Examensarbete

HÖGSKOLAN I SKÖVDE

ANVÄNDNING AV EVOLUTIONÄRA ALGORITMER FÖR ATT UPPTÄCKA BALANSPROBLEM I STRATEGISPEL.

USAGE OF EVOLUTIONARY ALGORITHMS TO DISCOVER BALANCE ISSUES IN STRATEGY GAMES.

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2015

Pontus Wallin

Handledare: Erik Sjöstrand
Examinator: Mikael Johannesson

Sammanfattning

Spelbalans är ett viktigt koncept inom spelutveckling, generellt sett så anses datorspel där det finns många olika strategier för att vinna, och där alla är effektiva, vara bättre än spel där det bara finns några få effektiva strategier. För att uppnå bra spelbalans krävs ofta att mycket tid med att testa spelet med speltestare för att hitta strategier som kan förstöra denna balans. Arbetet undersöker om man kan snabba upp speltestningsprocessen genom att låta en evolutionär algoritm automatiskt undersöka ett spels strategirymd, och sedan dra slutsatser på hur spelet skulle ändras för att förbättra spelbalansen. Resultatet pekar på att metoden kan hantera extremfall, men förlorar effektivitet vid en viss gräns. I framtiden kan algoritmens effektivitet förbättras genom att förbättra sättet algoritmen tar ekonomiska faktorer i åtanke.

Nyckelord: Evolutionära Algoritmer, Spelbalans, Speltestning.

Innehållsförteckning

1 Introduktion

Realtidstrategispel (RTS) är en spelgenre som karaktäriseras av att spelaren ska bygga en bas, och ifrån denna bas bygga antal enheter som ska användas för att slå ut sin motståndare. En stor del av spelbalans inom RTS är att se till att de olika enheterna är väl balanserade mot varandra, dvs. alla enheter spelar en viktig roll för att utföra en viss strategi, eller för att kontra en annan strategi. Om spelaren anser att en enhet är starkare eller svagare än andra minskar spelarens vilja att använda sig av strategier där många olika enheter används. Eftersom många RTS använder sig av ett stort antal enheter, som alla har olika egenskaper kan det vara svårt att upptäcka många av dessa balansproblem.

Frågeställningen det här arbetet försöker besvara är om det är fördelaktigt att använda evolutionära algoritmer som ett hjälpmedel för att upptäcka balansproblem i strategispel, eftersom traditionella metoder såsom speltestning har vissa begränsningar.

För att testa den här frågeställningen har ett spel skapats, samt en evolutionär algoritm som kan testa spelet. Algoritmen och spelet kommer att utvecklas sida vid sida och denna arbetsprocess kommer att dokumenteras.

Spelet som har skapats är ett enkelt strategispel av typen ”sidoskrollande” RTS, där de enda val en spelare kan göra är i vilken ordning han ska bygga sin enheter. Detta för att ett sådant spel lämpar sig väl för att använda evolutionära algoritmer på.

Den evolutionära algoritmen slumpar fram ett antal byggordningar som sedan evolveras. Algoritmen använder sig av samevolution, vilket innebär att de evolverade byggordningarna tävlar mot varandra och evolveras sida vid sida. Om en grupp lösningar hölls sig till en dominant strategi och den andra gruppen inte hittade en taktik som kontrade denna, ansågs speletet vara grovt obalanserat. Om bägge grupperna lyckades kontra varandra, men inte utnyttjade alla enheter som spelet har till förfogande, ansåg de enheter som inte används vara för svaga.

När sådana problem upptäcktes ändrades balansen i spelet att ändras. De viktigaste iterationerna av denna process dokumenterades med förklaringar över vad som ändrades och varför.

2 Bakgrund

I det här kapitlet beskrivs de termer som är centrala för mitt arbete. Vad ett RTS är för något, vad spelbalans inom ett RTS innebär och en beskrivning av vad en genetisk algoritm är för något.

2.1 RTS

RTS står för Realtidstrategispel och RTS-genren har varit en stark spelgenre, särskilt bland PC-spelare, ända sedan Westwood Studios startade *Command and Conquer*-serien i mitten av 90-talet. RTS spel brukar ses ifrån ovan och handla om att hantera resurser för att bygga olika byggnader och enheter som sedan används för att slå ut motståndaren.

I Command & Conquer: Tiberian Dawn (Westwood studios, 1995) tar spelaren rollen som överbefälhavare över en av två fiktionella faktioner. Spelet har en enda resurs, Tiberium, som måste skördas av speciella fordon som kallas för skördare för att spelaren ska kunna bygga nya enheter och byggnader.

Figur 1 Command & Conquer (Westwood Studios, 1995) var ett av de första spelen som populariserade realtidsstrategi-genren.

I Command & Conquer bygger spelaren en bas, bestående av byggnader som kraftverk, baracker, fabriker och raffinaderier. Ekonomin i spelet består av att speciella skördare hämtar tiberium ifrån spelvärlden och get detta till raffinaderierna, spelaren kan sedan genast använda tiberiumet för att bygga spelets olika enheter och byggnader. Den militära delen av spelet består av att använda enheterna man har byggt för att bekämpa sin motståndare. Spelet möjliggör många taktiker, t.ex. att attackera fiendens bas direkt, att attackera hans sårbara skördare, eller att försöka slå ut fienden egna militärenheter.

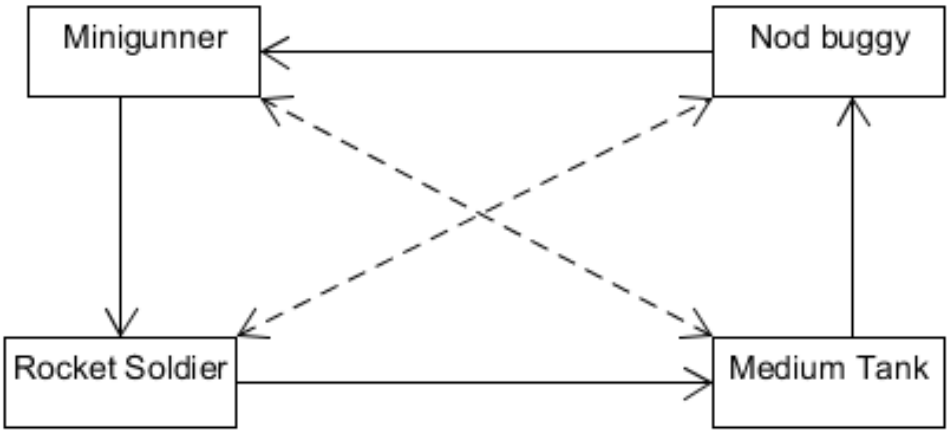
2.1.1 Spelbalans i RTS

Strategispel brukar vara balanserade efter principen att olika enheter har olika roller och att dessa roller kontrar varandra. Adams och Dormans (2012) kallar detta för *ortogonal enhetsdifferentiering*.

> Ideally, every type of unit in a real-time strategy game should be unique in some way and not just a more powerful (but otherwise identical) version of another unit.
>
> Adams & Dormans, 2012, s.142

Nedan illustreras detta med en liten delmängd av enheterna i Command & Conquer som exempel. Den billigaste enheten i spelet är en ”Minigunner”. Detta är en infanterienhet vars roll är att slå ut infanteri, men som är dålig på att slå ut fordon. Det logiska sättet att kontra detta är att köpa ett fordon som är specialiserat på att slå ut infanteri, som en ”Nod buggy”. För att kontra detta bör vi använda ett fordon som är bra på att slå ut andra fordon, exempelvis en ”Medium Tank”.

Stridsvagnen är inte bra på att slå ut infanteri, det logiska är då att bygga pansarvärnsinfanteri, i form av enheten ”Rocket Soldier”. Men eftersom en ”Rocket soldier” är dålig på att försvara sig mot annat infanteri är den i sin tur svag mot en ”Minigunner”.



Figur 2 Fyra enheter ur Command & Conquer och hur de kontrar varandra. Enkelriktade pilar representerar fall där en enhet har hög effektivitet mot en annan. De dubbelriktade streckade pilarna är fall där bägge enheterna har mer likvärdig styrka mot varandra.

Eftersom de flesta strategispel har någon form av resurshantering är en enhets kostnad också en faktor. Enheter brukar nämligen ha kostnader som ekonomiska resurser och produktionstid och vissa enheter kanske inte är tillgängliga om spelaren inte bygger en viss byggnad först, eller spenderar resurser på någon sorts forskning, för att låsa upp den.

Den generella principen är att enheterna ska vara kostnadseffektiva lösningar för sina roller och inga andra roller. Om en raketsoldat inte ensam är tillräckligt stark för att slå ut en stridsvagn är det ok, förutsett att stridsvagnar är såpass dyra att spelaren kan bygga flera raketsoldater för varje stridsvagn hans motståndare kan bygga.

Det är viktigt att balansera enheterna mot varandra för om en enhet A är för svag uppmuntras spelaren att bygga andra enheter istället för denna, och kanske särskilt de enheter som det är meningen att A ska kunna kontra. Om en enhet är för stark uppmuntras däremot spelaren att bygga den här enheten på bekostnad på andra. Jaffe & Miller(2012) förklarar med ett schackliknande spel som exempel:

> Consider a chess-like game, where one design goal is that all the pieces must work in unison. A designer might consider a rebalance in which the movement radius of one of the pieces p is extended. This has the potential to impact other aspects of the game indirectly, for example rendering useless some other piece q. This can be hard to predict using reason alone.
>
> Jaffe & Miller m.fl., 2012, s 2

Balansproblem kan leda till att ett spel där många olika strategier ska vara effektiva devolveras till att endast ett fåtal strategier är effektiva. Dessa strategier kallas för dominanta strategier.

> A dominant strategy is a course of action that is always the best one available to a player in all circumstances (It doesn't guarantee that the player will win; it's just his best option.) [...]. Games with a dominant strategy aren't any fun to play, because the players end up doing the same thing over and over again.
>
> Adams & Dormans, 2012 s. 128

Utöver att dominanta strategier gör ett spel mer repetitivt gör det också att spelet verkar orättvist. Om en ny spelare stöter på fientliga stridsvagnar och svarar med att bygga enheter som ska kunna kontra stridsvagnar så kan spelaren bli frustrerad om den här strategin inte är effektiv.

Det är brukligt att använda sig av mänskliga speltestare för att identifiera balansproblem. Leigh, Shonfeld & Louis (2008) menar att denna metod har vissa begränsningar eftersom det tar tid att spela ett spel och en människa bara kan utforska en liten del av ett spel varje gång han spelar. Mänskliga speltestare kostar därför mycket i form av tid och pengar och kan inte undersöka alla möjliga strategier.

Enligt Leigh m.fl. (2008) är matematiska formler en annan metod man kan använda sig av, men detta är ofta svårt om det finns komplicerade interaktioner mellan alla spelreglerna.

2.1.2 Sidscrollande RTS

Sidscrollande RTS är en undergenre till RTS-genren. Skillnaden mellan denna typ av RTS och vanliga RTS är att dessa spel ses ifrån sidan. Första spelarens bas finns på ena sidan av bana och andra spelarens bas på den andra och på grund av spelets perspektiv, finns det endast en väg mellan baserna. Spelen brukar ha låg fokus på manövrering (om detta ens är möjligt) och alla enheter som byggs går helt enkelt ifrån sin bas, till fiendens bas, och attackerar alla fiendeenheter som är i vägen.

Armor Alley (Information Access Technologies, 1990) ett av de tidigaste spelen i genren. Spelet var en hybrid mellan ett actionspel och ett strategispel, i spelet styr spelaren en helikopter, samtidigt som han kan kalla in stridsvagnar, infanteri och andra markenheter

enheter. Enheterna kommer in till spelområdet ifrån vänster sida av skärmen och går åt höger och attackerar allt i vägen.

Figur 3 Armor Alley (Information Access Technologies, 1990) är en sidescroller/RTS-hybrid. Spelaren använder sin helikopter för att förgöra motståndarens enheter och kan också kalla in förstärkningar i form av olika markenheter. Skärmbilden är tagen ifrån en nyversion av spelet, utvecklat av Scott Shiller, 2013.

I modern tid har många spel med liknande koncept skapats, men metoderna spelaren har för att påverka striden direkt brukar vara i någon form av magier eller liknande förmågor.

2.2 Evolutionära algoritmer

Evolutionära algoritmer är en teknik som bygger på att lösa problem genom att evolvera fram bättre och bättre lösningar ifrån ett antal ursprungliga lösningar.

Buckland (2002) förklarar att på samma sätt som varelser evolveras över många generationer för att bli bättre anpassade för sin miljö så kan man med evolutionära algoritmer evolvera fram lösningar till vissa sorters problem.

Allt liv på jorden är resultatet av en lång process av evolution där individer som är bättre anpassade för sin miljö har större chans att överleva och föra sina gener vidare. Dessa gener finns i form av kemiska bindningar i varje varelses kromosomer.

I varje cell i en levande varelse finns det kromosomer och kromosomerna består i sin tur av gener, som är ordnade i DNA-strängar. En varelses kromosomer är ritningarna för vilka egenskaper den varelsen har. Majoriteten av dessa egenskaper är ärvda ifrån varelsens föräldrar, men vissa egenskaper kommer ifrån slumpvisa mutationer. Kromosomerna man hittar i en enda cell håller all information som är nödvändig för att återskapa varelsen och detta kallas för varelsens genom (Buckland, 2002).

Buckland menar också att evolutionära algoritmer är fördelaktiga eftersom man inte behöver veta hur ett problem ska lösas, utan endast hur man kan beskriva potentiella lösningar på ett sätt som den evolutionära algoritmen kan utnyttja.

2.2.1 Generell princip

Principen är att man ska kunna koda alla steg i en potentiell lösning som en sorts digital kromosom. Kromosomen kan ses som en ritning, eller en instruktionslista, för lösningen. När man har listat ut ett sätt att koda kromosomerna så skapar man en befolkning som består av ett visst antal lösningar. Dessa består typiskt av slumpvis genererade instruktioner. Dessa kromosomer kan ses som individer som bildar den första generationen. Lösningarna testas sedan och en ny generation skapas baserat på de bättre individerna ifrån den förra generationen. När den nya generationen bildas är det brukligt man lägga till slumpelement i form av mutationer, för att förhindra att lösningarna hamnar i ett lokalt optimum (Obtiko, 1998).

