Examensarbete

HÖGSKOLAN I SKÖVDE

VÄGPLANERING I DATASPEL MED HJÄLP AV ARTIFICIAL BEE COLONY ALGORITHM

PATHFINDING IN COMPUTER GAMES BY USING ARTIFICIAL BEE COLONY ALGORITHM

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2015

Jessica Lee

Handledare: Peter Sjöberg
Examinator: Mikael Johannesson

Sammanfattning

Artificial Bee Colony Algorithm är en algoritm som tidigare tillämpats på många numeriska optimeringsproblem. Algoritmen är dock inte vanligt förekommande vad gäller vägplanering i dataspel. Detta arbete undersöker ifall algoritmen presterar bättre än A* på fyra olika testmiljöer eftersom A* är en av de mest använda algoritmerna för vägplanering i dataspel och således en bra referens. De aspekter som jämförs vid mätningarna är algoritmernas tidsåtgång samt längden på de resulterande vägarna.

En riktad slumpgenerering av vägar har implementerats till algoritmen som gör att den inte fungerar på djupa återvändsgränder. Algoritmen har också en roulettehjulsselektion samt egenskapen att kunna generera slumpade grannvägar till de som skapats hittills.

Resultaten visar att Artificial Bee Colony Algorithm presterar betydligt sämre än A* och att algoritmen därför inte är en bättre algoritm för vägplanering i dataspel. Algoritmen har dock potential till att prestera bättre och fungera på återvändsgränder om man förbättrar dess slumpgenerering av vägar.

Nyckelord: Artificial Bee Colony Algorithm, A*, vägplanering, AI, spel

Innehållsförteckning

1 Introduktion

I dagens läge innehåller de flesta moderna dataspelen någon form av artificiell intelligens. En mycket vanligt förekommande form av artificiell intelligens är vägplanering. Många olika sökalgoritmer har utvecklats speciellt för vägplanering, men man försöker än idag finna nya metoder. Därför är det inte ovanligt att allmänna algoritmer som ursprungligen inte är avsedda för vägplanering utforskas inom detta område.

En algoritm vid namn Artificial Bee Colony Algorithm är ursprungligen skapad med avsikten att appliceras på numeriska optimeringsproblem (Karaboga, 2005). Den har sedan dess utforskats på en stor bredd olika former av optimeringsproblem, däribland vägplanering inom robotik (Bhattacharjee, Rakshit & Goswami, 2011; Liang & Lee, 2014; Ma & Lei, 2010). Utforskning på algoritmen inom vägplanering för dataspel är dock bristande. Därför fokuserar detta arbete på Artificial Bee Colony Algorithm inom kategorin vägplanering för dataspel. Eftersom algoritmen inte ursprungligen är skapad för vägplanering har en variant anpassad för vägplanering implementerats.

En av de vanligaste algoritmerna för vägplanering i dataspel är A* som snabbt finner den billigaste möjliga vägen som existerar i en miljö (Hart, Nilsson, & Raphael, 1968). På grund av detta har Artificial Bee Colony Algorithm jämförts med A* för att avgöra ifall algoritmen är en bra algoritm för vägplanering i dataspel.

För att ta reda på om Artificial Bee Colony Algorithm är mer lämpad för vägplanering i dataspel än A* har båda algoritmerna testats på fyra olika miljöer som är vanligt förekommande i spel. Sedan har dessa algoritmer jämförts med varandra med avseende på deras tidseffektivitet samt hur många förflyttningar deras resulterande vägar kräver.

2 Bakgrund

Detta kapitel innehåller bakgrundsinformation som är nödvändig för att lättare förstå sig på det som tas upp i resterande kapitel. Kapitlet omfattar information om Artificial Bee Colony Algorithm, A* samt vägplanering.

2.1 Svärmintelligens

Uttrycket ”Swarm intelligence”, som översätts till svärmintelligens, skapades av Gerardo Beni och Jing Wang 1989 (Beni & Wang, 1993). Svärmintelligens innebär att ett system inte har en enda central enhet som ensamt tar beslut, utan att ett flertal enheter istället tar ett gemensamt beslut för hela svärmen tillsammans. Varje enhet har dock sin egna lokala intelligens och arbetsuppgift och med hjälp av globalt informationsutbyte mellan dessa olika enheter kan svärmen effektivisera det gemensamma arbetet.

Konceptet svärmintelligens används inom fältet artificiell intelligens. Sedan Beni och Wang framställde uttrycket har ett stort antal olika varianter av svärmintelligens utvecklats i form av algoritmer. Dessa algoritmer är mestadels baserade på svärmbeteenden i naturen, bland annat på beteendet hos myror, fåglar (Hinchey, Loyola Coll, Sterritt, & Rouff, 2007) och honungsbin (Karaboga, 2005).

2.2 Artifical Bee Colony Algorithm

Artificial Bee Colony Algorithm (Karaboga, 2005), även förkortat ABC, är en form av optimeringsalgoritm som skapades av Dervis Karaboga år 2005. Algoritmen är en form av svärmintelligens som är baserad på beteendet hos honungsbin som söker efter nektar till sina bon och kan appliceras på numeriska optimeringsproblem.

2.2.1 Beteendet hos bin i det verkliga livet

En modell skapad av Tereshko visar hur bin beter sig i det verkliga livet när de söker efter föda. Modellen består utav tre viktiga komponenter (Tereshko, 2000):

Matkällor
Födan som bin söker efter till sin kupa, finner de i matkällorna. Vilken matkälla som är den bästa beror bland annat på avståndet från matkällan till kupan, hur näringsrik matkällans nektar är och hur lätt det är att utvinna energin från denna nektar. Alla dessa faktorer kan göras om till ett enda gemensamt värde som avvägs av de olika värdena.

Arbetarbin
Ett arbetarbi, eller ett så kallat ”Employed bee”, är ett bi som samlar nektar från en specifik matkälla som biet just då är associerat med, för att sedan hämta det till bikupan. Biet erhåller information om sin matkälla, bland annat matkällans plats och energiinnehåll. Denna information delar biet med sig av till de andra bina som väntar i bikupan.

Arbetlösa bin
Arbetslösa bin, eller så kallade ”Unemployed bees”, är bin som letar efter matkällor att utforska. Det finns två typer av arbetslösa bin: Utforskarbin, även kallade ”Scout bees”, som slumpmässigt letar efter matkällor, och åskådarbin, även kallade ”Onlooker bees”, som letar efter matkällor baserat på informationen de får från arbetarbin.

Syftet med att ha olika typer av bin, som var och en har en egen specifik uppgift, är att öka chansen för att de hämtar så mycket näringsrik nektar som möjligt på så lite tid som möjligt. I bikupan finns det en sektion som kallas för dansgolvet. Detta är en av de viktigaste delarna av bikupan eftersom detta är var utbytet av information mellan bina sker. När ett arbetarbi hämtat nektar från matkällan och återvänt till sitt bo kan den välja att utföra en så kallad ”waggle dance” på dansgolvet. Ju bättre matkälla biet har funnit, desto längre dansar det. På detta sätt delar biet med sig av information om sin nuvarande matkälla.

Åskådarbin håller sig till dansgolvet för att sedan följa med ett slumpvalt dansande arbetarbi till dess matkälla. Eftersom de arbetarbin som har de allra bästa matkällorna dansar längre än de med sämre matkällor, kommer de att närvara längre på dansgolvet. Detta kommer leda till att dansgolvet har fler bin associerade med starka matkällor än svaga. På detta sätt ökar chanserna för att åskådarbina väljer en bra matkälla och att fler bin sammanlagt söker sig till den bästa möjliga matkällan (Tereshko, 2000).

2.2.2 Algoritmen

Med hjälp av Tereshkos modell har Karaboga framställt en algoritm, Artificial Bee Colony Algorithm, som finner lösningar på numeriska optimeringsproblem på samma sätt som honungsbin finner näringsrik och lättåtkomlig nektar. I denna algoritm initieras en bikoloni varav hälften av kolonin består utav arbetarbin och den andra halvan består utav åskådarbin. Eftersom varje arbetarbi är associerat med en unik matkälla finns det lika många matkällor som arbetarbin. Varje matkälla representerar en möjlig lösning till det numeriska problemet. Mängden nektar och kvalitén på denna matkälla representerar kvalitén på den möjliga lösningen, det vill säga hur optimal den är för problemet.

När ett arbetarbi överger sin matkälla blir biet till ett utforskarbi, som slumpmässigt letar efter en ny matkälla, det vill säga att den slumpar fram en ny möjlig lösning. Karaboga sammanfattar algoritmen enligt pseudokoden i ***Figur 1*** (Karaboga, 2005):

```
Skicka utforskarbina till deras utvalda matkälla

UPPREPA

        Skicka arbetarbina till deras associerade matkälla och bestäm värdet på nektaren

        Beräkna sannolikheten att en matkälla blir vald av ett åskådarbi baserat på deras
        nektarvärde

        Avbryt utforskningen av matkällorna som bina överger

        Slumpa fram utforskarbinas nya matkällor och skicka dem till dessa

        Memorera den bästa matkällan som hittills hittats

TILLS (Kraven är mötta)
```

Figur 1. Pseudokod för Artifical Bee Colony Algorithm.

För varje upprepande cykel, det vill säga varje så kallad ”loop”, sker tre viktiga steg: I det första steget skickas arbetarbin till deras utvalda matkälla och fitnessvärdet på denna matkällas nektar räknas ut, i det andra steget väljer åskådarbina en utav de valda matkällorna och skickas till en närliggande matkälla, och i det tredje steget skickas utforskarbin till en ny slumpad matkälla.

När en ny cykel sker återvänder ett arbetarbi inte tillbaka till exakt samma matkälla som det gjorde den förra cykeln, utan letar istället efter en matkälla som är närliggande till den förgående matkällan. För ett numeriskt problem innebär detta att man gör någon mindre förändring i sin nuvarande lösning och på så vis skapar en grannlösning. Visar sig den nya matkällan, det vill säga grannlösningen, ha ett bättre fitnessvärde än den förra, överger biet den gamla matkällan och ersätter den med den nya. Visar sig grannlösningens fitnessvärde istället vara lika bra eller sämre, överger biet den nya matkällan och behåller sin gamla.

Arbetarbin har alla ett gemensamt gränsvärde, en så kallad ”limit”. När ett arbetarbi har sökt efter närliggande matkällor lika många antal gånger som gränsvärdet utan att ha ersatt den gamla matkällan en enda gång med en ny grannlösning, är matkällan färdigutforskad och arbetarbiet överger sin matkälla för att bli till ett utforskarbi.

När ett åskådarbi väljer en utav de nuvarande arbetarbinas matkälla är sannolikheten stor att biet väljer en matkälla av hög kvalité, men ingen garanti (Tereshko, 2000). På samma sätt fungerar Karabogas algoritm. Varje matkälla har ett fitnessvärde och när ett åskådarbi väljer en matkälla baseras det på en selektionsoperator där sannolikheten för att en matkälla blir vald ökar ju högre dess fitnessvärde är. Liksom arbetarbin, memorerar åskådarbin matkällans position och söker sig till en närliggande matkälla. Har matkällan ett högre fitnessvärde ersätts den gamla matkällan av den nya, annars behåller biet den gamla. När cykeln är slut sparas lösningen med det högsta fitnessvärdet undan och resten glöms bort.

När utforskarbin väljer en matkälla avgörs detta helt av slumpen. På grund av detta har utforskarbin billig sökkostnad men också låg genomsnittlig kvalité på matkällorna. Det finns dock alltid en liten chans att utforskarbin finner en rik matkälla som de andra bina aldrig skulle hittat. Artificial Bee Colony Algorithm innehåller alltså både global sökning efter nya lösningar och lokal sökning inom de existerande lösningarna som verkar mest lovande, där utforskarbin står för det globala och arbetarbin och åskådarbin står för det lokala (Karaboga, 2005).

Algoritmen består av sammanlagt tre parametrar som kan justeras beroende på vilket problem man tillämpar algoritmen på. Dessa tre parametrar är bikolonistorleken, alltså hur många arbetarbin och åskådarbin som existerar, cyklerna, alltså hur många cykler algoritmen ska upprepas i, och gränsvärdet som varje arbetarbi har.

2.2.3 Fördelar över andra algoritmer

Ett problem många andra optimeringsalgoritmer, däribland A*, Particle Swarm Intelligence och genetiska algoritmer, har är att de för vissa problem lätt kan fastna i samma lokala spår som lutar åt en optimal lösning men inte riktigt når fram hela vägen (Xu, Duan, & Liu, 2010). Eftersom Artificial Bee Colony Algorithm delvis sätter fokus lokalt men också har sina utforskarbin som söker efter nya lösningar utanför det lokala spåret när ett arbetarbi överstigit sitt gränsvärde, minskar det risken för att algoritmen fastnar i samma spår för länge.

När man applicerat ABC-algoritmen på ett flertal optimeringsproblem har algoritmen dessutom visat bättre resultat än många populära optimeringsalgoritmer med avseende på komplexitet (Singh 2008; Xu, Duan, & Liu, 2010).

2.2.4 Relaterad forskning

Sen Karaboga skapade Artificial Bee Colony Algorithm har den även testats på ett flertal olika problem. I en undersökning där man använde ABC för att räkna fram en fördelning av vikter till en fackverkskonstruktion fick man en fungerande lösning hela 100% av alla försök. Dessutom var dessa lösningar alltid lika bra eller bättre än lösningarna man fick fram av andra de andra algoritmerna som testades (Sonmez, 2010).

