Examensarbete

HÖGSKOLAN I SKÖVDE

GENETISKA ALGORITMER KONTRA TILLSTÅNDSMASKINER I INDIVIDCENTRERADE EKOSYSTEM I DATORSPEL

Effektivitet, Flexibilitet och Anpassningsförmåga

GENETIC ALGORITHMS VERSUS STATEMACHINES IN INDIVIDUAL CENTERED ECOSYSTEMS IN GAMES

Efficiency, Flexibility and Adaptability

Examensarbete inom huvudområdet Datalogi
Grundnivå 30 högskolepoäng
Vårtermin 2014

Hampus Gustafsson

Handledare: Erik Sjöstrand
Examinator: Henrik Gustavsson

Sammanfattning

Målet med detta verk var att undersöka vilken AI-teknik som är mest lämpad att styra beteende i ett artificiellt ekosystem. En tillståndsmaskin och en genetisk algoritm har implementerats. Dessa AI-tekniker har jämförts med avseende på tre olika aspekter: effektivitet, anpassningsbarhet och flexibilitet. För att kunna genomföra tester har individer och en experimentmiljö implementerats. Testerna gjordes på fyra olika banor. Resultatet från undersökningen talar för att GA är mer fördelaktig att använda än tillståndsmaskiner. Mätningarna visade på att GA tog längre tid att implementera än tillståndsmaskinen, dock är GA generellt mer robust och flexiblare, vilket visade sig när nya beteenden lades till. GA presterar även bättre på merparten av testerna för anpassningsbarheten. Tillståndsmaskinen var effektivare. När hänsyn togs till antalet anrop respektive algoritm fick, visade det sig att skillnaden var betydligt mindre. Resultatet kan användas som grund till utförliga undersökningar i framtida forskning. Implementationen kan anpassas till andra spel, innehållande ekosystem.

Nyckelord: Artificiella Ekosystem, Genetisk Algoritm, Tillståndsmaskin, Artificiell Intelligens

Innehållsförteckning

1 Introduktion 1

2 Bakgrund 2

2.1 Spelet 2

2.2 Artificiella Ekosystem 3

2.2.1 Maskininlärning 4

2.2.2 Deterministisk AI 4

2.2.3 Indeterministisk AI 5

2.3 Tillståndsmaskiner 5

2.4 Evolutionära Algoritmer 6

2.4.1 Genetiska Algoritmer 7

2.4.2 Genom 8

2.4.3 Fitness 8

2.4.4 Selektion 9

2.4.5 Överkorsning 10

2.4.6 Genetiska Algoritmer och dess potential 12

3 Problemformulering 13

3.1 Metodbeskrivning 14

3.1.1 Experimentmiljö 14

3.1.2 Individer 14

3.1.3 Utvärdering 15

4 Genomförande 16

4.1 Experimentmiljö 16

4.1.1 Aspekter från naturligt ekosystem 17

4.2 Den genetiska algoritmen 18

4.2.1 Fitnessfunktionen 18

4.2.2 Selektion 20

4.2.3 Överkorsning 21

4.2.4 Mutation 21

4.2.5 Representation av Actions 22

4.2.6 Regelsystemet 23

4.3 Tillståndsmaskinen 25

4.4 Individer 28

4.4.1 Ledaren 28

4.4.2 Beteendefunktioner 28

4.4.3 Individernas Egenskaper 29

4.4.4 Varierande Variabler 31

4.5 Systemets Struktur 32

4.6 Framtida utveckling av spelet 34

5 Utvärdering 36

5.1 Presentation av undersökning 36

5.1.1 Anpassningsförmåga 36

5.1.2 Effektivitet 37

5.1.3 Flexibilitet 38

5.2 Analys 38

5.2.1 Anpassningsförmåga 38

5.2.2 Effektivitet 41

5.2.3 Flexibilitet 45
5.3 Slutsatser 46
6 Avslutande diskussion 48
6.1 Sammanfattning 48
6.2 Diskussion 48
6.3 Framtida arbete 50
Referenser 51

1 Introduktion

Artificiell intelligens (AI) har funnit i spel sedan lång tid tillbaks. Detta kan ses i spel som exempelvis Pacman (Namco, 1980). AI är ett element av gameplay. Rollings och Adams (2004) säger att det inte finns en universal definition av gameplay, enligt dem är gameplay ”the result of a large number of contributing elements”. De menar också på att varje individ har sin egen uppfattning av vad gameplay är. Därför definieras i detta arbete att gameplay är allt förutom den grafiska och auditiva presentationen av spelet. I datorspelsindustrin har AI många användningsområden. AI används exempelvis till interaktion, att hitta möjliga vägar i en definierad miljö, lärande, flockbeteende, beslutstagande och anpassa svårighetsgrader.

I Sweetser och Wiles (2002) text *Current AI in Games: A Review* sammanfattar de dagens behov av AI. De menar på att är en trend inom AI som börjar skifta, att datorspelsanvändarna börjar bli trötta på förutsägbar AI. Sweetser och Wiles förespråkar för att spelutvecklare i framtiden bör fokusera på avancerad, realistisk AI, snarare än att förädla grafiken ytterligare, vilket skulle ge datorspelanvändarna den underhållning de kräver.

Indeterministiska AI-tekniker genererar beteenden för individer där beteendet har en viss nivå av obestämdhet och oförutsägbarhet. Detta skapar en större potential för ett AI som är mer interaktivt och levande. Ett exempel på ett sådant AI är en icke spelbar individ som lär sig och anpassar sig efter spelarens fightingtekniker (Bourg & Seemann, 2004).

Oftast används evolutionära algoritmer tillsammans med andra AI-tekniker. Ett exempel på detta är att automatiskt designa Decision Trees med hjälp evolutionära algoritmer (Barros et al., 2013). I detta projekt används en genetiska algoritm, som är en kategori av evolutionära algoritmer, för att styra individer. En individ i detta fall ses som något som kan uppfatta miljön runt omkring med hjälp av någon form av sensorer och agera i denna miljö för att uppfylla sina mål. Russel och Norvig (1995) använder ordet agent för att beskriva något som kan uppfatta den intilliggande miljön med sensorer och agera utifrån detta. I resten av rapporten kommer individ användas istället for ordet agent.

Detta projekt undersöker två olika AI-tekniker, genetisk algoritm och tillståndsmaskin som ska styra beteendet hos individuella livsformer i ett individcentrerat ekosystemsspel. Dessa AI-tekniker kommer vägas mot varandra med avseende på dessa tre kriterier, individers anpassningsbarhet, algoritmernas effektivitet med avseende på processortid, och antalet anrop respektive algoritm får samt hur flexibla algoritmerna är att arbeta med.

I implementationsdelen illustreras ekosystemets grund. Viktiga aspekter tas upp som individernas tillstånd och handlingar samt hur dessa fungerar. Implementationen förklarar även hur de två AI algoritmerna används för att styra beteendet hos individerna i ekosystemet. Detaljer om hur individers handlingar styrs, hur tillstånd uppdateras och hur jämförelsen mellan algoritmerna går till med avseende på kriterierna som nämndes tidigare.

2 Bakgrund

Detta kapitel beskriver bakgrunden till denna rapport. Sektion 2.1 ger en inblick i spelet som kommer producerat parallellt tillsammans med detta projekt samt sammarbetet med företaget Balthazar. Sektion 2.2 kommer ge en beskrivning av artificiella ekosystem och datorspel som har implementerat någon form av artificiellt ekosystem, samt beskriva några av uttrycken som nämndes i introduktionen, nämligen, maskininlärning deterministisk och indeterministisk AI. Sektion 2.3 kommer gå in på djupet av tillståndsmaskiner. Sektion 2.4 beskriver evolutionära algoritmer generellt och går djupare in på genetiska algoritmer med dess termer.

2.1 Spelet

Parallellt med examensarbetet skapas ett spel åt företaget Balthazar. Balthazar är ett ”science-center” i Skövde som är riktat till grundskoleelever. Dock får vem som helst komma förbi och interagera med alla lärorika ting som finns utspridda i deras lokaler. Spelet skapas tillsammans med två grafiker, en musiker, och en till programmerare.

Efter några möten med Balthazar fanns ett gemensamt beslut om vilken typ av spel Balthazar ville ha. Spelet i fråga bygger på ett individcentrerat ekosystems. I ekosystemet finns det individer av olika livsformer, dynamisk växtlighet och dynamiskt väder.

Spelet skall vara en del av en monter som personer har möjlighet att interagera med. De ska ha möjlighet att påverka ekosystemet på tre olika nivåer, lokalt, globalt, och på individ-nivå. Den lokala nivån påverkas genom att plantera ut växtlighet. Den globala nivån påverkas genom att höja eller sänka hela världens temperatur samt den individuella nivån påverkar fortplantningen av individerna genom att spelaren har möjlighet att ändra någon egenskap hos barnet som skapas under fortplantningen. Tanken är detta kombinerat skall lära grundskolebarn hur ekosystem fungerar.

Spelet som implementerades är ett tvådimensionellt interaktivt individcentrerat ekosystem. Spelet använder spelmotorn Unit3D (Unity Technologies, 2014). Unity3D valdes därför att den har stöd för många plattformar och är enkel att använda vilket underlättade mycket under implementationsfasen. Spelet utvecklades till det mobila operativsystemet Android.

Individerna i spelet kommer vara dynamiska. I en editor innan spelets start skall det vara möjligt att skapa en egen individ till ekosystemet. När spelet startar kommer det finnas en flock av den skapade individen, samt ett antal slumpmässigt genererade individer vilket kommer skapa grunden för ekosystemet. Alla individer har ett antal fördefinierade attribut som påverkar utseendet, överlevnad, och vilken del av den dynamiska miljön individen kommer trivas i. Dessa attribut kommer användas för att hantera individens beteende. Exempelvis om attributet för hunger är under en viss nivå skall individen leta efter mat samt äta när den hittar maten. Dessa attribut är:

- Attributet storlek kommer vara direkt kopplat till hur mycket hälsa individen har.
- Metabolism definierar hur snabbt en individ blir hungrig och hur lång tid individen måste äta för att bli ”mätt”.
- Kamelegenskaper fungerar på samma vis som metabolism, vilket styr hur snabbt individen blir törstig och hur mycket vatten en individ kan lagra.

- Törst anger nivån av törst, när denna nivå är under en viss nivå anses individen som törstig.
- Hunger fungerar på samma vis som törst.
- Påfågelsfaktor påverkar hur populär individen är hos andra individer vilket är avgörande när förökning av individer sker.
- Päls är ett attribut som bestämmer vart i värden en individ trivs. Om individen har päls kommer individen föredra kalla klimat, om individen är utan päls alls kommer individen föredra varmt klimat.
- Snabbhet bestämmer hur snabbt en individ kan röra sig i miljön.

2.2 Artificiella Ekosystem

I detta delkapitel kommer artificiella ekosystem beskrivas generellt. Först kommer en enkel introduktion till historien bakom artificiella ekosystem där en definition av naturliga ekosystem fastställs samt en beskrivning av hur ett sådant fungerar. Därefter ges en beskrivning av artificiella ekosystem och hur de är länkade till naturliga ekosystem. Till sist kommer en beskrivning av spel som bygger på artificiella ekosystem.

När forskare insåg att datorer var nog kraftiga för att använda dem till datorsimulation av biologiska fenomen, behövde de inte längre förlita sig på dyra experiment i laboratorier eller utsätta djur i naturen för studier. Datorsimulation används till att studera många kända biologiska fenomen. Som Andrews, Polack, Sampson, Timmis och Coles (2008) skriver, har simulation generellt två mål: vissa är skapade tillsammans med forskare för att förbättra forskares förståelse för naturliga system, samt andra system är byggda för att skapa eller undersöka alternativa verkligheter. Deras studie går ut på att bidra till den vetenskapliga förståelsen för immunologi med hjälp av simulation. Detta är ett av många användningsområden för datorsimulation om biologi.

Sjörs, Söderström och Ulfstrand (2014) definierar ekosystem som ett ekologiskt system innefattande allt levande och dess livsmiljö inom ett område. I sin bok om ekologi och dess element beskriver Clarke (1954) läran om samfundet mellan djur, växter och deras miljö, som vetenskapen av ekologi. Miljön i detta fall inkluderar djur och växters inverkan på varandra samt miljöns fysiska aspekter som landskapet, vatten, jord, öken eller berg. Clarke beskriver också att inget djur eller växt lever som en helt isolerad individ, utan de lever tillsammans och påverkar varandra direkt eller indirekt i ett komplext system. I grund och botten beskriver Clarke ett ekosystem som ett komplext samspel mellan energi, producenter, konsumenter, nedbrytare och transformerare. Producenter kan vara växlighet som producerar frukt, bär eller liknande och tar energi från någon ljuskälla som solen. Konsumenter kan ses som djur som äter producenternas frukt. När producenter och konsumenter dör bryts de ner av bakterier, nedbrytarna. När nedbrytarna har omvandlat producenten eller konsumentet till ett visst material kommer transformerarna in, andra typer av bakteria. Transformerarna omvandlar materialet från nedbrytarna till näringsämnen för växter. Detta skapar en cirkel i ekosystemet.

För att ett ekosystem skall fungera måste det finnas en balans. Exempelvis när antalet djur överskrider växtlighetens produktion kommer fler djur dö på grund av att de inte kan konsumera tillräckligt för att överleva, det sker en skiftning i systemet. Först minskar antalet växter till en kritisk nivå. När denna nivå är nådd kommer antalet döda djur öka på grund av underkonsumtion. När antalet döda djur ökar, får växtligheten en chans att föröka sig. Detta

i sin tur gör att levande djur åter igen ökar i antal. Dessa skiftningar kommer tillslut minska och ekosystemet kommer nå ett tillstånd av balans. Holling (1987) definierar detta som ”equilibrium (balance of nature)”.

Artificiella ekosystem tar konceptet av naturliga ekosystem och skapar olika datormodeller av dessa. Datormodeller i sin tur används för att simulera olika scenarion som kan utspela sig i ett visst ekosystem. I sitt arbete om artificiella ekosystem modellerar Rönkkö (2007) sitt ekosystem med hjälp av ett partikelsystem i en tredimensionell värld. Han definierar allt i sitt ekosystem som komponenter bestående av en eller flera partiklar. Alla partiklar är fysiskt baserade, de har också minne och någon form av reaktivt beteende. Med denna modell lyckas Rönkkö få med många aspekter av vad Clarke (1954) definierar som ett ekosystem. Det finns även ett rovdjur och bytesdjurbeteende i Rönkkös artificiella ekosystem. Detta system kan ses som ett individcentrerat ekosystem där varje individ består av flera partiklar. Varje individ har fysikaliska aspekter som position, hastighet och så vidare, samt att de har något beteende och minne.

Creatures (Millennium Interactive, 1996) är ett spel som tar aspekter från ett individcentrerat ekosystem. Varje individ har en position i världen, de har ett beteende och någon form av minne. Spelet går ut på att uppfostra en varelse som är identifierat som Norns. Varje individuell varelse under spelets gång lär sig att överleva, interagera med miljön och prata. Användaren har ansvaret att underhålla, lära och föröka Norns. Norns intelligens bygger på artificiella neurala nätverk (ANN). Buckland (2002) beskriver grundligt i sin bok *AI Techniques For Game Programming* hur ANN fungerar. Kombinationen av Norns med dess omgivning skapar ett individcentrerat ekosystem där Norns är själva individen i ekosystemet.

Ett annat spel som tar sin inspiration från naturens ekosystem är SimLife av Maxis Software Inc. (1992). Detta spel innefattar många områden inom ekosystem. Spelet har varierande terräng och klimat med ett antal arter av växtlighet, växtätare och köttätare. I detalj kan djurs DNA ändras.

2.2.1 Maskininlärning

Inlärning är ett mångfacetterat fenomen. Processen inlärning inkluderar anskaffandet av deklarativ kunskap, utveckling av motorik och kognitiv kunskap genom instruktioner eller övning, organiseringen av ny kunskap till en generell, effektiv representation samt förnimmandet av ny fakta och teorier genom observation och experimenterande (Carbonell, Michalski & Mitchell, 1983). Sedan datorerans start har forskare strävat efter att implementera och simulera dessa fenomen i datorer. Att implementera och simulera dessa fenomen är ett av de mest komplexta och invecklade målen inom AI. Läran och datormodelleringen av inlärningsprocesser skapar ämnet maskininlärning. Witten och Frank (2005) tar upp andra synsätt på maskininlärning. Maskininlärning är alltså grenen av AI som behandlar området inlärning och kunskap inom datoriserade miljöer. Maskininlärning är en del av evolutionära algoritmer (Whitey, 2001).

2.2.2 Deterministisk AI

Enligt Bourg och Seemann (2004) kan AI delas in i två kategorier, deterministisk och indeterministisk. De beskriver en deterministisk AI som att ha ett deterministiskt beteende eller handlingssätt. De nämner att beteendet eller handlingssättet är fördefinierat och förutsägbart. Cohen (1979) beskriver också deterministiska algoritmer på liknande sätt.

Beteendet byggs upp av fördefinierade rutiner och regler som individen följer. Dessa rutiner och regler är anpassade för att den icke spelbara individen skall hantera alla möjliga scenarion som kan ske under spelets gång. Ett exempel på en deterministisk AI är en icke spelbar individ som är programmerad att röra sig mot ett mål tills den når det. Bourg och Seemann (2004) påpekar också att eftersom individer inte lär sig eller utvecklas under spelets gång upplevs deras beteende som förutsägbart samt att detta minskar spelets underhållningsvärde.

Att implementera denna typ av AI i ett realtids-strategispel skulle kräva enorma mängder arbete för att komma på alla tänkbara situationer ett sådant AI kan utsättas för (Johnson & Wiles, 2001). En indeterministisk AI skulle lämpa sig bättre för en sådan typ av spel.

2.2.3 Indeterministisk AI

Den andra kategorin av AI är indeterministisk AI. Bourg och Seemann (2004) och Cohen (1979), beskriver den kategorin som motsatsen till deterministisk. Enligt dem har en indeterministisk AI en viss nivå av ovisshet och oförutsägbarhet. Nivån av ovisshet grundar sig i hur vilken typ av AI-teknik som används samt hur väl tekniken är förstådd. Cohen diskuterar dock kring generella algoritmers indeterminism snarare än direkt referera till AI-tekniker. Dock beskriver Bourg, Seemann och Cohen indeterminism på samma sätt i sina arbeten.

Ett exempel på AI som är indeterministisk är en icke spelbar individ som lär sig under tidens gång och anpassar sig till hur en spelare spelar spelet. I sin artikel om hur adaptivt AI förbättras med evolutionär inlärning ger Ponsen (2004) ett bra exempel på en implementation av ett indeterministiskt AI. Han grundar sin undersökning i att skapa ett bättre och anpassningsbart AI i ett realtids-strategispel. I realtids-strategispelet har han skapat ett AI som använder sig av informationen från exempelvis föregående sammandrabbning med en motspelare för att evolvera fram en ny strategi och öka chanserna i nästkommande sammandrabbning.

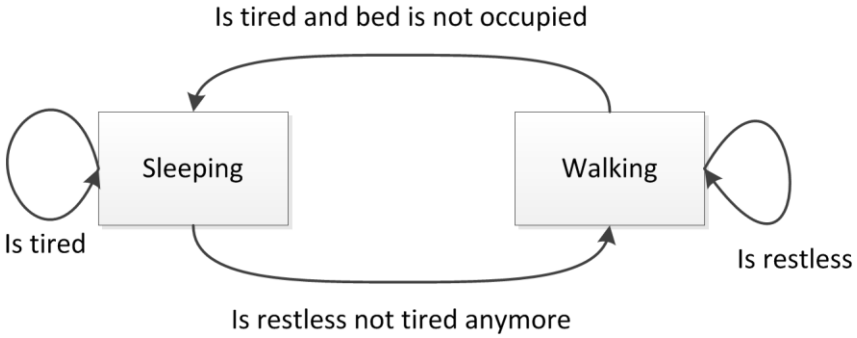
Som nämndes i introduktionen förekommer indeterministiska tekniker endast i enstaka fall. Detta på grund av att det finns en viss rädsla hos dataspelsutvecklarna över oförutsägbarheten hos indeterministiska AI-tekniker. Sweetser och Wiles (2002) nämner i sin artikel att den mest frekventa tekniken som spelutvecklare använder är tillståndsmaskiner, därför att de går snabbt att implementera, lätta att förstå och tillräckligt generell för att använda i många scenarion.

2.3 Tillståndsmaskiner

Mealy (1955) har en teknisk och formell definition av vad en tillståndsmaskin är. Enligt honom är en tillståndsmaskin en omvandlare som opererar på en ändlig samling av input-symboler, som besvarar input-symbolerna med en samling output-symboler samt har ett ändligt antal tillstånd. Alla möjliga input-output par tillsammans med alla möjliga tillstånds-övergångar för varje input-symbol skapar beteendet hos tillståndsmaskinen. Enligt Mealy kan en tillståndsmaskin definieras av en fem-tupel, (Q, I, Z, δ, ω), där Q är alla möjliga tillstånd, I är alla möjliga input-symboler, Z är alla output-symboler, δ är alla tillstånds-övergångar kopplade till någon input-symbol, det vill säga kombinationen av någon input I och något tillstånd Q ger det nya tillståndet Q, δ är alltså en mappning till detta, ω liknar δ dock kopplar ω ett input I och ett tillstånd Q som ger ett output Z.