För enkelhets skull är kromosomerna ofta kodade som bitsträngar med en viss längd. En slumpgenererad kromosom som är en byte i storlek kan tillexempel se ut så här: 10011011.

När Ponsen m.fl. (2006) undersökte användandet av evolutionära algoritmer för att utveckla AI för strategispelet Wargus (The Wargus Team, 2002-2011), representerade varje kromosom en statisk strategi. D.v.s. kromosomerna bestod av listor med olika spelhandlingar. De använde sig inte utav bitsträngar, utan kodade istället generna som bokstäver och siffror.

I Buckland (2002) beskrivs stegen som en evolutionär algoritm tar för att evolvera fram en befolkning av lösningar.

1. Lösningarna testas genom att man ”avkodar” kromosomerna till instruktioner som kan köras i testmiljön.
2. Två lösningar väljs ut, låt oss kalla dessa lösning A och lösning B. De lösningar som presterade bra under testningen har större chans att väljas i den här processen.
3. Kromosomerna i lösning A och B kombineras genom en process som kallas för överkorsning. En ny lösning skapas, som ärver den kombinerade kromosomen.
4. Slumpvis valda element av kromosomen ändras (muteras).
5. Steg 2 till 4 repeteras tills vi har tillräckligt många nya lösningar för att ersätta den tidigare generationen.

   Den här processen körs tills algoritmen avbryts (t.ex. efter ett visst antal generationer, eller när en tillräckligt bra lösning har hittats.).

2.2.2 Selektion och fitness

Selektion är processen som bestämmer vilka lösningar som ska kombineras. Det finns olika selektionsmetoder, men eftersom meningen är att man ska evolvera bättre och bättre lösningar handlar alla av dessa om att de lösningar som är bättre ska ha större chans att bli utvalda än de lösningar som är sämre.

Varje lösning har ett värde associerat med sig så att algoritmen förstår vilka lösningar som är bättre än andra. Detta värde kallas för lösningens Fitness. Fitness kan räknas ut på olika sätt beroende på problemet i fråga, man kan använda en definition där höga fitnessvärden är något positivt och låga eller negativa fitnessvärden är negativt. Om problemet är någon sorts optimeringsproblem kan andra sätt att räkna Fitness vara användbara. T.ex. om målet är att

hitta en minimal väg mellan två punkter, så kan inversen av antalet steg som tagits, eller hur många steg en viss lösning har tagit jämfört med den sämsta lösningen som genererades, vara lösningens fitness.

Ibland använder man sig av elitism när väljer ut lösningar. Detta innebär att några av de allra bästa lösningarna hålls oförändrade och kopieras till nästa generation, detta kan hjälpa en evolutionär algoritm att hitta bra lösningar snabbare (Buckland, 2002).

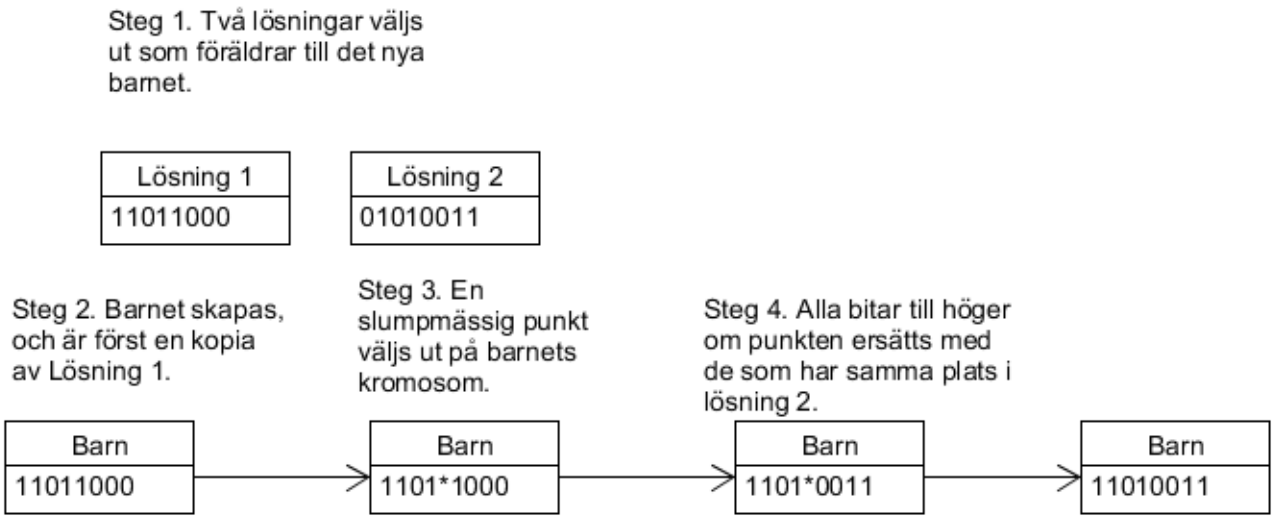
2.2.3 Överkorsning och mutation

Lösningarna kombineras genom en process som kallas för överkorsning. Det finns olika sätt att utföra överkorsning. Generellt så väljs två lösningar ut, och de kan ses som föräldrar till nya lösningen. Den nya lösningen kan kallas för de två föräldrarnas barn.

Obitko (1998) beskriver flera olika modeller.

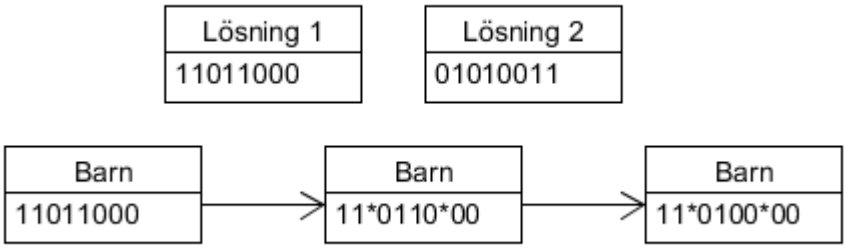
Envägsöverkorsning är den enklaste metoden. Algoritmen väljer ut en punkt i den ena kromosomens bitsträng och ersätter alla bitar som kommer efter (eller före) den punkten, med bitarna ifrån den andra lösningens kromosom. Tvåvägskorsning är samma sak, fast två punkter väljs ut.

Följande illustration är ett exempel på envägsselektion, där selektionspunkten är den fjärde biten:



Figur 4 En ny lösning skapas genom att korsa två andra lösningar genom envägsöverkorsning.

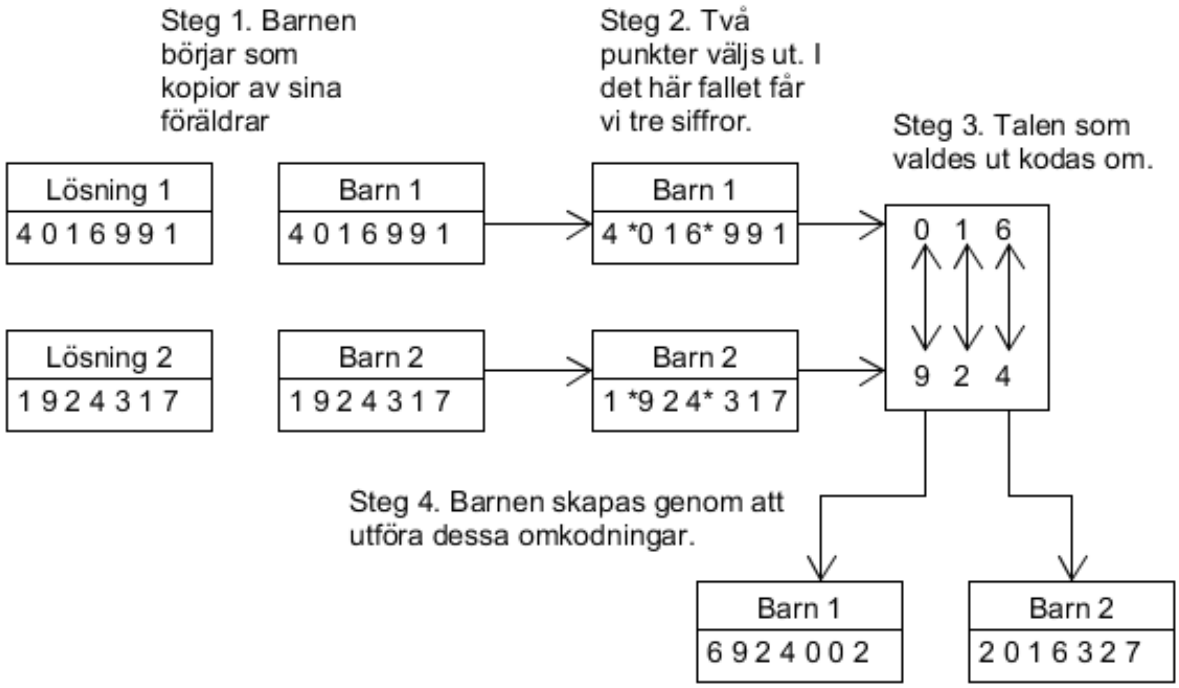
Tvåvägsselektion, där selektionspunkterna är den andra och den sjätte biten:



Figur 5 En ny lösning skapas genom att korsa två andra lösningar genom tvåvägsöverkorsning.

Det finns också varianter där flera slumpvisa selektionspunkter använder och varianter där den nya individens gener räknas ut matematiskt, beroende på dess föräldrars bitar.

Buckland (2002) beskriver en mer avancerad metod som fungerar på heltalskromosomer och som kallas för Partially Mapped Crossover (PMX). PMX handlar om att man ska koda om generna (som i det här fallet är heltal) i kromosomerna. Det enklaste sättet att förklara metoden är med ett exempel:



Figur 6 Två nya lösningar skapas genom Partially Mapped Crossover.

Enligt Črepinšek, Liu & Mernik (2013) så är en varierad befolkning ett krav för att undvika att en befolkning hamnar i ett lokalt optima. Det krävs en bra balans mellan att *utnyttja* det som verkar vara mest optimalt för tillfället (i en evolutionär algoritm, att korsa de bästa lösningarna) och att *utforska* fler lösningar genom att lägga till variation. Detta gäller inte bara evolutionära algoritmer, utan alla sorters sökalgoritmer.

> Every search algorithm needs to address the exploration and exploitation of a search space. Exploration is the process of visiting entirely new regions of a search space, whilst exploitation is the process of visiting those regions of a search space within the neighborhood of previously visited points. In order to be successful, a search algorithm needs to establish a good ratio between exploration and exploitation.
>
> Črepinšek m.fl.,2013, s 35

Ett sätt att uppnå variation är genom mutationer. En mutation är någon form av slumpartad ändring på en kromosom och används för att skapa variation i befolkningen man evolverar. Buckland (2002) nämner att ett vanligt sätt att mutera en bitsträng är att slumpmässigt växla ett litet antal bitar i kromosomen, vilket innebär att varje bit i kromosomen löper en lite risk att ändras ifrån en etta till en nolla eller vise versa. Han menar också att de flesta mutationer har negativa effekter på en lösnings fitness, men att negativa mutationerna brukar evolveras bort medan de positiva mutationerna består. Trotts detta finns risken alltid att vissa bra egenskaper försvinner.

Låg mutation innebär alltså att de positiva egenskaperna som redan finns i befolkningen utnyttjas mer. Dock är risken högre att man på grund av denna brist på variation missar många bra egenskaper som inte finns i den nuvarande befolkningen. Hög mutation innebär att fler av dessa egenskaper upptäcks, men risken är högre att bra egenskaper som redan finns i befolkningen försvinner.

2.2.4 Samevolution

När lösningar ska testas mot varandra (dvs. en lösnings framgång är beroende på en annans, som när två spelstrategier testas mot varandra) är det lämpligt att använda sig av samevolution. Samevolution innebär att man evolverar flera populationer sida vid sida. D.v.s. Population A testas mot Population B, A evolveras utifrån resultatet av testet, och sedan testas Population B mot Population A. Processen repeteras, och de två populationerna hamnar i en sorts kapprustning mot varandra.

Leigh, Schonfeld & Louis (2008) beskriver samevolution som en delmängd av Evolutionära Algoritmer som är utformad för att optimera problem där en lösnings fitness är kopplade till andra lösningars fitness. Enligt Leigh m.fl.(2008) så kan man med samevolution undersöka ett spels ”strategirymd” snabbare än man kan med mänskliga speltestare.

> Evolutionary algorithms tend to converge towards strategies that win all the time. Coevolution has the advantage of being able to play games faster than humans and searching the strategy space implicitly in parallel.
>
> Leigh m.fl.,2008, s. 1564

Hassan & Rafie (2010) beskriver samevolution på ett liknande sätt, och hävdar att evolutionära algoritmer kan hitta spelstrategier på expertnivå, utan hjälp från människor.

> Co-evolution, through the context of the evolutionary computation literature, defines the fitness of an individual based on his competitors and collaborators

> situation as well as the environment. Within the paradigm of evolutionary computations, expertgame-playing strategies have been explored and discovered without the need for human expertise, and the manual discovery of new invented strategies.
>
> Hassan & Rafie, 2010, s.1321

2.3 Relaterade arbeten

2.3.1 Leigh m.fl. (2008)

Leigh m.fl. (2008) använde evolutionära tekniker för att balansera tre olika spelstrategier i ett spel de skapat själva.

Spelet kallades för CaST (Capture, Steal, Territory) och var ett tvåspelarspel som gick ut på att varje spelare styr ett rymdskepp och får poäng genom att hämta lådor, som återskapas i mitten av banan, till sin bas, och genom att ta territorium. Banan är indelad i rutor, och en spelare konverterar automatiskt en ruta territorium till sin färg genom att flyga över den. Ett tredje sätt att få poäng på är att stjäla lådor som motståndaren håller i, och ta dessa lådor till sin egen bas. Spelreglerna leder till tre centrala strategier:

Crate Running - Spelaren hämtar lådan i mitten av banan, släpper din i sin bas, och hämtar sedan en ny låda.