Algoritmen har också applicerats på handelseresandeproblemet med goda resultat (Karaboga & Gorkemli, 2011).

Vid flera undersökningar där man letat efter effektiva sätt att navigera en robot genom ett plan utan att den krockar i väggar på vägen, har svärmintelligens varit av intresse (Liang & Lee, 2014). Detta har undersökts noggrannare, bland annat med hjälp av ABC, och gett en hel del goda resultat (Ma & Lei, 2010). Vid en av dessa undersökningar testade man även att styra roboten med hjälp av en differentialevolutionsalgoritm och en partikelsvärmsalgoritm. Resultaten visade att ABC snabbare fick fram vägar som krävde färre förflyttningar än de andra algoritmerna (Bhattacharjee, Rakshit, & Goswami, 2011).

2.3 Vägplanering

Vägplanering är något som existerar i vår vardag. När man vill gå, cykla eller köra till en destination har man oftast en planerad rutt för att ta sig dit via den kortaste möjliga vägen. Sedan flera år tillbaka har vägplanering varit av intresse även inom robotik, där man försökt efterlikna vägplanerandet hos människor och applicera det på robotar i form av artificiell intelligens (Brooks, 1983). Vägplanering i artificiell intelligensform är även något som används i många moderna dataspel.

I ett arbete om vägplanering, “A Formal Basis for the Heuristic Determination”, tar författarna till artikeln upp två olika sätt att prioritera vägplanering: den matematiska metoden, där man garanterat vill få fram den bästa möjliga vägen som existerar, oavsett kostnad, och den heuristiska metoden, där man vill få fram en så bra väg som möjligt som man kan på det billigaste möjliga sättet, även om det innebär att man kanske inte alltid får den bästa möjliga vägen (Hart, Nilsson, & Raphael, 1968).

2.3.1 Vägplanering i dataspel

Eftersom vägplanering i dataspel sker i realtid finns det ett krav på att vägplaneringen inte tar så pass lång tid på sig att det påverkar spelupplevelsen negativt. I vissa spel ändras villkoren på en karta dynamiskt. Exempelvis så kan hinder som inte funnits tidigare, plötsligt blockera en öppen väg och göra skillnad på var en karaktär får och inte får gå. På grund av detta måste en ny vägplanering ske som utgår från de nya villkoren. Detta medför att en ny väg måste sökas fram med en snabb algoritm eftersom långa söktider skapar en fördröjning varje gång förhållandena ändras och en ny väg måste hittas (Graham, McCabe, & Sheridan, 2005).

I turbaserade spel, som till exempel Heroes of Might and Magic (New World Computing, 1995), är vägplaneringen sällan dynamisk och längre söktider är därför inte lika kritiska. I

spel som dessa är det dessutom inte viktigt att sökalgoritmen alltid finner den kortaste möjliga vägen, eftersom spelkaraktären ändå kommer nå fram till sin destination innan det är motståndarens tur.

I spel som t.ex. Dota 2 (Valve Corporation, 2012), är det högt krav både på att vägplaneringen går fort, samt att en så kort väg som möjligt hittas. Om algoritmen för vägplaneringen istället får fram en lång omväg, kan detta ge en orättvis nackdel för spelaren eftersom spelarens karaktär i så fall kommer tar längre tid på sig att nå sin destination. Detta kan till exempel leda till att motståndaren hinner fly från spelaren, eller att spelaren inte hinner fly från motståndaren i tid, och därmed avgöra spelet.

Eftersom spel med vägplanering har betydligt fler uppgifter än bara vägplanering, som till exempel utritning av grafik, kollisionskontroller och undansparning av statistik, sker denna vägplanering på en begränsad mängd minne och CPU-resurser (Botea, Müller, & Schaeffer, 2004). Det är därför viktigt att vägplaneringen tar upp så lite prestanda och minne som möjligt.

2.3.2 A*

Bland de många sökalgoritmer som skapats för vägplanering är en utav de vanligaste en algoritm vid namn A*. Denna algoritm använder sig av heuristik och söker alltid igenom den mest lovande vägen först (Hart, Nilsson, & Raphael, 1968). A* får alltid fram den billigaste möjliga vägen.

Skaparna av algoritmen beskriver den enligt pseudokoden i ***Figur 2*** (Hart, Nilsson, & Raphael, 1968):

```
1) Lägg startnoden i den öppna listan och räkna ut dess värde för f
2) Välj noden i öppna listan vars värde av f är minst och sätt den till den ”aktuella
   noden”. Lägg sedan noden i den stängda listan
3) Om den aktuella noden = destinationsnoden, avsluta algoritmen, fortsätt annars till
   steg 4
4) För varje grannod den aktuella noden har:
        Om grannoden finns i den stängda listan eller är en förbjuden nod:
                Ignorera noden
        Annars:
                Om grannoden inte finns i den öppna listan:
                        Lägg till den i den öppna listan och räkna ut dess F-, G- och
                        H-värden. Sätt sedan den aktuella noden som grannodens förälder

                Annars:

                        Om kostnaden för att nå grannoden från den aktuella noden är
                        billigare än kostnaden för att nå grannoden från noden i
                        listan, ersätt grannodens föräldernod till den aktuella noden.
                        Matcha kostnaden för F, G och H med den nya föräldern.

5) Upprepa steg 2
```

Figur 2. Pseudokod för A*.

A*-algoritmen inleds med att lägga till startnoden i en lista som kallas för den öppna listan. Denna lista innehåller noder som är möjliga att besöka men som ännu inte har genomsökts.

Innan en nod läggs till i denna lista räknar man ut dess värde för F. Värdet på F beräknas genom att ta kostnaden det tog att nå noden, adderat med den uppskattade kostnaden det tar att nå slutdestinationen från noden (Se **Formel 1**).

$$F = H + G$$

Formel 1. Addition för att få värdet F. Där G är kostnaden för att ta sig till noden och H är den estimerade kostnaden för att ta sig från den nuvarande noden till destinationsnoden.

När man lagt startnoden i den öppna listan startar en cykel i algoritmen som upprepas tills man nått slutdestinationen. Denna cykel inleds med att man väljer ut den nod i den öppna listan med lägst F-värde. Denna nod är nu den så kallade aktuella noden. Den aktuella noden läggs till i en annan lista som kallas för den stängda listan. Denna lista innehåller alla noder som man redan besökt och existerar för att samma nod inte ska besökas mer än en gång.

För varje grannod som den aktuella noden har sker en villkorscheck: Om grannoden inte är en nod som är tillåten av besöka, till exempel en väggnod, eller redan finns i den stängda listan, ignorerar man grannoden helt, annars sker ett av följande alternativ: Om grannoden inte finns i den öppna listan kalkyleras dess F-värde fram samt information om vilken föräldernod den har och noden läggs sedan till i den öppna listan. Om grannoden redan finns i den öppna listan jämförs grannoden med noden i listan. Är den nya noden billigare ersätts den gamla noden i listan med den nya, annars ignorerar man den nya noden. Är den aktuella noden inte destinationsnoden upprepar man cykeln efter detta steg.

När den aktuella noden tillslut är destinationsnoden har algoritmen funnit en färdig väg och iterationen upphör. Eftersom slutnodens förälder är undansparad i den öppna listan kan man enkelt finna den kompletta vägen genom att följa varje nods förälder tills man når startnoden. Är den öppna listan tom utan att man nått fram till destinationsnoden innebär det att det inte finns någon giltig väg på planen från startnoden till destinationsnoden.

3 Problemformulering

Detta kapitel tar upp frågeställningen för arbetet och förklarar anledningen till att frågeställningen har undersökts. Kapitlet har även en metodbeskrivning som tar upp vad som har behövts göras för att kunna genomföra undersökningen, samt hur resultaten har bedömts för att kunna svara på frågeställningen.

3.1 Frågeställning

Detta arbete avser att utforska huruvida Artificial Bee Colony Algorithm lämpar sig för vägplanering i dataspel. Algoritmen har utforskats inom robotik, men betydligt mindre forskning går att finna på algoritmen inom spelsammanhang (Se 2.2.4). Ett vanligt förekommande problem inom spelutveckling är vägplanering. Man vill hela tiden finna algoritmer för vägplanering som har så lite tidsåtgång som möjligt men som samtidigt ger bra resulterande vägar. Därför ligger fokusen för detta arbete på utvecklingen av en vägplanering som använder sig av ABC.

En mycket vanligt förekommande algoritm som används för vägplanering i dataspel är A*, eftersom den alltid får fram den kortaste möjliga vägen i en miljö på väldigt kort tid (Se 2.3.2). På grund av detta har arbetet använt A* som en referens på en bra algoritm för vägplanering i dataspel.

Ett experiment och mätningar på de två algoritmerna har gjorts för att få ett resultat som visar om ABC presterar bättre än A* eller inte och för att på så vis kunna svara på om algoritmen lämpar sig för vägplanering i dataspel. Syftet med detta arbete kan summeras i en frågeställning:

Är Artificial Bee Colony Algorithm en bättre algoritm för vägplanering i dataspel än A?*

I 2.3.1 tas krav på vägplanering i dataspel upp. Genom att undersöka om en algoritm för vägplanering uppfyller dessa krav kan man få en uppskattning om algoritmen är bra för vägplanering i dataspel eller inte. Det som tas upp i kapitlet är främst hur viktigt det är att tidsåtgången inte är för lång i dataspel då det kan påverka spelupplevelsen negativt om en märkbar fördröjning sker mellan varje sökning efter nya vägar. Därför har en mätning på tidsåtgången genomförts. Det tas även upp att vägarna som fås fram helst ska vara så korta som möjligt. Enligt en artikel av Hart, Nilsson & Raphael, använder sig vägplaneringar i dataspel alltså av en heuristisk metod (Hart, Nilsson, & Raphael, 1968) för att få fram vägar. Därför är det viktigt att tidsåtgång prioriteras över att få fram den allra kortaste möjliga vägen.

3.2 Metodbeskrivning

För att kunna undersöka hur bra de två algoritmerna fungerar till vägplanering i dataspel har en spelmiljö för experiment skapats. Sedan har själva implementeringen av vägplaneringen skett. Efter att båda dessa delar genomförts har ett antal test skett och resultaten har sedan mätts med avseende på tidseffektivitet och väglängd.

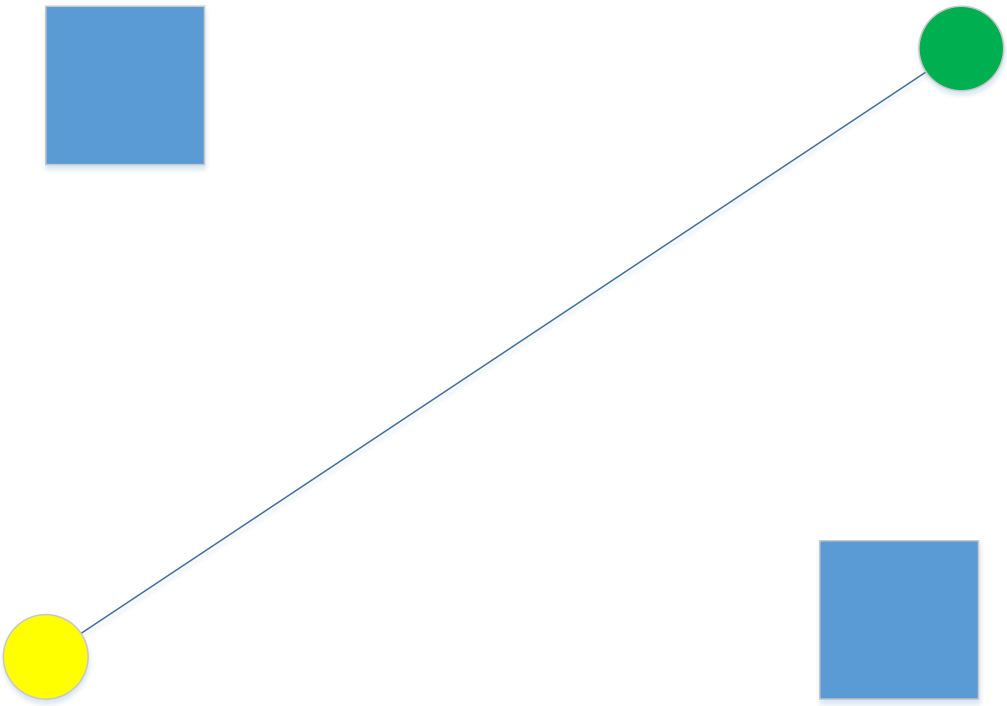
3.2.1 Utveckling av experimentmiljö

För att kunna applicera Artificial Bee Colony Algorithm på vägplanering behövs en spelmiljö där faktiska vägar kan existera. Spelmiljön för denna undersökning är i 2D eftersom