Det finns även en annan modell som ej beskrevs ovan, Moores tillståndsmaskin (Moore, 1956). Gill (1960) har gjort en jämförelse av Mealys och Moores modeller. Där resultatet av undersökningen visade på att modellerna är ekvivalenta. Gill menar på att något system som kan representeras av den ena modellen kan också representeras av den andra. Gill använder ett ekvivalensteorem för att bevisa modellernas ekvivalens. En skillnad som Gill påträffade var att respektive modell hade olika definitioner. Tillståndsmaskinen i detta arbete kommer bygga på Mealys definition av en tillståndsmaskin, den som beskrevs ovan.

Som Lee och Yannakakis (1996) nämner, kan en tillståndmaskin modelleras som ett diagram. Figur 1 illustrerar ett sådant diagram, vilket motsvarar ett gå-och-sova-beteende för en individ. I figuren finns det två tillstånd som individen kan befinna sig i, Sleeping och Walking. Individen har också input från sig själv i form av känslor som trötthet och rastlöshet. Miljön ger också input till individen genom att berätta för individen att det finns plats i sängen om individen är trött. Ett tillståndsbyte sker om exempelvis individen känner sig sömnig och vill sova när den befinner sig i tillståndet Walking. Om individen fortfarande känner sig rastlös och befinner sig i tillståndet Walking kommer en loop ske och individen stannar i samma tillstånd.



Figur 1 Enkel tillståndsmaskin för ett gång-och-sova-beteende

2.4 Evolutionära Algoritmer

Glymour (2001) argumenterar för i sin artikel om selektion, indeterminism och evolutionära teorier att evolution är indeterministiskt och eftersom evolutionära algoritmer tar sin inspiration från naturlig evolution är det också naturligt att se dessa algoritmer som indeterministiska. Johnson och Wiles (2001) beskriver i sin artikel om datorspel med intelligens att evolutionära algoritmer (EA) består av en familj av algoritmer som tar inspiration från den biologiska evolutionen. De nämner också att EA använder den adaptiva förändringen hos en population av individer för att lösa problem eller modellera komplexa system. Detta kan vara lösningar som att skapa funktioner för att förespå den finansiella marknaden, eller styra beteendet hos en icke spelbar individ i ett spel.

Mitchell och Taylor (1999) skriver i sin artikel *Evolutionary Computation: An Overview* att problem som att förespå den finansiella marknaden kräver komplexa lösningar och att lösningarna till dessa problem oftast för invecklade för att en mänsklig programmerare skall uppfinna en direkt lösning. De menar på att det bästa sättet för att lösa ett sådant problem är att skapa ett antal enkla regler, och tilldela medel till ett system som sedan skall kunna evolvera fram en lösning. Med hjälp av dessa regler är det möjligt att skapa komplexa beteenden, en form av artificiell intelligens, vilket inte skulle funka i samma utsträckning om tillståndsmaskiner användes för samma ändamål. Jones (2002) talar för att EAs har ett brett

användningsområde samt kan enkelt tillämpas på specifika problem. Några av problemexemplen han tar upp är förusägelse av makromolekylär struktur, molekylär dockning, kvantmekanik och många fler.

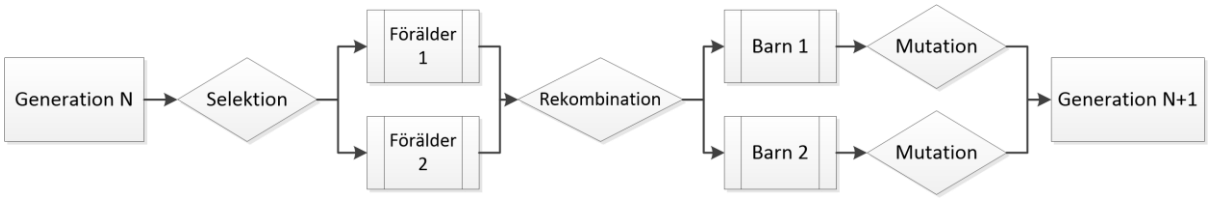
2.4.1 Genetiska Algoritmer

En typ av EA som har används i spel är genetiska algoritmer (GA) (Johnson och Wiles, 2001). GA är en teknik som tar sin inspiration från naturen. Tekniken baserar sina mekanismer på fortplantning och naturlig evolution. På samma sätt som djur utvecklas över många generationer för att bli mer framgångsrika i att överleva och fortplanta sig, utvecklas GA över tid för att komma fram till en lösning, eller lösningar beroende på kontext, för ett specifikt problem (Buckland, 2002). Det finns fyra centrala begrepp inom GA:

- Genom representerar data som skall evolveras, en kombination av genom och fitnessvärde beskrivs i detta projekt som en organism samt en samling organismer kalls population.
- Fitnessvärde bestämmer hur framgångsrik ett genom är med avseende på problemet som skall lösas.
- Selektion använder sig av genomens fitnessvärde för att utse vilka genom som skall fortplanta sig och föra vidare genomens gener till nästkommande generation.
- Överkorsning är steget efter selektion. Överkorsning kombinerar två genom att plockar ut ett antal gener ur båda genomen för att skapa nya barn vilket skapar den nya generationen. Under överkorsningen kan det ske en mutation på något av barnen. Detta gör att några av generna i barn-genomen förändras slumpartat. Figur 2 illustrerar den generella strukturen som beskrevs ovan.

Figur 2 bygger på stegen Jones (2002) nämner sin artikel när han beskriver hur en GA fungerar. Här nedan är listan som. Jones nämner även att evolutions-aspekten i GA börjar vid steg 3. Där steg 3 syftar på att välja två föräldrar som kommer reproducera nya individer, Detta är den erkända GA, skapad av Holland (1992) .

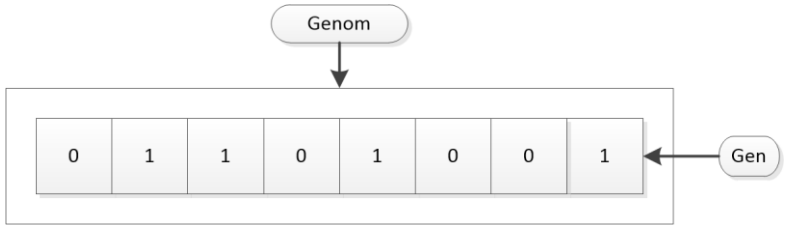
1. En population av individer initieras med slumpade värden.
2. Fitness-värden beräknas per individ.
3. En selektionsalgoritm väljer två föräldrar av nuvarande populationen.
4. Med en vald sannolikhet kan en överkorsning ske, två nya barn skapas som kommer ingå i den nya generationen.
5. Med en vald sannolikhet kan respektive barn muteras.
6. Nya populationen kommer nu ersätta den nuvarande.
7. Om något avslutande tillstånd har nåtts, avslutas algoritmen, annars gå tillbaka till steg 3.



Figur 2 Generell struktur för GA

2.4.2 Genom

För att lösa ett givet problem måste det finnas en representation av strukturen för lösningen Jones (2002). Det är detta genom används till. Genom består av en serie gener. Dessa gener representeras oftast av binära bitar, dock kan andra representationer som heltal förekomma. Figur 3 illustrerar en binärrepresentation av ett genom med åtta gener.



Figur 3 En binärrepresentation av ett genom

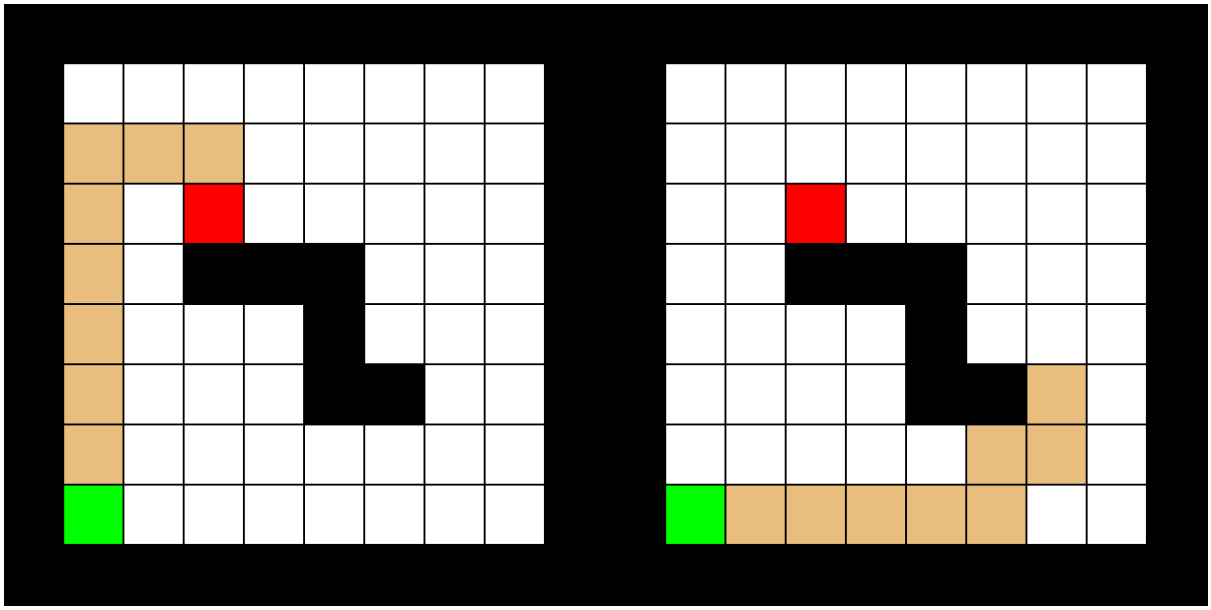
För att ett genom skall fungera i ett spel, måste det finnas ett sätt att översätta bitarna från binärrepresentationen till en lösning. Buckland (2002) ger ett bra exempel på detta. I hans exempel delar han upp bitarna i par. Där vare genpar representerar ett håll en karaktär skall gå. Han har låtit ”00” vara ”North”, ”01” vara ”South”, ”10” vara ”East” samt ”11” vara ”West”. Ett genom skapar då ett gångbeteende som består av en serie riktningar. Denna serie kan sedan användas för exempelvis leda en spelarkaraktär igenom en labyrint eller dylikt. Detta är dock ett av många beteenden genom kan representera.

2.4.3 Fitness

Som tidigare nämndes fungerar GA på liknande sätt som vanlig evolution. Det finns en population av organismer, där varje organism består av ett genom och ett fitnessvärde. Denna population evolveras sedan över tid och skapar nya generationer. På så sätt genereras en lösning till ett problem.

Fitnessvärde är ett mått på hur framgångsrik en organism är. Högre fitnessvärde motsvarar en mer framgångsrik organism. Organismer med högre fitnessvärde har större chans att sprida sina gener till nästkommande generation.

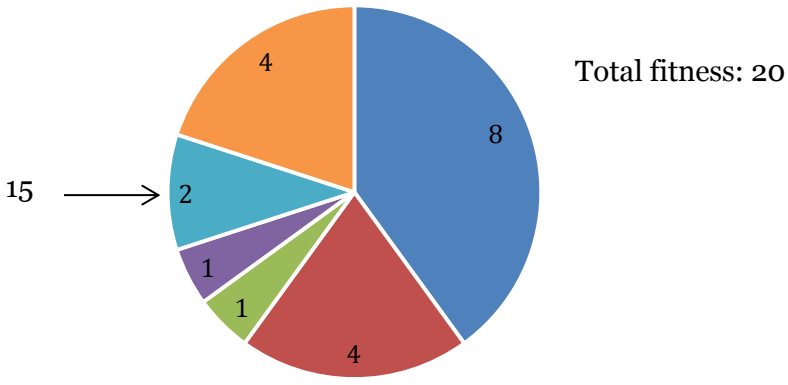
Beroende på problem som skall lösas finns det olika sätt att representera fitnessvärde. När GA används för att hitta kortaste vägen genom en labyrint, kan fitnessvärdet vara direkt relaterat till hur nära lösningen har kommit ett givet mål. När avståndet blir mindre för en specifik lösning ökar dess fitnessvärde. Figur 4 Illustrerar en förenklad version av en labyrint, med två genererade lösningar från en GA. Den vänstra lösningen har högre fitness därför att den hamnade närmare målet och anses vara mer framgångsrik än den högra lösningen som hamnade mycket längre ifrån målet.



Figur 4 GA för att hitta närmaste vägen till mål.

2.4.4 Selektion

Grundidén med selektion är att selektion skall på något vis relatera till genomens fitness (Jones, 2002). Den mest grundläggande implementationen av selektion är en så kallad rouletthjuls-selektion. Rouletthjulsselektion (RS) använder en sannolikhetsdistribution vilket selektionssannolikheten är direkt proportionell till en given organisms fitnessvärde. Figur 5 ger ett enkelt exempel på RS, där varje färg motsvarar en organism, storleken på cirkelsektorn symboliserar en organism fitnessvärde relativt till hela populationens gemensamma fitnessvärde som i detta fall är 20.



Figur 5 Rouletteselektion, organismer med högre fitness får större del av cirkeln

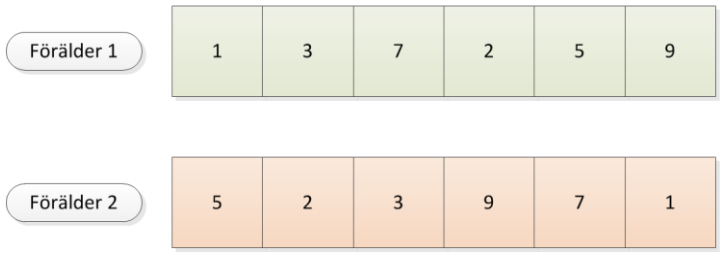
RS fungerar som ett rouletthjul, ett tal kommer slumpvis väljas. Alla heltal som cirkeln innefattar, i detta fall 1 – 20, kommer ha lika stor chans att bli vald. Notera att organismer med högre fitness har större chans att bli vald för selektion, därför att de innefattar större antal nummer. Dock är det inte säkert att de blir valda. Exemplet i Figur 5 visar på denna observation, där det slumpade talet blev 15, vilket hamnar på en organism med fitnessvärdet 2 som är betydligt lägre än det högsta fitnessvärdet 8. Detta skapar en sorts evolution där

även de svagare har en chans att överleva samt att de organismer med högre fitnessvärde får en bättre överlevnadschans.

Jones (2002) påpekar dock att GA kombinerat med RS ger ingen garanti till att organismer med högt fitnässvärde finns kvar i senare generationer. Dock nämner Jones att problemet går att lösa med hjälp av elitism. Som Jones beskriver bygger elitism på att kopiera den bästa organismen, oförändrat, från den nuvarande populationen till den nya populationen. Detta gör att det alltid finns en bra organism i den nya populationen, vilket underlättar GAs sökning efter en optimal lösning.

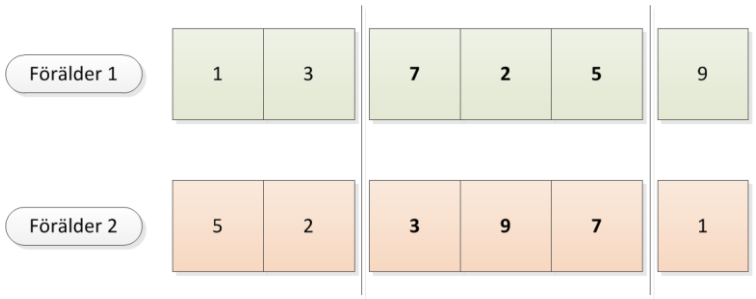
2.4.5 Överkorsning

Överkorsning kombinerar två föräldrar för att skapa två nya barn. Under överkorsningsfasen kan även barnen muteras. Detta skapar en del av nya generationen. När två föräldrar har valts genom selektion skall de rekombineras. Denna process kallas för överkorsning. Överkorsning fungerar på så vis att vissa av generna i första föräldern byter plats med de hos den andra föräldern, vilket resulterar att två nya barn skapas. Där barnens gener kommer från båda föräldrarna. En populär överkorsnings-teknik är Partially-Mapped Crossover (PMX) (Buckland, 2002). I exemplet nedan är generna kodade i vanliga heltal. Figur 6 visar två möjliga föräldrar. Jones (2002) nämner att denna överkorsningsmetod är framtagen för att bara passa sekvenser av gener där dubbletter ej existerar, vilket kan noteras när PMX illustreras här nedan.



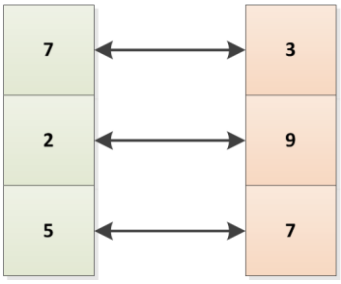
Figur 6 Två möjliga föräldrar som representerar genom av heltal.

För att implementera PMX måste två överkorsningspunkter väljas slumpmässigt. I exemplet valdes en överkorsningspunkt efter andra och femte genen. Separationerna illustreras med streck i Figur 7.



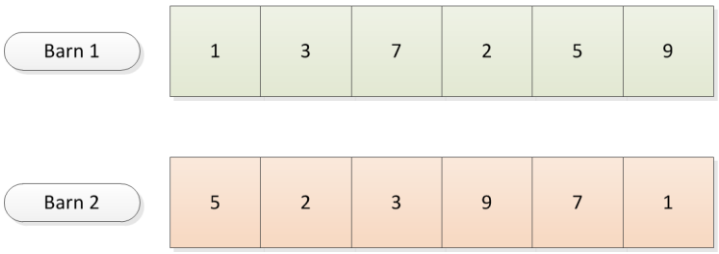
Figur 7 Överkorsningspunkt efter gen nr 2 och efter gen 5 är vald

Notera sedan siffrorna mellan strecken. Dessa siffror kommer nu ”mappas” till varandra, vilket illustreras i Figur 8. Här är det viktigt att det inte finns dubbletter.



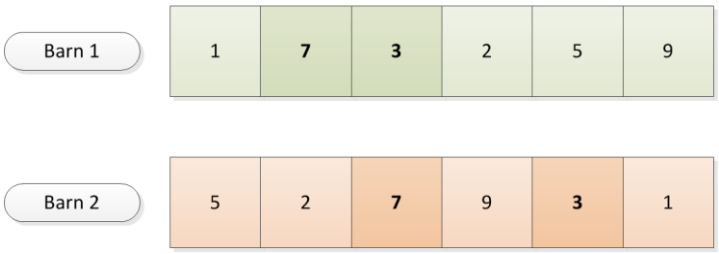
Figur 8 Mappning mellan nummer från förälder 1 och förälder 2

Det sista steget är att iterera genom barnens siffror och byta ut de nummer till de som motsvarar något nummer i mappningen ovan. Först skapas två nya barn som är direkta kopior av sina föräldrar, detta visas i Figur 9. Därefter stegas varje nummer igenom.



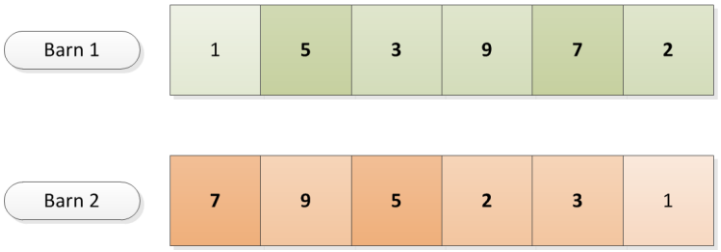
Figur 9 Barn som är direkt kopior av sina föräldrar

Första steget är att byta ut alla förekommande treor med sjuor och sjuor med treor. Detta illustreras i Figur 10, där de utbytta numren markeras med fetstil samt bakgrunden är mer saturerad.



Figur 10 Alla treor byts till sjuor samt alla sjuor byts till treor.

När listan med nummer i Figur 8 har itererats igenom är de två nya barnen klara. Figur 11 de nya barnen. Notera i att talet sju har bytts ut två gånger, vilket kan hända PMX. Detta markeras i Figur 11 genom att ange en ännu mer saturerad bakgrund.



Figur 11 Barn efter färdigt byte av alla siffror.

Efter en överkorsning finns det chans att vissa av barnens gener muteras. Detta händer innan de nya barnen placeras i en ny generation. Liknande överkorsning finns det olika tekniker för att mutera barn. En intuitiv teknik är The Exchange Mutation Operator (EM) (Buckland, 2002). Vilket liknar PMX genom att två punkter i ett genom väljs. De gener som finns på punkternas platser byts ut mot varandra. Detta illustreras i Figur 12. Mutationen sker på barn 1 från Figur 11, där generna med siffrorna ett och två har valts att byta plats.

Figur 12 Barn 1 muteras med EM tekniken där 2 och 1 byter plats.

Jones (2002) diskuterar kring sannolikheten med överkorsning i sitt arbete om GA. Han nämner exempelvis att en för hög överkorsningssannolikhet gör att algoritmen tenderar att generera en suboptimal lösning. Jones påpekar även att en för liten sannolikhet kommer göra att algoritmen tar för lång tid att närma sig en lösning.