Crate Stealing - Spelaren väntar på att den andra spelaren ska hämta lådan i mitten av banan, stjäl den ifrån den andra spelaren och hämtar den till sin bas.

Grab Territory - Spelaren försöker att konvertera så många rutor som möjligt och prioriterar att den andra spelarens rutor först.

Crate Running kontrar Grap Territory, eftersom den ger mer poäng vid matchens slut än vad Grab Territory gör. Crate Stealing kontrar Crate Running, eftersom den strategin hindrar spelaren från att hämta lådan. Grab Territory kontrar Crate Running, eftersom poängen som tjänas genom att konvertera territorium inte kan blockas med den strategin.

Spelarna styrdes av evolverade tillståndsmaskinsbaserade AI-agenter, som växlade mellan de tre olika strategierna. Agenternas genom höll, bland annat, två parametrar för varje spelstrategi. En parameter som bestämmer hur länge agenten kommer att vara kvar i den strategin, och en parameter som bestämmer sannolikheten att agenten övergår till just denna strategi när den är färdig med en annan strategi. Agenterna delar helt enkelt upp sin tid mellan de tre olika strategierna, utefter instruktionerna som genomet ger dem.

Samevolution användes. D.v.s. Agenterna delades upp i två befolkningar som turades om att tävla mot varandra i en sorts kapplöpning. Om befolkning A hittar en stark strategi så kommer befolkning B att hitta en stark motstrategi, befolkning A hittar sedan en strategi mot denna strategi, osv. Utöver detta så sparades de bästa individerna ifrån vardera befolkningen i en hall of fame och vissa individer därifrån användes i testningen, för att försäkra att de evolverade agenterna inte bara hade bra Fitness mot befolkningen den tävlade mot i nuläget, utan också de bästa dittills hittade lösningarna.

Strategierna balanserades mot varandra genom att ändra på värden som tiden de tog att plocka upp en låda, tiden det tog att stjäla en låda ifrån den andra spelaren, och hur många

poäng man tjänade på att hålla mer territorium än sin motståndare. Strategiernas effektivitet mot varandra visualiserades med en heatmap.

2.3.2 Ponsen m.fl. (2006)

Ponsen m.fl. (2006) utvecklade AI till spelet Wargus genom att använda sig av dynamisk skriptning, där AI:n dynamiskt använder sig av taktiker ifrån en stor databas av sådana. De använde evolutionära metoder för att skapa nya taktiker som kunde läggas till i denna databas.

Varje kromosom representerar en statisk lösning. Kromosomerna består av ”states” som i sin tur består av en serie gener. Fyra olika sorters gener användes (”combat”, ”build”, ”economy” och ”research”) och varje gen representerade en spelhandling.

Nya kromosomerna genererades genom fyra genetiska operationer. ”State”-överkorsning, vilket väljer ut två föräldrar och korsar deras ”states”. Genersättningsmutation, som väljer ut en förälder och ersätter slumpvis valda gener. Genpartisk mutation, som väljer ut en förälder och muterar parametrarna för redan existerande ekonomiska eller militära gener. Samt randomisering, som skapar en ny slumpvis kromosom.

En kromosoms Fitness berodde på de två lagens militära poäng och tiden matchen höll på. Vid högre militärpoäng gav högre fitness, vid vinst som vid förlust. Med vid förlust användes även tiden AI-agenten klarade sig, om den klarade sig längre innan den förlorade fick den högre poäng. Ponsen kom fram till slutsatsen att evolutionära metoder kan evolvera fram högkvalitativa taktiker.

3 Problemformulering

3.1 Syfte

Syftet är att undersöka hur evolutionära algoritmer kan användas för att hitta balansproblem i ett RTS.

Att försöka balansera ett spel matematiskt kan vara nästan omöjligt på grund av de komplicerade interaktionerna mellan spelreglerna (Leigh m.fl., 2008) och enligt Jaffe m.fl. (2012) är spelbalansering ett av de mest tidskrävande stegen i speldesignprocessen. Detta eftersom det görs genom många cykler av att göra små förändringar och speltesta efter varje förändring.

Eftersom speltestning är tidskrävande så ställde jag följande generella fråga:

Hur kan automatiserad speltestning med hjälp av evolutionära algoritmer underlätta balanseringsarbetet i ett realtidsstrategispel?

Denna frågeställning undersöktes genom att skapa ett simpelt strategispel, samt en evolutionär algoritm som använder sig av samevolution med två befolkningar. Sedan strävade jag efter att balansera spelet med hjälp av denna algoritm.

Målen jag prioriterade när jag balanserade spelet var att:

1. Ingen enhet eller strategi ska vara dominant. D.v.s. algoritmen favoriserar inte en enda enhet.
2. För varje enhet eller strategi så ska det finnas en annan enhet/strategi som effektivt motverkar denna. D.v.s. om en befolkning favoriserar enhet A så ska den andra befolkningen hitta motstrategier mot enhet A.
3. Ingen enhet ska vara så svag att den är överflödig. D.v.s. algoritmen ska inte ignorera vissa enheter.

När sådana balansproblem identifierades av algoritmen ändrade jag på en egenskap på en av enheterna och testa spelet igen i syfte att se om balansen blev bättre, samt undersöka om nya balansproblem uppkom.

3.2 Metodbeskrivning

Ett sidscrollande RTS har skapats, tillsammans med en evolutionär algoritm som är anpassad till den. Spelet och algoritmen utvecklades sida vid sida och utvecklingsprocessen dokumenterades.

Spelets grundläggande spelmekaniker färdiggjordes innan den evolutionära algoritmen skapades, och balansen i den första versionen av spelet var baserad på matematik, känsla och några få manuella speltester. Den evolutionära algoritmen skapades därefter och efter den punkten baserades det fortsatta balanseringsarbetet på information ifrån algoritmens simuleringar.

Den evolutionära algoritmen bygger på samevolution där två populationer evolveras sida vid sida. Där barje individ i den ena befolkningen spelar en match mot varje individ i den andra befolkningen. Individerna tilldelas fitnesspoäng baserat på hur bra de gjorde ifrån sig i dessa

matcher. Att vinna en match snabbt ger högst poäng, att vinna en match långsamt ger inte lika hög poäng. Att förlora en match långsamt ger låg poäng, och att förlora en match snabbt ger lägre poäng. Individens fitness är då summan av poängen den har fått på varje match.

Leigh m.fl. beskriver vissa mönster som pekar på dålig balans.

> if dominating strategies exist in the game, coevolution will likely find them and converge. If the game is balanced with proper counter strategies, then the two populations will never stop adapting to each other.
>
> Leigh m.fl., 2008

Tanken var att när spelet är balanserat så kommer de två populationerna aldrig att sluta anpassa sig efter varandra. Om en befolkning hittar en stark strategi men den andra inte hittar en stark strategi som kontrar denna är denna strategi dominant. Om bägge befolkningarna ignorerar en viss enhet kan den enheten räknas som överflödig och behöver stärkas för att göra enheten relevant, i vissa fall kanske tillochmed tas bort ifrån spelet. Om befolkningarna håller sig till endast ett fåtal enheter bör balansering ske så att alla enheter används flitigt i framtida simulationer.

Arbetet var en process som repeterades tills problemen nämnda ovan minimerades. Processen i fråga bestog av tre steg:

1. Testa spelet med den evolutionära algoritmen.
2. Undersök testdatan och gör ändringar. Ändra antingen en viss egenskap på en viss enhet i syfte att balansera spelet, eller förbättra den evolutionära algoritmen om nödvändigt.
3. Förklara problemet som identifierades i steg 1 och åtgärderna som togs för att lösa det i steg 2.

På grund av att det finns slumpelement i algoritmen så kördes algoritmen tre gånger för varje speltest. Tanken var att den här processen skulle utföras i 10 iterationer genom arbetes förlopp och att de viktigaste iterationerna skulle dokumenteras.

Leigh m.fl. (2008) dokumenterade endast den första iterationen, den sista iterationen och tre iterationer däremellan. Den första iterationen visade spelets ursprungsbalans och de fyra följande beskrev kronologiskt hur spelets balans utvecklades. Den sista iterationen visade på hur välbalanserat det slutliga spelet blev.

Enligt Dormans & Adams (2012) är det en bra idé att bara ändra en egenskap i taget när man speltestar, annars vet man inte vilken av ändringarna som gav vilket effekt på hela spelets balans. Det är också fördelaktigt att börja med stora förändringar, för att se att om förändringen i fråga har någon effekt alls.

3.2.1 Metoddiskussion

Eftersom vi inte visste i förväg vilka strategier som är starkast och inte har något sätt att beskriva hur en stark strategi kunde byggas så passade evolutionära algoritmer bra. Enligt Buckland(2002) har evolutionära algoritmer en fördel i det att man inte behöver veta hur ett problem ska lösas.

Enligt Leigh m.fl. (2008) har evolutionära algoritmer vissa fördelar över mänsklig speltestning, eftersom algoritmen kan spela spel många gånger snabbare och undersöka ”strategirymden” parallellt. De påstår också att en samevolutionär algoritm troligtvis kommer att hitta dominanta strategier, om dessa existerar.

Evolutionära algoritmer kan också vara en bättre lösning än att använda sig av matematiska metoder, enligt Jaffe .m.fl. (2012) bör man ifrågasätta sådana metoders giltighet, eftersom de ofta är godtyckligt uträknade.

Leigh m.fl. (2008) påstår att nuvarande AI-tekniker inte kan ersätta mänskliga speltestare helt och hållet, eftersom en riktig människa kan hitta faktorer som en Fitnessfunktion kanske missar.

En annan nackdel med metoden är att spelet som testas har stora begränsningar i form av dess simplicitet. I och med att enheterna bara möter en motståndare i taget, och alltså inte kan ”svärma” runt fienden om de är flera än de, kommer enheter som är testade i min miljö inte nödvändigtvis vara balanserade i ett vanligt RTS. Mitt arbete ska därför inte ses som ett verktyg för att balansera andra spel, utan snarare som ett ”proof of concept”.

3.3 Speldesign

Spelet kommer är ett tvåspelarspel där varje spelare äger var sin bas på var sin sida av banan. Spelarna tjänar automatiskt in resurser under spelets gång och använder dessa resurser för att bygga militära enheter, som var och en har ett visst syfte. När en enhet har skapats rör den sig mot motståndarens bas och stannar till och attackera de av motståndarens enheter som den möter på vägen. Om enheten når fiendebasen börjar enheten göra skada på den. Målet med spelet är att förstöra sin motståndares bas.

Ett sidscrollande RTS utvecklades istället för ett mer traditionellt RTS, eftersom sidscrollande RTS har vissa begränsningar som gör det fördelaktigt att anpassa en evolutionär algoritm till den sortens spel. Faktorer såsom manövrering och resurshantering behöver inte räknas in i de lösningar som den evolutionära algoritmen genererar. En match kan då enkelt representeras av två enhetsbyggordningar som exekveras mot varandra.

Varje enhet har någon egenskap som gör den till ett intressant val över de andra, beroende på vilken strategi spelaren använder. Det kommer att finns enheter som är duktiga på närkamp, det finns enheter med projektilvapen, det finns en enhet vars roll är ha hög hälsa och agera som en sköld åt andra enheter.

Produktion av enheter tar en viss tid (räknat i antal uppdateringar, inte realtid), beroende på enheten i fråga. Detta är delvis ännu en egenskap som kan ändras för att balansera de olika enheter, men finns också av mer tekniska skäl. Utan en produktionstid för varje enhet så skulle en algoritm (eller en människa) kunna bygga enheter så snabbt att de skapas för nära varandra, eller tillochmed i varandra.

3.4 Algoritmdesign

Algoritmen använder sig av samevolution av två populationer som båda startar med slumpmässigt genererade genom. En metod liknande den som användes av Leigh m.fl. (2008) kommer att användes.

Varje Individ i Population A testas mot varje individ i Population B. Individerna i population A evolveras sedan en generation framåt och Populationen B testas mot den nya Populationen A, så att Population B i sin tur kan evolveras ett steg. När Population A har avancerats en generation, testas Population B mot Population A på precis samma sätt.

Kromosomerna består av ett antal heltal, som tillsammans bildar en byggordning, varje heltal representerar en viss enhet. Fitness beror på tid. D.v.s. om en lösning vinner får den hög fitness, men ännu högre om den vinner under kort tid. Om en lösning förlorar får den alltid lägre fitness än en lösning som vinner, men fitnessen blir ännu lägre om den förlorar på kort tid.

Detta kan beskrivas med ekvationen:

F = T_max/T * K– vid vinst.

F = T/T_max * K – vid förlust.

F = K – vid oavgjort resultat.

Där F är fitnesspoängen, T är tiden matchen tog, T_max är maxtiden en match får ta och K är en konstant.

Två populationer bestående av 20 individer vardera användes i algoritmen, och elitism implementerades genom att de två bästa lösningarna i varje population kopierades över till nästa generation utan förändringar. Övriga individer skapas genom överkorsning av två individer som väljs ut med hjälp av rouletthjulselektion, samt en låg mutationsrisk på varje element. Algoritmen kördes i 100 generationer.

Algoritmen sparar information om hur ofta de olika enheterna förekom under hela simulationen, samt sparar de 10 bästa byggordningarna i en ”Hall of Fame”. Programmet genererar en textfil som sparar ner information om hur ofta varje enhetstyp användes, för varje befolkning och varje generation, denna information importerades till Microsoft Excel för att generera grafer som illustrerade hur de olika befolkningarna evolverades. En annan textfil genererades, som skriver ut Hall of Fame för körningen, samt generell statistik om hur ofta de olika enheterna förekommer i Hall of Fame, och i hela epoken.