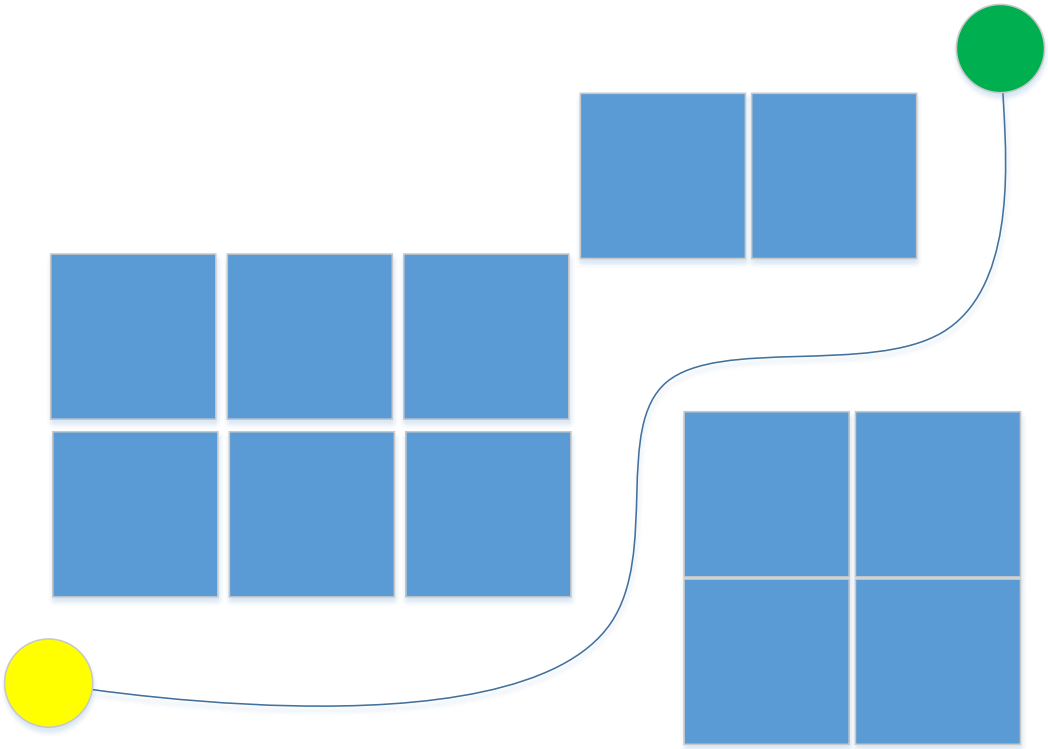
karaktären är markbunden och inte kan justera position i Y-led. Perspektivet för miljön är i fågelperspektiv, det vill säga att man ser vägen ovanifrån (Se **Figur *3*** och **Figur *4***). Detta eftersom det tillåter en att tydligt se all terräng på en och samma gång och därmed ger en bra överblick över hela kartan. Miljön består utav terräng som karaktären kan gå på, men också terräng som karaktären inte får gå på, så kallad förbjuden terräng.

Ett miljöverktyg har implementerats som gör att man kan man placera förbjuden terräng vart man vill på kartan och även hur mycket förbjuden terräng man vill genom att klicka på noderna i kartan. Klickar man på en nod som är fri terräng görs den om till förbjuden terräng. Klickar man på en nod som är förbjuden terräng görs den istället om till fri terräng. När man sedan byggt färdigt banan kan man spara undan den och söka efter vägar på den med hjälp av de två algoritmerna. Man kan även placera en startkoordinat och en destinationskoordinat vart som helst på terräng som inte är förbjuden.

En karta med hinder kan se ut på flera möjliga sätt. Den kan bestå utav extremt få hinder och ha ett nästintill öppet plan (Se ***Figur 3***). Den kan också bestå utav en stor mängd hinder som gör det mer besvärligt att ta sig fram (Se ***Figur 4***). För att vara säker på att en vägplaneringsalgoritm är dynamisk och inte bara anpassad för en typ av karta är det viktigt att man testar algoritmen på flera banor med olika upplägg av hinder. Därför är det bra att man kan placera ut förbjuden terräng vart man vill på kartan i denna spelmiljö. Genom att testa algoritmen på banor med öppna plan och även testa algoritmen på banor med mycket hinder får man en tydlig bild över ifall den fungerar bättre på vissa typer av miljöer än andra. Algoritmerna har testats på fyra olika miljöer som byggts upp med hjälp av miljöverktyget: Ett öppet plan utan hinder, en skogsmiljö, en stadsmiljö och en labyrintmiljö.



Figur 3. Karta med relativt öppet plan. Gul punkt är startnoden och grön punkt är destinationsnoden. Alla blå kvadrater är hinder. Linjen mellan punkterna visar en möjlig väg från startnod till destinationsnod.



Figur 4. Karta med många hinder. Gul punkt är startnoden och grön punkt är destinationsnoden. Alla blå kvadrater är hinder. Linjen mellan punkterna visar en möjlig väg från startnod till destinationsnod.

3.2.2 Utveckling av implementation

En vägplaneringsvariant av ABC har implementeras. Varje möjlig väg algoritmen utvärderar representerar en möjlig matkälla (Se 2.2.2). Ju färre förflyttningar vägen kräver från början till slut, desto bättre fitnessvärde har den.

I 2.2.2 tas bikolonistorlek, cykler och gränsvärden upp. Dessa parametrar är justerbara. Att enbart testa ett fast värde på parametrarna vid utförandet kan ge ett orättvist resultat eftersom det kan finnas potential för bättre resultat hos andra värden. Därför har tester gjorts på implementationen där parametrarna börjat från sitt lägsta möjliga värde och sedan ökat med 1 efter varje körning. Testerna har gjorts på samtliga fyra testmiljöer. De parametervärdena som har gett bäst resultat för en specifik miljö har sedan applicerats på den slutgiltiga undersökningen som presenteras i 5.1. För att avgöra vad som klassas som bäst resultat, har resultatet jämförts med A*-algoritmens resultat. Eftersom tidsåtgången prioriteras över väglängden väljs parametervärdena som ger en så kort väg som möjligt utan att överstiga tidsåtgången A* får fram med marginal.

Eftersom A* är en av de mest använda algoritmerna för vägplanering i dataspel är den en bra algoritm att jämföra ABC med. Därför har en vägplanerare som använder sig av A*-algoritmen implementerats. Vägen som fås fram av denna algoritm har sedan jämförts med vägen som fås fram av ABC. Denna jämförelse går till så att man räknar hur många förflyttningar som krävs i vägen A* får fram, för att sedan jämföra det med antalet

förflyttningar som krävs i vägen ABC-algoritmen får fram. Den algoritm vars väg kräver minst förflyttningar är den som tagit fram den bästa vägen utav de två algoritmerna.

3.2.3 Utvärdering av resultat

Efter att spelmiljön har byggts färdigt och algoritmerna implementerats har båda algoritmerna testats 1000 gånger var på varje testmiljö. Resultaten redovisas i 5.1. Vägarna som har fåtts fram av algoritmerna har evaluerats genom att jämföra deras längd med varandra, samt genom att observera hur vägarna ser ut, där till exempel onödiga omvägar får en väg att se sämre ut.

Tidseffektiviteten har mätts genom att med hjälp av en timer, mäta tiden det tar för algoritmen från att starta vägplaneringen till att nå slutdestinationen och bygga den slutgiltiga vägen.

Det är svårt att avgöra om tidsåtgången för ABC är en önskad tidsåtgång utan att ha något att jämföra den med. Därför har tidsåtgången för ABC att jämförts med tidsåtgången det tar för A* att söka fram en väg på exakt samma testmiljö.

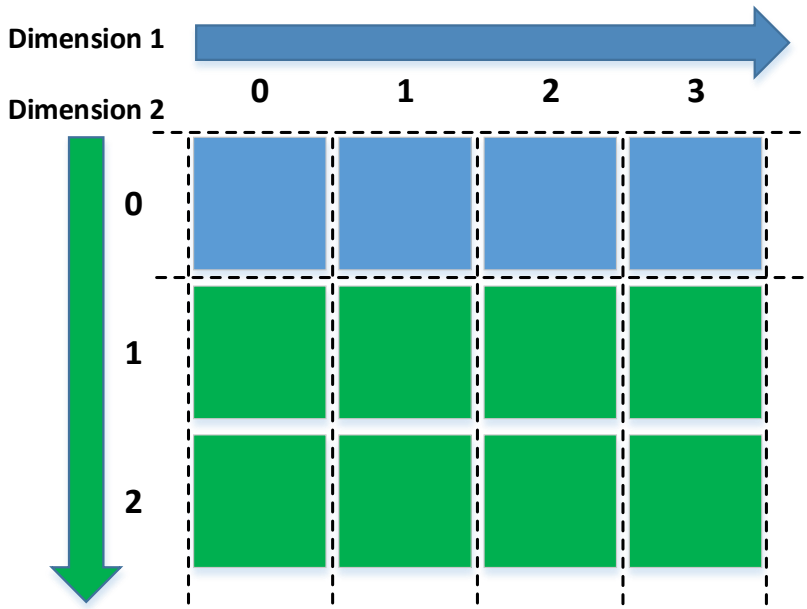
4 Implementation

Detta kapitel går in på utvecklingen av implementationen för arbetet. Kapitlet innehåller en beskrivning av experimentmiljön som algoritmerna ska testas på. Beskrivningar av hur de två algoritmerna skapats tas upp samt diverse problem som dykt upp på vägen. Även en analys över arbetets utvärderbarhet går att finna i kapitlet.

4.1 Experimentmiljö

Experimentmiljön är implementerad i programmeringsspråket C# i utvecklingsmiljön Unity 5 (Unity Technologies, 2015). Unity erbjuder verktyg för att bygga upp scener, knappar och utritning av grafik på ett snabbt och enkelt sätt. Med hjälp av dessa verktyg kan man enkelt och dynamiskt konstruera olika typer av banor och rita ut vägarna som sökalgoritmerna får fram. Eftersom arbetets fokus ligger på sökalgoritmerna och inte den grafiska aspekten, finns det ingen anledning till att lägga ner en stor mängd tid på implementering av banor och grafik. Därför är Unity en passade utvecklingsmiljö för arbetet. Genom att utnyttja Unitys verktyg för grafikutritning sparar man tid som istället kan läggas på att implementera sökalgoritmerna.

Miljön som algoritmerna testats i byggs upp av en tvådimensionell array, där ena dimensionen representerar x-axeln och den andra representerar y-axeln. Varje index-steg i arrayen representerar en koordinatförflyttning i dimensionens axel och varje element är en nod i kartan (Se ***Figur 5***). Arrayens element är av datatypen bool. False innebär att noden är mark och True innebär att den är hinder. För att kunna representera denna nivå grafiskt skapas en till array med identiska dimensioner som istället innehåller Unity-typen material. För varje False i bool-arrayen innehåller material-arrayen ett markmaterial på motsvarande index. För varje True är materialet på motsvarande index ett väggmaterial.



Figur 5. En tvådimensionell array. Den blå raden är den första dimensionen och de gröna raderna är den andra dimensionen. Varje kvadrat är ett element. Siffrorna representerar elementets index.

När programmet körs ritas elementen i material-arrayen ut på samma koordinat som deras index i arrayen. Klickar man på en nod i banan ändras både boolen och materialet på noden i båda arrayer till dess motsats. På detta sätt kan man enkelt bygga varierande banor att testa algoritmerna på och samtidigt se hur de ser ut. Banorna som byggs upp går att spara för framtida användning med hjälp av biblioteket Json.Net (Newton-King, 2006). Banorna sparas då i en txt-fil och kan sedan läsas in igen vid körning med hjälp av Json.Net. Exempel på hur en bana kan se ut kan man se i ***Figur 6***.

Figur 6. Exempel på en bana skapad och utritad i Unity 5. De ljusgråa partierna är mark och de svarta partierna är hinder.

Den enda informationen som används från experimentmiljön när algoritmerna söker efter vägar är bool-arrayen. Detta är en billig metod eftersom algoritmerna bara behöver applicera koordinaterna på arrayens index. Att komma åt ett element i en array när man vet dess index är betydligt billigare än att till exempel söka igenom en hel länkad lista efter element man vill komma åt.

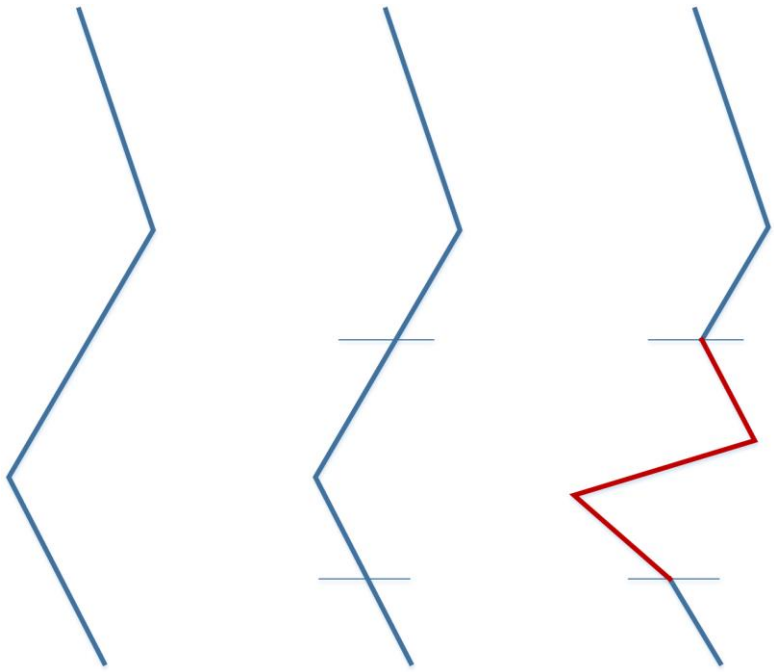
4.2 Implementering av Artificial Bee Colony Algorithm

För att få en bra överblick över hur hela algoritmen ska konstrueras upp har en pseudokod (Danaraj, 2010) och källkod (Karaboga & Basturk, 2007) använts som inspiration. Algoritmen för detta arbete har sedan modifierats för att vara anpassad efter vägplanering med hinder.