2.4.6 Genetiska Algoritmer och dess potential

Johnson och Wiles (2001) nämner i sin artikel om datorspel med intelligens att dessa tekniker oftast tar mycket processorkraft och är för långsamma för att producera ett användbart resultat. Myers (2007) nämner dock i sin artikel om hur en GA kan användas för att skapa olika strategier i ett kortspel, att det finns potential för GA. Han skriver att tidskomplexiteten, den tid det tar för att beräkna en lösning med algoritmen, är för stor i stora mer komplexa spel. Men han menar på att när datorerna blir mer och mer parallella, dvs. att datorerna kan dela upp beräkningar till flera beräkningselement, och om en lösning hittas för att använda parallellismen på ett bra sätt finns det en chans att GA kan användas i komplexa spel. Goldberg och Holland (1988) talar också för att det finns potential i GA. De skriver att GA och evolutionärt baserad maskininlärning oftast kritiseras från fel perspektiv. Kritiken tar sin grund från naturlig evolution och ignorerar skillnaden mellan tidsskalorna av naturliga och artificiella system. De går även in på att GA är av parallell natur med avseende på att de använder ett distribuerat set av värden som används för att generera nya set av värden. Detta betyder att som Myers (2007) nämner att dessa algoritmer blir mer användbara när datorer får en ökad parallellism.

3 Problemformulering

Detta projekt avser att undersöka två olika typer av AI-tekniker för att styra beteendet hos individer i ett individcentrerat ekosystem i spel. Dessa två tekniker är GA och tillståndsmaskiner. Fokus kommer ligga på vilken algoritm som ger bäst förutsättning för individers överlevnad i en fördefinierad spelarmiljö och hur naturligt individerna beter sig i miljön. Algoritmernas effektivitet kommer jämföras med avseende på hur mycket prestanda de tar av datorresurser och vilken algoritm som ger bäst flexibilitet. Dessa begrepp kommer att förklaras mer ingående i den kommande metodbeskrivningen.

Sweetser och Wiles (2002) nämner några problem med tillståndsmaskiner. De säger att individer som har ett beteende baserat på tillståndsmakiner oftast blir förutsägbara. Detta kan liknas i att det är lätt att förutse ett mönster när en individ alltid sover om den är trött istället för att börja springa för att höja blodcirkulationen, eller ta en kaffe. Ett annat problem de nämner med tillståndsmaskiner är att de ofta tenderar att vara dåligt strukturerade, vilket minskar flexibiliteten och svåra att underhålla i framtiden.

Bakgrundskapitlet nämnde att det är många som kritiserar GA för att vara långsamma. Dock finns det personer som hävdar att det finns potential för GA (Myers, 2007) samt Sweetser och Wiles (2002) nämner några spel som har implementerat GA. Tanken är att anpassa en GA för att styra beteendet hos individer i spelet som beskrevs i bakgrundskapitlet och testa dess potential. Om detta ger ett givande resultat kan framtida forskning använda resultatet till att generallisera en lösning som går att applicera på fler spel än spelets som beskrevs i bakgrundskapitlet.

Chaudhry, Khan och Khan (1999) skriver i sitt arbete *A Genetic Algorithm for Flexible Job Shop Scheduling* att det går att göra en universal GA, där grund-GA-rutinen är tillräckligt robust att den kan tillämpas på flera problem. Detta skapar en viss flexibilitet hos GA, i den mån att det är möjligt att använda samma grundrutin på fler än ett problem. Detta till skillnad från tillståndsmaskiner som oftast är skapade för att hantera ett problem (Lee & Yannakakis, 1996; Mealy, 1955). En hypotes är att med samma grundbeteende kommer en GA vara mer flexibel än en tillståndsmaskin i den mån att när nya beteenden läggs till är det lättare att anpassa GA till förändringen än tillståndsmaskinen.

Något som också noterades tidigare är att GA är mindre effektiva än tillståndsmaskiner på grund av hur respektive teknik är designad. Istället för att direkt föra vidare information, som tillståndsmaskiner är designade, måste GA först evolvera fram en lösning till ett problem och därefter tolka resultatet med ett antal definierade regler, (Beyer, Schwefel & Wegener, 2002; Rudolph, 1994). Evolveringsteget i GA kräver i detta fall mer processorkraft än tillståndsmaskiner. Detta skapar hypotesen att med samma grundbeteende är tillståndsmaskiner mer effektiva än GA.

Frågeställningen som härleddes av denna problemställning är följande: Finns det en fördel att använda genetiska algoritmer kontra tillståndsmaskiner för att styra beteende i individcentrerade ekosystem i spel med avseende på vilken algoritm som skapar bäst förutsättningar för individers överlevnad, beräkningsmässig effektivitet och flexibilitet?

3.1 Metodbeskrivning

Experiment är den vetenskapliga metod som valts för att utföra arbetet. Experimentet kommer genomföras genom att jämföra en deterministisk och en indeterministisk teknik med avseende på hur väl de fungerar för att styra beteendet hos individer i ett individcentrerat ekosystem och vilket teknik som kommer göra att individer överlever längst. Jämförelsen kommer även ta hänsyn till algoritmernas effektivitet, med avseende på hur mycket processorkraft de tar att hantera en individs beteende och hur många individer systemet kan innefatta innan hanteringen av beteendet tar för mycket kraft från datorns resurser. Förutom detta kommer algoritmernas flexibilitet och utbyggbarhet jämföras, det som jämförs då kommer vara hur väl algoritmerna fungerar när mer beteenden läggs till i systemet och hur komplext det är att hantera många olika beteenden.

För att möjliggöra experimentet har en fördefinierad miljö skapats. Ett antal individer har definierats. Algoritmerna som skall styra individernas beteende skall vara tillräckligt modulära för att enkelt kunna byta och testa båda algoritmerna i samma scenario.

Förmågan hos den som programmerat implementationen och verktyg som har används till implementationen måste finnas i åtanke då utvärderingen läses.

3.1.1 Experimentmiljö

För att experimentet skall vara möjligt måste det finnas en fungerande experimentmiljö. Miljön skall innehålla tillräckligt många egenskaper från ett naturligt ekosystem för att skapa någon form av automatisk drift i systemet. Miljön skall vara enkelt utformad för att det skall vara lätt att experimentera och skapa olika scenarion i ett ekosystem. Miljön kommer med hjälp av spelmotorn Unity3D (Unity Technologies, 2014) implementeras. Miljön modelleras helt oberoende av beteendealgoritmerna för att inte påverka jämförelsen mellan algoritmerna. För hantering av beteende och generell implementation av miljön används programmeringsspråket C#.

Den testmiljön som har valts innefattar dock inte hela spelet utan kommer vara en kontrollerad miljö. Miljön kommer innefatta ett antal fördefinierade individer med en fördefinierad öppen tvådimensionell värld. I världen kommer det finnas vatten, föda och växtlighet samt kalla och varma områden.

3.1.2 Individer

De individer som kommer existera i experimentmiljön kommer ha ett antal grundläggande attribut som anger om en individ är bytesdjur eller rovdjur, hur väl individen tål kyla, hur lång tid det tar innan individen blir hungrig och så vidare. Egenskaperna måste vara oberoende av beteendealgoritmerna.

Beteendealgoritmerna kommer vara så modulära som möjligt. Detta betyder att det skall vara lätt att ändra vilken AI-teknik, (GA eller tillståndsmaskin), som kommer hantera beteendet för en individ i ett visst scenario.

Att bestämma när och hur individen skall agera på ett visst vis kommer vara fokus i implementationen. Vilket kommer ligga till grund för jämförelsen mellan AI-teknikerna som skall styra beteendet hos individerna.

3.1.3 Utvärdering

Utvärderingen kommer jämföra de två valda AI-teknikerna utifrån de valda kriterierna anpassningsförmåga, effektivitet och flexibilitet. Första kriteriet, anpassningsförmågan, har jämförts för att adressera hur väl algoritmerna utför sin uppgift. Det finns ett antal sätt att testa anpassningsförmågan. Ett sätt är att ta in data från simulationen i form av storleken på populationen och hur storleken har ändrats med tiden samt hur höga populationens fitnessvärde är (Buckland, 2002; Eiben, Marchiori & Valkó, 2004). Nackdelen med att observera data på detta vis är att det inte går att avgöra hur naturligt individerna beter sig. Bara deras överlevnadsförmåga och framgång går att observera. Ett annat sätt att utföra utvärderingen är att ta in människor som kan analysera beteendet hos individerna och om de uppfattas naturligt (Johnson, Wiens, Milne & Crist, 1992). Detta skulle dock kräva ett stort antal människor för att få ett ordentligt pålitligt resultat. Därför kommer utvärderingen av individernas anpassningsförmåga grunda sig i populationsstorlek, individers livslängd, samt hur högt individens fitnessvärde är.

Det andra kriteriet effektivitet kan jämföras på två vis, antigen genom algoritmanalys (Beyer, Schwefel & Wegener, 2002), vilket kan göras utan att implementera algoritmen i ett program. Nackdelen med detta är att det är lätt att missa avgörande faktorer, speciellt i algoritmer för beteendehantering där algoritmens komplexitet oftast gör att faktorer missas. Det är svårt att hitta vilka faktorer som påverkar algoritmens tidskomplexitet. Algoritmanalys ger också bara ett teoretiskt värde på hur effektiv den är, det anger inte hur väl algoritmen presterar under körning. Det andra sättet är att praktisk mäta algoritmernas effektivitet (Eiben, Marchiori & Valkó, 2004). Praktisk mätning har dock nackdelen att det är svåra att upprepa, vilket gör det vetenskapligt svårt att fastställa en mått för effektivitet. Främst beror detta på vilken typ av maskin testet appliceras på, samt de parametrar som används för att bygga systemet under implementationen. För att få ett pålitligt resultat måsta många test göras. I detta projekt används dock praktisk mätning för att fastställa en algoritms effektivitet.

Det sista kriteriet jämför AI-teknikernas flexibilitet. För att bestämma teknikernas flexibilitet har den praktiska implementationen undersökts. Där fokus låg på att studera vad som behövdes ändras och läggas till när respektive algoritm anpassades till en förändring. Att jämföra implementationstiden för respektive algoritm har övervägds. Dock på grund av att detta mått är mycket beroende av erfarenheten hos den som implementerar blir resultatet allt för subjektivt och därmed oanvändbart.

4 Genomförande

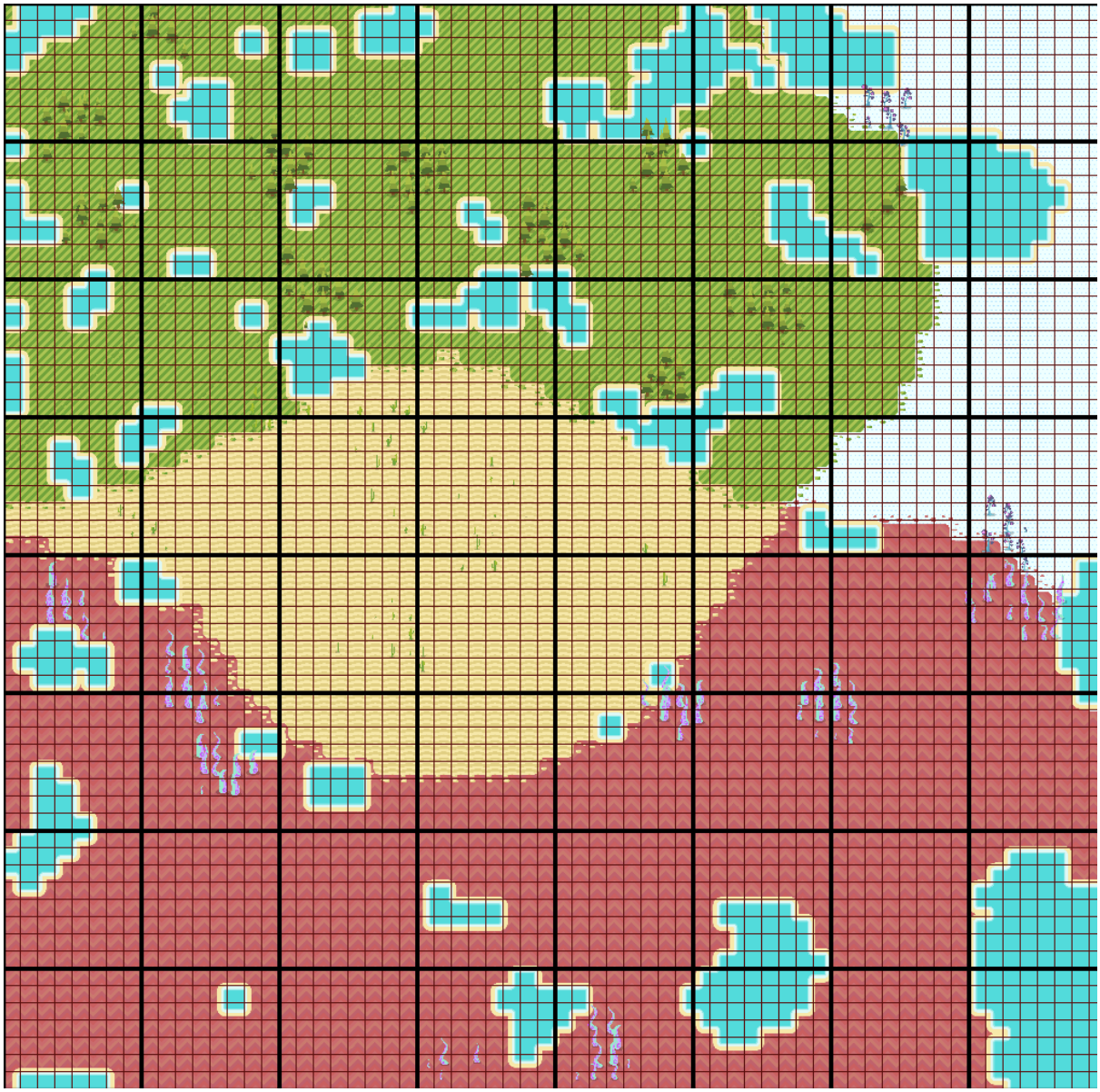
Detta kapitel kommer diskutera designval som tagits under implementationsfasen, samt ge en djup beskrivning av implementationen av respektive del, experimentmiljö och individers beteende.

4.1 Experimentmiljö

Världen individerna lever i består av olika typer av rutor. I dessa rutor kan olika växter, vatten och individer existera. Figur 13 illustrerar en värld med dessa beståndsdelar. Vilken typ av ruta det är bestäms av vilken temperatur och luftfuktighet det är på den rutan. Exempelvis är en ökenruta, i mitten, torr och varm. Notera även att öknen inte har sjöar, vilket beror på att öknen är en mycket torr plats. En grön skogsruta i norr är varm och fuktig. Till skillnad från öknen har skog mycket högre luftfuktighet vilket också medför att skogen har sjöar i vegetationsområdet. Skog och öken är exempel på vegetationsområde som finns i spelvärlden. Det finns även vinter som illustreras i nordöst, samt tundra i söder. Individerna kommer sedan anpassa sig till miljöerna genom att de strävar efter att gå till det vegetationsområde som passar bäst för dem.

I denna rapport definieras en ruta som en av de små kvadraterna i Figur 13. Detta rutmönster navigera individerna efter för att hitta till vatten, mat och andra individer. I Figur 13 finns även större kvadrater. De större kvadraterna definieras som en världsbit. Tanken med världsbitar är att det minskar antalet rutor individerna måste gå igenom för att söka efter mat, vatten eller andra individer. Istället för att individerna letar igenom alla rutor som existerar på banan hämtar en individ de närmsta världsdelarna och söker igenom de rutor som existerar i dessa världsdelar. Detta är ett optimeringsbeslut som togs för att underlätta när många individer söker efter föda.

Miljön är slumpvist genererad, dock kan seed sättas för att generera samma värld. Seed fungerar som nycklar för en slumpgenerator. Om en slumpgenerator får samma seed kommer generatorn alltid generera samma serie av värden. På så vis kan världar genereras slumpvist, fast generatorn kommer alltid generera samma värld beroende på vilket seed generatorn får. Genereringen av miljön med dess träd och sjöar är skapad av en annan programmerare som jobbar på samma spel som detta projekt grundar sig i.



Figur 13 Experimentmiljö med rutor (mörkröda linjer) och världsbitar (svarta linjer)

4.1.1 Aspekter från naturligt ekosystem

Från Clarkes (1954) bok om ekologi och Rönkkös (2007) arbete om artificiella ekosystem samt diskussion i den grupp som tillsammans skapar ekosystemsspelet. Från detta har ett antal viktiga aspekter för ekosystemet tagits fram. Dessa aspekter ligger till grund för hur programmet har implementerats:

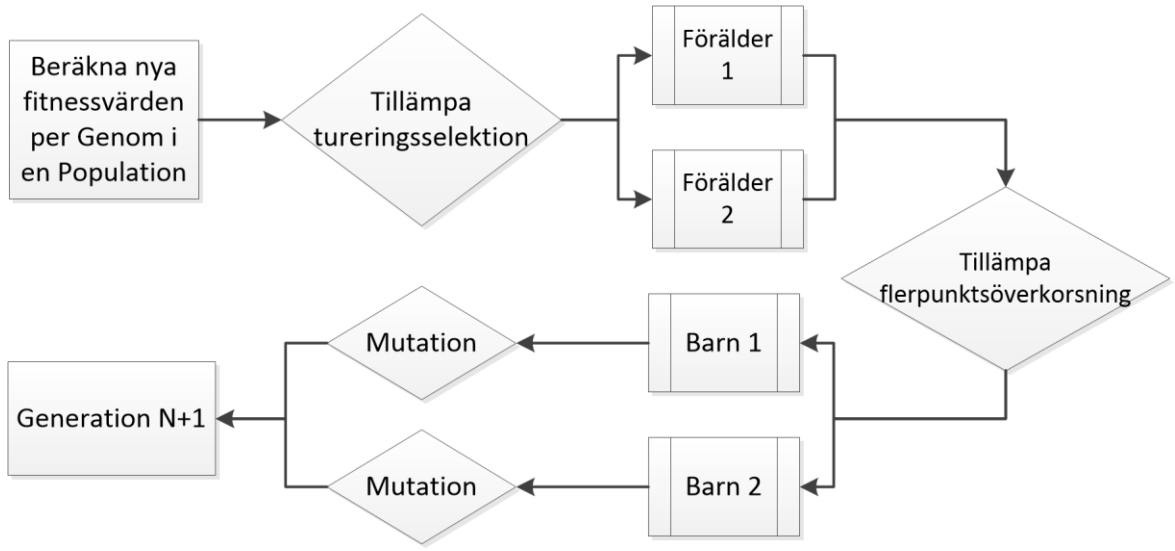
- *Mat*: Mat i form av växter och andra individer.
- *Vatten*: Sjöar som individer kan dricka ifrån.
- *Temperatur*: Denna aspekt avser att skapa en global påverkan på individerna. Om individerna befinner sig i ett för kallt klimat skall deras beteende styra dem till ett varmare klimat. Temperatur bestämmer också vilken typ vegetationsområde en ruta är definierad som.
- *Fuktighet*: Bestämmer hur mycket vatten ett vegetationsområde har.

- *Individuellt Beteende*: Beteende som gör att individerna kan leva i sin miljö. Alla individers beteenden beskrivs i kapitel 4.4.2 (Beteendefunktioner).
- *Miljöpåverkan*: Individerna äter av växter.

Att bestämma vilka beteenden individerna skall ha i ekosystemet är en avgörande del av implementationen. Beteendena måste vara utformade så att de tar nytta av alla ekosystemspecifika aspekter som nämndes tidigare. Det skall finnas ett äta-beteende som innefattar både växter och kött, dvs. det skall finnas växtätare och köttätare. Alla individer skall också ha möjlighet att dricka vatten, samt kunna söka efter ett klimat som passar individen bäst.

4.2 Den genetiska algoritmen

Den GA som implementerades i detta projekt fungerar som Jones (2002) beskrivning av Hollands (1992) GA, som togs upp i kapitel 2.4.1. Det finns en population som initieras från start med slumpade värden. En population i detta fall är en samling organismer. Varje organism är en kombination av ett genom och ett fitnessvärde. Genomet i sin tur innehåller en serie gener. En gen representeras som ett heltal. Efter initieringen beräknas ett fitnessvärde per organism i en population. Därefter används turneringsselektion för att ta ut två av de bästa organismerna från en delmängd av populationen. Dessa organismer används sedan för att generera två nya barn till en ny generation med hjälp av flerpunktsöverkorsning. Under överkorsningen kan det även ske en mutation. Mutationen i detta fall ändrar ett heltalsvärde på en vald gen till ett nytt slumpat värde. När de nya barnen är skapade, börjar algoritmen om från turneringsselektionen, tills en ny population är genererad. Då ersätts den gamla populationen med den nya. Figur 14 illustrerar flödet under en hel evolutionsepok när en ny population av genom skall evolveras.



Figur 14 Flödesdiagram över en evolutionsepok

4.2.1 Fitnessfunktionen

En av de viktigaste delarna i GA är fitnessfunktionen. Buckland (2002) nämner också att fitnessfunktionen är svårast att definiera i en GA på grund av att det oftast finns många mål en organism måste uppfylla. I detta projekt måste fitnessfunktionen vara anpassad så att individerna överlever så lång tid som möjligt i någon miljö som individerna skall överleva i.