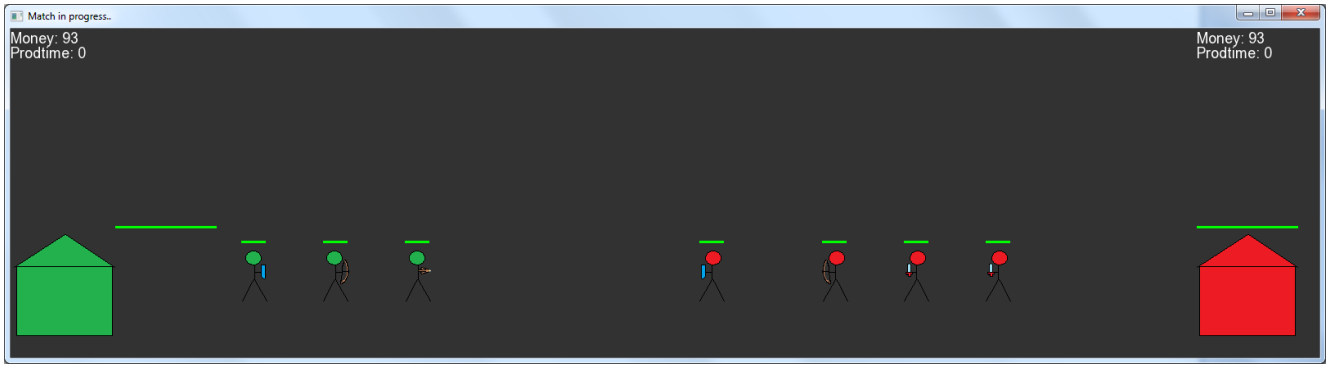
4 Genomförande

4.1 Spelet.

I spelet tar de två spelarna kontroll över var sin basbyggnad, vilka är placerade på var sin sida av skärmen. Basbyggnaderna genererar pengar, som spelarna använder för att bygga militära enheter. När en enhet byggs så dras pengar av ifrån spelaren ekonomi och enheten dyker upp utanför basen, när en produktionstimer har gått ner till noll. Enheterna går sedan mot motståndarens bas och attackerar den närmaste fiendeenheten, om denna är inom räckvidd.

Målet är att bekämpa motståndarens enheter för att nå, och förgöra, motståndarens bas. Om det inte finns några militära enheter kvar i spelvärlden, och ingen av baserna håller på att bygga nya, så leder detta till oavgjort. En väldigt sällsynt bugg, som gör baserna oförstörbara, leder också till oavgjort i vissa fall.

Spelet programmerades i C++ och grafikbiblioteket SFML användes för att rendera spelvärlden med hjälp av enkla sprites. Enheterna är färgkodade. Gröna enheter tillhör lag ett och röda enheter tillhör lag 2.



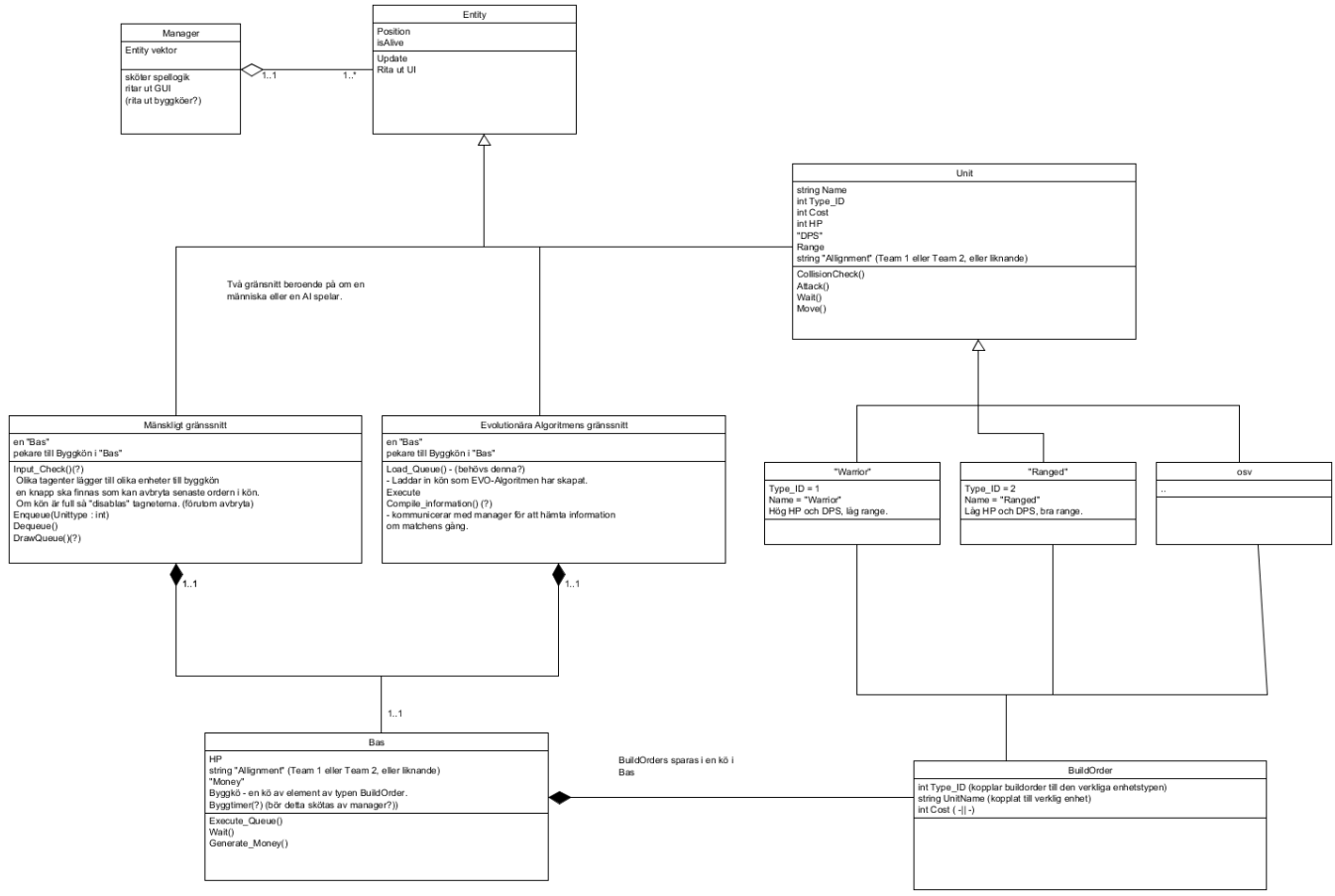
Figur 7 Skärmbild av spelet som visar upp lagens respektive baser och de enheter som har byggts. Texterna I övre vänstra och högra hörnen visar hur mycket pengar varje lag har och, om en enhet håller på att byggas, hur många uppdateringar det dröjer tills nästa enhet är färdig

I nuläget så kan spelet inte spelas av en mänsklig spelare. Den evolutionära algoritmen bestämmer vilka enheter som ska byggas och varje match i spelet spelas mellan två byggordningar som algoritmen har skapat. Byggordningarna laddas in i baserna, beroende på vilken befolkning byggordningen i fråga tillhör. D.v.s. en byggordning ifrån befolkning 1 laddas alltid in till basen som tillhör lag 1, och vice versa. Byggordningarna laddas in via en komponent som kallas för EVO_Base, som är tänkt som gränssnitt mellan den konkreta implementationen av baserna (Concrete_Base) och den evolutionära algoritmen.

Anledningen till att speciellt gränssnitt utvecklades för algoritmen var att ursprungligen var tänkt att även ett annat gränssnitt skulle utvecklas, för att göra det möjligt för en mänsklig användare att spela spelet. Detta utvecklades aldrig, eftersom det i slutändan inte ansågs nödvändigt att göra spelet spelbart av en användare.

Byggordningarna sparas som köer av objekt med typen ”BuildOrder”. En ”BuildOrder” kan ses som en beställning efter en enhet. En sådan beställning har en kostnad, produktionstid och namn, associerat med sig, beroende på vilken enhet det är som ska beställas. Varje bas är ansvarig för att gå igenom byggordningen den har fått, och kommer att bygga enheterna en efter en, förutsatt att spelaren har tillräckligt med pengar. Om spelaren inte har tillräckligt med pengar så stoppas byggandet upp tills spelaren har tillräckligt.

Detta preliminära klassdiagram skapades under designstadiet som en ritning för hur spelet skulle designas:



Figur 8 Preliminärt klassdiagram. Notera att en ”BuildOrder” inte är en hel byggordning, utan endast en enstaka ”beställning”, en det som när ordet ”byggordning” används så menas egentligen flera ”BuildOrders” i en kö.

Alla spelobjekt i (dvs. enheterna och baserna) ärver ifrån basklassen Entity, så att spellogiken för alla spelobjekt kan skötas genom detta objekt. Spelloopen, och spellogiken, sköts av komponenten Manager, som håller alla spelobjekt i en vektor med spelobjekt av typen Entity. Manager uppdaterar spelobjekten och håller koll på spelvärldens status, bland annat om något lag har förlorat, om något objekt ska elimineras ifrån spelet, om två objekt kolliderar med varandra och vilka enheter som är inom räckvidd för andra enheter.

De olika eftersom enheterna i spelet har ganska liknande beteenden, varje enhet behöver hålla koll på information som sin position och sina stridsegenskaper. Alla enheter beter sig också på samma sett vid kollision, d.v.s. enheten stannar om den kolliderar med ett objekt framför sig själv, och rör sig ett steg framåt vid varje uppdatering, om ingen kollision sker. Det enda

som differentierar enheterna ifrån varandra är att deras attribut (hälsopoäng, attackräckvidd, kostnad osv.) har olika värden.

På spelmekanisk nivå kan spelet delas upp i två delar, stridssystemet och ekonomisystemet.

4.1.1 Stridssystemet.

Spelet kollar avståndet mellan alla enheter utför attacker om någon enhet är inom räckvidd för någon annan enhet. Enheterna attackerar alltid den fiende som är närmast sig själv. När en enhet har tagit emot mer skada än den har i hälsopoäng så elimineras den ifrån spelet. Enheterna kan också göra skada mot motståndarens bas. Om det inte finns några enheter alls kvar på spelplanen och om bägge spelarna har utfört hela sin byggordning så blir matchen oavgjord.

I början av utvecklingen så var förgörelse av fiendens bas det enda sättet att vinna Eftersom majoriteten av matcherna slutade med att en sida fått slut på enheter och det enda som skedde de sista sekunderna i dessa matcher var att de överlevande enheterna ifrån motståndarlaget gick mot basen och förgjorde den utan motstånd så skapades ett nytt sätt att vinna.

När en spelare har förlorat alla sina enheter (och inga nya kan byggas) kommer den spelaren att vinna om den andra spelaren har minst en enhet kvar på spelplanen. Detta minskade tiden varje match tog lite granna och det löste också en sällsynt bugg där enheterna vägrade att attackera motståndaren bas, även om den var inom räckvidd.

För att hantera detta fall kollade spelet hur länge det dröjt innan senaste utförda attacken och om inga attacker utfördes på 10 000 uppdateringar så blev matchen oavgjord. Genom att lägga till det här nya vinstvillkoret så eliminerades byggen, eftersom enheterna aldrig har problem med att attackera andra enheter.

I början utfördes attackerna av enheterna själva när dessa uppdaterades i spelloopen, men systemet ändrades så att alla attacker utförs samtidigt i början av varje uppdatering, för att försäkra att alla enheter för en chans att utföra attacken. Detta för att eliminera risken att vissa enheters attacker utförs en uppdatering före andra, vilket ledde till att de enheter som inte råkade få förtur kunde elimineras innan deras sista attack utfördes.

Varje enhet har följande egenskaper:

Hälsopoäng – hur mycket skada enheten tål innan den elimineras ifrån spelet.

Attackskada – hur mycket skada enheten gör för varje attack.

Attackförsening – antalet uppdateringar som enheten måste ”vila” innan nästa attack.

Räckvidd – Över hur långt avstånd enheten kan attackera, mätt i pixlar.

Kostnad – Kostnaden för att bygga denna enhet.

Produktionstid – Tiden, i antal uppdateringar, som det tar från att en enhet har blivit betalad för, tills att den skapas och är med i spelet.

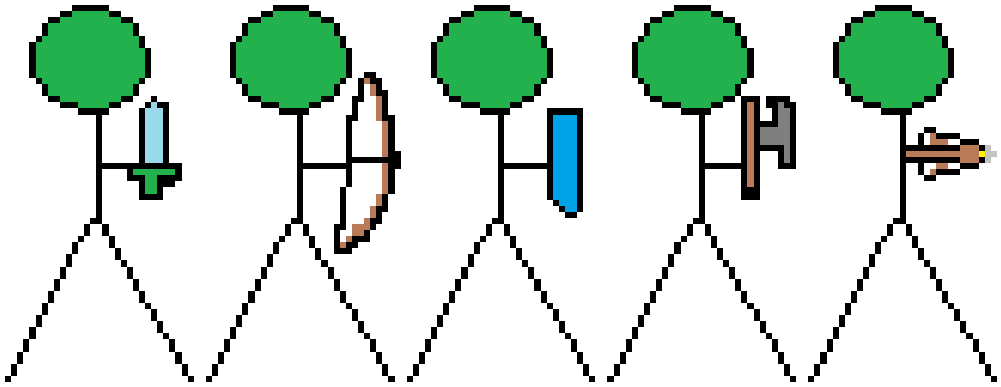
Rörelsehastighet – Hur många pixlar enheten rör sig mot fiendens bas varje uppdatering. (I nuläget är denna 1 pixel, för alla enheter.).

Attackförsening lades till för att ge fler möjligheter att balansera enheterna och ge dem olika roller. T.ex. för att skapa en enhet som gör mycket skada på en gång, men som kanske förlora i det långa loppet mot en enhet som tål den första attacken.

Vissa begränsningar har gjort att några enheter som skulle ha varit med i designstadiet inte är med i den färdiga versionen. Spelet har inga flygande enheter, på grund av tidsbrist. Spelet har inte någon enhet som är specialinriktad på att göra mycket skada mot fiendens bas, eftersom detta ansågs överflödigt på grund av att nästan alla matcher i praktiken redan är vunna innan förlorarens bas tar skada.