För att kunna skapa en komplett ABC-algoritm har slumpgenerering av vägar implementerats, som tas upp i detalj i 4.2.1. Slumpgenereringen tar emot en startnod och destinationsnod som input och genererar fram en slumpad väg mellan dessa två noder. En

simpel roulettehjulsselektion har även implementerats för att representera selektionen för åskådarbina (Se 2.2.2), som baserar oddsen på antal noder en väg innehåller. Ju färre noder vägen har, desto kortare är vägen, och desto högre är chansen att denna väg blir vald utav åskådarbina. Denna selektionsmetod har valts eftersom det är den mest använda metoden vid undersökningarna i 2.2.4.

En viktig del i ABC är att kunna generera grannlösningar till de existerande lösningarna. I en studie där man använder ABC för vägplanering (Eldrandaly, AbdelAziz, & Hassan, 2015) har författarna valt att skapa grannvägar genom att slumpmässigt välja en sektion i den ursprungliga vägen och sedan återanvända samma funktion som genererade den ursprungliga vägen. Man låter startnoden vara första noden i den utvalda sektionen och destinationsnoden vara den sista noden i sektionen. På så sätt slumpas en ny väg ut inom den lokalt utvalda sektionen, vilket gör grannvägen till en likadan väg som den ursprungliga, fast med en modifikation på den utvalda sektionen. ***Figur 7*** visar hur detta kan se ut. Samma metod har använts i detta arbete, eftersom det tillåter en att återanvända samma funktion istället för att skriva en ny funktion som i stort sett uppfyller samma syfte.



Figur 7. Demonstration över hur grannvägar skapas. Den vänstra linjen är den ursprungliga vägen. De två horisontella linjerna på vägen i mitten är den slumpmässigt valda sektionen som ska justeras och den röda linjen i vägen till höger är den slumpade vägen mellan den valda sektionen. Den högra vägen är en grannväg genererad från den vänstra vägen.

Sammanfattningsvis fungerar algoritmen på så sätt att vägar slumpgenereras och sparas undan i en lista. Arbetarbina genererar sedan grannvägar till var och en av dessa. En girig selektion sker mellan grannvägen och den ursprungliga vägen, där vägen som innehåller minst noder, det vill säga den kortaste vägen, väljs. Om grannvägen är den kortaste vägen ersätter den alltså den ursprungliga vägen i listan. Åskådarbina väljer sedan en väg baserat

på roulettehjulsselektionen och skapar en grannväg till den valda vägen. Även här sker en girig selektion. Efter varje cykel sparas den hittills bästa funna vägen undan i en variabel. Om en väg i listan inte har ersatts av en grannväg efter lika många cykler som arbetarbinas gränsvärde, ersätts denna av ett utforskarbi som slumpgenererar en helt ny väg.

4.2.1 Slumpgenerering av vägar

Som nämnts i 2.2.2 består Artificial Bee Colony Algorithm utav tre typer av bin som har varsin uppgift: Arbetarbin, åskådarbin och utforskarbin. Varje bi är beroende av slumpgenerering. Arbetarbiet initierar en slumpad lösning och skapar sedan en slumpad grannlösning till denna lösning. Åskådarbiet väljer ett arbetarbis lösning genom roulettehjulsselektion och skapar också en slumpad grannlösning till den valda lösningen. Utforskarbiet slumpar fram en ny lösning när ett arbetarbi överstigit sitt gränsvärde av antal försök. För att kunna applicera ABC på vägplanering där varje lösning representeras av en väg behövs alltså någon form av slumpgenerering som genererar vägar. Algoritmens syfte förloras om alla vägar som initieras är identiska.

Ett sätt att slumpgenerera vägar är genom att använda sig av sökalgoritmen Random Walk (Wales & Sanger, 2015) som tar fram vägar genom att börja från startnoden och slumpa varje nästa steg som tas tills nästa nod som besöks är destinationsnoden. Algoritmen får alltid fram en unik och fullständig väg, förutsatt att den tilldelas oändligt med söktid. Eftersom söktiden kan bli oändlig är detta dock ingen optimal lösning för dataspel, som är beroende av tidsåtgång (Se 2.3.1).

I en studie där man ville undersöka vägplanering på öppna plan med hjälp av ABC, tas samma problem med slumpgenerering av vägar upp (Eldrandaly, AbdelAziz, & Hassan, 2015). Där har man valt att minska på tidsåtgången för att initiera vägar genom att skapa en riktad slumpgenerering av vägar. Detta minskar söktiden markant eftersom algoritmen hela tiden rör sig riktad åt destinationsnoden och avståndet mellan nuvarande noden och destinationsnoden minskar med tiden. Eftersom denna metod ger en optimerad slumpgenerering av vägar så skapas en metod som är inspirerad av denna metoden för detta problem.

Slumpgenereringen av vägar för detta problem fungerar på så sätt att den räknar ut riktningen mellan nuvarande noden och destinationsnoden. Om grannoden i den riktningen är en marknod sätts denna nod till den nuvarande noden. Är grannoden i riktningen istället en förbjuden nod, slumpas en annan riktning ut, tills den funnit en grannod som är en marknod. Denna nod sätts då till den nuvarande noden. Beteendet upprepar sig tills en grannod är samma nod som destinationsnoden. När destinationsnoden är funnen har en komplett väg blivit funnen.

Ett stort problem med denna metod är att det kan leda till att algoritmen fastnar i återvändsgränder. Om en djup korridor (Se ***Figur 8***) blockerar vägen i riktningen mellan den nuvarande noden och destinationsnoden, kommer algoritmen att röra sig in i korridoren tills den stöter på väggen. Vid väggen kommer en ny riktning att slumpas fram som tillslut sätter den nuvarande noden i riktning från destinationsnoden. Efter detta kommer riktningen mellan den nuvarande noden och destinationsnoden att räknas ut igen, vilket kommer leda till att algoritmen rör sig mot väggen igen. Beteendet kommer upprepa sig i all oändlighet och en färdig väg kommer aldrig att bildas. För att lösa detta problem behövs en

algoritm som tar hänsyn till dessa extremfall. Detta kan dock leda till nya problem och högre tidsåtgång, som i sin tur måste lösas.

Figur 8. Exempel på en återvändsgränd. Den gula cirkeln är den nuvarande noden och den gröna är destinationsnoden. De svarta linjerna är väggar. Den blå pilen visar hur den gula noden kommer att förflytta sig.

Efter mycket avvägningar och försök till bättre slumpgenerering av vägar har beslutet tagits att algoritmen bara kommer att testas på banor utan återvändsgränder. Fokusen för detta arbete ligger i vägplanering med Artificial Bee Colony Algorithm och inte slumpgenerering av vägar. Att lägga all fokus och arbetstid åt slumpad väggenerering leder till att de viktiga delarna av arbetet blir lidande. Det är således inte värt mödan.

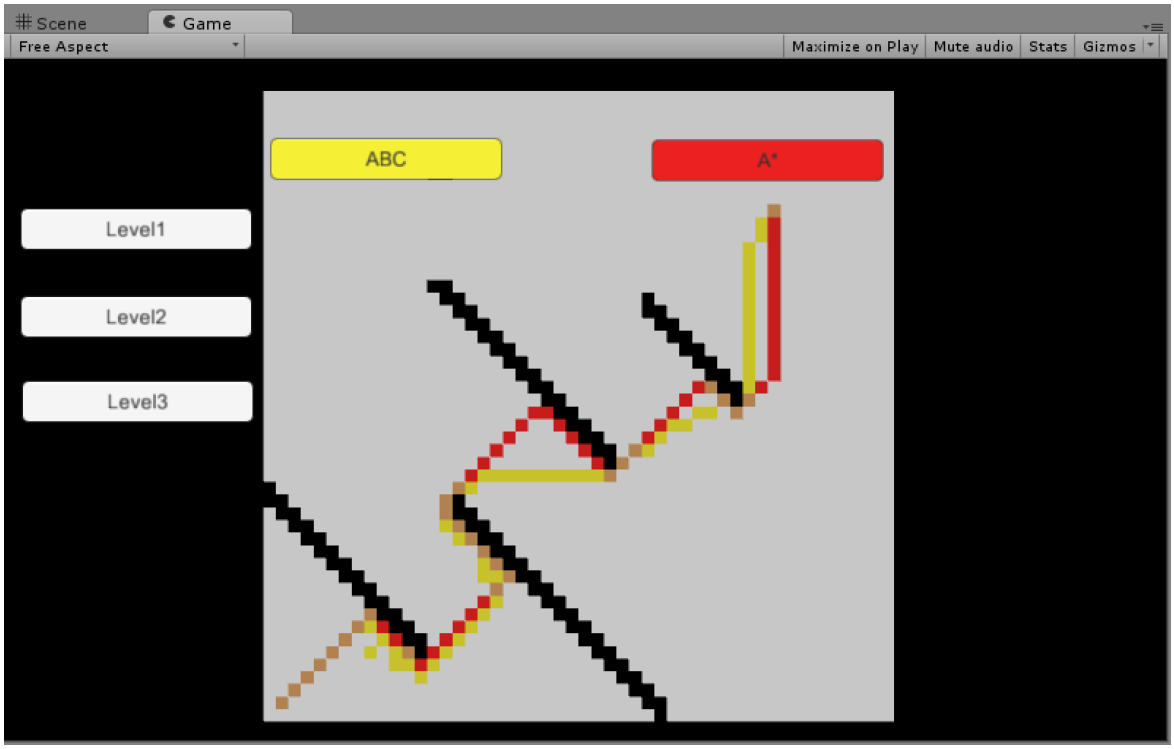
4.3 Implementering av A*

A* är en vanligt förekommande algoritm och en stor mängd dokumentation och exempelkod finns tillgänglig på internet. För att implementera algoritmen har bland annat ***Figur 2*** använts som inspiration, men även en webbartikel vid namn *A* Pathfinding for Beginners* (Lester, 2005).

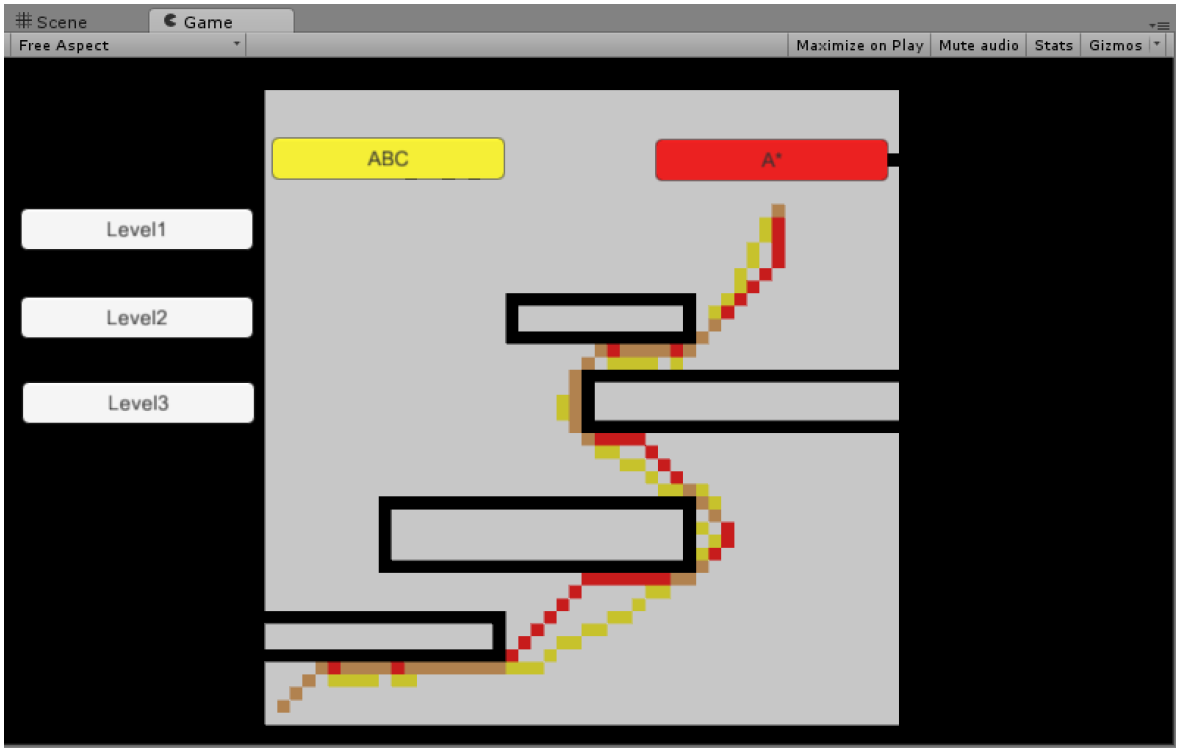
En svårighet med att implementera en A*-algoritm är att få den så optimal som möjligt. Algoritmen består utav en lista med noder som ska sökas igenom varje loop. Den noden som väljs ut från listan är alltid noden med lägst F-värde, som tas upp i 2.3.2. För att avgöra vilken nod som har lägst F-värde är ett alternativ att söka igenom alla noder i listan och jämföra deras F-värde med varandra, för att sedan hämta ut den med lägst F-värde. Ett annat alternativ är att sortera noderna baserat på F-värdet innan insättning så att det högsta värdet hamnar längst fram i listan och det lägsta värdet hamnar längst bak. När en nod ska hämtas ut väljs då den sista noden i listan, som alltid är den med lägsta F-värdet. Det första alternativet innebär att ingen förskjutning sker i listan vid insättning eftersom element sätts in längst bak i listan. Dock innebär det att både sökning och eventuell förskjutning av elementen sker vid uttagning från listan eftersom alla element som är bakom det utvalda elementet måste förskjutas ett steg framåt. Vid det andra alternativet tar man dock ut det sista elementet vid uttagning, vilket innebär att det inte sker några förskjutningar vid uttagning, utan enbart eventuellt vid insättning. Det andra alternativet är det mest optimala och används därför vid detta arbete.