För att anpassa fitnessfunktionen till individen så att den överlever längst är fitnessfunktionen direkt kopplad till individens varierande variabelvärden, dessa värden kommer beskrivas i senare kapitel. Tanken är att GA skall försöka skapa en serie actions som är optimalt för individen vid en viss tidpunkt. Om en individ är hungrig skall fitnessfunktionen hjälpa till så att en serie actions innehåller åtminstone en action som gör att individen äter. Om individen istället är mätt skall fitnessfunktionen undvika att evolvera en serie actions som innehåller action för att äta mat. Här nedanför är en pseudoversion av fitnessfunktionen som beräknar fitnessvärde för varje genom i en population.

```
private void calculate_population_fitness(Creature creature)
{
      // Get the values from the individual
      health      = creature.health;
      thirst      = creature.thirst;
      hunger      = creature.hunger;
      body_temp   = creature.body_temp;


      population.fitnessvalue = 0;
      for(every gene in population-genome[n])
      {
            switch ((gene[n]).get_value)
            {
            case 0:
                  if(creature.body_temp < 100)
                        body_temp += 50;
                  else
                        body_temp -= 10;
                  break;
            case 1:
                  if(creature.body_temp > 100)
                        body_temp += 50;
                  else
                        body_temp -= 10;
                  break;
            case 2:
                  hunger = 100 / creature.hunger;
                  break;
            case 3:
                  thirst = 100 / creature.hunger;
                  break;
            case 4:
                  if(health < 100)
                        walking += 100;
                  else
                        walking += 100;
                  if(hunger >= 100)
                        health += 50;
                  break;
            case 5:
                  walking += 40;
                  break;
            case 6:
                  walking += 40;
                  break;
            default:
                  break;
            }
      }
```

```
      population.fitnessvalue += health;
      population.fitnessvalue += thirst;
      population.fitnessvalue += hunger;
      population.fitnessvalue += body_temp;
      population.fitnessvalue += walking;
      population.fitnessvalue += reproduction;
      population.fitnessvalue += population-genome[n].fitness;

      // Store the best genome for later Elitism.
      if(population-genome[n] > fitness)
      {
            fitness = population-genome[n].fitness;
            fittest_genome = population-genome[n];
      }
}
```

4.2.2 Selektion

Selektion är en viktig del av GA. Selektion bestämmer vilka två genom som skall korsas och skapa en delmängd av en ny generation genom. Valet av selektionsmetod spelar stor roll. Exempelvis om alltid de bästa individerna utses och nya generationer skapas med dessa, finns en risk att systemet fastnar på en icke-optimal lösning och kommer aldrig ifrån den. Det är också viktigt att mindre bra genom finns med i nya generationer. Detta ger en variation bland gener, vilket skapar en större chans att systemet når en optimal lösning (Buckland, 2002).

Under implementationen har två selektionsmetoder övervägds: Turneringsselektion och roletthjulsselektion med elitism. Tanken är att få en blandad population av både bra och dåliga genom för att kunna hitta en lösning på ett effektivt sätt. Både turnerings- och roletthjulsselektion har dessa egenskaper.

Roletthjulsselektion beskrevs i bakgrunden, där Jones (2002) nämnde att elitism kombinerat med roletthjulsselektion kan öka chansen till att en nära optimal lösning evolveras fram. Dock nämner Buckland (2002) att roletthjulsselektion kräver ett antal förberäkningar samt ett steg för att skala om fitnessvärden. Buckland argumenterar för att alltid överväga turneringsselektion i implementationen innan andra selektionsmetoder övervägs. Buckland menar att denna metod är mycket effektivare än roletthjulsselektion på grund av att metoden inte behöver göra förberäkningar och skalning.

Utifrån detta så valdes turneringsselektion som selektionsmetod. Här nedanför är en pseudoversion av den implementerade turneringsselektionsmetoden. Metoden går ut på att först ange hur många tävlande som skall finnas i turneringen. For-loopen kommer sedan köra lika många varv som det finns tävlanden. Under varje varv hämtas en slumpartad organism från den nuvarande populationen. Organismens fitnessvärde jämförs med den nuvarande bästa fitnessvärdet. Om organismens fitnessvärde är högre än det nuvarande sparas indexet för denna organism undan. När selektionen är klar så returneras organismen med det högsta fitnessvärdet från den delmängd som tävlade under selektionen.

```
private Genome tournament_selection(int NUM_COMPETITORS)
{
      double      best_fitness      = 0.0;
      int         best_genome       = 0;

      for(int i = 0; i < NUM_COMPETITORS; ++i)
      {
```

```
            int genome_index = RandomValue(0 – POPULATION_SIZE);
            if(population[genome_index].fitness > best_fitness)
            {
                  best_genome       = genome_index;
                  best_fitness      = population[genome_index].fitness;
            }
      }
      // Return best genome
      return population[best_genome];
}
```

4.2.3 Överkorsning

Tanken var att överkorsning av individer skulle använda Partially-Mapped Crossover (PMX) (Buckland, 2002), som beskrevs i kapitel 2.4.5, vilket också implementerades till en början. Senare i utvecklingen upptäcktes det dock att individer berikas av dubbletter. Växter i ekosystemet har en begränsad nivå av näring vilket gör att om individer är mycket hungriga måste de äta flera växter för att återställa sin hungernivå. På grund av detta berikas systemet av dubbletter med actions i en serie. På grund av att PMX är till för överkorsning utan dubbletter så implementerades istället en enklare version. Denna överkorsningsmetod kallas för flerpunktsöverkorsning. Metoden illustreras med pseudokod här nedan. Alla gener hos *mum* och *dad* itereras igenom. Om *RandomValue* är under ett *CROSSOVER_RATE* skall *baby_0* få pappans gen och *baby_1* få mammans gen. Annars händer det motsatta. Detta sker för varje gen mamman och pappan har, vilket till slut har skapat två nya barn.

```
private void crossover_multi_point(Genome mum,
                                   Genome dad,
                                   Genome baby_0,
                                   Genome baby_1)
{
      for(every gene in mum and dad)
      {
            if(RandomValue < CROSSOVER_RATE)
            {
                  baby_0 gene = dads gene;
                  baby_1 gene = mums gene;
            }
            else
            {
                  baby_0 gene = mums gene;
                  baby_1 gene = dads gene;
            }
      }
}
```

4.2.4 Mutation

Mutation har hållits enkel redan från början. Buckland (2002) nämner många varianter av mutitionsalgoritmer. Många av dem ändrar positioner på gener i ett genom, vilket kan vara bra om genomen inte skall innehålla dubbletter av gener eller om generna som initierades från start alltid skall finnas kvar, det vill säga, att ingen gen skall ersättas av en ny slumpartad gen. Dock valdes i detta projekt en enkel mutationsalgoritm. Denna algoritm beskrivs i pseudokod här nedanför.

```
private void mutate(Genome genome)
{
      for(every gene in genome)
```

```
	{
		random = RandomValue(0 - 1);
		if(random < MUTATION_RATE) // Should gene mutate?
		{
			// Alter a genes value
			gene = RandomValue(0 - MAX);
		}
	}
}
```

Tanken med denna mutationsalgoritm är att ge en möjlighet att mutera en gen på så vis att det kan förekomma dubbletter av samma gen. Detta sker med hjälp av att sätta en specifik gen till ett nytt slumpat värde. *MAX* i detta fall bestämmer det största värde som kan *RandomValue* kan generera. *MAX* är också maxantalet av actions som finns. Det vill säga, om det finns sju actions är *MAX* sju.

4.2.5 Representation av Actions

Innan GA implementerades har ett antal designval gjorts. Ett viktigt designval som gjordes var hur GA skall representera actions i form av gener. Reeves och Rowe (2003) nämner i sin bok att det vanligaste sättet att representera gener är i binär form. Detta övervägdes till en början,

Tabell **1** illustrerar hur alla actions skulle representeras med binära tal. En evolverad lösning från GA i form av ett genom med sju gener skulle kunna se ut såhär i binär form:

000 | 010 | 100 | 110 | 101 | 001 | 000

Tabell 1 En binärrespesentation av varje action.

Identification	**Action**
000	Find Warmer Place
001	Find Cooler Place
010	Find Food
011	Find Water
100	Walk Randomly
101	Follow Leader
110	Idle

Dock efter vidare forskning av GA valdes den binära representationen bort. Detta på grund av att regelsystemet, som beskrivs i senare kapitel, skulle behöva avkoda en evolverad binärlösning varje gång en individ skall utföra en ny action. Detta är något som Buckland (2002) påpekar i ett av hans egna projekt. Buckland nämner att en binärrepresentation skulle ge ett onödigt avkodningssteg. Buckland ger också exempel på att en mer intuitiv representation i form av heltal. Bäck (1996) har även gjort en grundlig jämförelse mellan binär och heltalsrepresentation av gener i GA. Bäck summerar resultatet i att en

binärrepresentation är mer komplext i många fall på grund av avkodningsprocesser jämfört med heltalsrepresentationen som oftast undviker avkodningsprocessen. Förutom detta nämner Bäck att tidskomplexiteten för ett system som använder binärrepresentation är exponentiell jämfört med en logaritmisk tidskomplexitet för heltalsrepresentationen. Utifrån denna information valdes istället att actions skall representeras av heltal. Tabell 2 ger en beskrivning av hur systemet i detta projekt mappar heltalen till actions.

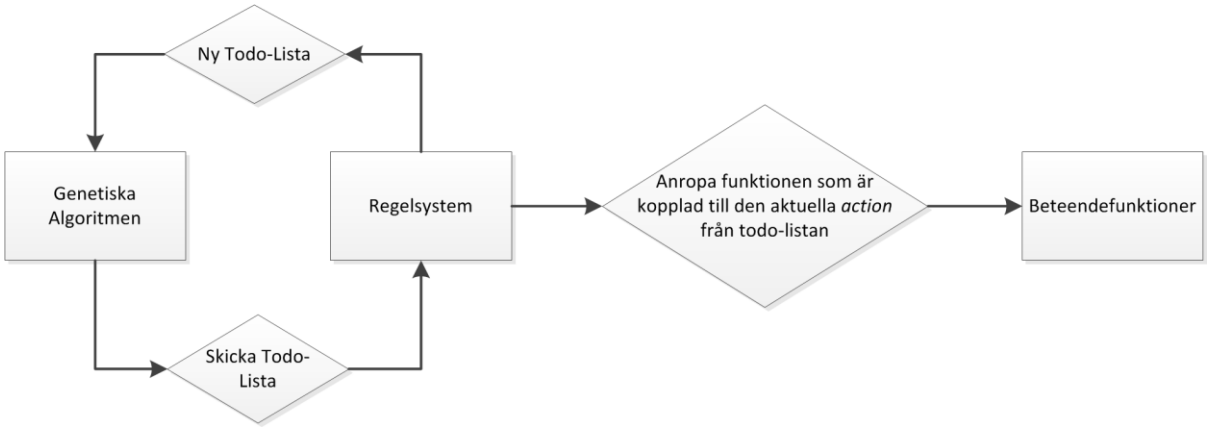
Tabell 2 Identifiktation och beskrivning av respektive handling individer kan ta.

Identification	**Action**
0	Find Warmer Place
1	Find Cooler Place
2	Find Food
3	Find Water
4	Walk Randomly
5	Follow Leader
6	Idle

Idén kommer från hur Buckland (2002) tolkar värden från sin GA. En skillnad med Bucklands lösning är att han evolverar serier av action- och tidsintervall-par och i denna lösning evolveras bara heltal. Tanken från början var att detta projekts beteendesystem också skulle stödja tidsintervall. Efter senare övervägning slopades dock tidsintervall. Detta grundar sig i att det vore underligt att en individ som exempelvis går mot ett kallare klimat samt avbryter och istället utför *Idle* istället. Istället valdes att individerna måste utföra ett helt action innan den kan utföra ett annat.

4.2.6 Regelsystemet

För att GA systemet skall fungera korrekt designades ett regelsystem som tolkar utdatan från GA. Regelsystemet illustreras i Figur 15 som också visar på hur kommunikationen mellan GA, regelsystemet och beteendefunktionerna sker. Regelsystemet kallar på en funktion från GA. Denna funktion kommer gå igenom en evolutionsepok, vilket illustrerades tidigare i Figur 14. När evolutionsepoken är klar returnerar funktion en evolverad serie av actions, ett så kallat genom. Därefter tolkar regelsystemet detta genom som en todo-lista. Denna todo-lista innehåller alla actions en individ måste genomföra. När alla actions är genomförda börjar allting om och en ny todo-lista genereras från GA.



Figur 15 Kommunikation mellan GA, Regelsystem och Beteendefunktioner

Utifrån detta implementerades funktionen *handle_todo* som illustreras i pseudokod här nedanför. Vad som kan noteras är att regelsystemet byggs upp av switch-case- och if-satser som talkar output-värden från GA. Notera också att det är pseudokod vilket bara skall ge en enkel beskrivning av regelsystemet. Vilken beteendefunktion som anropas är helt beroende på vilket heltalsvärde som hämtas från *todo_list*.

```
void handle_todo()
{
      if(action_index > length of todo_list)
      {
            set fetch_new_todo_list to true
            reset action_index to 0
            return;
      }


      switch(todo_list[action_index])
      {
      case action_0:
            status = call behavior function 0;
            break;
            .
            .
            .

      case action_n:
            status = call behavior function n;
            break;
      }

      if(status == FunctionDone)
            increase action_index;
}
```

Något som inte var helt arbiträrt var hur många heltal en lösning som evolveras av GA skulle vara. Om det finns för lite actions i en lösning måste GA köras ofta, vilket kan resultera i att systemet går långsamt. Detta är oacceptabelt därför att projektet avser att skapa en GA för ekosystem i spel samt att spel oftast är skapade för köras i realtid. Att skapa för långa lösningar är heller inte att föredra på grund av GA i detta fall avser att evolvera en serie actions som skall vara anpassade efter vilka behov en individ har vid en specifik tidpunkt. GA kollar hela tiden på individens varierande värden för *health, hunger, thirst* och

body_temp för att bestämma en individs behov. Dessa värden förändras under tid. Vilket gör att om en för lång lösning evolveras från GA, kommer denna lösning föråldras och är efter ett tag ej anpassad till individens behov. Utifrån detta valdes då att en lösning skall innehålla lika många actions som det finns möjliga actions. Det vill säga om det finns sju actions skall GA evolvera en lösning som innehåller en serie av sju gener. Detta gör att det finns möjlighet att utföra alla definierade actions i systemet, samt att en ny todo-lista inte behöver hämtas för ofta.

4.3 Tillståndsmaskinen

Den andra AI-tekniken som implementerades är tillståndsmaskinen. Ett antal tillstånd definierades. Valet gjordes att definiera tillstånd som skall vara grundliga och enkla att förstå, men också innefatta de viktigaste tillstånden en individ i ett ekosystem kan existera i. Tillstånden är också anpassade efter vilka aspekter som togs fram för ekosystemet. Här nedanför är alla tillstånd som har definierats för detta projekt:

- *FOOD_HUNGRY:* Individens hungernivå är låg.
- *HEALTH_WOUNDED*: Individens liv ligger på en låg nivå.
- *TEMP_COLD:* Kroppstemperaturen för en individ är för låg.
- *TEMP_HOT:* Kroppstemperaturen för en individ är för hög.
- *WATER_THIRSTY:* Vatten nivån hos en individ är låg .
- *WALK_RANDOMLY:* Individen går omkring i ett område utan något mål.
- *FOLLOWING_LEADER:* En individ är för långt ifrån sin ledare.
- *IDLE:* Individen står stilla på en position.

I tillståndsmaskinen är det också möjligt att göra ett tillståndsbyte från ett tillstånd till alla de andra definierade tillstånden. Dessa tillståndsbyten bestäms av vilket event som har tillkallats på en specifik tillståndsfunktion. Det är också dessa event som bestämmer vad en individ skall göra vid en viss tidpunkt. Här nedanför är en summering av alla event som kan ske:

- *ENTER:* Tillkallas när individen byter från ett tillstånd till ett annat.
- *EXIT:* Tillkallas när individen byter från ett tillstånd till ett annat.
- *FIND_FOOD:* Säger till systemet att individen skall börja söka efter mat.
- *FIND_WATER:* Säger till systemet att individen skall söka efter vatten.
- *WALK_RANDOMLY:* Individen skall börja gå slumpartat till olika ställen.
- *FIND_TEMP_COOLER:* Individen har för hög kroppstemperatur och måste hitta ett varmare klimat.
- *FIND_TEMP_WARMER:* Individen har för låg kroppstemperatur och måste hitta ett kallare klimat.
- *FOLLOW_LEADER:* Individen är för långt ifrån sin ledare och måste gå till ledaren.
- *IDLE:* Individen skall bara stå på sitt ställe.

För varje tillstånd finns det även en funktion som hanterar tillståndet. Här nedanför är ett exempel på tillståndsfunktionen som hanterar *FOOD_HUNGRY*. De resterande funktionerna, som hantera de andra tillstånden, ser ut som funktionen här under. Funktionen får ett av eventen som beskrevs tidigare. Beroende på vilket event det är så anropas respektive *case*-block i switch-case-satsen. Om exempelvis funktionen får in eventet

FIND_TEMP_COOLER skall funktionen byta tillstånd från det nuvarande, *FOOD_HUNGRY,* till *TEMP_COLD*. Annars om funktionen får in eventet *FIND_FOOD* skall blocket anropa beteendefunktionen som gör att individen börjar söka och äta mat.

```
public void handle_food_hungry(BehaviorManager.Events e)
{
	switch(e)
	{
	case BehaviorManager.Events.ENTER:
		break;
	case BehaviorManager.Events.EXIT:
		break;
	case BehaviorManager.Events.FIND_FOOD:
		if(_creature.carnivore)
			_status = _behavior.find_meat(_creature);
		else
			_status = _behavior.find_food(_creature);
		break;
	case BehaviorManager.Events.FIND_TEMP_COOLER:
		_manager.change_state(BehaviorManager.State.TEMP_COLD);
		break;
	case BehaviorManager.Events.FIND_TEMP_WARMER:
		_manager.change_state(BehaviorManager.State.TEMP_HOT);
		break;
	case BehaviorManager.Events.FIND_WATER:
		_manager.change_state(BehaviorManager.State.WATER_THIRSTY);
		break;
	case BehaviorManager.Events.WALK_RANDOMLY:
		_manager.change_state(BehaviorManager.State.WALK_RANDOMLY);
		break;
	case BehaviorManager.Events.FOLLOW_LEADER:
		_manager.change_state(BehaviorManager.State.FOLLOWING_LEADER);
		break;
	case BehaviorManager.Events.IDLE:
		_manager.change_state(BehaviorManager.State.IDLE);
		break;
	default:
		_manager.change_state(BehaviorManager.State.IDLE);
		break;
	}
}
```

Systemet, som hanterar övergångar mellan tillstånd och transporteringen av event, bygger på tre funktioner. Den första funktionen, *handle*, används för att transportera vidare events till funktionen *dispatch*. Funktionen *dispatch* består av en switch-case-sats som liknar funktionen *handle_food_hungry* som beskrevs tidigare. Dock istället för att kolla på event kollar denna funktion på vilket tillstånd en individ befinner sig i. Det är också denna funktion som bestämmer vilken tillståndsfunktion ett event ska transporteras till. Hanteringen av tillstånden bygger på designmönstret *State* (Gamma, 1995). Funktionen *dispatch* illustreras här nedanför.

```
private void dispatch(BehaviorManager.Events e)
{
	switch(_handler.state)
	{
	case BehaviorManager.State.FOOD_HUNGRY:
		_handler.handle_food_hungry(e);
		break;
	case BehaviorManager.State.HEALTH_WOUNDED:
```

```
            _handler.handle_health_wounded(e);
            break;
      case BehaviorManager.State.TEMP_COLD:
            _handler.handle_temp_low(e);
            break;
      case BehaviorManager.State.TEMP_HOT:
            _handler.handle_temp_high(e);
            break;
      case BehaviorManager.State.WATER_THIRSTY:
            _handler.handle_water_thirsty(e);
            break;
      case BehaviorManager.State.WALK_RANDOMLY:
            _handler.handle_walk_randomly(e);
      break;
      case BehaviorManager.State.FOLLOWING_LEADER:
            _handler.handle_finding_leader(e);
            break;
      case BehaviorManager.State.IDLE:
            _handler.handle_idle(e);
            break;
      default:
            _handler.handle_idle(e);
            break;
      }
}
```

Den sista funktionen *transit* hanterar övergångar mellan olika tillstånd. Om exempelvis funktionen *handle_food_hungry* får ett annat event än *FIND_FOOD,* kommer handle_food_hungry anropar funktionen *change_state* och skickar vidare eventet levererades till *handle_food_hungry*. Det *change_state* gör är att anropa *transit* som i sin tur anropar *internal_transit*. I *internal_transit* transporteras eventet *EXIT* till den nuvarande tillståndsfunktionen. Där efter byts tillståndet till ett nytt och eventet *ENTER anropas på den nya tillståndsfunktionen.*

```
private void internal_transit(BehaviorManager.State to_state)
{
      dispatch(BehaviorManager.Events.EXIT);
      _handler.state = to_state;
      dispatch(BehaviorManager.Events.ENTER);
}
```

Under implementationen av tillståndsmaskinen har ett nytt tillstånd lagts till. Tillståndet *FOLLOWING_LEADER*. Det var inte tänkt från början att detta tillstånd skulle existera. Det var oklart om det fanns tid över att implementera hanteringen av detta tillstånd. Dock när tillståndsmaskinen var till stor del implementerad fanns det tid över att implementera hanteringen av tillståndet *FOLLOWING_LEADER*.

