarez, A. & E. Nacke, L. (2022). Much Realistic, Such Wow! A Systematic Literature Review of Realism in Digital Games. I *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22).* Association for Computing Machinery, New York, NY, USA 30 April - 05 May 2022, Article 190, s. 1–21. doi.org/10.1145/3491102.3501875

Suurballe, J. W., & Tarjan, R. E.(1984). A quick method for finding shortest pairs of disjoint paths. *Networks*, 14 (2), s. 325-336. doi.org/10.1002%2Fnet.3230140209

Swift, N. (2017). Easy A* (star) Pathfinding. *medium.com/@nicholas.w.swift* [blogg], 28 februari. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 [2025/02/17]

Unity (2025). Building a NavMesh.

https://docs.unity3d.com/2021.3/Documentation/Manual/nav-BuildingNavMesh.html [2025/02/17]

Unity (2025). Rigidbody.

https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Rigidbody.html [2025/02/12]

Yang, Y., & Cai, Q. (2023). Research on the A-star Algorithm Based on Path Finding. I *2023 IEEE 3rd International Conference on Data Science and Computer Application (ICDSCA)*. Dalian, China 27-29 October 2023, s. 199-202. doi-org/10.1109/ICDSCA59871.2023.10392293

Zhao, J., Xu, Q., Zlatanova, S., Liu, L., Ye, C., & Feng, T. (2022). *Weighted octree-based 3D indoor pathfinding for multiple locomotion types*. International Journal of Applied Earth Observation and Geoinformation, 112, 102900. doi.org/10.1016/j.jag.2022.102900

Zengzhen, M., Hongjian, X., & Chonghua, H. (2023). *Path planning of indoor mobile robot based on improved A* algorithm incorporating RRT and JPS*. AIP Advances 13, 045313. doi.org/10.1063/5.0144960

Appendix A – Längden av vägen för karta 1 och tiden för alla algoritmer, 0 betyder att den inte klarade av.

PathLenght 1:		NavMesh:	A*Nav:	A*Quad	Theta*Nav	Theta*Quad	Phi*Nav	Phi*Quad
91.92878		24.24036	19.54026	19.70026	18.16022	18.96	18.66024	18.02022
89.75799		23.50035	18	18.36023	17.74022	16.98	17.36021	18.54023
75.79688		21.02029	15.94017	17.2802	14.34014	14.74	14.34014	15.44016
18.74659		7.039994	5.559995	6.559994	6.239995	6.86	6.239995	0
76.85953		14.94015	14.72015	13.60012	13.68012	14.82	13.68012	14.30014
87.19173		27.14043	21.4003	21.3403	20.02027	20.36	19.58026	20.46028
48.79683		11.32007	9.240021	7.559993	11.46007	0.00	11.46007	9.240021
113.3072		19.88026	18.50023	18.68024	18.20023	0.00	18.24023	0
73.83827		21.5003	16.06018	15.60017	14.56014	0.00	14.88015	0
96.8895		18.64024	19.40025	16.44019	17.1002	18.98	15.44016	0
35.96251		6.199995	5.659995	5.199996	5.679995	5.40	5.679995	0
56.24559		10.74006	9.420025	8.560005	10.74006	0.00	10.74006	8.840012
100.8372		15.66017	15.00015	14.96015	14.24014	14.04	14.62014	16.58019
36.30431		12.26009	10.08004	10.26004	0	0.00	9.420025	10.10004
92.23045		18.30023	17.1802	15.86017	14.56014	16.26	14.56014	15.10015
31.79295		6.799994	6.399994	5.559995	6.519994	6.76	6.519994	6.039995
106.8791		21.2403	18.32023	17.0202	16.00018	17.48	15.86017	0
8.568337		1.599999	1.479999	1.479999	1.479999	1.48	1.479999	1.479999
43.29597		12.22009	10.06004	10.10004	10.48005	9.86	9.460026	9.900036
80.44616		22.56033	0	18.24023	15.50016	15.92	15.50016	16.8802
51.42573		11.38007	9.900036	9.400024	9.540028	7.84	9.540028	7.839993
64.98624		22.06031	14.94015	17.34021	14.34014	14.60	14.32014	15.42016
65.66196		23.28034	19.26025	17.42021	17.34021	16.66	16.70019	0