4.4 Utvärderbarhet

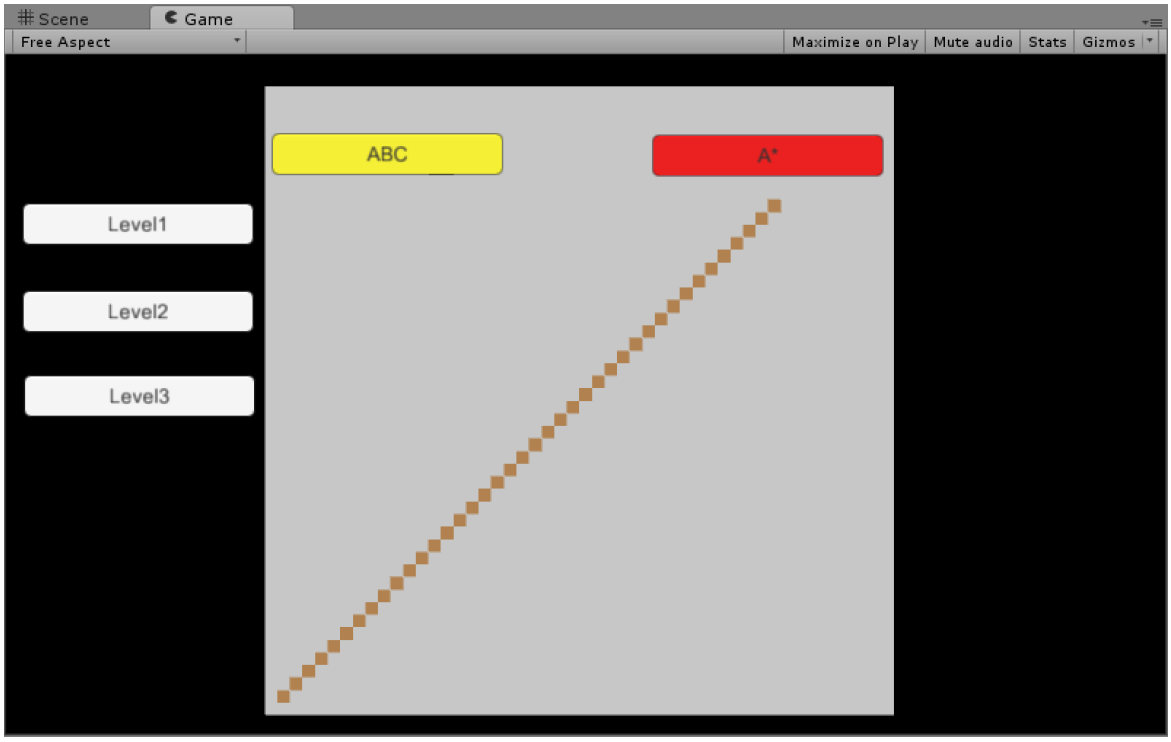
En pilotversion av utvärderingen har implementerats för att testa arbetets utvärderbarhet. I denna version är tre olika banor sparade i txt-filer. Genom att klicka på respektive algoritms knapp körs den valda algoritmen på banan och när algoritmen är färdig ritas vägen ut i algoritmens knappfärg. Exempel på hur det ser ut när de tre olika banorna testats i pilotstudien går att se i ***Figur 9***, ***Figur 10*** och ***Figur 11***.



Figur 9. Körning av de två algoritmerna på bana 1. Gult är vägen ABC får fram, rött är vägen A* fram. Brun färg är där de två algoritmerna korsar varandra över samma nod. De svarta partierna på banan är hinder.



Figur 10. Körning av de två algoritmerna på bana 2. Gult är vägen ABC får fram, rött är vägen A* fram. Brun färg är där de två algoritmerna korsar varandra över samma nod. De svarta partierna på banan är hinder.



Figur 11. Körning av de två algoritmerna på bana 3. Bana 3 är ett öppet plan utan några hinder och de två algoritmerna får fram exakt samma väg.

Som figurerna visar får algoritmerna inte alltid fram samma vägar som varandra. På banor med hinder får ABC även fram olika resultat varje gång den körs eftersom den använder sig av slumpgenerering. A* får däremot alltid fram samma väg hur många gånger den än körs. Redan vid pilotstudien är det tydligt att ABC tar längre tid på sig än vad A* gör på banor med mycket hinder. Detta sker dock på en bikolonistorlek på 50, ett gränsvärde på 10 och 10 cykler (Se 2.2.2) och på ett begränsat antal banor. För den slutgiltiga undersökningen som presenteras i 5.1, har dessa parametrar justerats enligt metoden på 3.2.2 för att få en bättre balans mellan dessa och en kortare tidsåtgång. Dessutom har algoritmerna provats på fler banor än dessa tre i den slutgiltiga undersökningen. Båda algoritmerna är dessutom i ett mer optimerat stadie än algoritmerna som använts vid pilotstudien.

För att få en mer exakt bild av hur stor skillnaden på tidsåtgången är, har en mer ordentlig tidmätning gjorts i kapitel 5. Där startar en klocka när algoritmen körs och stannar när algoritmen fått fram den slutgiltiga vägen.

Den grafiska utritningen tillåter en att klart och tydligt se ifall vägarna är estetiskt tilltalande eller inte, samt estimera hur långa vägarna är. För att lättare få en exakt längd på vägarna skrivs antalet element den färdiga vägen innehåller ut i den slutgiltiga mätningen.

I kapitel 3 nämns det att frågeställningen besvaras med avseende på tidsåtgången, väglängden och hur de resulterande vägarna ser ut. Denna pilotstudie visar att frågeställningen går att besvara med hjälp av denna implementation och att arbetet således är utvärderbart.

5 Utvärdering

I detta kapitel listas resultat från den slutgiltiga undersökningen. Delvis presenteras resultaten på mätningarna men också figurer som demonstrerar hur de resulterande vägarna ser ut. Kapitlet innehåller en analys som tar upp samband mellan resultaten och förklarar även varför resultaten ser ut som de gör. Kapitlet avslutas med en slutsats som kopplar resultaten till frågeställningen som tagits upp tidigare.

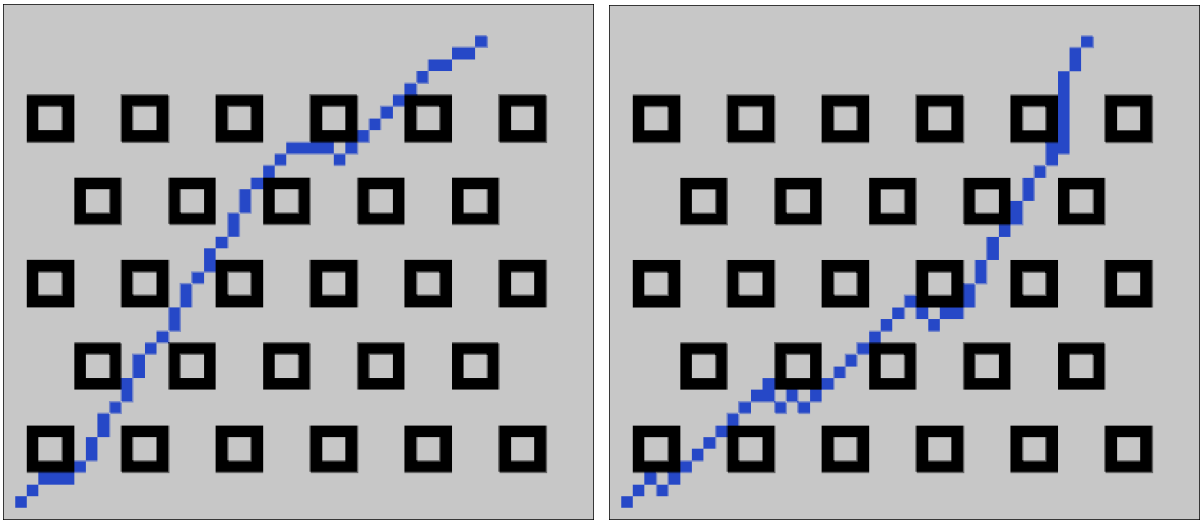
5.1 Presentation av undersökning

Till denna undersökning har de fyra miljöerna som tas upp i 3.2.1 byggts upp med hjälp av miljöverktyget som skapades. För att undvika specialfall och få en rättvis och generell mätning har vardera algoritm körts 1000 gånger var på varje spelmiljö. Exekveringstiden i millisekunder och den resulterande vägens längd i antal förflyttningar från start till slutnod, har sparats undan för varje testkörning. Medelvärdet för tidsåtgången och väglängden från de 1000 körningarna har sedan räknats ut för att få fram det genomsnittliga resultatet för en körning. De resulterande medelvärdena går att se i ***Tabell 1***.

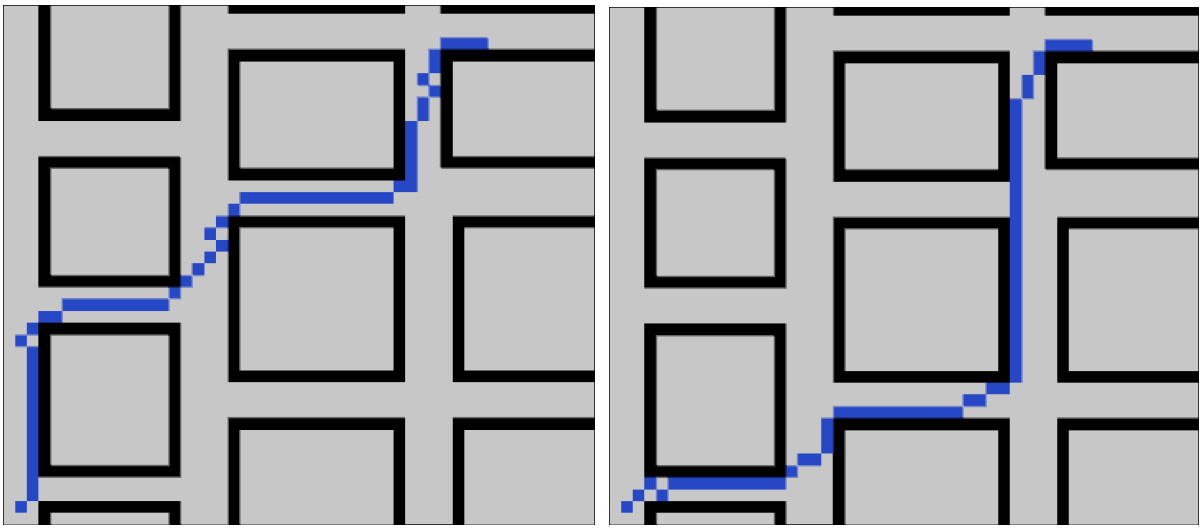
Som nämnts i 2.3.2 får A* alltid fram den kortaste möjliga vägen som existerar. Den har därför alltid fått fram samma identiska väg för en specifik bana i varje testfall som körts i denna undersökning. Eftersom ABC använder slumpgenerering och rouletteselektion (Se 4.2) får den i de flesta fallen fram unika vägar vid varje körning (Se ***Figur 12*** och ***Figur 13***). Parametrarna för ABC som tas upp i 2.2.2 har justerats enligt metodbeskrivningen i 3.2.2 tills de fått en så kort väglängd som möjligt utan att få en exekveringstid som överstiger exekveringstiden för A* så pass mycket att det blir en märkbart stor skillnad.

Tabell 1. Resultaten efter 1000 körningar av algoritmerna på varje testmiljö. De listade siffrorna är medeltalet för de 1000 resulterande talen. Exekveringstiden representeras i millisekunder och väglängden representeras i antal förflyttningar.