För tillståndsmaskinen innebär detta att en ny tillståndfunktion måste implementeras samt att för alla de andra tillståndsfunktionerna måste ett nytt *case*-block läggas till som hanterar skall kalla på funktionen *change_state* för att byta till tillståndsfunktionen för *FOLLOWING_LEADER*.

Tanken med tillståndet *FOLLOWING_LEADER* är att försöka gruppera individerna i ekosystemet i flockar så att de inte blir för separerade. Detta är ett tillstånd som är nödvändiga i ekosystemet, dock inte avgörande för att skapa ett fungerande ekosystem (Clarke, 1954).

4.4 Individer

Individerna är grunden för detta arbete. De val som har påverkat arbetet mest är vilka beteendefunktioner individerna har att välja mellan, vilka egenskaper individerna skall ha samt vilka variabler som skall påverkas under tidens gång.

4.4.1 Ledaren

I varje flock finns det en ledare. Denna ledare skall samla en flock och göra så att individer håller sig inom ett område runt ledaren.

Ledaren evalueras hela tiden under spelets gång. Detta är viktigt att om en ledare skulle dö måste det finnas en ny individ som tar över rollen efter ledaren. Här nedanför är ett exempel på hur en ledare evalueras. En ledare evalueras genom att iterera genom alla individer i en flock och kolla på dess fitnessvärden. Fitnessvärdet beräknas genom att skapa en summa av individens *speed*, *adaptation* och *age*. Individen med högst fitnessvärde blir ledare över flocken.

```
for(every creature in herd)
{
      fitness = creature.speed + creature.adaptation + creature.age;
      if(fitness > highest_fitness)
      {
            highest_fitness = fitness;
            leader = creature;
      }
}
```

4.4.2 Beteendefunktioner

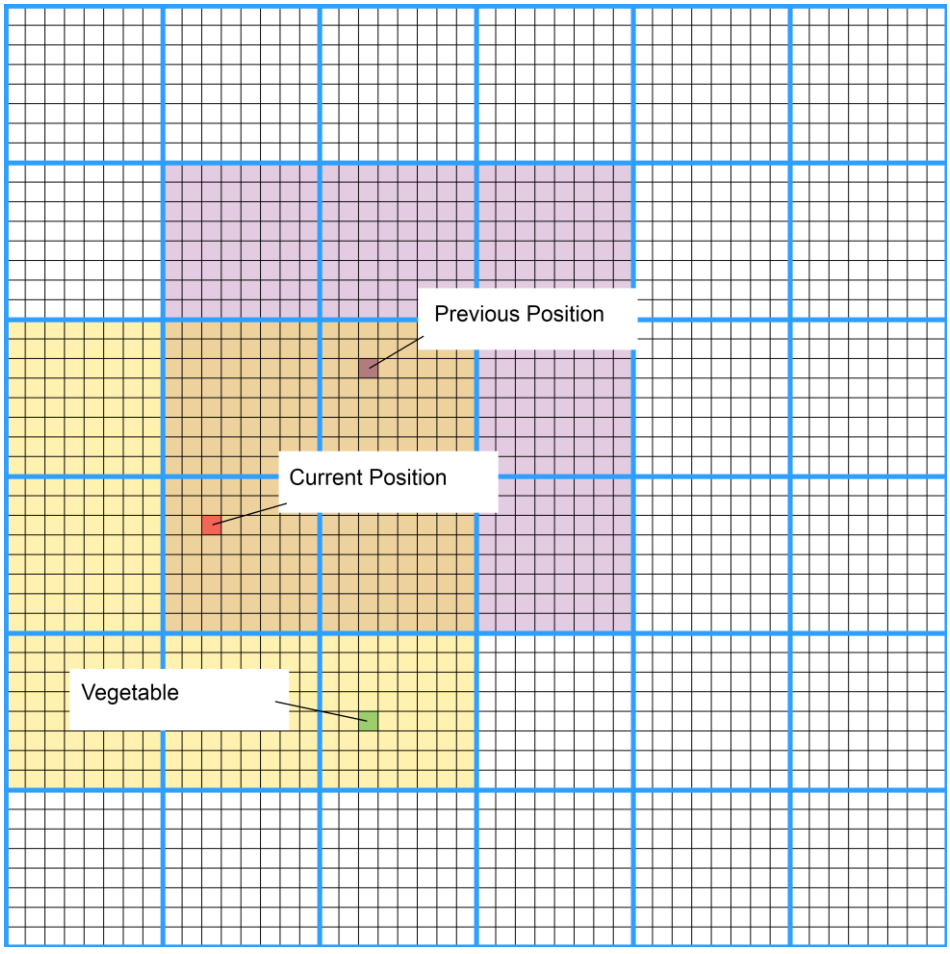
Vissa beteenden har slagits ihop för att underlätta kommunikationen mellan beteendealgoritmerna och de existerande beteendefunktionerna. Ett exempel på ett beteende som har slagits ihop är *find_food*. Detta beteende bygger på att först hitta en växt i de närmaste världsbitarna, därefter går individen mot växten samt äter den. Alla dessa delar anropas som en serie när GA eller tillståndsmaskinen anropar funktionen *find_food*. Själva strukturen för beteendefunktionerna blir mer komplex, men beteendehanteringen hos tillståndsmaskinen och den genetiska algoritmen blir lättare på grund av att det är färre funktioner att anropa. Eftersom det blir lättare att implementera respektive algoritm minskar risken för problem under implementationen. Det minskar också risken för oönskade värden när undersökningen görs, samt att förståbarheten för respektive algoritm håller sig på en bra nivå.

- *find_warmer_place:* Individen söker i de nio närmaste världsbitarna och försöker hitta den ruta som har en varmare grad än vad individen står på nu, samt går dit.
- *find_cooler_place:* Individen söker på samma sätt som *find_warmer_place* dock försöker individen hitta en ruta som är kallare än den individen står på just nu.
- *find_food / find_meat:* Beroende om individen är köttätare eller växtätare kommer individen kalla på funktionen *find_meat* respektive *find_food*.
    - *find_meat:* Individen söker efter en växtätare, om växtätaren har ett lägre styrkevärde kommer individen kunna döda växtätaren. Om individen hittar en sådan växtätare kommer individen gå till denna växtätare, döda den samt äta upp köttet. För enkelhetensskull ingår all denna funktionalitet i samma

funktion, gå, döda och äta. Detta gör det mindre komplext att designa tillståndsmaskinen och genetiska algoritmen på grund av att de inte behöver ta hänsyn till att det finns en funktion för respektive funktion, gå, döda och äta.

    - *find_food:* Individen söker efter den närmsta växten i de nio närmsta världs-bitarna. Om individen inte skulle hitta tar den en position i någon av de åtta världs-bitarna runt den världsbiten individen existerar i just nu. Detta illustreras i Figur 16, bilden illustrera också områdena som individen söker i på respektive position. En världs-bit i detta fall motsvaras en blå fyrkant. Där en blå fyrkant innehåller 64 rutor.

- *find_water:* Denna funktion fungerar ungefär som *find_food*. Individen söker i de nio närmsta världsbitarna för att hitta vatten. Om den inte hittar vatten i de nio närmsta går den till någon slumpad ruta i någon av de åtta yttre världsbitarna.
- *walk_randomly:* Individen hittar en slumpartad ruta som individen sedan går till.
- *follow_leader:* Individen söker efter sin ledare samt går till ledaren
- *idle:* Individen står stilla på sitt ställe och fördriver tiden.



Figur 16 Illustration av individers sätt att hitta mat i en miljö av tiles och chunks.

4.4.3 Individernas Egenskaper

Under implementationen av individen har ett antal designval gjorts rörande vilka egenskaper en individ skulle ha. En egenskap som nämndes i bakgrunden var *påfågelfaktor*. Denna egenskap skulle påverka hur populär en individ är i en flock. En individ med högre

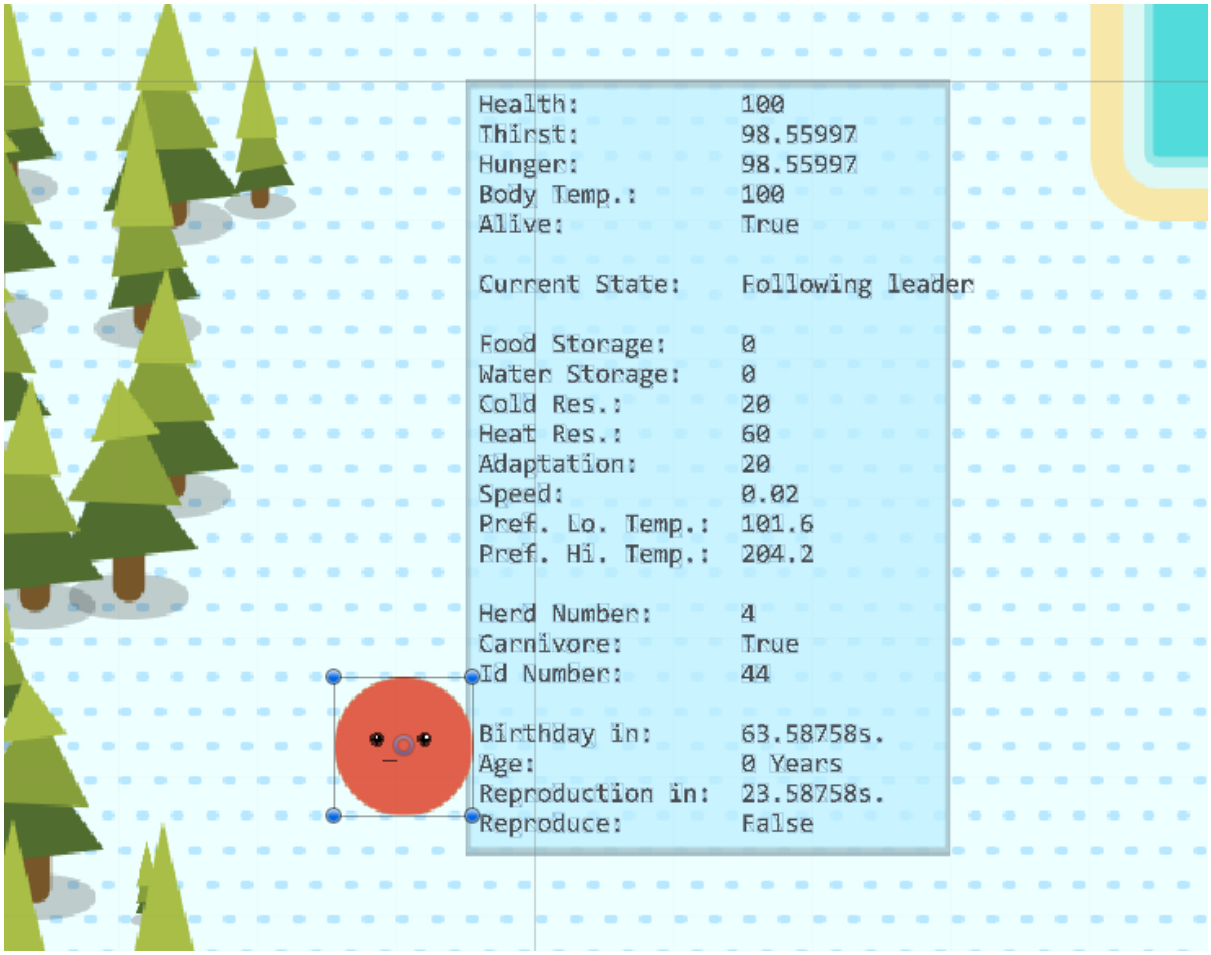
påfågelvärde skulle ha högre chans att föra vidare sina egna gener i en flock. Dock på grund av tidsbrist implementerades ej någon reproduktionsprocess hos individerna. Detta gjorde också att egenskapen *påfågelfaktor* också blir oanvändbar. En annan egenskap som nämndes i bakgrunden var *päls*. Egenskapen *päls* finns med dock har *päls* fått ett annat namn, vilket ger en mer generell beskrivning av vad egenskapen gör, nämligen *Cold Resistance*. Egenskapen *Heat Resistance,* som är motsatsen till *Cold Resistance,* valdes att implementeras. Tanken var från början att *Cold Resistance* skulle agera som både *Heat Resistance* och *Cold Resistance* i den mån att om en individ har lite i *Cold Resistance* klarar individen varma klimat bättre. Dock på grund av hur systemet för generering av egenskaperna, när flockar i världen initieras med individer, är byggt fungerar det inte att ha två egenskaper som *Cold Resistance* och *Heat Resistnace*. En annan egenskap som valdes bort var egenskapen *storlek*, denna egenskap skulle bestämma hur mycket hälsa en individ har. Dock ansågs denna egenskap som ett onödigt tillägg i ekosystemet som skulle försvåra utvärderingen av resultat senare i undersökningen på grund av att de skapar en ytterligare variabel som kan ändra en individs överlevnadsförhållanden. På grund av detta slopades denna egenskap. Här nedanför är en lista på alla definierade egenskaper en individ har i spelet:

- *Food Storage:* Denna egenskap beskriver hur lång tid en individ kan klara sig utan att bli hungrig, samt hur snabbt individen äter. Med ett högre värde kan individen klara sig längre utan mat samt individen kan äta snabbare.
- *Water Storage:* Denna egenskap funkar som *Food Storage*. Dock är ämnad för att hantera hur väl individen kan hålla sig utan vatten, samt hur snabbt individen dricker.
- *Cold Resistance:* Ett högre värde av denna egenskap gör att individen kan klara sig i kallare klimat.
- *Heat Resistance:* Ett högre värde av denna egenskap gör att individen kan klara sig i varmare klimat.
- *Adaptation:* Detta är en komplex egenskap i sig, därför att den bestämmer hur snabba individer är. Om individen är en växtätare fungerar även denna egenskap som ett försvarsvarde samt om individen är en köttätare fungerar denna egenskap som ett attackvärde. Med ett högre *adaptation*-värde klara exempelvis växtätare ifrån att bli uppäten av köttätare.

Ett designval som gjordes för alla individer var att hålla en gemensam maxgräns på hur mycket poäng som kan sättas ut på varje egenskap. Till en början skulle ett värde mellan 0 – 100 slumpvist sättas per varje egenskap. Detta skulle dock göra att överlevnadsförhållandena, per individ, blir för slumpartade. I stället skapades pool med fem poäng, som kan sättas på varje egenskap. Med hjälp av en slumptalsgenerator sätts dessa poäng ut på några av egenskaperna. Efter att dessa fem poäng har satts ut på respektive egenskap konverteras dessa från poäng till värden. Ett poäng är värt 20. Totalt kan en egenskap vara värd 100. Utifrån detta bestäms hur bra en individ är på exempelvis klara kyla och så vidare.

En avvägning som gjordes är att utseendet inte är kopplade till egenskaperna som är listades tidigare. Tanken är att huvud, ben, hals, kropp och svans skall vara kopplade till någon av egenskaperna. Köld-skydd skulle exempelvis ge en kroppsdel päls. Detta gör det lätt att se vilka egenskaper en individ har utifrån dess utseende. Dock på grund av tidsbrist utelämnades detta i den slutliga implementationen och ett mer formellt fönster med text

bredvid individen implementerades. Detta fönster innehåller all information om individens egenskaper och variabler. Detta fönster illustreras i Figur 17.



Figur 17 Individual med informationsfönster

4.4.4 Varierande Variabler

Från de aspekter som nämndes i kapitel 4.1.1, som togs fram av Clarkes (1954) bok om ekologi och Rönkkös (2007) arbete om artificiella ekosystem, har variablerna här nedan definierats. Dessa variabler är till för att styra individens beteenden. På så vis gör dessa variabler att individen använder miljön den lever i för att hålla en bra nivå på sina värden.

- *health:* Denna variabel påverkas när variabelvärdet för *hunger*, *thirst* eller *body_temp* är under ett visst värde. När något variabelvärde är under en viss gräns, påverkas *health* negativt genom att *health* sakta minskar. För att individen *health* skall sluta påverkas negativt måste individen tillfredsställa sig själv genom att utföra någon av den tidigare nämnda funktioner. På så vis ökas variabelvärdet för *hunger, thirst* eller *body_temp* och individens *health* återgår sakta till sitt maxvärde. När köttätarna äter av växtätarna är det *health* som bestämmer hur mycket en köttätare kan äta från individen. Detta betyder att köttätarens variabelvärde för *hunger* ökar mindre om en växtätarens *health* är minimalt.
- *thirst:* Denna variabel påverkas hela tiden av tiden. Efter ett tidsintervall kommer *thirst* ligga under en minimalgräns som bestämmer om individen är törstig eller inte. När denna gräns är nådd måste individen utföra beteendefunktionen *find_water* för

att tillfredsställa variabelvärdet för törst. När individen utför denna funktion kommer individen dricka tills *thirst* når sitt maxvärde.

- *hunger:* Denna variabel är lik *thirst*. Variabeln minskar med tiden samt när *hunger* når ett minimivärde måste individen hitta mat genom att utföra *find_food* eller *find_meat* om individen är växtätare respektive köttätare.
- *body_temp:* Denna variabel påverkas också över tid.

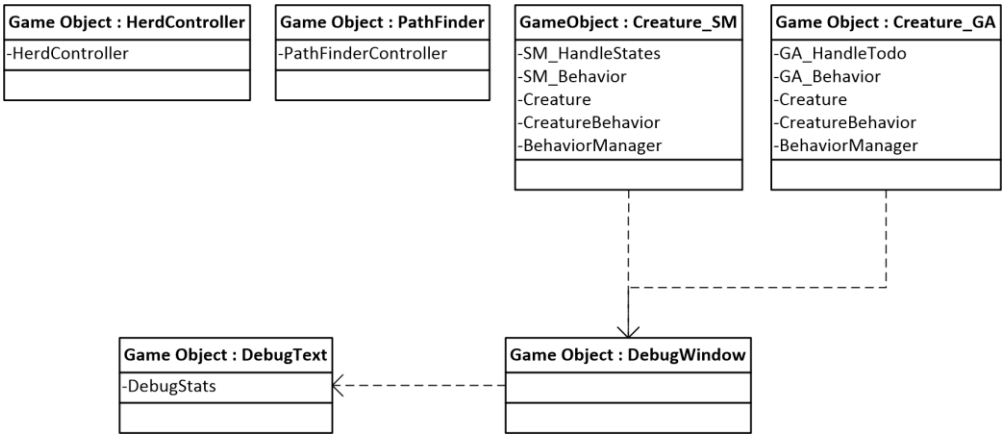
4.5 Systemets Struktur

Experimentmiljön är skapad i Unity3D (Unity Technologies, 2014) där C# har använts som programmeringsspråk för utbyggnad och modifiering av motorn.

Många av klasserna ärver från Unity3Ds egna klass *Monobehavior*. Denna klass innehåller egenskaper från den bakomliggande motorn som Unity3D bygger på. Detta gör att vissa variabler går att visa i editorfönstret när programmet startas i Unity3Ds editormiljö. Klasserna som ärver från Monobehavior måste också sitta på *GameObjects* för att de skall exekvera. GameObjects är en abstrakt container som kan innehålla Unity3Ds egna objekt som textur-renderare, positionering, Monobehavior-skript och så vidare. Att kunna visa variabler i editormiljön underlättar mycket när klasser skall felsökas. Unity3D innehåller också rendering av texturer, ett enkelt sätt att hantera positionering, fysik och kollisionshantering samt att editormiljön är användarvänligt. Detta ligger till grund varför projektet utvecklas med hjälp av Unity3D.