46.84919		10.28004	8.640007	9.180019	7.779993	0.00	0	0
88.58733		27.50044	22.16032	22.54033	20.78028	21.32	20.78028	21.6003
66.80157		24.50037	13.72012	17.76022	16.66019	16.54	17.0202	18.08022
71.2737		22.64033	16.36018	18.04022	15.84017	15.78	15.84017	16.16018
48.71053		9.360023	7.219994	8.560005	8.860012	0.00	8.860012	0
53.8313		22.88033	14.30014	15.26016	14.10013	0.00	0	14.48014
80.20617		28.50046	22.36032	22.76033	19.38025	20.44	19.14025	19.84026
88.91864		22.56033	18.04022	0	16.48019	16.56	16.46019	15.78017
79.51485		21.76031	16.72019	17.66021	14.82015	15.24	14.82015	15.86017
44.62295		11.00006	9.440025	8.500004	9.860035	8.78	9.860035	9.22002
60.67817		21.4203	14.78015	15.00015	0	0.00	0	14.80015
50.47541		7.159994	5.759995	6.679994	6.779994	9.04	6.779994	9.040016
87.72792		18.46023	15.64017	14.68015	13.84013	15.08	13.38012	0
113.7273		19.74026	19.22025	18.50023	18.00022	18.16	18.08022	20.74028
29.21242		7.419993	6.199995	7.019994	6.139995	0.00	6.139995	0
70.32703		21.3803	15.50016	16.9602	14.60014	15.24	14.60014	15.70017
81.15532		23.82035	18.42023	18.26023	17.30021	18.30	17.74022	18.38023
93.71103		20.36028	16.8602	15.44016	15.28016	16.52	14.86015	0
51.5084		5.099996	5.099996	5.059996	5.279995	5.28	5.279995	5.279995
37.34106		11.04006	9.240021	9.420025	8.199997	9.12	8.600006	9.480026
116.7585		21.84031	19.98027	21.82031	0	0.00	19.68026	0
70.1363		22.34032	16.60019	16.32018	16.02018	16.74	15.66017	0
78.74504		15.82017	14.92015	14.12013	14.18013	15.28	14.18013	14.66014
17.04852		2.739998	2.539998	2.619998	2.619998	2.62	2.619998	2.619998
51.74826		9.240021	8.980015	8.740009	9.120018	0.00	9.120018	9.080017
93.88232		15.00015	14.76015	14.56014	13.86013	15.44	13.86013	15.98018
72.05516		21.88031	17.40021	15.74017	0	14.58	14.80015	17.62021

94.67616		17.86022	17.84022	16.16018	14.70015	16.40	14.70015	15.46016
74.27013		22.88033	17.0202	17.0402	15.20016	16.00	15.04015	16.04018
51.29558		21.3003	13.22011	13.84013	12.8601	0.00	0	14.98015
96.13853		16.9802	16.12018	13.38012	0	0.00	15.34016	0
48.72142		10.16004	9.120018	8.159996	8.239998	8.14	8.239998	9.22002
95.50945		19.28025	17.68021	17.60021	18.00022	0.00	17.68021	0
18.96965		7.099994	5.479995	0	6.299994	0.00	6.299994	0
79.58115		15.86017	15.04015	14.14013	14.60014	15.60	14.60014	15.06015
11.78867		2.139998	2.099998	2.099998	2.099998	2.10	2.099998	2.099998
85.88306		19.76026	15.32016	13.50012	13.34011	16.16	12.9201	16.16018
99.51286		17.70021	16.26018	16.70019	16.32018	15.36	16.08018	18.20023
22.71809		7.919993	6.779994	7.019994	0	6.24	0	6.239995
80.71453		14.70015	13.86013	13.22011	0	14.10	0	13.60012
61.6563		23.44035	17.62021	18.06022	0	16.92	16.78019	0
85.62064		18.48023	15.64017	14.56014	13.84013	15.14	13.60012	16.04018
99.8049		15.34016	15.18016	15.12016	14.26014	14.22	14.26014	16.22018
36.87966		13.02011	10.16004	10.36005	0	0.00	9.580029	11.26007
96.49301		18.00022	17.36021	16.04018	14.40014	16.18	14.40014	15.22016
82.55385		23.84035	18.40023	18.50023	17.0802	18.50	17.74022	18.54023
84.24249		16.20018	15.86017	14.48014	15.36016	16.20	15.36016	15.36016
87.75073		27.14043	21.4803	21.5603	19.98027	20.40	18.92024	20.30027
88.57423		22.56033	0	0	16.74019	19.08	16.66019	17.92022
81.52962		0	16.18018	0	15.46016	16.38	16.8802	0
26.70117		6.299994	5.379995	5.279995	5.339995	6.54	5.339995	6.539994
65.95176		19.62026	0	15.48016	13.16011	13.60	13.18011	13.22011
83.80644		27.76044	22.06031	21.4403	20.52028	20.32	19.68026	20.04027
60.35952		14.24014	0	10.46005	10.82006	12.88	10.82006	10.94006