	ABC Exekveringstid	A* Exekveringstid	ABC Väglängd	A* Väglängd
Öppet plan	0,1 ms	5,78 ms	40	40
Skog	10,46 ms	8,23 ms	51,39	49
Stad	191,22 ms	101,53 ms	68,23	64
Labyrint	940,9 ms	335,58 ms	111,17	68



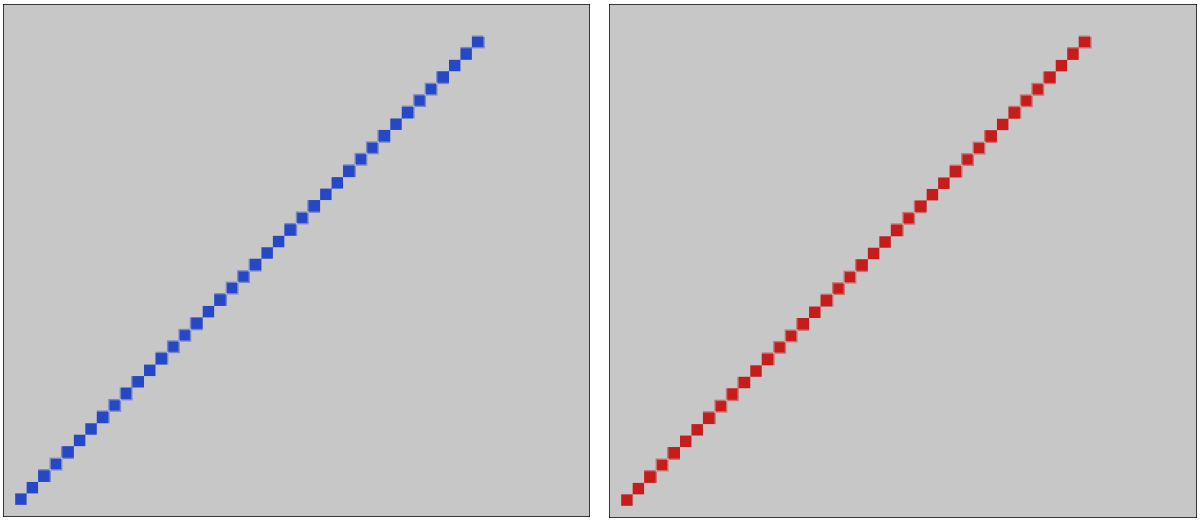
Figur 12. Exempel på hur de resulterande vägarna ABC får fram kan variera på samma miljö.



Figur 13. Exempel på hur de resulterande vägarna ABC får fram kan variera på samma miljö.

5.1.1 Öppet plan

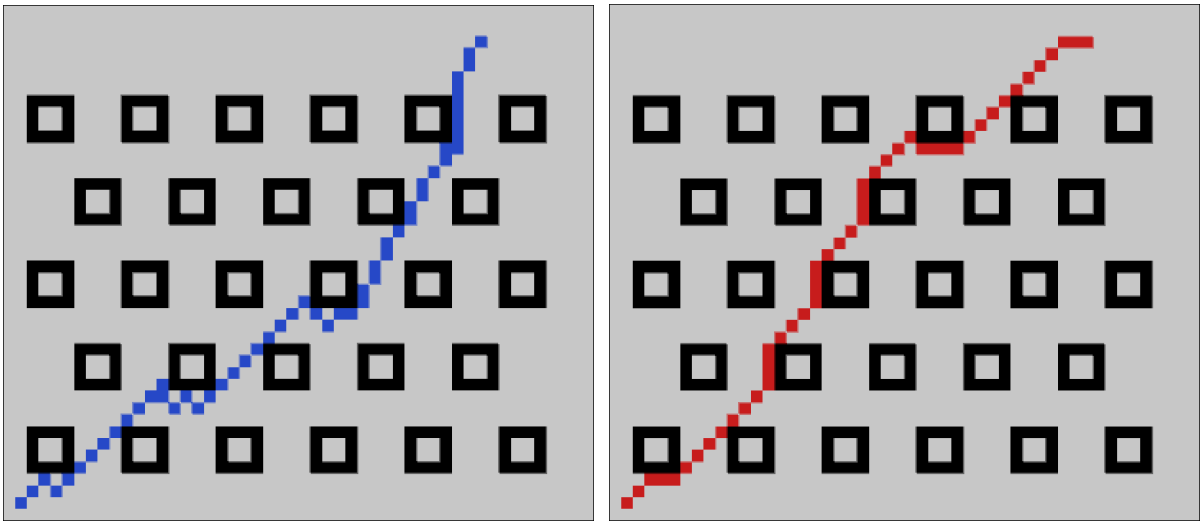
På det öppna planet får båda algoritmerna alltid fram samma resulterande väg på 40 förflyttningar (Se ***Figur 14***) oavsett vilka parametrar som sätts på ABC-algoritmen. Detta beror på att det inte finns några hinder i miljön och att slumpfunktionen som tas upp i 4.2.1 därför aldrig behöver kallas på i ABC-algoritmen. Parametrarna för ABC har satts så lågt som möjligt eftersom alla alternativa lösningar ABC räknar fram i den öppna miljön kommer att se likadana ut (Se 4.2.1). Bikolonin har därför fått en storlek på 2 bin som körts i 1 cykel med ett gränsvärde på 1. Eftersom parametrarna har väldigt låga värden och ingen slumpgenerering krävs har algoritmen fått en väldigt kort exekveringstid med ett genomsnitt på 0,1 millisekunder. A* har fått en lite längre genomsnittlig tid på 5,78 millisekunder.



Figur 14. De resulterande vägarna på det öppna planet. Den blå vägen till vänster är framställd med hjälp av ABC och den röda till höger är framställd med hjälp av A*.

5.1.2 Skog

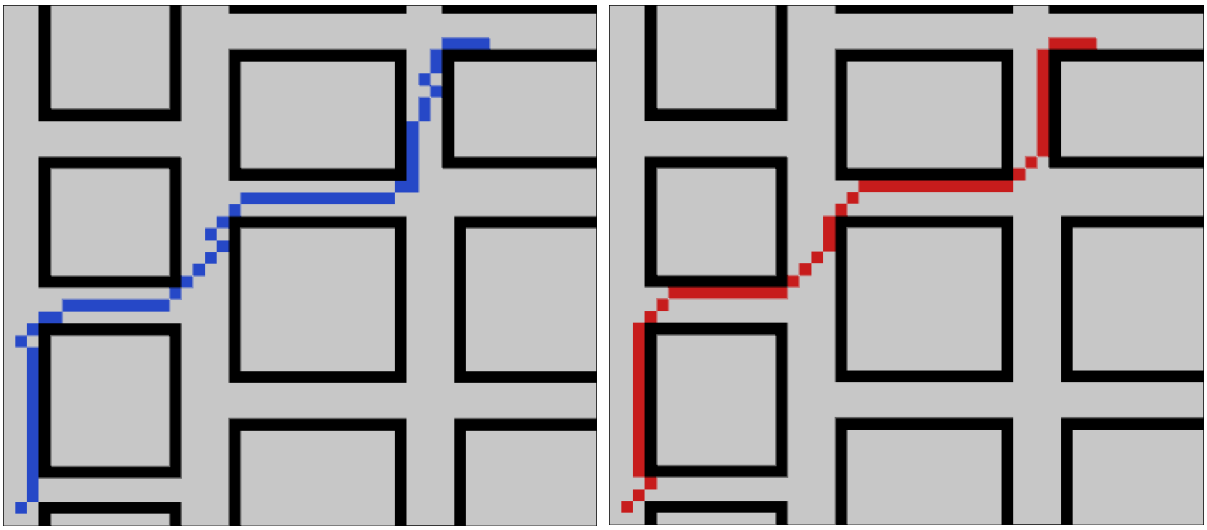
När algoritmerna körs på skogsmiljön skiljer sig resultaten åt något. ***Figur 15*** visar ett exempel på en väg ABC har fått fram och även vägen som A* alltid får fram i miljön. För resultatet i figuren och ***Tabell 1*** har ABC en bikoloni på storlek 12 med 7 cykler och ett gränsvärde på 4. A* får både kortare vägar och en kortare tidsåtgång än ABC på denna miljö med en väg på 49 förflyttningar och en genomsnittlig tidsåtgång på 8,23 millisekunder, medan ABC får en genomsnittlig väglängd på 51,39 förflyttningar och en genomsnittlig tidsåtgång på 10,46 millisekunder.



Figur 15. Exempel på resulterande vägar på skogsmiljön. Den blå vägen till vänster är framställd med hjälp av ABC och den röda till höger är framställd med hjälp av A*.

5.1.3 Stad

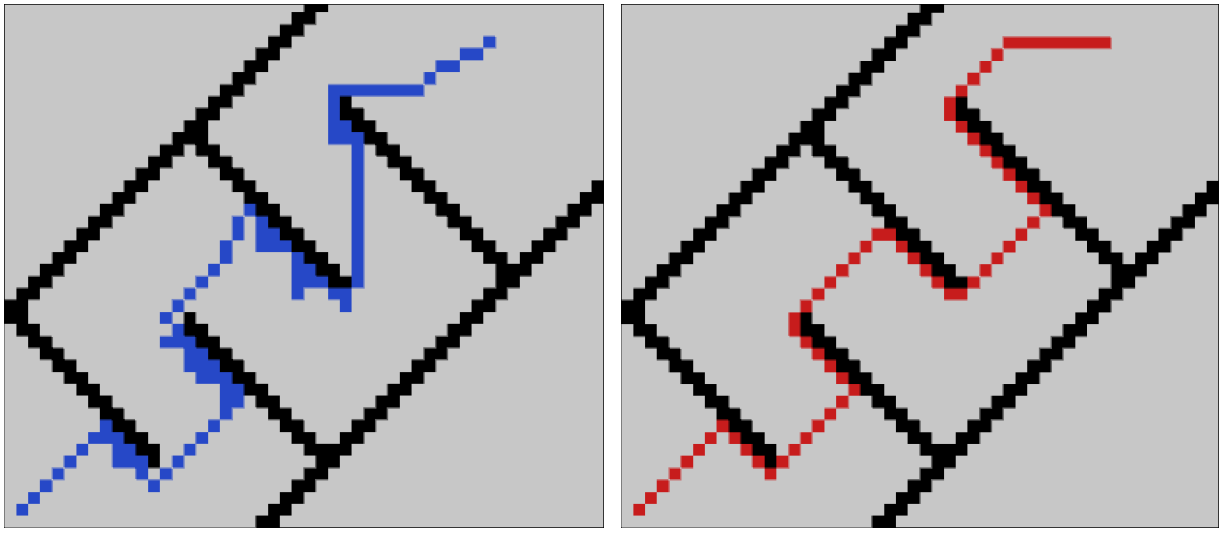
På stadsmiljön skiljer sig resultaten åt lite mer än på skogsmiljön. I denna miljö presterar ABC fortfarande sämre än A*, men med större marginal. Här får ABC en genomsnittlig tidsåtgång på 191,22 millisekunder och en genomsnittlig väglängd på 68,23 förflyttningar. A* får däremot en genomsnittlig tid på 101,53 millisekunder och en genomsnittlig väglängd på 64 förflyttningar. Trots att A* presterar mycket bättre än ABC får båda algoritmerna fram en väg som är estetiskt tilltalande på denna miljö, som man kan se i ***Figur 16***. För denna miljö har ABC en bikoloni på storlek 26 som körs i 22 cykler med ett gränsvärde på 12.



Figur 16. Exempel på resulterande vägar på stadsmiljön. Den blå vägen till vänster är framställd med hjälp av ABC och den röda till höger är framställd med hjälp av A*.

5.1.4 Labyrint

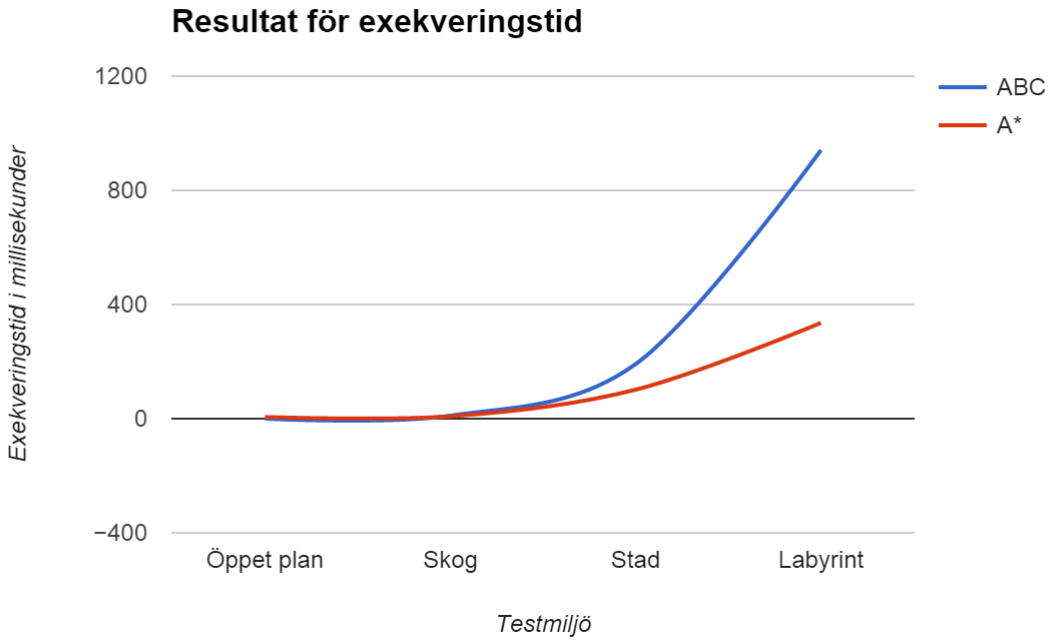
I labyrintmiljön får båda algoritmerna en väldigt hög tidsåtgång. A* presterar dock betydligt bättre än ABC med en genomsnittlig tidsåtgång på 335,58 millisekunder och en väglängd på 68 förflyttningar. Med en bikoloni på 12 bin som körs i 12 cykler med ett gränsvärde på 7, får ABC en genomsnittlig tidsåtgång på hela 940,9 millisekunder och dessutom en genomsnittlig väglängd på 111,17 förflyttningar. Skillnaderna mellan de två algoritmerna är som störst på denna miljö. Som man kan se på ***Figur 17*** får ABC inte bara långa vägar, utan även vägar som går i omvägar och som är estetiskt otilltalande. Dessa omvägar sker när algoritmen krockar i hinder eftersom det är där slumpgenereringen sker. Det är möjligt att skapa kortare vägar med färre omvägar på denna miljö genom att öka värdet på parametrarna, men det kräver att tidsåtgången blir ännu högre än den redan är.