De enheter som finns i spelet är:

1. Warrior. En flexibel enhet, som inte ska vara speciellt bra, eller dålig, på någon speciell roll.
2. Archer. Långdistansenhet. Bågskyttar kan slå ut enheter som inte tål mycket skada innan de kommer i närkamp. Men om en bågskytt hamnar i närkamp mot en annan enhet så ska den vara ineffektiv.
3. Shieldbearer. Närkampsenhet som tål mycket skada, men inte kan göra mycket skada tillbaka. Sköldbärarens roll är att ta emot attacker, så att andra enheter har chans att göra skada innan de faller.
4. Axeman. En aggressiv krigare som gör mycket skada, och attackerar snabbt. Men tål inte mycket skada själv.
5. Crossbowman. Dyr och kraftig långdistansenhet. Gör mycket skada per skott, men skjuter långsamt.



Figur 9 Sprites för de fem olika enheterna. Från vänster till höger: Warrior, Archer, Shieldbearer, Axeman, Crossbowman.

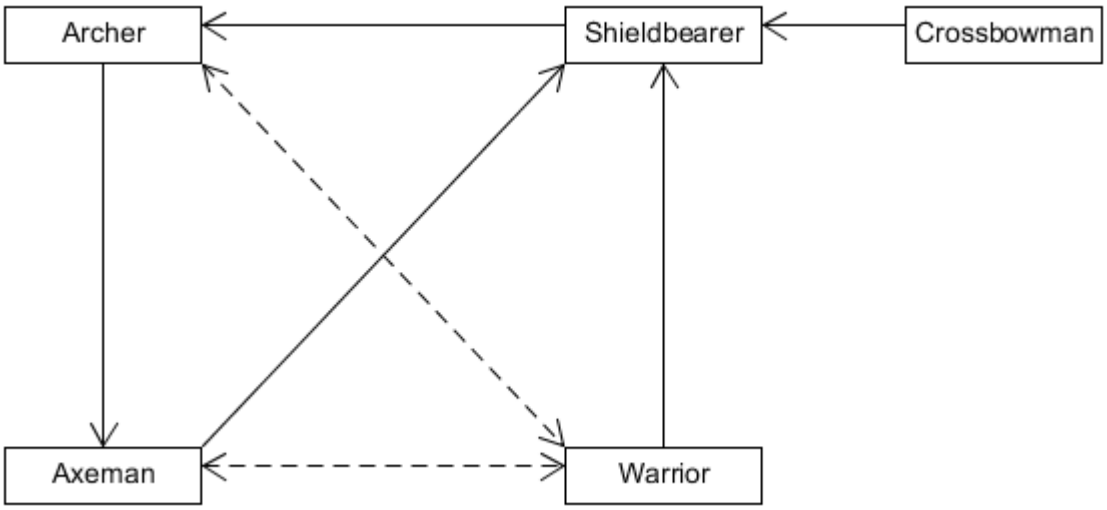
Bågskyttar ska vinna mot Yxmän, eftersom dessa inte hinner komma i närkamp innan de faller, men förlora mot sköldbärare, då dessa tål mer, och hinner komma i närkamp.

Yxmän ska vinna mot sköldbärare, eftersom sköldbärare har låg attackskada och yxmän har hör attackskada.

Armborstmän är dyra ”superenheter” som garanterat förgör andra enheter med en enda attack, men har låg attackhastighet och högt pris. De kan därför kontras genom att bygga flera, billiga enheter. Eftersom armborstmän gör extremt hög skada är dessa bäst använda mot Sköldbärare.

Krigare ska vara en generell närkampsenhet, som är ett flexibelt alternativ till yxmän eller sköldbärare.

Dessa enheter har alla unika roller och deras egenskaper bygger på principen om ortogonal enhetsdifferentiering som beskrevs i bakgrundkapitlet till detta arbete (Adams och Dormans, 2012). De har alla en unik roll i spelet, möjligtvis med undantag av Warrior eftersom dennas unika roll är att vara ett flexibel, men inte specialiserad på något.



Figur 10 Enheterna i spelet och hur de kontrar varandra. Enkelriktade pilar representerar fall där en enhet har hög effektivitet mot en annan. De dubbelriktade streckade pilarna är fall där bägge enheterna har mer likvärdig styrka mot varandra. Crossbowman kan kontras med större antal av vilken enhet som helst förutom Shieldbearer.

4.1.2 Ekonomisystemet.

Spelet har ett enkelt ekonomisystem där bägge spelare gradvis tjänar in pengar, i ett intervall på en pengaenhet vid varje uppdatering. De olika enheterna i spelet har olika kostnader associerade med sig och kan inte börja byggas innan spelaren har tillräckligt mycket pengar för att täcka hela kostnaden. Syftet med det här systemet är att göra spelet mer komplex och att ge möjlighet att balansera enheter utefter ekonomiska faktorer.

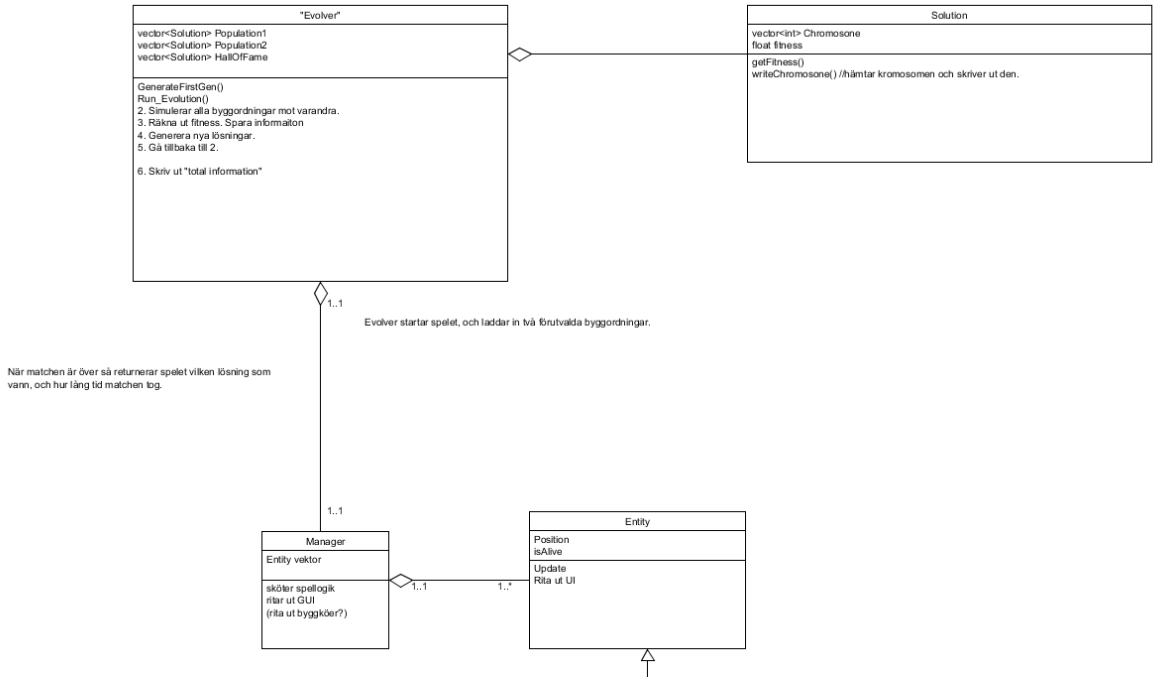
Eftersom spelarna konstant tjänar pengar så finns det ingen gräns på hur många enheter som kan användas under en match, och därmed inte heller hur lång tid en match kan ta. Så en gräns sattes på hur mycket pengar som kan genereras, denna gräns är i nuläget 1000 pengaenheter.

När en enhet ska byggas så dras dessa pengar ifrån spelarens ekonomi och en produktionstimer startas. När produktionstiden är slut skapas enheten i fråga och blir aktiv i spelvärlden. Produktionstimern och mängden pengar som en spelare har ritas ut på skärmen.

4.2 Algoritmen.

Från början var det tänkt att separera algoritmen och spelet till två olika program och låta dessa kommunicera med varandra. Men detta skulle vara svårt att göra i praktiken eftersom

algoritmen måste kunna spela många matcher så snabbt som möjligt och bedömningen gjordes att med två olika program så hade testerna tagit längre tid. Det kändes också mer intuitivt att bygga algoritmen utifrån spelet, eftersom alla matcherna körs i en loop inom algoritmen.



Figur 11 Klassdiagrammet expanderades, och detta användes sedan som ritning för algoritmen och dess integration med spelet.

Komponenten Evolver är central för den evolutionära algoritmen. Evolver sköter evolutionsprocessen och håller lösningarnas genom. Processen startas genom att ett antal slumpgenererade lösningar skapas. Dessa lösningar är uppdelade i två populationer och skapar sedan en spelinstans (genom att skapa ett objekt av typen Manager) genom vilken den kör sina simulerade speltester. Utöver detta sparar Evolver de tio bästa lösningarna i en speciell population som kallas för ”Hall of Fame”. ”Hall of Fame” är bara en lista på de bästa lösningarna, och deltar inte direkt i samevolutionen.

Individerna som testas är av typen Solution. En Solution (lösning) består av en vektor av heltal, som är lösningens kromosom. Varje heltal representerar en enhet i spelet, så genomet blir en byggordning. Denna byggordning matas in i spelet när det är den lösningens tur att testas. Solution håller också en del information, som sin egen fitness, hur många matcher den spelat och hur många av dessa som den har vunnit.

Evolver höll ursprungligen dessa befolkningar, och Hall of Fame, som vektorer av lösningar. Detta visade sig ha brister, eftersom det hade varit bra att kunna sortera ordningen lösningarna förekommer i populationen, ta bort lösningar som har låg fitness (ifrån Hall of Fame) räkna upp hur ofta olika enheter förekommer i populationerna, m.m. En ny klass skapades därför, Population, för att hålla lösningarna och för att ge smidig åtkomst till dessa funktioner. Även Hall of Fame ändrades till typen Population.

Algoritmen fungerar så att varje lösning i befolkning ett spelar mot varje lösning i befolkning två. Lösningarnas fitness uppdateras sedan beroende på om den vann matchen eller inte, samt hur lång tid matchen tog, räknat i antal uppdateringar. Sedan kollar algoritmen om någon av dessa lösningar har tillräckligt hög fitness för att gå med i ”Hall of Fame”. Relevant information för alla lösningar i befolkningen skrivs sedan ut i en outputtextfil.

Bägge befolkningarna evolveras sedan. Majoriteten av lösningarna genereras en efter en genom att välja ut två föräldrar via rouletthjulsselektion. Den nya lösningen skapas genom envägsöverkorsning av föräldrarnas genom, samt en 1 % mutationsrisk (vilket kan ändras under testningens gång). Efter att de nya befolkningarna har skapats så får de individerna spela nya matcher mot varandra, och resten av processen repeteras.

Eftersom rouletthjulsselektion är en urvalsprocess som är delvis slumpbaserad så finns det en risk att vissa genom inte kommer vidare till nästan generation, trotts att de har högre fitness än andra genom. Elitism har implementerats i syfte att se till att dessa individer inte förloras (Buckland, 2002, sid 137). Detta innebär att ett visst antal av de allra bästa lösningarna kopieras över i sin helhet till nästa generation. I nuläget är detta antal två eliter per befolkning.

Programmet är färdigt när ett visst antal generationer har skapats. Detta, samt hur många individer som ska finnas i varje befolkning, och hur många enheter som varje lösning består av, ställs in av användaren innan algoritmen börjar köras. När algoritmen är färdig så skrivs information för det totala förloppet (”Epoken”) ut i outputtextfilen.

Ett problem som noterades är att lösningarnas storlek inte är baserade på pris. D.v.s. byggordningarna genereras inte baserat på hur mycket pengar som kan genereras under en match. Därför måste man hålla koll på byggordningarnas storlek kontra enheternas pris. Om byggordningarna är för stora så kommer de sista enheterna aldrig att byggas, vilket gör att de sista instruktionerna i en lösning är värdelösa för algoritmen. Om byggordningarna är för små så kommer många matcher att avslutas utan att alla resurser har använts, om bara billiga enheter används, och då kommer algoritmen att favorisera enstaka dyrare enheter över flera billiga.

Eftersom det är komplicerat att evolvera byggordningar baserat på ekonomiska faktorer så beslutades det att två gränser skulle sättas. Den maximala mängden som kan tjänas in för en spelare under match sattes till 1000 enheter. Samt att ingen enhet får kosta mindre än 50 enheter pengar.

Med de gränserna kan man räkna med att en spelare aldrig kan bygga fler än 20 enheter och under simulationerna så genereras alltså lösningar/byggordningar med exakt 20 enheter.

4.3 Utvärderingsprocessen.

Följande metod kommer att användes för att utvärderaalgoritmens effektivitet:

Spelet testades tio gånger, efter varje gång så analyserades algoritmens output och om något balansproblem upptäcktes så gjordes en ändringar på spelet, i syfte att förbättra spelbalansen. Svaga enheter gjordes starkare, starka enheter gjordes svagare.

Excel användes för att representera resultaten ifrån testerna med hjälp av grafer. Innehållet i Hall of Fame efter varje körning sparades i en separat textfil. Innan testerna började dokumenterades algoritmens inställningar och spelets ursprungliga balans.

Jag använder en dokumenteringsmetod som liknar den som Leigh .m.fl.(2008) använde. De tester som anses mest intressanta (inklusive det första och det sista testet) skrevs ner i slutrapporten. Ändringarna som skedde dokumenterades, resultatet av körningen visas upp, och ett resonemang för hur spelets enheter ska ändras lades upp.

En pilotstudie har utförts för att undersöka hur algoritmen hanterar ett extremfall.

4.3.1 Pilotstudien.

Spelets balanserades så att enheten ”Warrior” skulle bli starkare än de andra enheterna. Warrior gör i det här exemplet 30 skadepoäng, med en attackförsening på 10 uppdateringar, vilket är mycket högre än alla andra enheter.