90.46529		23.62035	17.32021	18.98024	17.2602	0.00	18.44023	0
94.38343		24.34037	18.32023	18.42023	18.34023	18.70	18.12022	19.22025
74.22211		13.86013	11.94008	11.14006	14.24014	0.00	14.24014	11.40007
91.91655		25.40039	19.44025	18.80024	0	0.00	18.84024	0
54.65364		22.42032	14.20013	15.22016	0	0.00	0	14.26014
63.92657		26.0204	19.60026	19.34025	0	17.94	17.2202	18.26023
38.94894		12.9201	10.16004	10.36005	0	10.02	9.700031	10.98006
89.3943		14.50014	14.10013	14.10013	13.38012	14.42	13.38012	15.06015
73.1731		22.18032	16.30018	16.48019	0	16.58	15.60017	16.38018
92.9612		21.98031	18.54023	16.84019	16.70019	16.98	16.74019	0
25.16104		5.179996	4.679996	4.459996	4.619996	4.80	4.619996	4.399996
99.95292		22.06031	18.48023	16.08018	15.64017	17.10	15.64017	17.1002
85.1002		28.56046	22.78033	22.72033	22.08031	23.82	22.08031	24.36037
52.54405		11.18007	9.800034	8.620007	9.360023	8.02	9.360023	8.019993
90.38417		18.06022	16.9602	15.72017	14.08013	15.94	14.08013	15.02015
44.55962		9.340023	7.739993	6.619994	8.520004	7.28	8.520004	7.279994
86.30673		20.88029	0	16.44019	15.78017	17.34	15.98018	16.9402
15.89773		2.559998	2.519998	2.539998	2.539998	2.54	2.539998	2.539998
44.21433		9.380024	8.480003	7.019994	9.340023	0.00	9.340023	7.859993
75.04568		22.64033	16.56019	18.04022	15.50016	15.82	15.50016	0
16.63491		9.960037	7.579993	7.559993	7.199994	0.00	7.199994	0
77.25452		14.62014	13.82013	13.28011	13.34011	14.36	13.34011	13.84013
68.6945		25.9604	19.92027	20.04027	0	18.44	17.76022	18.14022
68.40803327	AVERAGE:	17.86481	18.17653	17.20753	26.10251	33.81	19.36892	34.3831

Appendix B – Längden av vägen för karta 2 och tiden för alla algoritmer, 0 betyder att den inte klarade av.

PathLenght 2:		NavMesh	A*Nav	A*Quad	Theta*Nav	Theta*Quad	Phi*rNav	Phi*Quad
115.1558		11.80008	15.62017	0	0	9.120018	0	9.120018
34.6174		3.919997	3.939996	4.159997	3.999996	3.999996	3.999996	3.999996
110.7946		10.02004	0	10.92006	8.299999	8.460003	8.520004	8.460003
78.76231		9.420025	13.28011	7.139994	6.639994	6.639994	6.639994	6.639994
69.70612		6.999994	11.68008	7.579993	6.619994	7.059994	6.619994	7.059994
2.868369		0.03999998	0.8999993	5.579995	0.8999993	0.8999993	0.8999993	0.8999993
73.26234		7.219994	14.00013	9.020016	6.779994	13.60012	6.779994	13.60012
83.71696		7.699993	7.679993	7.939993	7.319993	7.359993	7.319993	7.359993
31.91877		4.579996	6.259995	4.759996	4.419996	4.739996	4.719996	4.739996
68.58035		6.259995	8.099995	6.699994	6.279994	6.559994	6.279994	6.559994
50.68735		5.539995	7.799993	5.799995	5.459995	5.419995	5.459995	5.419995
76.65503		7.499993	11.24007	7.159994	6.939994	0	6.939994	0
100.3358		10.68005	9.700031	10.94006	7.759993	7.759993	7.759993	7.759993
77.33571		8.039993	11.36007	8.440002	7.799993	0	7.799993	0
79.39004		8.119995	9.20002	8.380001	8.219997	7.179994	8.219997	8.960014
58.6265		5.979995	8.500004	8.239998	5.719995	5.759995	5.719995	5.759995
57.89859		5.299995	5.319995	5.299995	5.239995	5.239995	5.239995	5.239995
76.5298		7.059994	12.18009	6.479994	6.379994	7.159994	6.379994	7.159994
76.25443		7.639993	12.06009	7.699993	7.359993	7.139994	7.359993	7.139994
120.8979		10.58005	20.88029	10.72005	9.180019	9.920036	9.180019	9.920036
45.59835		5.539995	8.32	5.619995	5.319995	5.359995	5.159996	5.359995