Figur 17. Exempel på resulterande vägar i labyrinten. Den blå vägen till vänster är framställd med hjälp av ABC och den röda till höger är framställd med hjälp av A*.

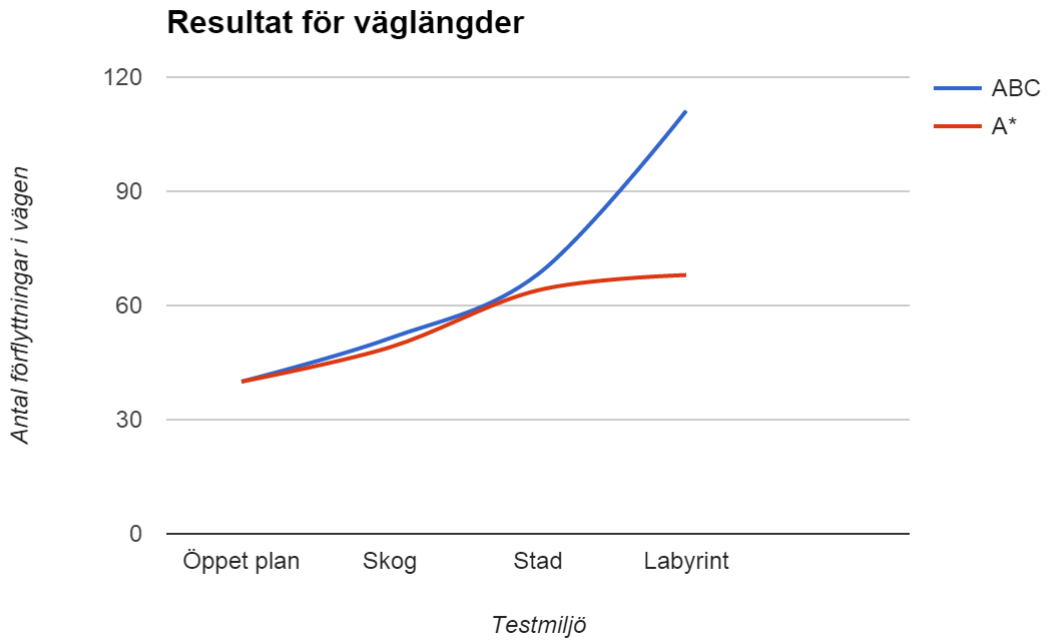
5.2 Analys

Resultatet från 5.1 visar att både ABC och A* presterar bättre på miljöer som är mer öppna och har små hinder, medan de presterar sämre på korridorliknande miljöer som har stora och breda hinder. Detta gäller både när det kommer till tidsåtgång och väglängder. Baserat på diagrammet i ***Figur 18*** skiljer sig tidsåtgången inte åt så mycket mellan algoritmerna på de mer öppna miljöerna. När hindrena blir bredare eskalerar dock tidsåtgången för ABC mycket mer än vad den gör för A*.



Figur 18. Diagram för tidsåtgången för de två algoritmerna baserat på resultaten i **Tabell 1**. Blå linje representerar ABC och röd linje representerar A*.

När det gäller den resulterande väglängden för de två algoritmerna ligger A* hela tiden lika bra eller bättre till än ABC enligt ***Figur 19***. Detta beror på att A* hela tiden får fram den kortaste möjliga vägen, medan vägen ABC får fram kan variera eftersom den framställs med hjälp av slumpgenerering när hinder existerar. Precis som med tidsåtgången eskalerar skillnaden mellan de två algoritmerna ju bredare hindrena i miljön blir.



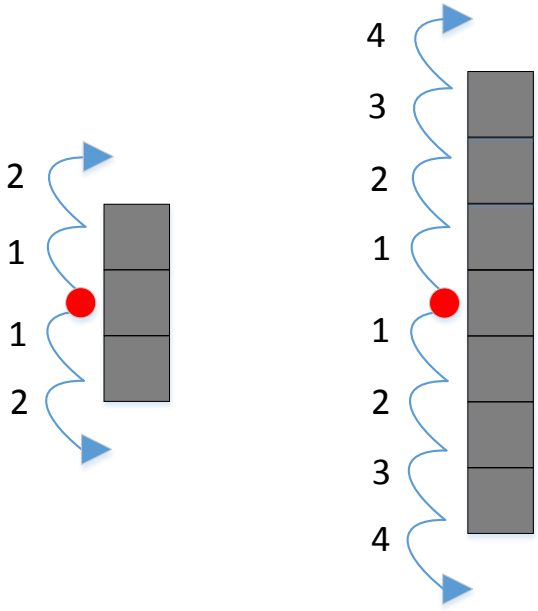
Figur 19. Diagram för resultaten av väglängder för de två algoritmerna baserat på resultaten i **Tabell 1**. Blå linje representerar ABC och röd linje representerar A*.

Trots att A* har fått en bättre genomsnittlig tidsåtgång än ABC vid mätningarna går det nästan alltid att få en lägre tidsåtgång med hjälp av ABC än A*. Detta genom att sätta låga värden på ABC:s parametrar. Detta innebär dock att vägarna kommer bli extremt långa vid miljöer som inte är helt öppna. Vägarna kommer dessutom vara estetiskt otilltalande på samma sätt som vägen i ***Figur 17***. Ju lägre parametrarnas värden är, desto kortare blir tidsåtgången, men desto längre blir vägarna vid miljöer där algoritmen stöter på hinder. Ju högre parametrarnas värden är, desto bättre blir de resulterande vägarna men till följd av en hög tidsåtgång som sträcker sig högt över A*. Eftersom A* alltid får den kortaste möjliga vägen är det inte värt att öka ABC:s parametrar så att algoritmens tidsåtgång överstiger tidsåtgången för A*, då vägarna aldrig kommer kunna bli kortare än vägarna A* får fram.

5.2.1 Slumpgenerering vid breda hinder

Anledningen till att ABC presterar avsevärt sämre på miljöer med stora och breda hinder än A* är att slumpgenereringen, som tas upp i 4.2.1, måste slumpa fram en förflyttning flera gånger på raken och att förflyttningarna dessutom måste resultera i en sekvens som gör att vägen leder ut ur hindret. Om algoritmen exempelvis krockar i ett hinder (Se ***Figur 20***) som fortsätter en nod åt höger och en nod åt vänster behöver den enbart slumpa sig förbi två noder, den som den krockat i och antingen noden till höger eller vänster. Blockerar hindret istället tre noder till höger och vänster behöver den slumpa sig åt rätt håll minst fyra gånger

på raken. Ju fler gånger på raken algoritmen måste slumpa förflyttningen åt ett specifikt håll, desto mindre är sannolikheten att det sker. Detta leder till att algoritmen behöver fler försök på sig att slumpa förflyttningarna, vilket leder till en högre genomsnittlig tidsåtgång.



Figur 20. Exempel på två olika hinder med en bredd på tre och en bredd på 7 och hur många förflyttningar algoritmen minst måste slumpa fram i en bestämd sekvens. Den röda punkten är vart krocken sker. De blå pilarna är förflyttningarna.

5.2.2 Prestandan på A*

Resultatet från denna undersökning baserar sig enbart på implementationen för detta arbete. I 5.1 visar resultaten att det tog över 300 millisekunder för A* att söka fram en väg på labyrintmiljön som är 50x50 noder stor. I en annan variant av A*, implementerad av Christoph Husse (Husse, 2010), har algoritmen en tidsåtgång på 300 millisekunder på en testmiljö på hela 512x512 noder. Det är alltså möjligt för A* att uppnå ännu bättre resultat än vad den uppnått från detta experiment. Detta innebär att ABC är desto sämre lämpad för vägplanering i spel än vad A* är. Samtidigt exkluderar inte detta möjligheten att en betydligt bättre variant av ABC kan implementeras och ge lika bra resultat som den bättre varianten av A*.

5.3 Slutsatser

I kapitel 3 tas en frågeställning upp som lyder: *Är Artificial Bee Colony Algorithm en bättre algoritm för vägplanering i dataspel än A*?* Denna fråga besvaras med avseende på tidsåtgången det tar för algoritmerna att genomföra sökningen av vägarna samt hur korta dessa resulterande vägar blir. Ju lägre tidsåtgång och ju kortare vägar, desto bättre är algoritmen.

I 5.1 presenteras resultaten från testningar som har utförts enligt metodbeskrivningen på 3.2. Resultaten visar att ABC presterar sämre på samtliga fyra miljöer algoritmen testats på,

förutom på miljön som saknar hinder. Att algoritmen får ett bättre resultat på denna miljö beror dock på att dess parametrar (Se 2.2.2) sätts så lågt som möjligt. Hade man enbart kört den riktade väggenereringen, som beskrivs i 4.2.1, en enda gång utan att ens använda ABC:s egenskaper hade samma väg fåtts fram men med ännu bättre tidsåtgång. Att detta blir ett bra resultat beror alltså inte på egenskaperna hos ABC, utan den riktade väggenereringen och har därför inget med ABC att göra.

Eftersom alla former av banor med korridorliknande återvändsgränder har fått slopas (Se 4.2.1) begränsar detta vilka typer av spel ABC ens kan användas till för vägplanering. A* kan däremot finna vägar på alla former av miljöer och är således en mer anpassningsbar och mer pålitlig algoritm för vägplanering i dataspel än ABC. Eftersom designen på många dataspel ändras under utvecklingens gång är det säkrare att använda A* ifall man senare vill lägga till återvändsgränder i ett spel som ursprungligen inte är tänkt att ha det.

Eftersom A* ger ett bättre resultat på alla miljöer förutom den miljön där ABC:s egenskaper knappt utnyttjas, kan man dra slutsatsen att ABC inte är en bättre algoritm för vägplanering i dataspel än A*, både när det gäller tidsåtgång och väglängd, men också vad gäller anpassningsbarhet och pålitlighet. Faktumet att det finns ännu mer effektiva varianter av A* än den varianten som använts för denna undersökning (Se 5.2.2) stödjer slutsatsen ännu mer.

6 Avslutande diskussion

Detta kapitel innehåller en sammanfattning av undersökningen från frågeställning till slutsats. Därefter finns en diskussion som diskuterar resultatet i ett större sammanhang utanför problemformuleringen. Kapitlet avslutas med en diskussion om en hypotetisk fortsättning på arbetet både i ett kort och långt perspektiv.

6.1 Sammanfattning

Syftet med detta arbete har varit att ta reda på om ABC är en bra algoritm för vägplanering i dataspel genom att jämföra den med A*, som är en av de vanligaste algoritmerna som används för vägplanering i dataspel. Detta eftersom man alltid vill finna mer effektiva algoritmer för vägplanering om det är möjligt. För att ta reda på ifall ABC är bättre än A* för detta problem har en experimentmiljö bestående av fyra vanliga spelmiljöer skapats: Ett öppet plan utan några hinder, en skog, en stad och en labyrint. De aspekterna som algoritmerna har jämförts på är tidsåtgången för att räkna ut vägarna på miljöerna samt antalet förflyttningar varje resulterande väg kräver för att nå destinationen. På grund av att ABC kräver slumpgenerering av vägar som kan ta oändligt mycket tid har en riktad variant av slumpgenereringen skapats. Detta leder dock till att algoritmen inte fungerar på vägar med återvändsgränder. På grund av detta har algoritmen inte testats på några miljöer med återvändsgränder.

Testerna visar att A* presterar lika bra eller bättre på samtliga miljöer förutom på det öppna planet utan hinder. Algoritmerna ger liknande resultat på mer öppna miljöer med få och små hinder men ABC presterar betydligt sämre än A* ju större hindrena är och ju mer korridorliknande miljöerna blir. Detta beror främst på hur slumpgenereringen hanteras vid kollision med hinder. Eftersom alla riktningarna slumpas fram krävs det att de slumpas i en sekvens som gör att man tar sig förbi hindret och inte återvänder tillbaka till det. Vid riktigt stora hinder blir det ännu svårare för slumpgenereringen att ske i rätt sekvens för att ta sig förbi hindret eftersom det kräver fler korrekta förflyttningar. Detta gör att både tidsåtgången ökar och att de resulterande vägarna blir längre och tar fler onödiga omvägar.