Enhetsegenskaper:

	Kostnad.	Produktionstid.	HP.	DMG.	Attackförsening.	Räckvidd.
Archer.	200	70	10	5	100	200 pixlar.
Axeman.	100	50	6	8	75	Närkamp.
Crossbowman.	400	100	6	30	200	300 pixlar.
Shieldbearer.	150	80	20	2	120	Närkamp.
Warrior.	100	50	20	30	10	Närkamp.

Spelinställningar:

Maximalt antal pengar som kan genereras = 1000.

Baserna genererade 1 pengaenhet per uppdatering.

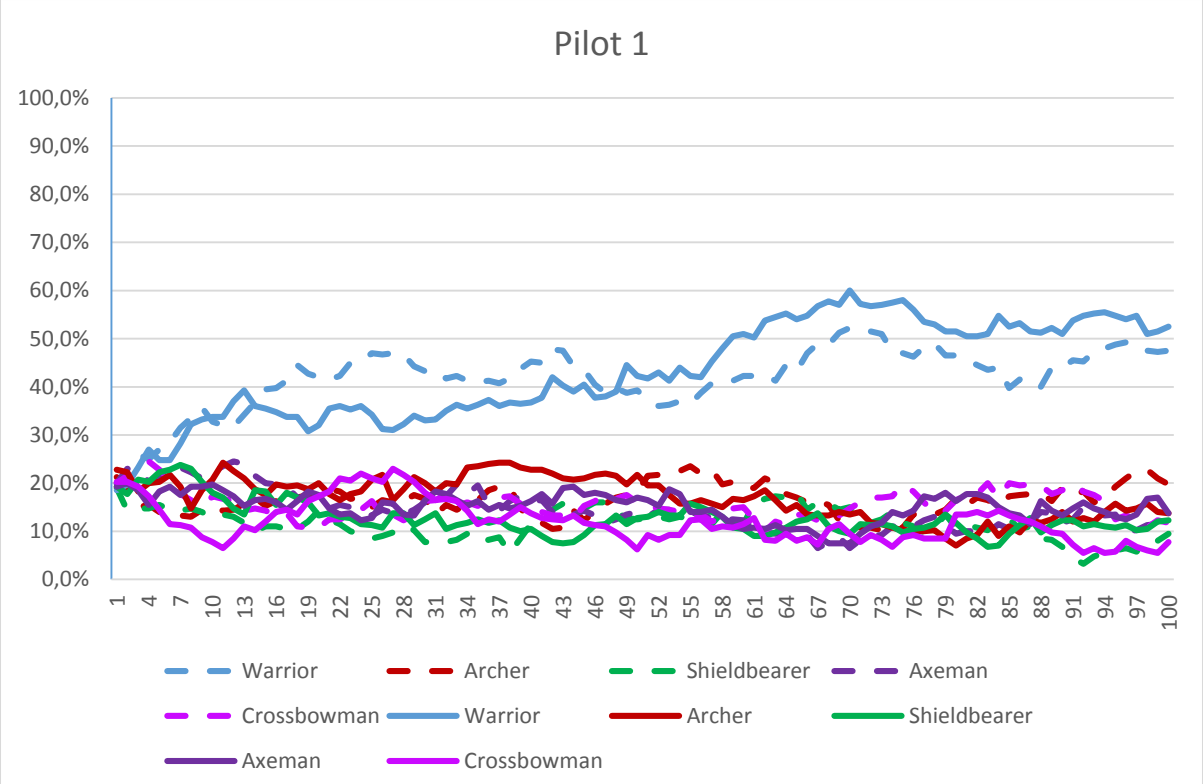
Algoritminställningar:

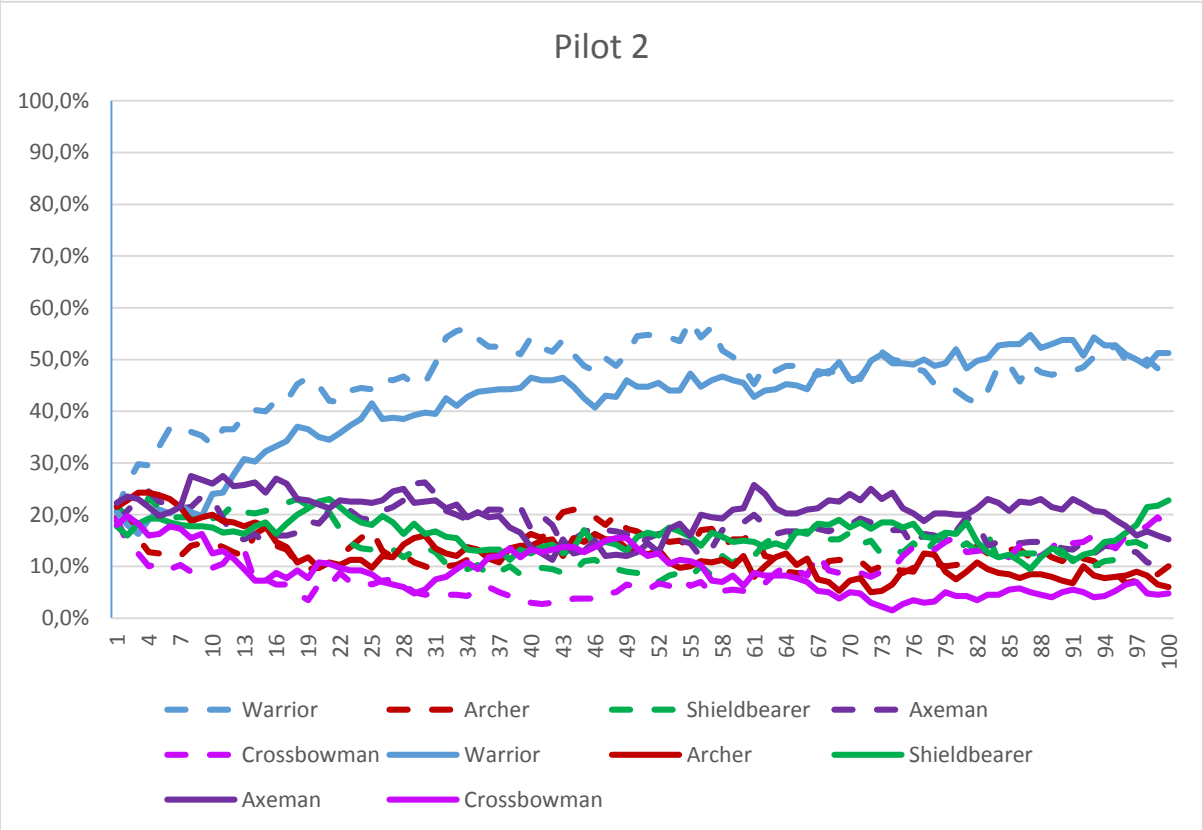
Algoritmen kördes tre gånger, under 100 generationer per gång. Varje befolkning innehöll 20 lösningar varje lösning bestod av 20 enheter.

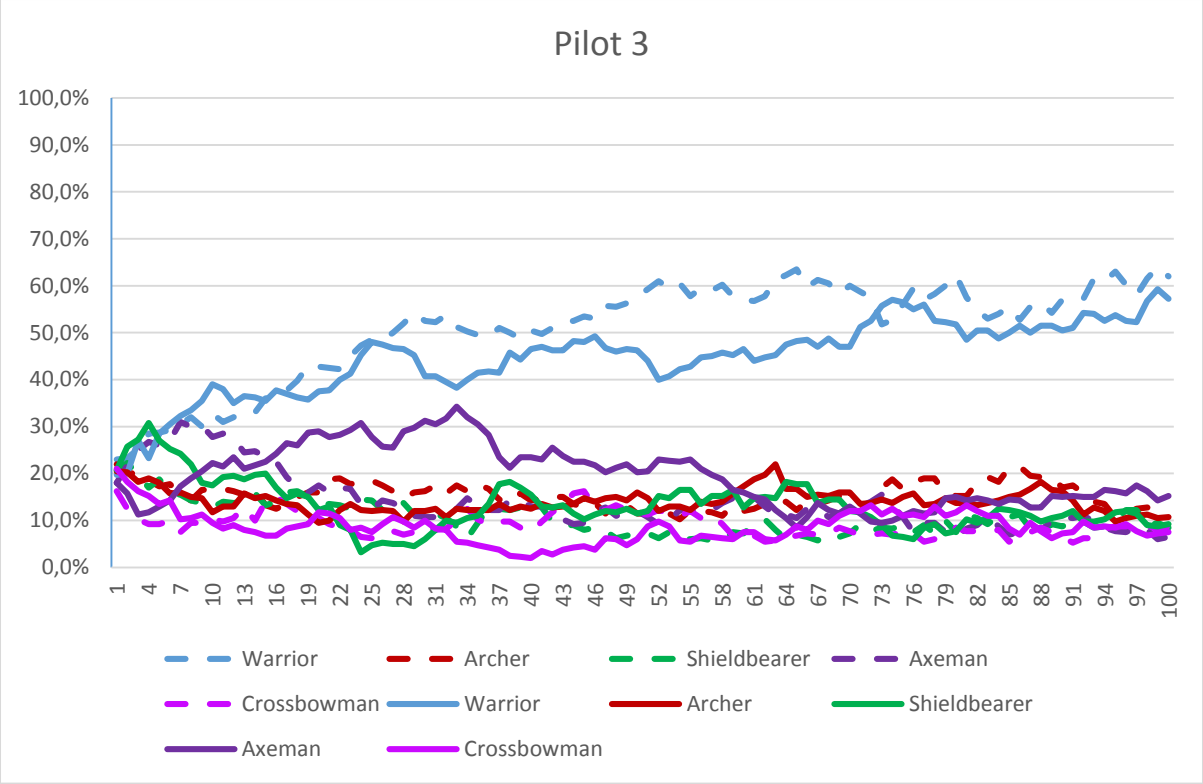
Mutationsrisken var 5 %.

Resultat:

Följande grafer genererades ifrån de tre körningarna. X-axeln är tidsförloppet (d.v.s. antal generationer) och y-axeln är procentuell användning. Solida linjer är ifrån befolkning 1 och streckade linjer är ifrån befolkning 2.


Pilot 1
100,0%
90,0%
80,0%
70,0%
60,0%
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Pilot 2
100,0%
90,0%
80,0%
70,0%
60,0%
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 12 Resultatet av pilotstudiens tre simulationer.

Graferna visar tydligt att algoritmen i alla tre fall lärde sig att använda enheten Warrior i mycket högre utsträckning än andra enheter. Bägge befolkningarna började snabbt att favorisera enheten Warrior, och efterssom ingen motstrategi mot enheten Warrior kunde hittas så fortsatte denna trend igenom hela simulationen.

Analys av de tre simulationernas ”Hall of Fame” visar på att enheten Warrior endast användes 30-40% av gångerna bland de lösningar med allra bäst fitness, vilket är lägre än andelen som används mot slutet av epokerna (c:a 50%).

Slutsats:

Alla tre körningar visar på att algoritmen favoriserar enheten ”Warrior”. Mot slutet av epokerna är användandet av Warrior nära, eller överstigande 50 % av det totala antalet enheter. Flera gånger under simulationerna överskrider användandet detta.

I Hall of Fame favoriseras inte enheten lika mycket. Detta beror troligtvis på att de lösningar som har fått högst poäng råkar vara bland de första lösningarna som använde enheten, och fick höga poäng efterssom dess motståndare inte hade hunnit anpassat sig än.

Slutsatsen är att algoritmen kan hantera extremfall, men eftersom enheten ifråga har mer än tio gånger högre attackskada än någon annan enhet så kan det tyckas att algoritmen borde favorisera enheten ännu mer. Trotts denna brist så ser man fortfarande att enheten Warrior används i mycket högre utsträckning än andra enheter.

Utvärdering

4.4 Presentation av undersökning

Undersökningen bestod av tio iterationer av testningar med den evolutionära algoritmen, enligt den beskrivna metoden. Ett urval av de tio iterationerna som kördes kommer härmed att presenteras, med samma format som pilotstudien.

Följande spelinställningar användes i samtliga simulationer:

Gränsen för hur mycket pengar som kunde genereras under en match sattes till 1000 pengaenheter. Dessa pengar lades till på spelarnas ekonomier i en takt av en pengaenhet per uppdatering.

Följande algoritminställningar användes:

Algoritmen kördes tre gånger, under 100 generationer per gång. Varje befolkning innehöll 20 lösningar, som vardera bestod av 20 enheter. Mutationsrisken sattes till 5 %

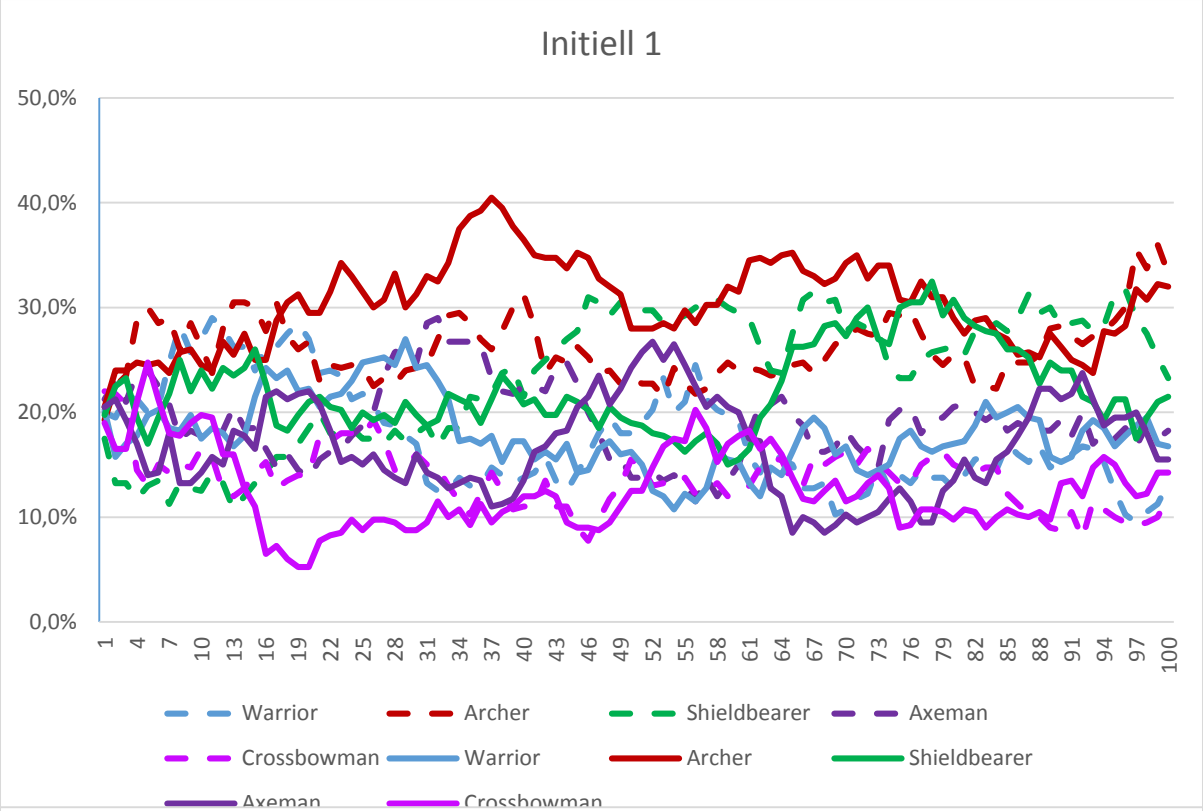
4.4.1 Ursprunglig konfiguration.