101.6406		9.360023	18.66024	9.820034	8.219997	8.740009	8.159996	8.840012
20.72712		2.919997	3.039997	2.979997	2.979997	2.979997	2.979997	2.979997
101.8387		9.900036	14.84015	10.16004	0	0	8.720009	0
127.0291		11.20007	17.0002	11.34007	9.380024	9.960037	9.380024	9.960037
38.73386		4.639996	4.519996	4.719996	4.519996	4.519996	4.519996	4.519996
115.3975		11.58007	21.10029	10.86006	0	0	0	0
40.01506		4.359996	6.239995	4.499996	4.299996	4.299996	4.299996	4.299996
105.2708		10.12004	15.74017	10.62005	8.740009	8.580006	8.940014	9.680031
5.01709		1.259999	1.159999	6.019995	1.159999	1.159999	1.159999	1.159999
39.53308		4.719996	4.739996	4.999996	0	0	0	0
114.6491		11.02006	20.88029	10.52005	9.680031	9.140018	9.680031	9.140018
48.9016		5.339995	8.800011	5.519995	5.339995	5.299995	5.339995	5.299995
108.6253		10.90006	14.68015	9.480026	8.800011	8.76001	8.940014	9.66003
34.50196		4.019997	5.379995	4.299996	4.159997	4.379996	4.159997	4.379996
47.81376		5.499995	5.559995	5.639995	5.479995	5.539995	5.479995	5.539995
87.86372		7.839993	8.059994	8.039993	0	7.519993	0	7.519993
83.5285		7.519993	12.8201	7.939993	8.099995	8.199997	8.159996	8.199997
106.5458		10.08004	18.38023	10.20004	0	9.500027	0	9.500027
12.28878		2.139998	2.159998	2.159998	2.159998	2.159998	2.159998	2.159998
55.1054		5.899995	5.879995	6.299994	5.899995	5.719995	5.899995	5.719995
33.70186		3.919997	6.939994	3.919997	3.959996	3.959996	3.959996	3.959996
82.78175		8.440002	12.10009	9.100018	8.279999	8.440002	8.279999	8.440002
77.72772		6.279994	6.279994	6.319994	6.279994	6.279994	6.279994	6.279994
26.75168		3.399997	3.859997	3.679997	3.499997	3.499997	3.499997	3.499997
68.30991		6.879994	12.00008	7.199994	6.479994	6.379994	6.479994	6.379994
67.81174		6.919994	10.32005	6.019995	6.599994	6.659994	6.359994	6.659994
68.69878		6.799994	10.72005	7.019994	6.759994	6.779994	6.759994	6.779994