Anledningen till att ABC presterar något bättre på öppna miljön är att det inte finns några hinder och att slumpgenereringen aldrig används. Eftersom slumpgenerering är en viktig del i ABC förstör detta hela syftet med ABC och samma väg hade kunnats få fram ännu snabbare enbart med den riktade vägplaneringen som används till slumpgenereringen.

Eftersom A* presterar bättre på alla miljöer utom den miljön där ABC:s egenskaper inte utnyttjas kan man dra slutsatsen att ABC inte är bättre än A* för vägplanering i dataspel.

6.2 Diskussion

Det finns flera sätt att implementera ABC för vägplanering i dataspel på. Varianten som är implementerad för denna undersökning är bara en av många möjliga. Man kan enbart dra slutsatsen att algoritmen inte ger ett bättre resultat än A* för varianten som implementerats för denna undersökning. Om man på något sätt lyckas implementera en variant av ABC som hanterar initieringen av vägar bättre än denna variant, som förklaras ingående i 4.2, så finns det möjligheter till att skapa en ABC-algoritm som ger bättre resultat än vid denna undersökning. Hur denna initiering kan förbättras förklaras mer ingående i 6.3.

Baserat på detta kan man alltså inte dra slutsatsen att ABC aldrig är bättre än A* för vägplanering i dataspel. Däremot kan man dra slutsatsen att denna lösning inte är det och att det stödjer tesen att ABC inte är en bra algoritm för vägplanering i dataspel. Att avfärda ABC helt och hållet för vägplanering i dataspel för all framtid baserat på en enda variant av algoritmen vore inte etiskt försvarbart, delvis mot Karaboga som uppfunnit algoritmen (Karaboga, 2005), men också mot forskare och spelutvecklare som vill utforska algoritmen i framtiden.

6.2.1 Skräpsamling

En sak man måste ta hänsyn till är att implementationen för denna undersökning är skriven i programmeringsspråket C#. Detta språk använder sig av så kallad skräpsamling, vilket innebär att programmet söker igenom minnet efter block som inte används längre och frigör blocket så att det kan användas till nya dataobjekt. Detta sker automatiskt och styrs inte av programmeraren men tar upp en stor del CPU-tid (Unity Technologies, 2015). Om skräpsamlingen tar upp mer CPU-tid i en utav algoritmerna än den andra kan detta ge algoritmen sämre tidsåtgång i jämförelse mot den andra algoritmen än om den hade implementerats i till exempel C++ eller Java. Det finns alltså en risk att resultatet har blivit försämrat på grund av C#.

6.2.2 Relaterad forskning

Den relaterade forskningen som tas upp i 2.2.4 ger enbart positiva resultat med ABC, vilket inte överensstämmer med resultatet från denna undersökning. Att Sonmez undersökning ger positiva resultat beror på att algoritmen tillämpas på ett problem som inte fokuserar på tidsåtgång. Artikeln tar dessutom upp att algoritmen presterar bra, men inte särskilt snabbt (Sonmez, 2010). Att ABC ger goda resultat på undersökningarna som tillämpar algoritmen på handelsresandeproblemet och vägplanering för robotar kan bero på att algoritmen enbart jämförs med algoritmer som vanligtvis inte används till vägplanering i dataspel, däribland genetiska algoritmer, partikelsvärmsalgoritmer och differentiella evolutionsalgoritmer (Bhattacharjee, Rakshit, & Goswami, 2011; Karaboga & Gorkemli, 2011; Liang & Lee, 2014; Ma & Lei, 2010). Ingen av dessa arbeten har jämfört ABC med A*.

Som nämnt i kapitel 3 har ABC jämförts med A* i detta arbete för att det är en av de vanligaste algoritmerna som används för vägplanering i dataspel. ABC har alltså bedömts kritiskt och objektivt av etiska själ för att inte ge en vilseledd och partisk bild av ABC. Detta minskar risken för att man sprider budskapet att algoritmen är bättre lämpad för vägplanering än den egentligen är och att till exempel spelutvecklare använder algoritmen till vägplanering i sina spel i felaktigt troende om att den kommer prestera bättre än A*.

6.2.3 Forskningsmetod och etik

Att använda sig av upphovsskyddat material utan tillåtelse är inte en etiskt korrekt forskningsmetod. Algoritmerna för denna undersökning har implementerats med hjälp av ett antal inspirationskällor som tas upp i 4.2 och 4.3. Dessa inspirationskällors upphovsmän är dock tydligt refererade till och trots att källorna har använts som inspiration, har implementationen för detta arbete skapats från grunden utan att kopiera källkod rakt av från dessa källor.

Eftersom inga försökspersoner har använts till detta arbete, har arbetet heller inte påverkat människors fysiska eller psykiska hälsa negativt. Då arbetet tas upp i detalj har den även en hög återupprepningsbarhet för forskare som eventuellt vill återupprepa arbetet. Vissa delar

av arbetet hade dock kunnat göras tydligare för ännu högre återupprepningsbarhet. Däribland tydligare specifikation på kodstruktur och hårdvaran som använts för denna undersökning.

Sammanfattningsvis kan man dra slutsatsen att arbetet har genomförts med etiska aspekter i åtanke.

6.3 Framtida arbete

Det svåraste hindret vid detta arbete har varit implementeringen av slumpgenereringen för ABC som tas upp i 4.2.1. Något som skulle kunna göras för framtida arbete är därför att lägga fokus på att slumpgenerera vägar på ett bättre sätt. Fokusen skulle delvis kunna ligga på att optimera slumpgenereringen. Minskar man tidsåtgången för att initiera en enda slumpad väg minskar man även tidsåtgången för hela algoritmen avsevärt eftersom initieringen körs flera gånger i algoritmen. En annan sak man även skulle kunna lägga fokus på är att lösa problemet med att initieringen av vägar inte fungerar på återvändsgränder. På så sätt skulle man även öka flexibiliteten av algoritmen som just nu bara lämpar sig för mer öppna miljöer. Om man lyckas förbättra dessa två aspekter skulle man kunna få fram en vägplanerare som ger mer varierade och mer estetiskt tilltalande vägar än A* utan att tidsåtgången blir lidande.

Något som även skulle kunna göras för framtida arbete är att prova algoritmen på fler och större miljöer än de fyra som använts för denna undersökning. Resultaten som fåtts fram baserar sig enbart på dessa miljöer. Testar man algoritmen på fler miljöer finns det en möjlighet att man finner ännu fler brister i den som går att åtgärda för att således få en ännu mer effektiv variant av algoritmen.

En annan sak som skulle kunna göras för att få en mer rättvis bedömning av resultatet är att låta testpersoner bedöma de resulterande vägarna algoritmerna får fram och peka ut vilken algoritms vägar dessa personer föredrar. På så sätt får man en tydligare bild över vilken algoritm som ger de mest estetiskt tilltalande vägarna.

Eftersom algoritmen för ABC som använts i denna undersökning ger ett sämre resultat än A* finns det ingen anledning att använda den för vägplanering i spel där syftet är att få fram en så billig väg som möjligt på så kort exekveringstid som möjligt. A* och många andra vägplaneringsalgoritmer får dock alltid fram samma identiska väg på en specifik bana hur många gånger de än körs. ABC genererar oftast en unik väg för varje körning vilket gör vägarna mindre förutsägbara. Vill man skapa ett dataspel med vägplanering där variation är viktigt för vägarna är ABC en alternativ lösning för detta i framtida spel.

Ett exempel på ett dataspel där varierad vägplanering är viktigt kan vara ett spel där flera AI-styrda spelkaraktärer ska springa längs samma miljö samtidigt. Springer alla exakt samma identiska väg kommer det att se väldigt robotstyrt och planerat ut, men om varje karaktär istället springer i sin egna unika väg kommer det att ge karaktärerna mer individualitet vilket bidrar till mer realism i spelet. Ett annat exempel där varierad vägplanering kan vara bra för spel är bosstrider. Om en boss i ett spel springer i ett bestämt mönster flera gånger kan spelaren efter ett tag börja förutse exakt hur bossen kommer röra sig framåt. Detta kan leda till att svårighetsgraden blir för låg. En vägplanering som genererar unika vägar varje gång ökar dock denna svårighet eftersom det gör det svårare för spelaren att förutse rörelsemönstret bossen har.

Om forskare någon gång i framtiden lyckas effektivisera denna algoritm så att den konkurrerar ut de andra vägplaneringsalgoritmerna, skulle den till och med kunna användas till framtida autopilotsystem för transportfordon. Detta är något som skulle gynna människor som inte har behörighet till att köra ett fordon eller som inte har tillgång till kollektivtrafik för att ta sig fram. Eftersom algoritmen dock använder sig av slumpgenerering kommer det alltid att finnas en risk, om än en väldigt liten, för att alla genererade vägar blir dåliga oavsett hur höga värdena på parametrarna som tas upp i 2.2.2 är. Detta skulle medföra enorma säkerhetsrisker för passagerarna på fordonet, vilket inte är positivt ur ett samhällsetiskt perspektiv.

7 Referenser

Beni, G., & Wang, J. (1993). Swarm Intelligence in Cellular Robotic Systems. *NATO Advanced Workshop on Robots and Biological Systems*. Tuscany: NATO.

Bhattacharjee, P., Rakshit, P., & Goswami, I. (2011). Multi-robot path-planning using artificial bee colony optimization algorithm. *2011 Third World Congress on Nature and Biologically Inspired Computing (NaBIC)* (ss. 219-224). Salamanca: IEEE.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near Optimal Hierarchical Path-Finding. *Journal of game development* , 7-28.

Brooks, R. A. (1983). Solving the find-path problem by good representation of free space. *IEEE Transactions on Systems, Man, and Cybernetics*, 190-197.

Danaraj, R. (2010). *Solution to Economic Dispatch by Artificial Bee colony Algorithm*. Hämtat från MATLAB Central: http://www.mathworks.com/matlabcentral/fileexchange/27125-solution-to-economic-dispatch-by-artificial-bee-colony-algorithm/content/ABC-eld/runABC.m

Dota 2 . (2012). Valve Corporation.

Eldrandaly, K. A., AbdelAziz, N. M., & Hassan, M. M. (2015). A Modified Artificial Bee Colony Algorithm for Solving. *Applied Mathematics & Information Sciences Natural Sciences*, 147–154.

Graham, R., McCabe, H., & Sheridan, S. (2005). Realistic Agent Movement in Dynamic Game Environments. *Changing Views – Worlds in Play*. Vancouver: Digital Games Research Association.

Hart, P., Nilsson, N., & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. 100-107: IEEE.

Heroes of Might and Magic: A Strategic Quest. (1995). New World Computing.

Hinchey, M., Loyola Coll, M., Sterritt, R., & Rouff, C. (2007). Swarms and Swarm Intelligence. *Computer*, 111 - 113.

Husse, C. (den 18 10 2010). *Fast A-Star (2D) Implementation for C#*. Hämtat från CodeProject: http://www.codeproject.com/Articles/118015/Fast-A-Star-D-Implementation-for-C den 16 05 2015

Karaboga, D. (2005). *An idea based on honey bee swarm for numerical optimization*. Erciyes University, Engineering Faculty, Computer Engineering Department.

Karaboga, D. (2005). *An idea based on honey bee swarm for numerical optimization*. Erciyes University, Engineering Faculty, Computer Engineering Department.

Karaboga, D., & Basturk, B. (2007). Artificial Bee Colony (ABC) Optimization Algorithm for Solving Constrained Optimization Problems. *Foundations of Fuzzy Logic and Soft Computing*, 789-798.

Karaboga, D., & Gorkemli, B. (2011). A combinatorial Artificial Bee Colony algorithm for traveling salesman problem. *2011 International Symposium on Innovations in Intelligent Systems and Applications (INISTA)* (ss. 50 - 53). Istanbul: IEEE.

Lester, P. (2005). *A* Pathfinding for Beginners*. Hämtat från Policy Almanac: http://www.policyalmanac.org/games/aStarTutorial.htm den 08 04 2015

Liang, J.-H., & Lee, C.-H. (2014). Efficient collision-free path-planning of multiple mobile robots system using efficient artificial bee colony algorithm. *Advances in Engineering Software*, 47–56.

Ma, Q., & Lei, X. (2010). Dynamic Path Planning of Mobile Robots Based on ABC Algorithm. *Artificial Intelligence and Computational Intelligence* (ss. 267-274). Sanya: Springer Berlin Heidelberg.

Newton-King, J. (2006). Json.NET.

Singh, A. (2008). An artificial bee colony algorithm for the leaf-constrained minimum spanning tree problem. *Applied Soft Computing*, 625–631.

Sonmez, M. (2010). Artificial Bee Colony algorithm for optimization of truss structures. *Applied Soft Computing*, 2406–2418.

Tereshko, V. (2000). *Reaction-Diffusion Model of a Honeybee Colony's Foraging Behaviour*. Berlin: Springer Berlin Heidelberg.

Unity. (2015). Unity Technologies.

Unity Technologies. (2015). *Understanding Automatic Memory Management*. Hämtat från Unity Manual: http://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html den 17 05 2015

Wales, J., & Sanger, L. (2015). *Random Walk*. Hämtat från Wikipedia: http://www.en.wikipedia.org/wiki/Random_walk den 20 03 2015

Xu, C., Duan, H., & Liu, F. (2010). Chaotic artificial bee colony approach to Uninhabited Combat Air Vehicle (UCAV) path planning. *Aerospace Science and Technology*, 535–541.