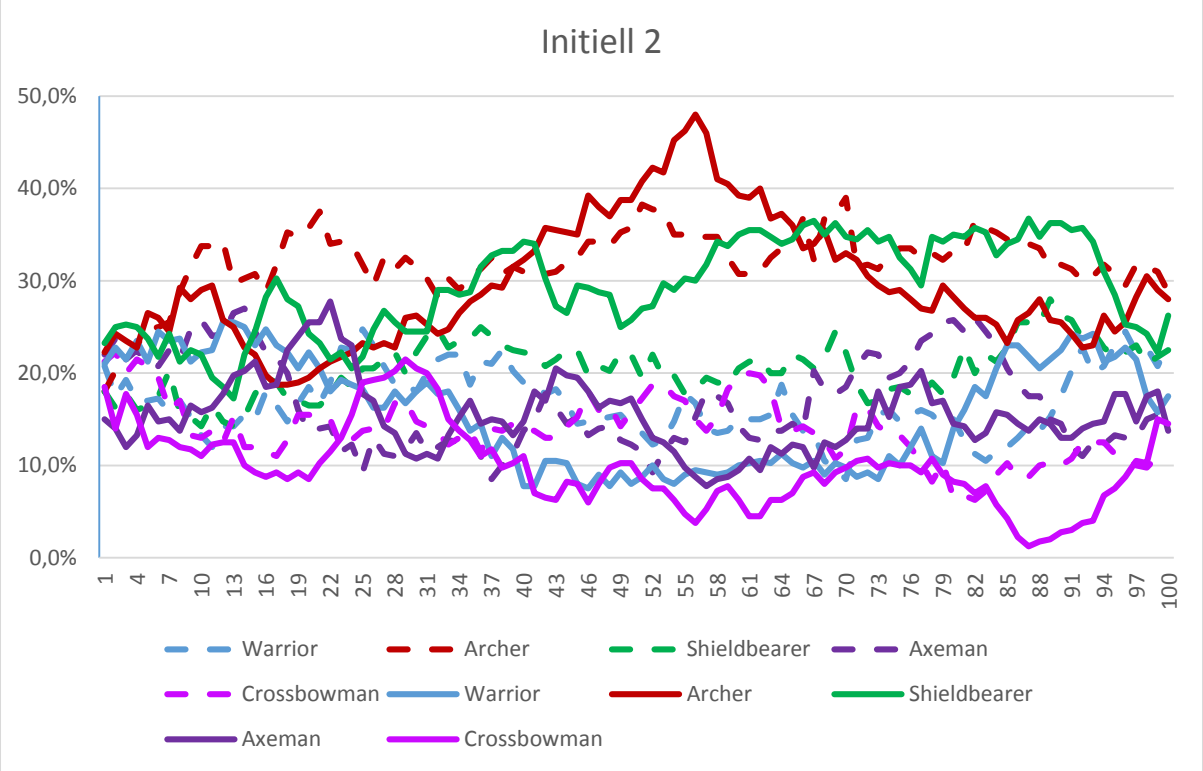
Enhetsegenskaper:

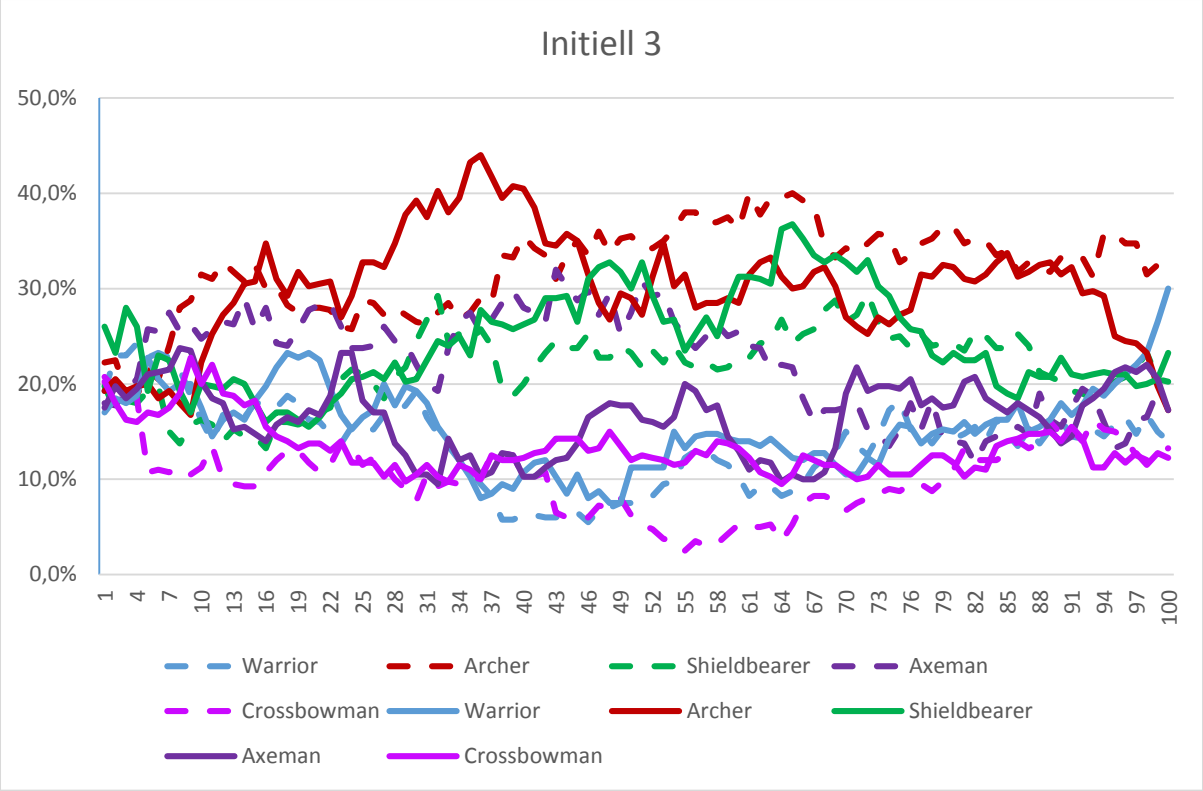
	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	120	200 pixlar.
Axeman.	100	50	7	8	75	Närkamp.
Crossbowman.	400	100	6	35	250	300 pixlar
Shieldbearer.	150	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Enheterna balanserades efter känsla och efter observation av ett antal speltester, för att enheterna skulle uppfylla rollerna som beskrivs i kapitel 4.1.1.

Resultat:


Initiell 1
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Initiell 2
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 13 Resultatet av tre initiala simulationerna.

Enheterna Archer och Shieldbearer är överrepresenterade. Användningen av enheten Archer når nästan 50 % vid enstaka punkter.

Shieldbearer kan vara överanvänd på grund av att denna enhet ska kontra Archer, och det verkar som befolkningarna till viss del anpassar sig genom att använda högt antal Shieldbearers, om deras motståndare använder ett stort antal Archers.

Det är dock svårt att veta om en ökning av antalet Shieldbearers beror på att algoritmen lär sig att enheten är bra på att kontra Archer, eller om algoritmen har lärt sig att enheten bara är allmänt stark.

Analys av simulationernas Hall of Fame pekar på att Archers används ofta i de bästa lösningarna. Med användning på 35 %, 28 % och 40 %. Användningen av Shieldbearer hamnade på endast 15-25%, vilket pekar på att Shieldbearer inte är för stark som enhet, utan bara användes ofta på grund av att den kontrar Archer.

Slutsats:

Archer användes väldigt ofta under de här simuleringarna, följt av Shieldbearer. Eftersom Shieldbearer inte förkom lika ofta i Hall of Fame, som under hela simulationen, så är den troligtvis inte särskilt stark i allmänhet, utan bara en bra kontring till Archer.

Trotts detta är det inte helt säkert att algoritmen har använt Shieldbearer som kontring till Archer, eller om den använder båda enheterna endast för att båda enheterna råkar vara starka.

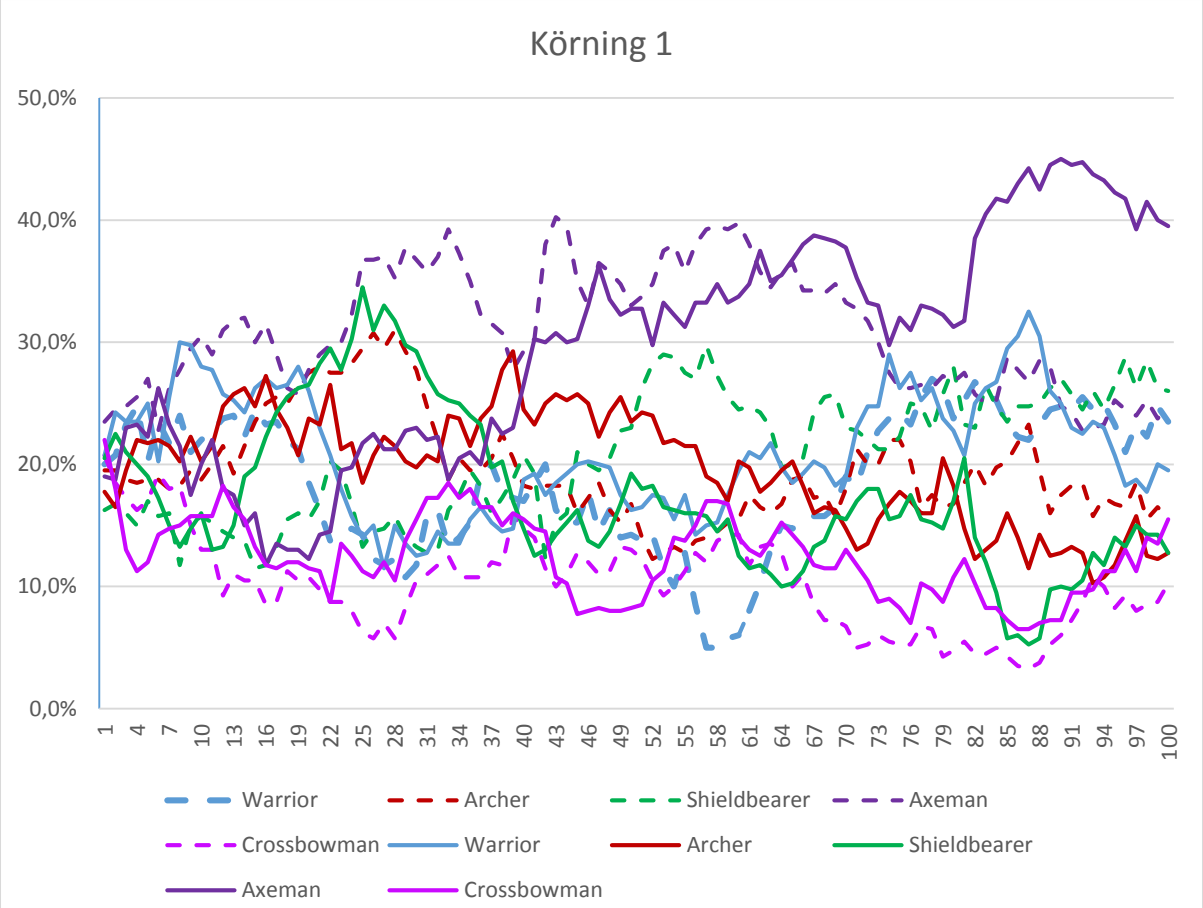
Eftersom analysen tydligt visade att enheten Archer var för stark så sänktes enhetens attackhastighet, ifrån att utföra en attack var 120:e uppdatering, till en var 180:e uppdatering.