95.49057		10.10004	10.70005	10.66005	7.479993	7.479993	7.479993	7.479993
33.39061		4.619996	4.619996	4.819996	4.279996	4.019997	4.279996	4.019997
70.33441		6.999994	7.559993	6.859994	0	6.599994	7.499993	0
15.6716		2.559998	2.859998	2.539998	2.539998	2.539998	2.539998	2.539998
92.65884		9.20002	13.44012	9.180019	8.820011	8.660007	8.820011	8.660007
106.0324		10.44005	12.14009	0	9.740032	10.48005	9.740032	10.48005
58.72271		6.159995	8.860012	6.699994	5.839995	5.899995	5.839995	5.899995
81.03605		8.360001	8.720009	10.22004	6.919994	7.339993	6.979994	7.339993
77.23014		9.780033	11.84008	7.239994	6.779994	6.879994	6.839994	6.879994
65.76559		6.799994	8.259998	8.900013	6.279994	6.859994	6.279994	6.859994
70.72461		6.959994	10.86006	6.099995	6.359994	6.819994	6.359994	6.819994
42.38382		5.039996	4.939996	5.299995	4.979996	5.259995	4.979996	5.259995
89.62118		8.820011	9.260021	0	8.299999	7.959993	0	7.959993
86.98882		7.339993	14.14013	7.959993	7.079994	7.259994	7.079994	7.259994
76.99412		8.76001	10.20004	8.119995	8.219997	7.119994	8.219997	7.119994
25.40422		3.399997	3.359997	3.419997	3.419997	3.419997	3.419997	3.419997
49.53925		5.739995	6.439994	5.739995	0	5.739995	5.999995	5.739995
96.80711		10.68005	9.820034	10.00004	7.559993	7.839993	7.559993	7.839993
76.42032		8.039993	11.36007	8.380001	7.839993	7.619993	7.839993	7.439993
74.80226		7.219994	7.739993	7.339993	7.039994	7.499993	6.759994	7.499993
14.01739		2.359998	2.279998	2.279998	2.279998	2.279998	2.279998	2.279998
62.7309		6.659994	6.639994	7.059994	5.819995	6.739994	5.819995	6.739994
3.041327		0.7799994	0.9199993	5.659995	0.9199993	0.9199993	0.9199993	0.9199993
38.89048		4.699996	4.699996	4.959996	4.619996	4.619996	4.619996	4.619996
101.6964		10.92006	19.18025	9.940037	0	0	0	0
74.58392		6.879994	11.02006	6.979994	6.659994	6.779994	6.659994	6.779994
97.05768		8.920013	17.70021	8.400002	7.779993	8.380001	7.779993	8.380001

4.056992		0.7799994	1.059999	5.439995	1.059999	1.059999	1.059999	1.059999
67.50361		6.439994	9.700031	6.939994	6.719994	6.379994	6.719994	6.379994
113.3576		11.52007	15.26016	0	9.120018	9.060017	9.120018	9.060017
29.14293		3.679997	3.599997	3.619997	3.619997	3.619997	3.619997	3.619997
90.32734		8.34	15.08015	8.520004	0	0	0	0
15.76186		2.559998	2.559998	2.559998	2.559998	2.559998	2.559998	2.559998
105.7054		10.20004	15.78017	9.560028	8.660007	8.520004	8.660007	8.520004
21.36095		3.099997	6.399994	3.039997	3.039997	3.039997	3.039997	3.039997
75.44564		7.859993	11.24007	8.440002	7.999993	8.32	7.999993	8.32
90.76943		9.280022	13.70012	9.500027	7.439993	7.819993	7.439993	7.819993
23.99051		3.239997	3.159997	3.179997	3.179997	3.179997	3.179997	3.179997
62.31559		6.099995	7.659993	8.199997	6.599994	5.899995	5.899995	5.899995
60.09948		6.899994	9.560028	5.819995	6.219995	6.219995	6.479994	6.219995
47.22139		4.759996	4.699996	4.799996	4.659996	4.659996	4.659996	4.659996
119.8351		11.12006	21.5403	10.84006	9.22002	9.280022	9.22002	9.280022
46.59206		5.799995	8.500004	5.479995	5.059996	4.739996	5.179996	4.739996
74.11153		8.079994	7.299994	7.579993	6.419994	7.159994	6.419994	7.159994
44.90657		5.139996	7.099994	5.159996	5.139996	5.399995	5.419995	5.399995
52.52108		6.159995	6.639994	5.619995	5.939995	6.139995	5.939995	6.139995
69.73574		6.099995	11.32007	5.999995	5.959995	5.959995	5.959995	5.959995
68.13332		7.779993	11.70008	7.639993	6.579994	7.399993	6.699994	7.399993
117.8127		11.12006	21.18029	10.78006	8.78001	10.54005	8.78001	9.980038
85.1762		9.840034	14.16013	9.780033	7.299994	7.779993	7.399993	7.779993
72.66624		7.179994	11.46007	7.839993	7.239994	7.159994	6.679994	7.159994
16.20029		2.559998	2.679998	2.579998	2.579998	2.579998	2.579998	2.579998
66.63189068	AVERAGE:	6.867809	10.52486	10.71641	15.4608	12.8534	13.5992	13.8188