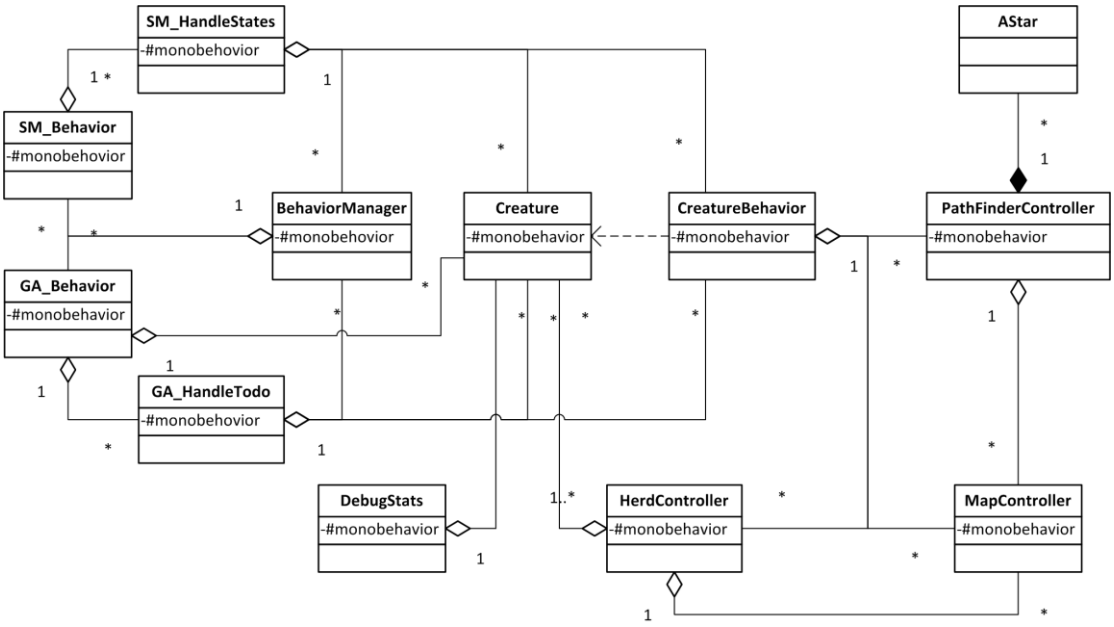
Redan från början var det bestämt att systemets delar som innehåller tillståndsmaskinen och GA skulle vara modulära. Vilket betyder att komponenter lätt kan bytas ut för att skapa en individ vars beteende styrs av GA eller en tillståndsmaskin. Det finns två olika GameObjects för respektive beteendealgoritm, *GA_Creature* (Genetic Algoritm Creature) och *SM_Creature* (State Machine Creature). Som illustreras i Figur 18 innehåller båda objekten ett antal skript. Scriptet *Creature* innehåller alla variabler som definierar en individ, det var dessa variabler som beskrevs tidigare i 4.4.4. Klassen *CreatureBehavior* innehåller alla beteendefunktioner som grundligt beskrevs i kapitel 4.4.2. *BehaviorManager* hanterar kopplingen mellan de beteendespecifika klasserna som *GA_Behavior* och *GA_HandleTodo* samt *SM_Behavior* och *SM_HandleStates*. Dessa klasser beskrevs i kapitel 4.2 respektive kapitel 4.3.

DebugWindow och *DebugText* är två GameObjects som implementerades senare under implementationen. På grund av att hur Unity3Ds editormiljö är konstruerad är det jobbigt att klicka på varje individ i miljön för att se deras värden i ett editorfönster. Istället implementerades ett debug-fönster som visar alla värden i text bredvid individen.



Figur 18 GameObjects och deras kopplingar

Figur 19 här nedanför ger en bild av hur kommunikationen fungerar mellan klasserna i systemet. Som nämndes tidigare ärver många klasser från Monobehavior, detta markeras i respektive klass som ”#monobehavior”. Alla klasser som har en vit romb på sig innehåller en referens till en annan klass. Kommunikationen sker bara från ett håll, i detta fall är det klassen med den vita romben som påverkar den klassen som ej har en vit romb på sig. Exempelvis *BehaviorManager* påverkar *SM_HandleStates, SM_Behavior, GA_Behavior* och *GA_HandleStates*. Det finns en klass som har en svart romb på sig, *PathFinderController*. Detta är klassen som sköter vägplaneringen för individerna. Där vägplaneringen använder klassen AStar, som bygger på A* search algorithm (2014). Till en början skulle inte individerna implementeras med vägplanering, utan de skulle gå i den riktningen som ett objekt befinner sig i tills målet är nått samt att kollisionshantering gör att individer undviker att korsa sjöar. Dock visade det sig att miljöerna blev för komplexa. Det fanns risk för att individerna fastnar på konkava delar av en sjö. På grund av detta implementerades vägplaneringsalgoritmen så att individerna kan hitta runt krokiga sjöar.



Figur 19 Klassdiagram för systemet

4.6 Framtida utveckling av spelet

Som nämndes tidigare utvecklas ett spel parallellt med examensarbetet. De delar av spelet som visades tidigare är en anpassad version som är bara till för examensarbetet. Här nere kommer ett antal bilder som illustrerar det fortsatta arbetet på spelet som utvecklas till företaget Balthazar.

Figur 20 illustrerar en individ som dricker vatten ur en sjö.

Figur 20 Individ dricker vatten ur sjö

Figur 21 Illustrerar en död individ. När individer dör i spelet går faller individen isär och blir en hög på marken.

Figur 21 En död individ som ligger i en hög

Figur 22 illustrerar två typer av individer.

Figur 22 Två typer av individer

Figur 23 illustrerar ett antal individer som lever i en kall miljö.

Figur 23 Individer i en kall miljö

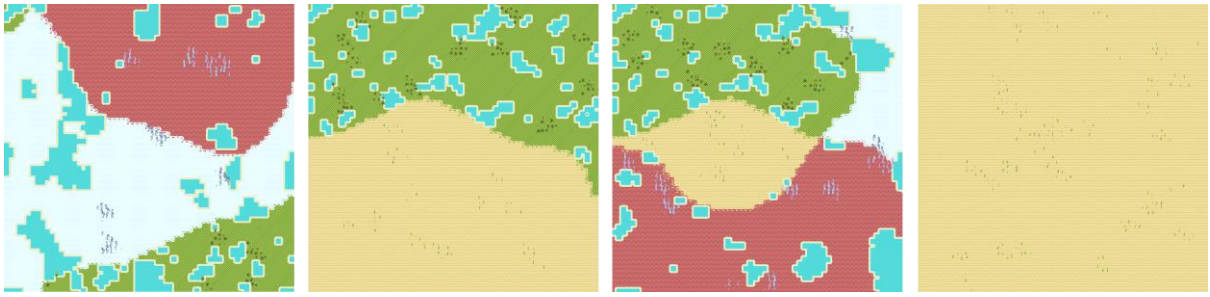
5 Utvärdering

Syftet med detta arbete var att undersöka de två AI-teknikerna, GA och Tillståndsmaskiner med avseende på tre olika aspekter: En mätning av hur väl individer anpassar sig till en miljö, en bedömning av hur lätt det var att implementera och anpassa algoritmerna till förändringar som skedde under projektets gång, samt en statistisk analys av effektiviteten hos respektive algoritm.

5.1 Presentation av undersökning

Figur 24 visar alla banor som användes under utvärderingen. Banorna valdes därför att de är varierande i sina utföranden. Variationen är bra för att det har frambringat ett slutresultat som hjälper till skapa en generell bild över hur bra respektive AI-teknik presterar med avseende på anpassningsbarhet. Från vänster till höger har banorna dessa namn:

- *Mostly Cold*: Banan är till stor del kall. Luftfuktigheten växlar dock, där den röda delen är torr och snön är fuktig.
- *Desert and Woods*: Detta är en varm bana. Även här växlar fuktigheten.
- *Varied*: Banan har alla fyra olika typer av vegetationsområden; Snö, tundra, skog och öken.
- *Desert*: En bana som är varm och torr.



Figur 24 Fyra testbanor som användes under utvärderingen

5.1.1 Anpassningsförmåga

Denna aspekt fokuserar på att testa hur bra respektive AI-teknik är på att styra individer i ekosystemet. Utförandet av detta test undersöker tre värden: Populationsstorlek, livslängd för individerna, och individernas fitness. Fitness i detta fall beräknas på föjande vis:

```
public float UpdateValue()
{
      float fitness = 0;
      float he = _health / 1000;
      float th = _thirst / 100;
      float hu = _hunger / 100;
      float bo = (100 +(Mathf.Min(100, _body_temp) -
                        Mathf.Max(100, _body_temp))) / 100;


      fitness = he + th + hu + bo;
      return fitness;
}
```

Fitness är ett värde på hur bra en individ mår. I detta fall är det en summa på hur törstig och hungrig individen är, hur nära individens kroppstemperatur är till den optimala temperaturen (100), samt hur mycket hälsa individen har. Alla värden är omskalade till en skala från 0 till 1 för att göra så att alla värden påverkar fitnessvärdet lika mycket. Törstighet, hunger och kroppstemperaturs kan max bli 100 och hälsan kan max bli 1000. Detta betyder att när dessa värden är omskalade till en skala från 0 – 1, kan fitness maximalt bli 4 per individ.

Det fitnessvärde som används för undersökningen är ett medelvärde av alla individer i populationen som testet utförs på. Detta för att få en generell bild av hur individerna mår i ekosystemet.

Anpassningsbarheten testades på alla banor som beskrevs tidigare. Där varje bana har testats med båda algoritmerna. På respektive bana utförs 20 tester per algoritm, vilket ger totalt 40 utförda tester per bana. Tiden på respektive bana går 6 gånger snabbare. Detta är nödvändigt för att göra alla tester inom en rimlig tid. I normal takt skulle vare test ta 60 min att utföra, dock på grund av att tiden går snabbare tar varje test 10min.

Värdena för livslängd, populationsstorlek och fitness sparas ner i en textfil som sedan används i Excel för att skapa diagram och göra beräkningar. Populationsstorleken och fitnessvärdet sparas ned till en textfil två gånger per sekund. Livslängden sparas ned till en textfil så fort en individ dör i ekossystemet.

5.1.2 Effektivitet

Fokus för denna aspekt är att testa respektive AI-tekniks effektivitet och ge en generell förståelse om hur effektiva AI-teknikerna är i relation till varandra. De värden som undersöks för GA är hur lång tid det tar att evolvera en ny lösning. För tillståndsmaskinen undersöks hur lång tid det tar att skicka vidare information mellan funktionerna i tillståndsmaskinen.

På grund av att det bara är algoritmer i respektive AI-teknik som undersöks och att respektive AI-teknik inte använder sig av funktioner som är kopplade till banorna, spelar det ingen roll vilken bana effektivitetstesten görs på. Alla tester görs i detta fall på banan *Varied*.

Context switch i ett system som stödjer samtida exekvering av processer, refererar till bytet mellan processer eller trådar på en processor. Context switch möjliggör att ett system kan exekvera flera program samtidigt. Dock skapar det oundviklig overheadkostnad (Ding, Li & Shen, 2007). Detta är något som måste hållas i åtanke när resultatet av effektivitetstesten studeras. På grund av varierande antal context switch per sekund, händer det ibland att mätvärdena i effektivitetstestet är onormalt stora. Detta är något som har tagits hänsyn till, därav har testet gjorts i 1000 iterationer där varje iteration är ett medelvärde av 6000 undansparade värden. Dessa 6000 värden är nummer på hur lång tid det tog att exempelvis skicka vidare information i tillståndsmaskinen eller evolvera en ny lösning i GA. Det medelvärdet hjälper till att minska de onormalt stora mätvärdena och ger på så vis ett bättre och mer läsbart resultat.

Värdena sparas ner på samma sätt som testet för anpassningsförmåga. För varje iteration sparas medelvärdet, av de undansparade 6000 värdena, till en textfil. Varje test använder respektive AI-teknik totalt 6 miljoner gånger.

Ett till test för att kolla antalet anrop respektive AI-teknik får per sekund har också genomförts. Detta test gjordes över 1000 iterationer, där en iteration tar en sekund. Varje iteration sparar undan hur många anrop respektive AI-teknik får. Detta värde sparas även ner till en textfil som används senare för beräkningar och diagramskapande.

Den hårdvara som används är en Dell Vostro 3550 bärbar dator. Den har en Core i5 2410M processor som är nedklockad till 700Mhz. Anledningen till att den är nedklockad är på grund av att tidtagaren i C# ej var högupplöst nog för att läsa värden när den kördes i normala hastigheten 2.5Ghz. Det hände ofta att tidtagaren gav noll-värden.

5.1.3 Flexibilitet

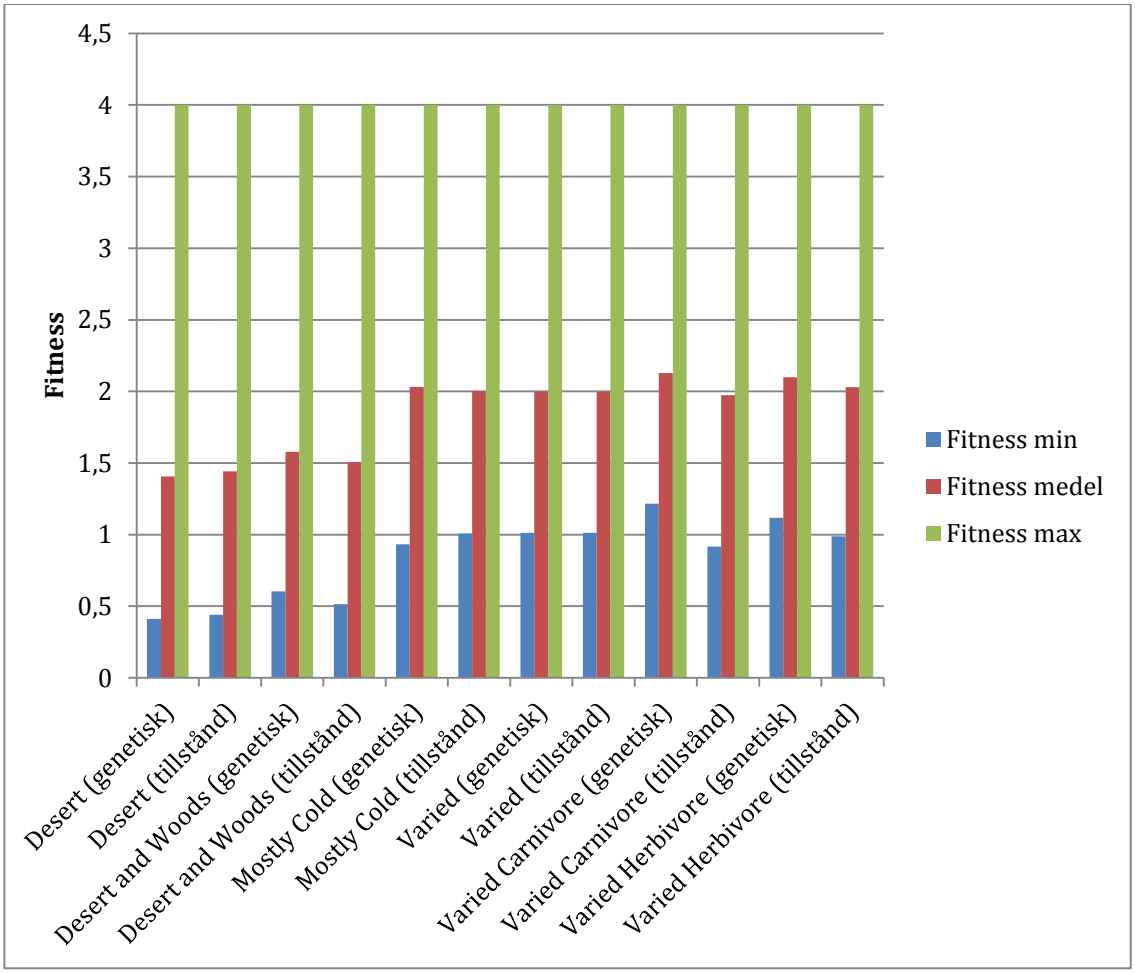
Flexibilitet syftar på att undersöka hur lätt det är att använda respektive AI-teknik. Flexibilitet kan inte kvantifieras på samma sätt som effektivitet och anpassningsbarhet på grund av att mäts på ett helt annat sätt. Flexibiliteten har istället undersökts genom att studera det praktiska arbetet som genomfördes för respektive AI-teknik. Fokus ligger på att undersöka svårighetsgraden av att implementera respektive AI-teknik. Detta genom att studera hur mycket som behöver ändras eller läggas till när AI-teknikerna byggs ut. En bildlig illustration har konstruerats för att visa på hur mycket kod som måste läggas till eller ändras för respektive AI-teknik när nya beteenden implementeras i systemet. Denna illustration har sedan studerats och resultatet av studien ger ett mått på hur flexibelt det är att använda respektive AI-teknik.

5.2 Analys

5.2.1 Anpassningsförmåga

Anpassningsförmågan hos respektive AI-teknik har testats på fyra olika banor: *Desert, Desert and Woods, Mostly Cold* och *Varied,* med avseende på tre olika aspekter: Fitness, livslängd och populationsstorlek. Banan *Varied* har testats på tre olika vis. Först testades banan med en jämn fördelning mellan köttätare och växtätare. I andra testet minskades antalet växtätare och antalet köttätare ökades. Det tredje testet minskades antalet köttätare och antalet växtätare ökades. Syftet var att visa på en viss generalitet för anpassningsbarheten för respektive AI-teknik. Genom att statistiskt observera dessa tre aspekter för anpassningsförmågan och jämföra dem mellan de båda AI-teknikerna uppnås ett mått för hur bra de bägge AI-teknikerna är på att styra individerna i ekosystemet.

I Figur 25 nedan illustreras medelvärdet för fitness av alla individer. Testet har genomförts med båda AI-teknikerna 20 gånger per bana. Notera att både tillståndsmaskinen och GA presterar nästan lika bra. Medelvärdet för GA är något högre på de flesta banorna än tillståndsmaskinen. Anpassningsförmågan hos GA beror till stor del på hur väl den lyckas evolvera serie av actions som är givande för individen. Testet visar på att GA lyckas evolvera serier av action som är lika givande som tillståndmaskinen. Just detta test står sig båda algoritmerna lika till varandra.

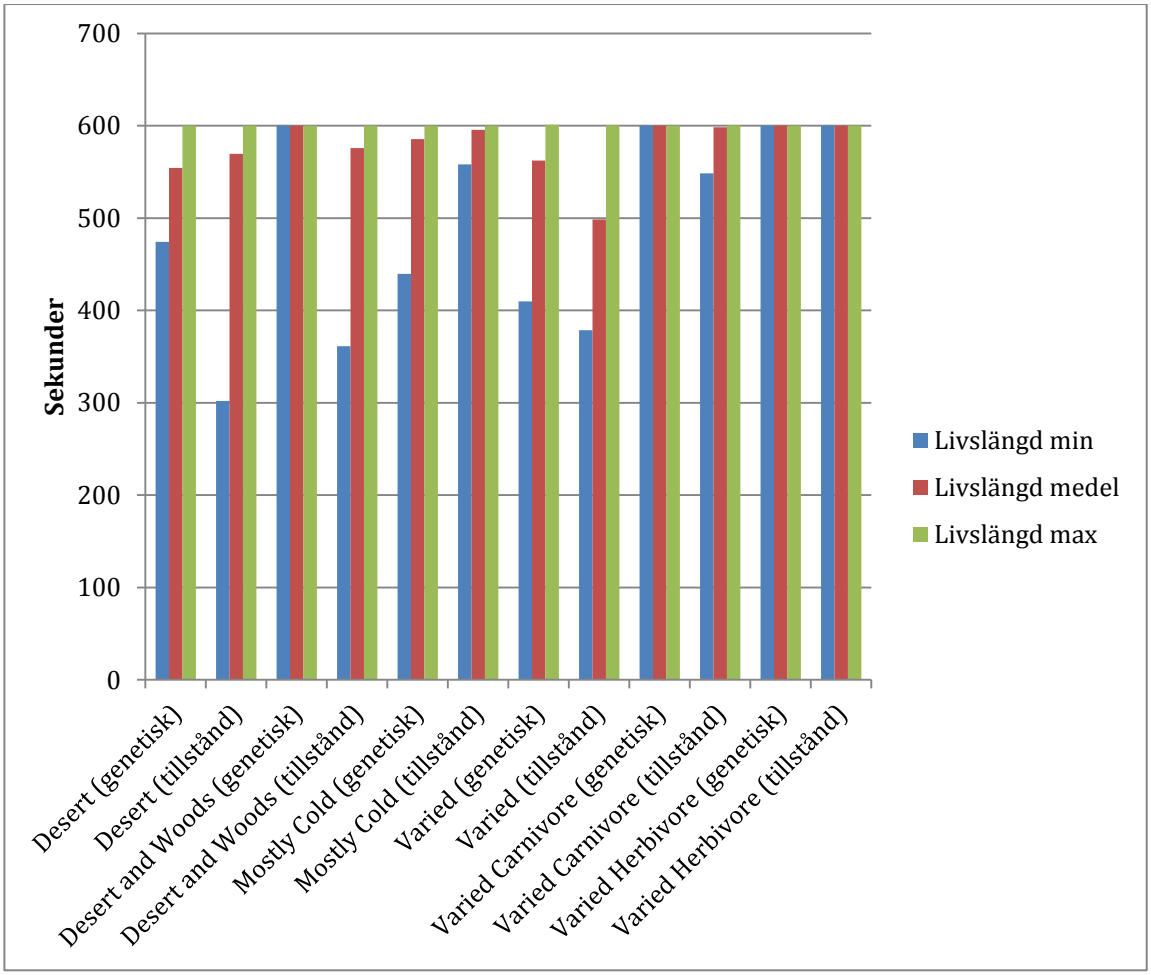


Figur 25 Medelvärde av fitnessvärden för respektive AI-teknik per bana

Testet ovan, för fitnessvärdena, berättar bara hur bra individerna mår. Testet visar inte hur många eller hur länge individerna överlever under testet. De två nästkommande tester har tagit hänsyn till detta.

Det andra testet nedan kollade på livslängden. Detta test syftade till att ge en mer vidgad vy av AI-teknikernas anpassningsförmåga. Testet fokuserar på att jämföra min-, medel- och max-livslängden för individerna i ekosystemet. Detta test har genomförts på alla banor som beskrevs tidigare, där varje bana har testats med båda AI-teknikerna.

Figur 26 illustrerar testets resultat. De banorna som har tre lika långa staplar och där staplarna står på 600 sekunder är test där alla individer har överlevt hela testet. Detta kan noteras på banan *Varied Herbivore, Varied Carnivore* och *Desert and Woods*. Här kan en skillnad noteras mellan tillståndsmaskinen och GA. Det syns tydligt på många banor att GA lyckas hålla individer vid liv längre än tillståndsmaskinen. Den enda banan som tillståndsmaskinen presterar bättre än GA är *Mostly Cold*. På banan *VariedHerbivore* presterar båda AI-teknikerna lika bra med avseende på livslängd hos individer. På banan *Desert* har tillståndsmaskinen högre medelvärde än GA. Dock har GA högre min-värde. Detta beror på att individer överlever till stor del längre hos tillståndsmaskinen än den genetiska värden, dock har en individ lyckats dö mycket tidigare när tillståndsmaskinen användes. Överlag presterar GA bättre än tillståndsmaskinen. Både min- och medel-värdet är högre på merparten av banorna för GA.

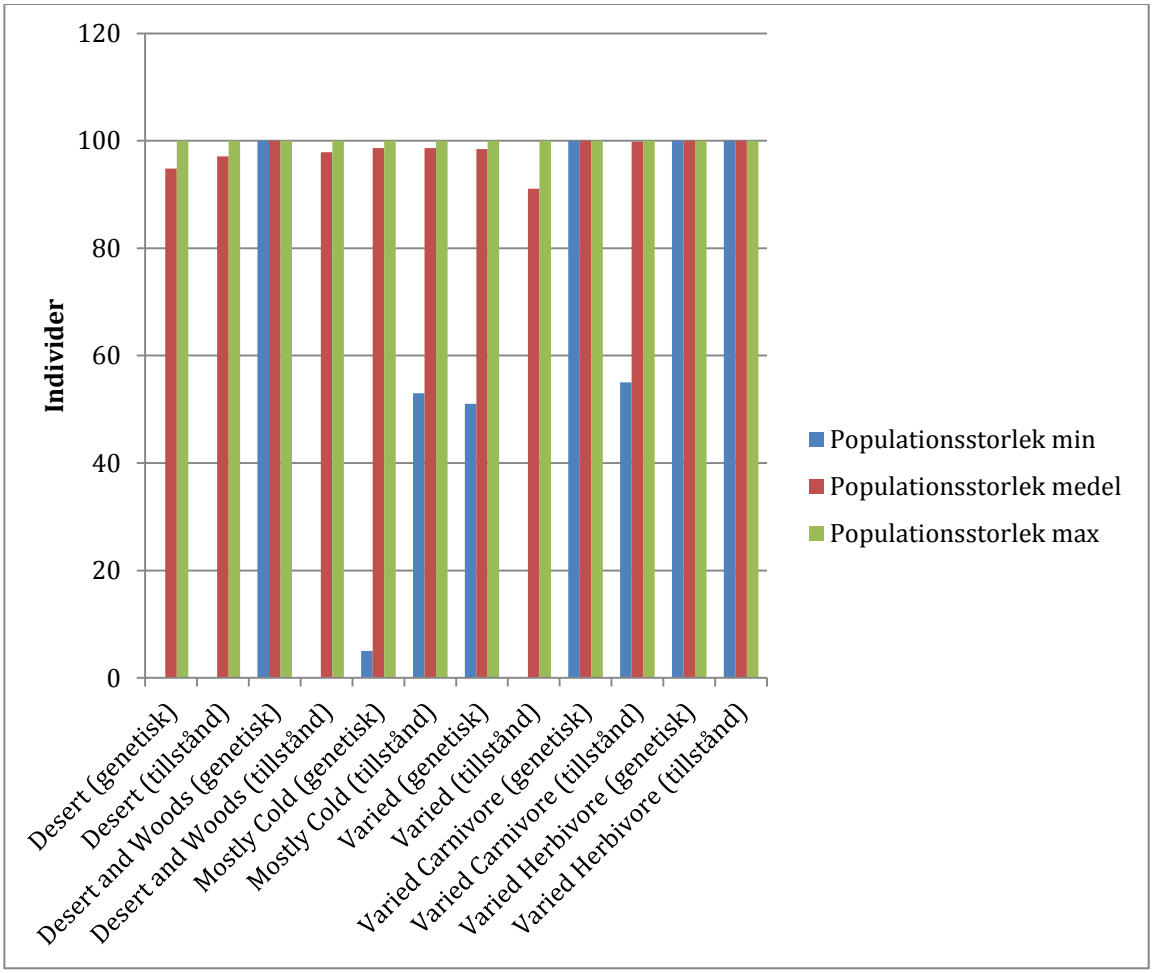


Figur 26 Livslängd för individer, värden utan kolumn betyder att individer överlevde hela testet

Det nästkommande test fokuserade på att jämföra min-, medel- och max-population för individerna i ekosystemet. Som föregående test har detta också genomförts på alla banor som beskrevs tidigare.

Figur 27 illustrerar testets resultat. Detta test är nära kopplat till föregående test som kollade på livslängden hos individerna. Notera att även här förekommer det tre fulla staplar på banorna, *Varied Herbivore, Varied Carnivore* och *Desert and Woods*. På förra testet noterades det att dessa banor bara innehöll individer som överlevde hela testet. Detta test gjordes samtidigt som föregående test. Vilket betyder att hela populationen är vid liv när testet är avslutat. Därav finns även här tre fulla staplar på respektive testbana.

Överlag är resultatet liknande som på det föregående testet. GA lyckas hålla fler individer vid liv merparten av banorna jämfört med tillståndsmaskinen. Två av sex banor presterar tillståndsmaskinen lite bättre än tillståndsmaskinen, på banan *Varied Herbivore* presterar båda AI-teknikerna lika, resten av banorna presterar GA bättre.

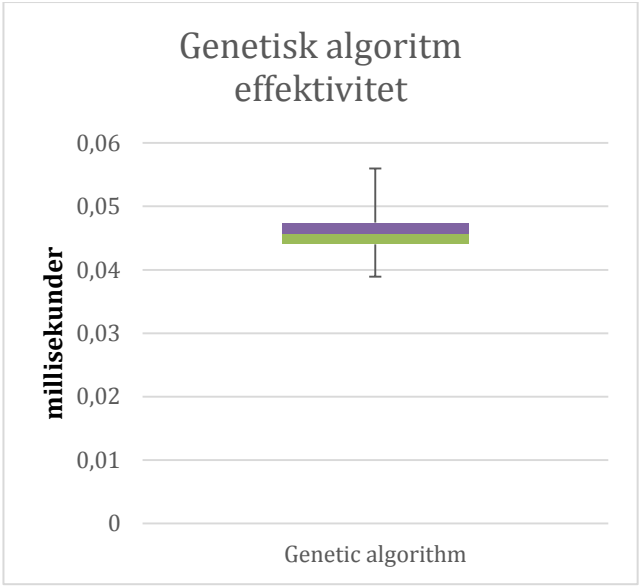


Figur 27 Populationsstorlek för respektive testbana

5.2.2 Effektivitet

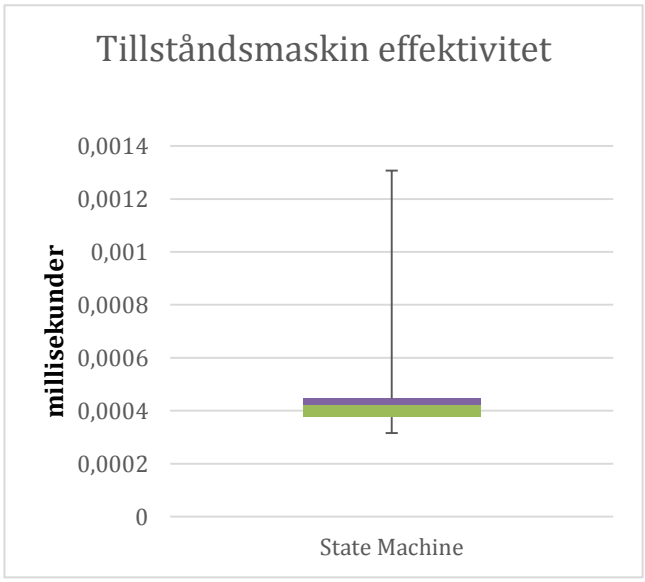
Fokus ligger i denna delundersökning på att beräkna en relativ effektivitet mellan tillståndsmaskinen och GA för att få en förståelse av skillnaderna mellan dess effektivitet. Fokus kommer också ligga på att mäta hur snabbt respektive teknik är på att exekveras. Effektiviteten med avseende på processortid kommer mätas i millisekunder. Snabbheten redovisas sedan med hjälp av lådagram. Delundersökningen kollar också på hur många anrop respektive AI-teknik får samt hur mycket processortid detta motsvarar när antalet anrop läggs ihop med processortiden AI-teknikerna tar under ett anrop.

Figur 28 illustrerar hur mycket processortid GA tar när en ny lösning evolveras. Lådagrammets lila och gröna områden innehåller de processortider som oftast förekommer när en ny lösning evolveras. I det här fallet tar det mellan 0,04 och 0,05 millisekunder att evolvera en ny lösning. I detta program evolveras en lösning alltid 10 iterationer vilket betyder att när en individ vill ha en ny lista av actions kommer det ta 0,4 – 0,5 millisekunder för att få en ny lista. De två pinnarna innehåller de mindre signifikanta värdena som ibland uppkommer. Slutet på dessa två pinnarna markerar minimumvärde och maximumvärde för tiden det tar att evolvera en ny lösning.



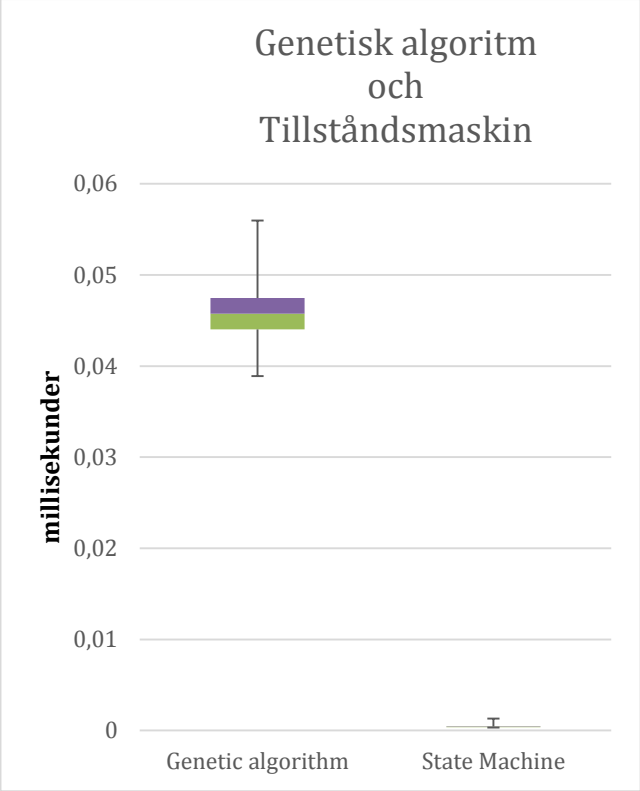
Figur 28 Effektivitetstest för den genetiska algoritmen för att se hur lång tid det tar att evolvera en ny population av actions

Figur 29 illustrerar hur mycket processortid tillståndsmaskinen tar när den skickar vidare information och ändrar tillstånd i tillståndsmaskinen. Eftersom det bara är information som skickas vidare mellan funktioner i tillståndsmaskinen används markant mindre processortid än GA.



Figur 29 Effektivitetstest för tillståndsmaskinen för att se hur lång tid det tar att skicka info mellan dess funktioner.

Figur 30 här nedanför illustrerar skillnaden mellan de två AI-Teknikerna med avseende på tiden det tar att exekvera respektive algoritm. Notera att GA tar längre tid att exekvera än tillståndsmaskinen.



Figur 30 En jämförelse mellan de två AI-Teknikerna.

För att få ett nummer på hur stor skillnaden är mellan algoritmerna jämfördes medianvärdet av båda algoritmerna med varandra. Tabell 3 här nedanför visar på att GA tar 0,045 millisekunder längre att exekvera än tillståndsmaskinen. Detta betyder att GA är ungefär 108 gånger långsammare än tillståndsmaskinen.

Tabell 3 Ratio och differens mellan respektive AI-teknik

Genetisk algoritm (medelvärde)	**Tillståndsmaskin (medelvärde)**	**Ratio**	**Differens**
0,046 ms	0,00042 ms	108 ggr	0,045 ms

Tabell 4 visar antalet anrop respektive AI-teknik tar per sekund, det tabellen illustrerar en tydlig skillnad mellan teknikerna. Tillståndsmaskinen anropas fler gånger än GA. Resultatet förenades sedan med medelvärdet av exekveringstiden för respektive AI-teknik. Notera att skillnaden mellan respektive AI-teknik minskar markant. GA är fortfarande långsammare än tillståndsmaskinen, dock på grund av att GA anropas färre gånger än tillståndsmaskinen är skillnaden mycket mindre. Tabell 5 visar resultatet av det kombinerade värdet från Tabell 3 och Tabell 4. Resultatet visar på att GA är 3 gånger långsammare än tillståndsmaskinen när hänsyn har tagits till hur många anrop respektive AI-teknik får per sekund.

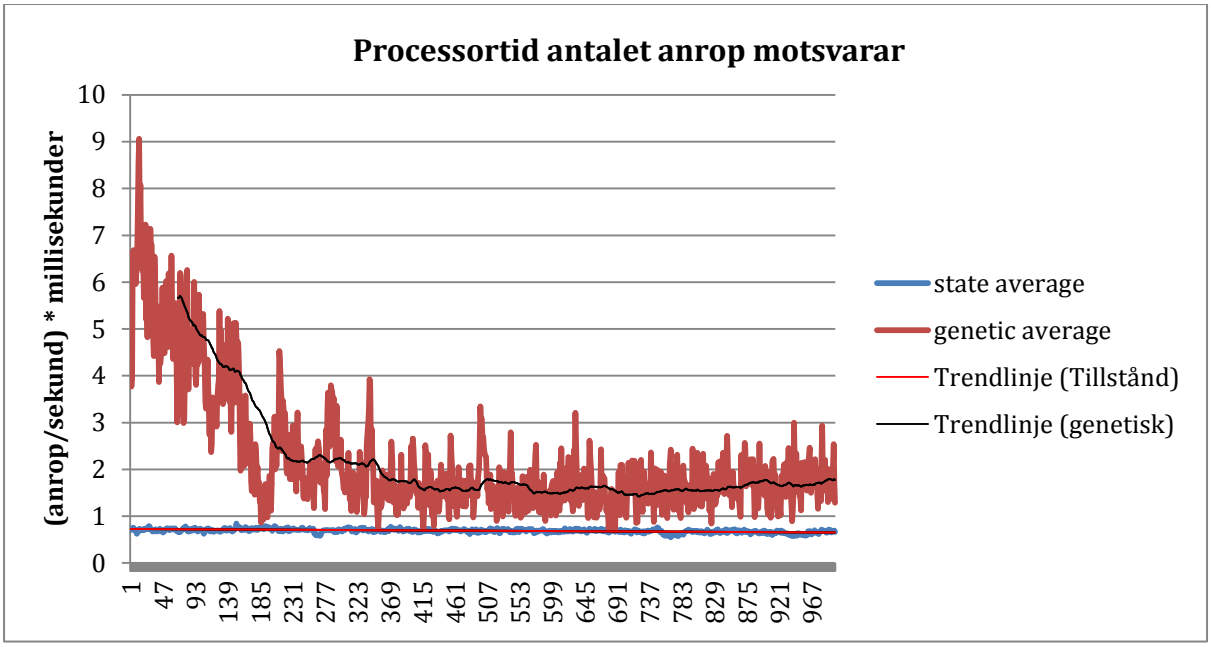
Tabell 4 Antalet anrop per sekund respektive AI-teknik tar

AI-Teknik	Genetisk Algoritm	Tillståndsmaskin
Medel anrop / sekund	48 anrop / sek	1626 anrop / sek

Tabell 5 Ration och differens mellan respektive AI-teknik med avseende på anrop per sekund och exekveringstid i millisekunder

Genetisk algoritm (anrop / sekund) * millisekunder	Tillståndsmaskin (anrop / sekund) * millisekunder	Ratio	Differens
2,21 (anr/sek)*ms	0,73 (anr/sek)*ms	3 ggr	1,48 (anr/sek)*ms

Figur 31 är en illustration hur effektiviteten mellan respektive AI-teknik står sig till varandra. Detta är ett test på 1000 iterationer där varje iteration sparar undan antalet anrop AI-teknikerna får per sekund. Antalet anrop har också multiplicerats med medelvärdet av exekveringstiden för respektive AI-teknik. Detta för att illustrera hur mycket processortid båda AI-teknikerna tar per sekund. Notera att GA i början tar mycket processortid. Detta beror på att alla individer i ekosystemet vill samtidigt hämta en ny lista från GA. Dock avtar detta värde efter ett antal iterationer, GA närmar sig hela tiden tillståndsmaskinens processortid. På grund av GA är beroende av individernas varierande variabler när en ny lista med actions skall evolveras, får individer olika listor med actions. Detta gör att vissa individer blir snabbare klar med sina listor än andra, vilket gör att individerna hämtar nya listor med actions oregelbundet från GA. På så vis sprids anropen till GA ut och antalet anrop minskar till en viss nivå som illustreras i Figur 31.



Figur 31 Diagram över effektivitet med avseende på hur många millisekunder av en sekund respektive AI-teknik tar

5.2.3 Flexibilitet

Det Figur 32 illustrerar är hur krävande respektive algoritm är att bygga ut. Till vänster illustreras en bild av tillståndsmaskinens kod, till höger illustreras GA. När ett beteende läggs till för individerna måste kod läggas till i respektive algoritm. Det bilderna illustrerar är vad som behövs lägga till eller ändras i båda algoritmerna. Notera skillnaden på antalet rader som behöver adderas när ett nytt beteende implementeras. Betydligt mycket mer kod behöver läggas till i tillståndsmaskinen än GA. I GA behövs bara en variabel ändras samt ett case-block läggas till. I tillståndsmaskinen måste alla ”handle-funktioner” ändras i för att hantera det nya beteendet samt en helt ny handle-funktion måste läggas till. Förutom detta måste även nya regler läggas till som hanterar bytet mellan det tillståndet som skall representera beteendet. Det är en stor fördel att arbeta med GA när nya beteenden läggs till, arbetet är mycket lättare än tillståndsmaskinen.

```
public int _NUM_ACTIONS = 7;



void handle_todo()
{
        if(_todo_index == _todo.Length - 1)
        {
                _todo_index = 0;
                _new_list = true;
                return;
        }

        switch(_todo[_todo_index])
        {
        case 0:
                _status = _behavior.find_warmer_place(_creature);
                break;
        case 1:
                _status = _behavior.find_cooler_place(_creature);
                break;
        case 2:
                if(_creature.carnivore)
                        _status = _behavior.find_meat(_creature);
                else
                        _status = _behavior.find_food(_creature);
                break;
        case 3:
                _status = _behavior.find_water(_creature);
                break;
        case 4:
                _status = _behavior.walk_randomly(_creature);
                break;
        case 5:
                _status = _behavior.follow_leader(_creature);
                break;
        case 6:
                _status = _behavior.idle(_creature);
                break;
        default:
                _status = _behavior.idle(_creature);
                break;
        }

        if(_status != CreatureBehavior.Status.BUSY)
                _todo_index++;
}
```

Figur 32 Förändring i respektive algoritm, tillståndsmaskin till vänster, genetiska algoritmen till höger.

5.3 Slutsatser

Effektivitetstestet fokuserade på att jämföra vilken av algoritmerna som exekverar snabbast. I bakgrundskapitlet togs en hypotes fram som sade att en tillståndsmaskin skulle vara effektivare än en GA med samma grundbeteende. Detta grundar sig i resultat som har tolkats utifrån Beyer, Schwefel och Wegener *(2002)* arbete om *How to Analyze Evolutionary Algorithms* samt Rudolph, 1994 arbete *Convergence Analysis of Canonical Genetic Algorithms*. Hypotesen stämmer helt med resultatet som testet gav. På både testet där AI-teknikerna exekveringslängd för ett anrop till respektive AI-teknik, samt den hur mycket processortid respektive AI-teknik tog på av en sekund, är resultatet att tillståndsmaskinen är effektivare. Om AI-teknikerna bara undersöks med avseende på hur lång tid respektive AI-teknik tar att exekvera en gång, är det stora skillnader mellan båda teknikerna. Dock om hänsyn tas till hur många anrop respektive AI-teknik får per sekund är skillnaderna mycket mindre.

Testet för anpassningsbarhet fokuserade på att jämföra hur väl respektive AI-teknik klarar av att styra beteendet hos individer i ekosystemet. Testet kollade på tre olika aspekter: fitness, livslängd och populationsstorlek. Testen gjordes på fyra olika banor, där en av

banorna varierande i antalet köttätare och växtätare. Syftet med variationen var att ge en generell uppfattning av hur väl AI-teknikerna presterade. Det första testet som innefattade en undersökning av fitnessvärden visade på att båda AI-teknikerna presterar jämbördigt. Det andra testet syftade på att undersöka hur lång tid respektive AI-teknik lyckas hålla individer vid liv. Här visades en skillnad mellan teknikerna. GA presterade generellt bättre på flera av banorna, där några banor var prestandan bättre hos tillståndsmaskinen. De var även likbördiga på en av banorna. Det tredje och sista testet visar hur många individer som överlever per test. Detta test visade liknande resultat som resultatet för livslängstestet.

Algoritmernas flexibilitet jämfördes genom att studera hur lätt de var att använda. Fokus låg på att jämföra hur mycket som behövde läggas till eller ändras i respektive algoritm när nya beteenden implementerades i systemet. När båda algoritmerna var korrekt implementerade och justerade, visade resultatet på att GA är mycket flexiblare att arbeta med än tillståndsmaskinen. När antalet beteenden ökar måste tillståndsmaskinen ta hänsyn till detta genom att lägga nya tillstånd som skall representera beteenden och för varje tillstånd måste det också finnas en funktion som hanterar detta tillstånd. Detta göra att tillståndsmaskinen flexibilitet minskar när antalet tillstånd ökar. GA behöver inte ta hänsyn till nya tillstånd på samma vis. Eftersom GA bara evolverar en serie av actions som representeras av heltal, behöver GA bara ta hänsyn till att ett nytt heltal kan finnas med i serien. Regelsystemet som tolkar utdatan från GA måste också ta hänsyn till att ett nytt heltal finns. För att anpassa regelsystemet till ett nytt beteende behövs bara ett case-block läggas till i switch-case-satsen som tolkar heltalen. I GA behövs alltid en variabel ändras samt ett nytt case-block läggas till när ett beteende adderas till systemet. Detta betyder flexibiliteten för GA alltid är konstant och ändras inte när nya beteenden läggs till.

Mätningarna talar för att GA är mest lämpad för att styra beteenden i individcentrerade ekosystem i spel. GA generellt mer robust och flexiblare i den mån att algoritmen är enklare att anpassa till förändringar i systemet som när nya beteenden läggs till vilket illustrerades i Figur 32. GA presterar även bättre på merparten av testen för anpassningsbarheten. Tillståndsmaskinen var effektivare, dock när hänsyn togs till antalet anrop respektive algoritm fick per sekund, var skillnaden att GA var 3 gånger långsammare än tillståndsmaskinen.

6 Avslutande diskussion

6.1 Sammanfattning

Målet med detta verk var att undersöka vilken AI-teknik som är mest lämpad att använda för styrning av beteende hos individer i ett artificiellt ekosystem. Två typer av AI-tekniker har implementerats, en deterministisk och en ickedeterministisk AI-teknik. Den deterministiska AI-tekniken är en tillståndsmaskin, den ickedeterministiska AI-tekniken är en GA. Dessa AI-tekniker har jämförts med avseende på tre olika aspekter: effektivitet, anpassningsbarhet och flexibilitet.

För att kunna genomföra tester har individer och en experimentmiljö implementerats. Testerna gjordes på fyra olika banor som genererades av en slumptalsgenerator där seeds har använts för att kunna genomför flera tester på samma bana. Applikationen som utvecklades har skapats på så vis att det enkelt skulle gå att byta vilken AI-teknik som styr beteendet hos individer.

Den frågeställningen som togs fram var: Finns det en fördel att använda genetiska algoritmer kontra tillståndsmaskiner för att styra beteende i individcentrerade ekosystem i spel med avseende på vilken algoritm som skapar bäst förutsättningar för individers överlevnad, beräkningsmässig effektivitet och flexibilitet. Resultatet från undersökningen talar för att GA är mer fördelaktig att använda än tillståndsmaskiner. Mätningarna visade på att GA är generellt mer robust och flexiblare i den mån att algoritmen är enklare att anpassa till förändringar i systemet, som när nya beteenden läggs till, vilket illustrerades i Figur 32. GA presterar även bättre på merparten av testen för anpassningsbarheten. Tillståndsmaskinen var effektivare, dock när hänsyn togs till antalet anrop respektive algoritm fick per sekund, visade det sig att GA anropades betydligt färre gånger. När antalet anrop och processortid kombinerades till ett gemensamt resultat, var GA generellt 3 gånger långsammare än tillståndsmaskinen vilket inte är mycket när hänsyn tas till fördelarna algoritmen ger med avseende på anpassningsbarhet och flexibilitet.

6.2 Diskussion

Som kapitel 2.4.6 tar upp finns det potential för GA. Johnson och Wiles (2001) nämnde i sin artikel om datorspel med intelligens att dessa tekniker oftast tar mycket processorkraft och är för långsamma för att producera ett användbart resultat. Detta stämmer dock inte helt om hänsyn tas till resultatet av denna undersökning. Resultatet visade på att när antalet anrop och processortid kombinerades till ett gemensamt resultat, var GA generellt 3 gånger långsammare än tillståndsmaskinen. Detta visar på att om GA implementeras och anpassas till korrekt till det ämnade problemet kan en GA användas utan problem.

Myers (2007) nämnde exempelvis i sin artikel om hur en GA kan användas för att skapa olika strategier i ett kortspel, att det finns potential för GA. Meyers nämnde också att tidskomplexiteten, den tid det tar för att beräkna en lösning med GA, är för stor i stora mer komplexa spel. Ekosystemsspelet i sig är komplicerat, dock är lösningarna som GA evolverar fram enkla i den mån att lösningarna bara består av en serie heltal. Några designval gällande optimering har även tagits, exempelvis slopades en binärrepresentation på grund av att de bara skulle ge ett onödigt avkodningssteg av en evolverad lösning, samt selektionsmetoden tornamentselektion valdes istället för rouletthjulsselektion på grund av

tornamentselektionens positiva egenskaper gällande effektivitet. Detta gör att GA går att köra i realtid och användas i individcentrerade ekosystems i spel.

Trovärdigheten hos resultatet går att diskutera. Det går säkerligen att få bättre resultat genom att köra testerna flera gånger öka antalet testbanor och variera populationsstorleken. Större populationsstorlek hade inneburit att systemet skulle gå långsammare på grund av att respektive algoritm skulle få ett ökat antal anrop till sig. Hänsyn måste dock tas till att detta projekt utfördes under en kort tidsperiod där mycket av arbetet gick till det teoretiska som metod och litteraturstudie samt en utvärdering av resultatet. Därav fanns inte tid över att utföra mer utförliga tester.

Det finns också en möjlighet att hitta bättre funktioner till GA som skapar bättre förutsättningar för individer så som att anpassningsförmågan blir bättre eller effektiviteten ökar. Om tid hade funnits skulle flera olika evolveringssessioner kunnat utföras, där olika funktioner inom GA testats och på så vis ta fram vilken fitness-, selektions- och mutations-metod som skulle passa bäst för problemet.

När respektive algoritm kan tänkas användas kan skilja beroende på projektets utformning. Om ett projekt syftar till att få ut en produkt så snabbt som möjligt och som kräver ett förutsägbart resultat och ej kräver en avancerad AI passar en tillståndsmaskin bra, då implementationstiden inte är lång och utvecklaren god kontroll över det slutliga beteendet. Dock om projektet kräver en produkt med en AI som är mer avancerad och om produkten är ett ekosystem som skall finnas i ett spel samt att utvecklaren i ett senare tillfälle kommer bygga ut produkten vilket gör att vissa delar av AI måste ändras, skulle en GA passa bättre, då GA är mer robust och flexibel samt ger bättre förutsättningar till att individen i världen utför rätt handlingar och överlever längst.

De samhälleliga aspekterna detta arbete kan besvara är få, dock kan arbetet som gjordes med GA användas som ett exempel på hur beteendet hos individer i artificiella ekosystem kan styras. Arbetet kan exempelvis användas i senare spelutvecklingssammanhang. Resultatet av undersökningen kan användas som grund till vidare forskning. Resultatet kan exempelvis lägga grunden till en GA som har uppgiften att stabilisera rymdraketer under uppskjut och landning. GA skulle i detta fall kunna styra ett antal stabiliseringsraketer som är monterade på rymdraketen. Utifrån ett antal orienteringsvärden, som berättar position och vinkel rymdraketen har, skulle GA kunna evolvera lösningar i form av serier handlingar. Där en handling kan exempelvis vara att starta eller stänga av en stabiliseringsraket. En GA skulle även kunna användas till att planera vägar för självstyrande bilar, samt hantera olika hinder och situationer som kan uppstå i olika sorters vägnät. I detta fall skulle en handling kunna vara att GA måste få bilen att bromsa eller svänga med en viss kraft. GA skulle även kunna optimera den självstyrande bilens avgasutsläpp genom att evolvera fram en lösning för en viss situation eller hinder som ger minst påverkan på miljön.

Programmet, som användes för undersökningen, implementerades med spelmotorn Unit3D (Unity Technologies, 2014). Detta sätt att genomföra en studie på är bra rent forskningsetiskt därför att Unity3D är en populär spelmotor som uppdateras frekvent och är även skapad för att kunna köras på flera plattformar, samt källkoden, mätresultaten och metoden för att ta fram resultaten är öppen för allmänheten. Detta är nog med publicerad information för att kunna återupprepa resultatet. Eftersom Unity3D uppdateras frekvent, samt att stöd för nya plattformar hela tiden läggs till, försäkrar detta en återupprepbarhet i framtiden.

6.3 Framtida arbete

Ett framtida arbete för detta verk kan göras i en mängd olika sätt. Som nämndes ovan kan skillnaden i resultatet skapas genom att undersöka algoritmerna mer och göra en grundlig utvärdering på vilka funktioner för GA som passar bäst till detta problem för detta projektarbete. Där ändamålet i detta fall är att styra individer i ekosystemet.

En annan sak som detta verk inte behandlar är hur naturliga individerna beter sig i miljön. För att utföra detta skulle ett antal andra artificiella ekosystem som anses innehålla individer som beter sig naturligt, analyseras. Utifrån detta skulle ett antal frågor definieras som tar hänsyn till individers naturliga beteenden. De definierade frågorna skulle sedan ges till försökspersoner, dessa skulle då svara på frågorna och ge motiveringar till svaren på frågorna. Resultatet kan sedan kvantifieras och analysers och en slutsats om hur naturliga individerna beter sig kan sedan fastställas. Detta är dock något som är ämnat åt framtida forskning. Testet skulle ytterligare kunna byggas ut till att hantera interaktion med världen från spelare som spelar ekosystemsspelet och utifrån interaktionen skulle spelarna kunna säga hur naturligt individerna reagerar i ekosystemet.

Något som också skulle kunna göras är att utvidga undersökningen ytterligare genom att ändra omfattningen på studien. Detta betyder att tester kan skapas för exempelvis försök där fokus ligger på att studera vilka förändringar som måste göras för att en hel population skall garanterat överleva ett test.

Det existerande ekosystemet skulle också kunna överföras till andra spel och på så vis genomföra ytterligare i form av annorlunda miljöer. Generellt sett skulle även GA eller tillståndsmaskinen användas i andra spelsammanhang, där fokus ligger på att anpassa nuvarande algoritmer till ett annat problem och utifrån detta få ett bredare perspektiv över prestandan av båda algoritmerna av anpassningsbarhet, effektivitet och flexibilitet. Något som inte har undersökts i detta verk är hur flexibelt det är att anpassa respektive algoritm till ett annan projekt i form av individer som har andra mål än att bara överleva i en fördefinierad miljö.

Som nämndes i diskussionen skulle resultatet från detta arbete kunna användas som grund till vidare forskning, så som rymdraketsforskning eller forskning för att ta fram självstyrda bilar. Ett exempel hur en GA skulle användas i rymdraketssammanhang, är styrning av stabiliseringsraketer vid uppskjut och landning av rymdraketer. Där GA evolverar fram lösningar i form av instruktioner som skall utföras. Där en instruktion startar eller stänger av en viss stabiliseringsraket. Ett annat exempel var att GA kan användas i självstyrande bilar för vägplanering, samt göra så att självstyrande bilar kan hantera hinder och speciella situationer. GA skulle även kunna användas i form av optimering för att bilarna skall ha så liten miljöpåverkan som möjligt.

Det vore även intressant om andra spelmotorer använder sig av arbetets resultat för att i framtiden skapa en generell GA som enkelt kan anpassas till olika områden innom spel som spelmotorn är anpassad för. Detta skulle exempelvis underlätta arbetet av att skapa en AI.

Referenser

A* search algorithm (2014) *Wikipedia: The Free Encyclopedia*. Tillgänglig på Internet: https://en.wikipedia.org/wiki/A*_search_algorithm. [Hämtad April 2, 2014].

Agarwal, S., Deb, K., Meyarvan, T. & Pratap, A. (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation,* 6(2). s. 182 – 197.

Andrews, P.S., Polack, F., Sampson, A.T., Timmis, J. & Coles, M. (2008) Simulating Biology: Towards Understanding What The Simulation Shows. I *Cosmos 2008: Complex Systems Modelling and Simulation, Proceedings of the 2008 Workshop on Complex Systems Modelling and Simulation*. York, UK, Luniver Press. s. 93 – 120.

Android (mobilt operativsystem) (2014) *Wikipedia: The Free Encyclopedia*. Tillgänglig på Internet: https://en.wikipedia.org/wiki/Android_(operating_system). [Hämtad Februari 15, 2014].

Barros, C.R., Basgalupp, M.P., Carvalho, A.C.P.L.F., Freitas, A.A. (2013) Automatic Design of Decision-Tree Algorithms with Evolutionary Algorithms. *Journal of Evolutionary Computation Winter 2013,* 21(4). USA, MIT Press. s. 659 - 684.

Beyer, H., Schwefel, H. & Wegener, I. (2002) How to Analyze Evolutionary Algorithms. *Journal of Theoretical Computer Science*. Elsevier. s. 101 – 130.

Bourg, D.M. & Seemann, G. (2004) *AI for Game Developers*. Sebastopol, California: O'Reilly Media.

Buckland, M. (2002) *AI Techniques For Game Programming*. USA, Premier Press. s. 90 - 230.

Buckland, M. (2004) *Programming Game AI by Example*. USA, Worldware Publishing Inc.

Bäck, T. (1996) *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford.

Carbonell, J.G., Michalski, R.S. & Mitchell, T.M. (1983) An Overview Of Machine Learning. I Carbonell, J., Michalski R.S. & Mitchell, T. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. TIOGA Publishing CO., Palo Alto, California.

Chaudhry, I.A, Khan, A.M. & Khan, A.A. (1999) A Genetic Algorithm for Flexible Job Shop Scheduling. WCE 2013, *Proceedings of 1999 IEEE International conference on Robotics and Automation,* 2(1). Detroit, MI. s. 1120 – 1125.

Clarke, G.L. (1954) *Elements Of Ecology*. United States of America, John Wiley & Sons.

Cliff, D. & Grand, S. (1999) The Creatures Global Digital Ecosystem. *Artificial Life 5. Proceedings of the fifth international workshop on the synthesis and simulation af living systems*. Massachusetts Institute of Technology. s. 77 – 94.

Cohen, J. (1979) Non-Deterministic Algorithms. *Journal of ACM Computing Surveys*, 11(2). s. 79 – 94.

Creature Labs (1996) *Creatures* [Datorprogram]. Creature Labs.

Ding, C., Li, C. & Shen, K. (2007) *Quantifying The Cost of Context Switch*. San Diego, CA.

Elias, R., Swenson, W. & Wilson, D.S. (2000) Artificial Ecosystem Selection. *Proceedings of the National Academy of Sciences of the United States of America,* 97(16). University of Massachusetts, Amherst, MA. s. 9110-9114.

Eiben, A.E., Marchiori, E. & Valkó, V.A. (2004) *Evolutionary Algorithms with On-the-Fly Population Size Adjustment*. Berlin, Springer-Verlag. s. 41 – 50.

Gamma, E. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Canada, Addison-Wesley. s. 305 – 314.

Gill, A. (1960) Comparison of finite-state models. *IEEE Transactions on Circuts and Systems I*. Berkeley, California.

Glymour, B. (2001) Selection, Indeterminism, and Evolutionary Theory. *Philosophy of Science,* 68(4). The University of Chicago Press. s. 518 – 535.

Goldberg, D.E. & Holland, J.H. (1988) Genetic Algorithms and Machine Learning. *Machine Learning 3*. Netherlands, Kluwer Academic Publishers. s. 95 – 99.

Holling, C.S. (1987) Simplifying The Complex: The Paradigms of Ecological Function and Structure. 30(2), *European Journal of Operational Research*. s. 139–146.

Holland, J. (1992) *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press.

Johnson, A.R., Wiens, J.A., Milne, B.T & Crist, T.O. (1992) Animal Movements and Population Dynamics in Heterogeneous Landscapes. *Landscape Ecology*, 7(1). Colorado State University, Academic Publishing. s. 63 - 75.

Johnson, D. & Wiles, J. (2001) *Computer Games With Intelligence*. Australia, The University of Queensland.

Jones, G. (2002) Genetic and Evolutionary Algorithms. *Encyclopedia of Computational Chemistry*. John Wiley and Sons, Ltd. USA. Tillgänglig på Internet: http://www.wiley.com/legacy/wileychi/ecc/samples/sample10.pdf [Hämtad: Mars 17, 2014]

Lamothe, A. (1999) *Tricks of the Windows Game Programming Gurus: Fundamentals of 2D and 3D Game Programming*. United States of America, HW Sams. s. 729 – 730.

Langton, C.G. (1995) *Artificial Life: An Overview*. MIT Press.

Lee, D. & Yannakakis, M. (1996) Priciples and Methods of Testing Finite State Machines: A survey. *Proceedings of the IEEE*, 84(8). AT&T Bell Labs. USA. s. 1090 – 1123.

Louis, S.J. & Miles, C. (2005) Playing to Learn: Case-Injected Genetic Algorithms for Learning to Play Computer Games. *IEEE Transactions on Evolutionary Computation*, 9(6). s. 669 - 681.

Maity, S.P. & Nandi P.K. (2007) Genetic Algorithm for Improvement in Detection of Hidden Data in Digital Images. *Proceedings of the Sixth International Conference on Advances in Pattern Recognition*. Singapore. World Scientific, Hackensack, New Jersey.

Maxis Software Inc. (1992) *SimLife* [Datorprogram]. Maxis Software Inc.

Mealy, G.H. (1955) A Method for Synthesizing Sequential Circuits. *Bell Systems Technical Journal*, 34. s. 1045 – 1079.

Mitchell, M. & Taylor, C.E. (1999) *Evolutionary Computation: An Overview*. Annual Review of Ecology and Systematics.

Moore, E.F. (1956) Gedanken-Experiments on Sequential Machines. *Automata Studies: Annuals of Mathematics Studies*. Princeton University Press. USA.

Myers, N. (2007) *Game Strategy Learning using Genetic Algorithms*.

Namco (1980) *Pacman* [Datorprogram]. Midway Games.

Ponsen, M. (2004) *Improving Adaptive Game AI with Evolutionary Learning*. Delft University of Technology.

Reeves, C.R. & Rowe, J.E. (2003) *Genetic Algorithms: Principles and Perspectives A Guide to GA Theory*. Dordrecht, Kluwer Academic Publishers.

Rollings, A. & Adams, E. (2004) *Andrew Rollings and Ernst Adams on Game Design*. Indianapolis, New Riders.

Rudolph, G. (1994) Convergence Analysis of Canonical Genetic Algorithms. *IEEE Transactions on Neural Networks*, 5(1). Germany. s. 96 – 101.

Russel, S.J. & Norvig, P. (1995) *Artificial Intelligence: A Modern Approach*. New Jersey, Alan Apt.

Rönkkö, M. (2007) An Artificial Ecosystem: Emergent Dynamics and Lifelike Properties. *Artificial Life 13. Proceedings of the thirteenth international conference on the simulation and synthesis of living systems*. United States of America, Michigan, MIT Press. s. 159 – 187.

Sweetser, P. & Wiles, J. (2002). *Current AI in Games: A Review*. *Australian Journal of Intelligent Information Processing Systems*. School of ITEE, University of Queensland. 8(1), s. 24 – 42.

Sjörs, H., Söderström, B. & Ulfstrand, S. (2014) *Ekosystem*. *Nationalencyklopedin*. Tillgänglig på Internet: http://www.ne.se/lang/ekosystem [Hämtad Februari 17, 2014]

Whitey, D. (2001) *An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls*. *Information and Software Technology*. Elsevier. 43(14), s. 817 – 831.

Witten, A.H. & Frank, E. (2005) *Data Mining: Practical Machine Learning Tools and Techniques,* 2. uppl. San Francisco. Elsevier.

Woodcock, S. (2000) *Game AI: The State of the Industry*. Tillgänglig på Internet: http://www.gamasutra.com/view/feature/131975/game_ai_the_state_of_the_industry.php?page=2

Unity Technologies (2014) *Unity3D* (Version: 4.3.4) [Datorprogram]. United States of America, San Francisco, Unity Technologies. Tillgänglig på Internet: http://unity3d.com/