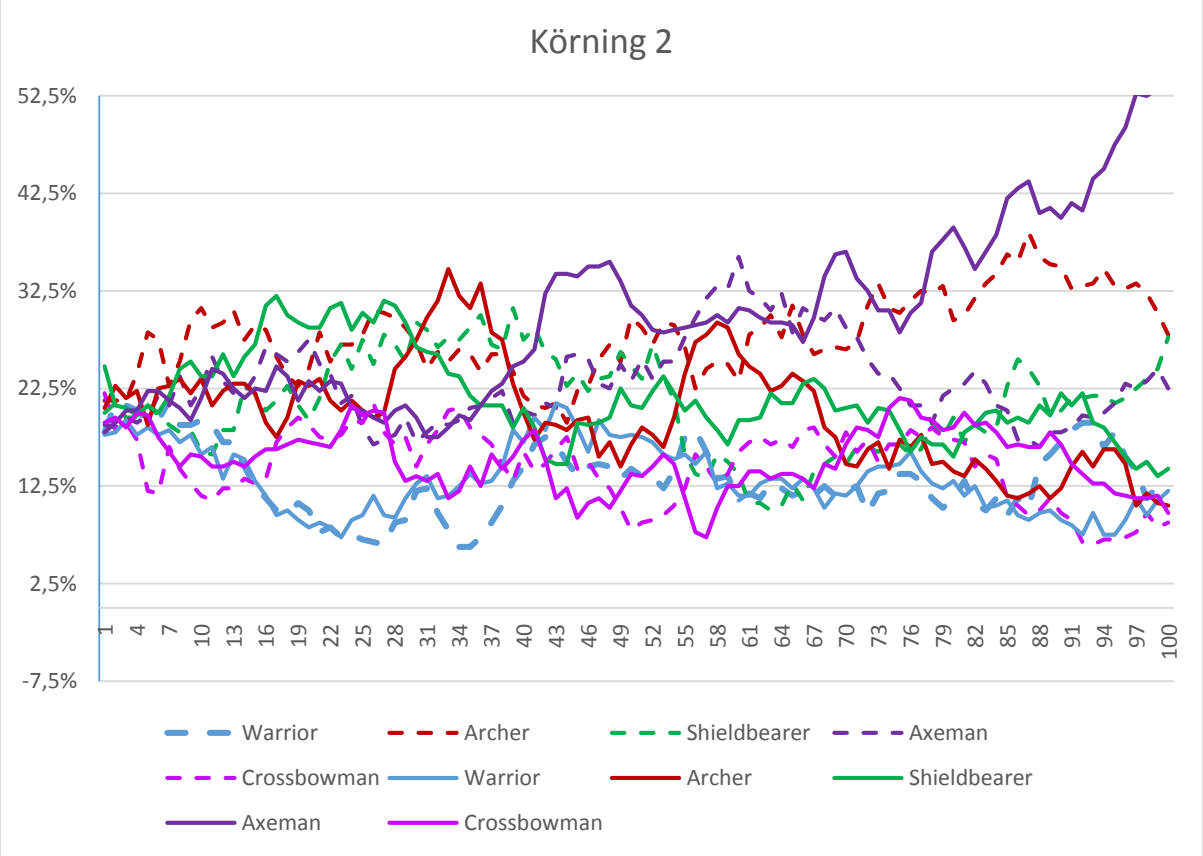
4.4.2 Iteration 1.

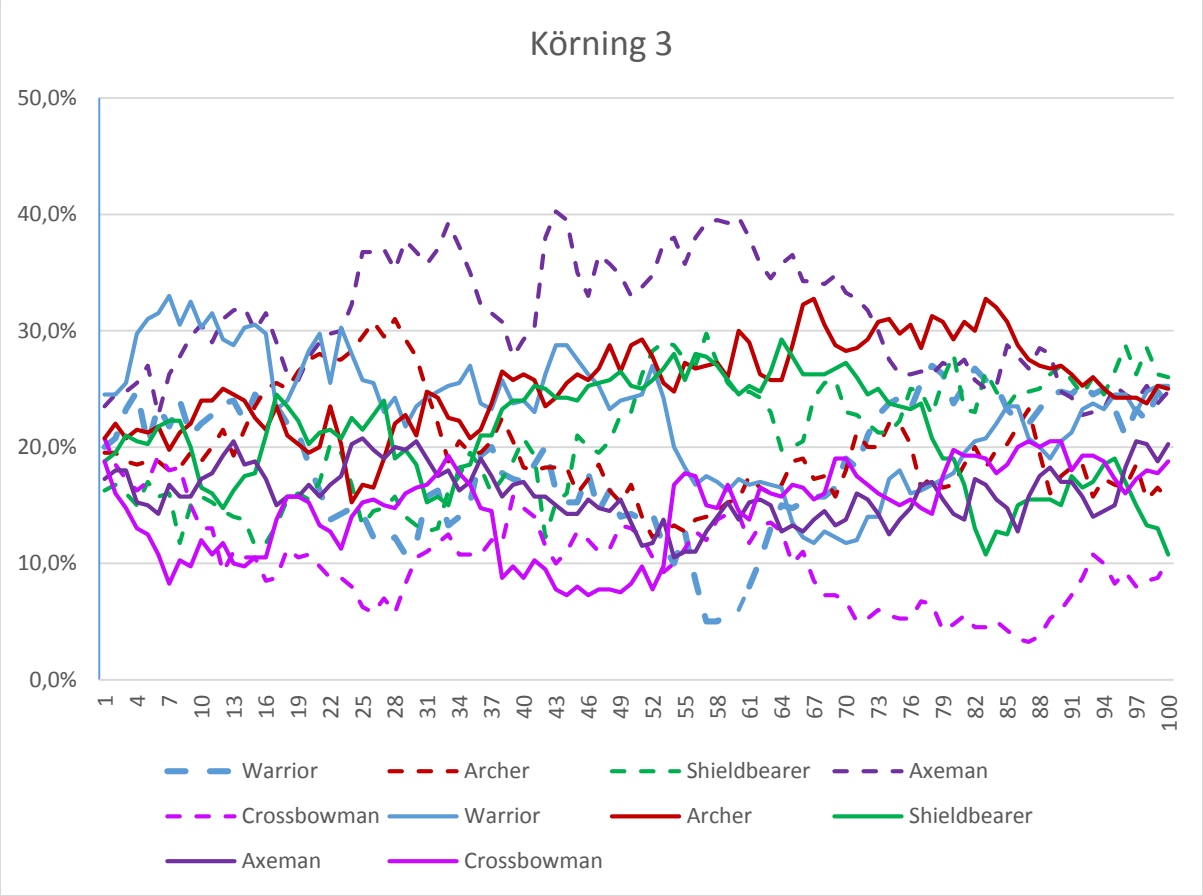
Enhetsegenskaper:

	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	**180**	200 pixlar.
Axeman.	100	50	7	8	75	Närkamp.
Crossbowman.	400	100	6	35	250	300 pixlar
Shieldbearer.	150	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Resultat:


Körning 1
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Körning 2
52,5%
42,5%
32,5%
22,5%
12,5%
2,5%
-7,5%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 14 Resultatet ifrån första Iterationen.

I körning två och tre kan man se att den ena befolkningen upptäckte att Axeman var en stark enhet, och att den andra därför började bygga Archers för att kontra detta. I körning ett kan man se att bägge befolkningarna fokuserade på Axeman.

I Hall of Fame för den ena körningen var Axeman dominant (44 %). I Hall of Fame för den andra användes Archer och Axeman ungefär lika ofta (28.5% och 29.5%), men i högre antal än de andra enheterna. I den tredje körningens Hall of Fame användes Archer oftast (31 %).

Slutsats:

Resultaten varierade kraftigt, men det är tydligt att Axeman och Archer är för starka. Vid ett tillfälle hamnar användningen av Axeman, för en befolkning, på över 50 %!

Eftersom Archer är enheten som kontrar Axeman så är det troligt att den relativt höga användningen av Archer beror på att Axeman är för stark, och att Archer är den bästa kontringen till Axeman. D.v.s. om Axeman försvagas så kommer detta resultera till att även Archer blir svagare.

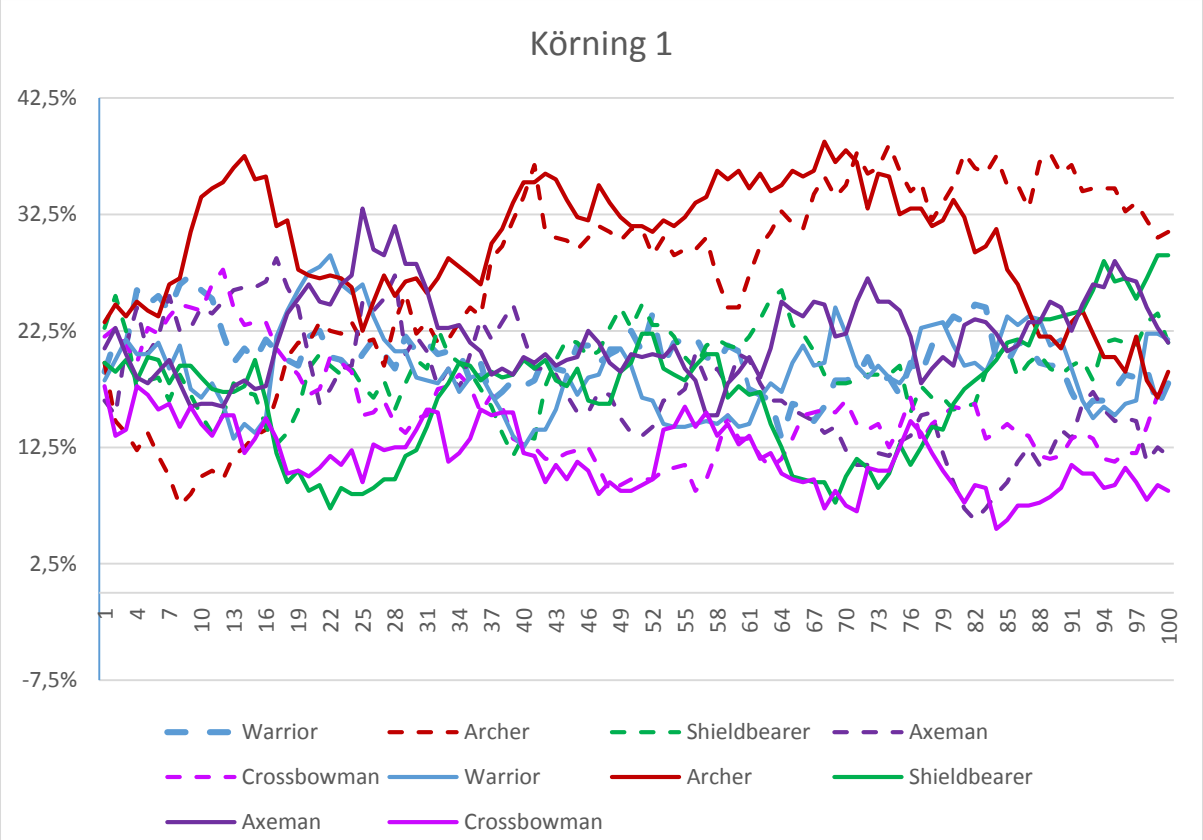
Inför nästa iteration kommer Axeman-dominerade strategier att göras svagare. Eftersom Axeman redan har väldigt låg hälsa, och eftersom Axemans viktigaste roll är att göra så mycket skada som möjligt, så tog beslutet att Axeman ska försvagas genom att ge enheten en prisökning på 20 pengaenheter.

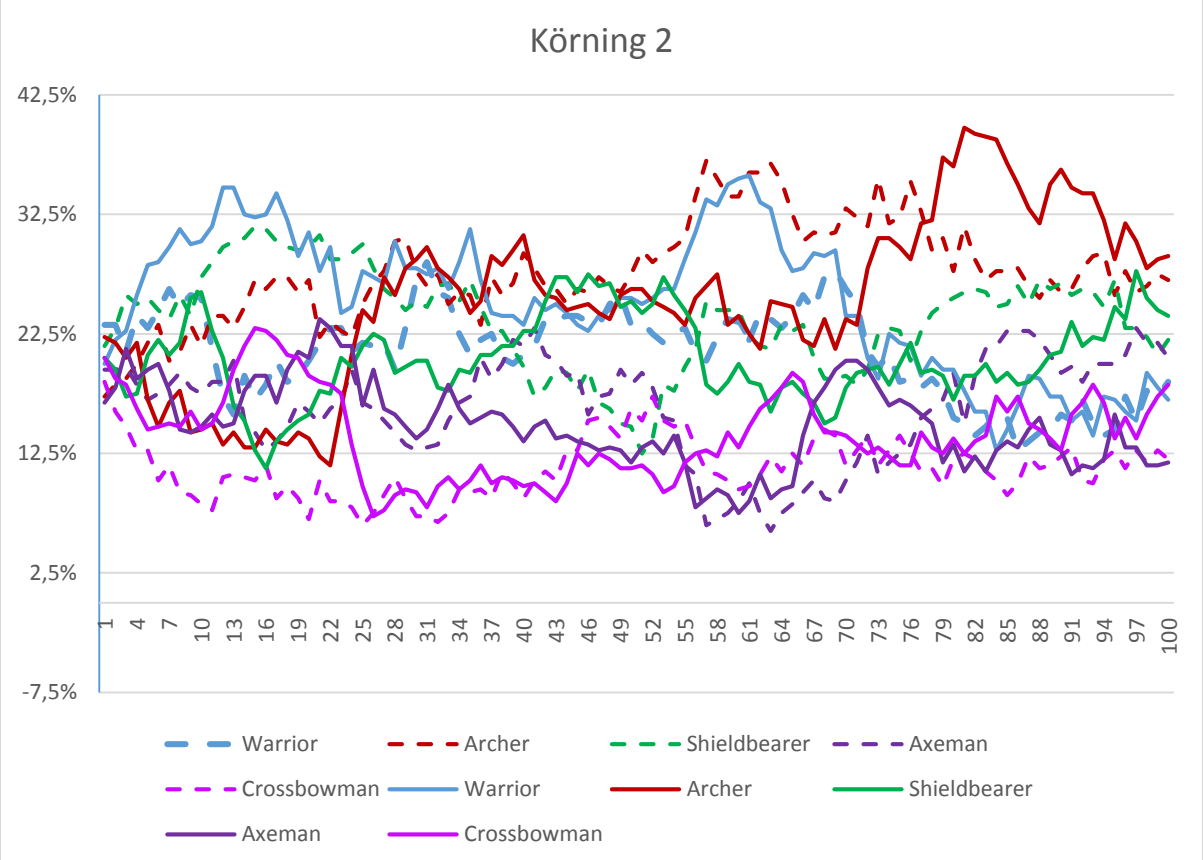
4.4.3 Iteration 2.

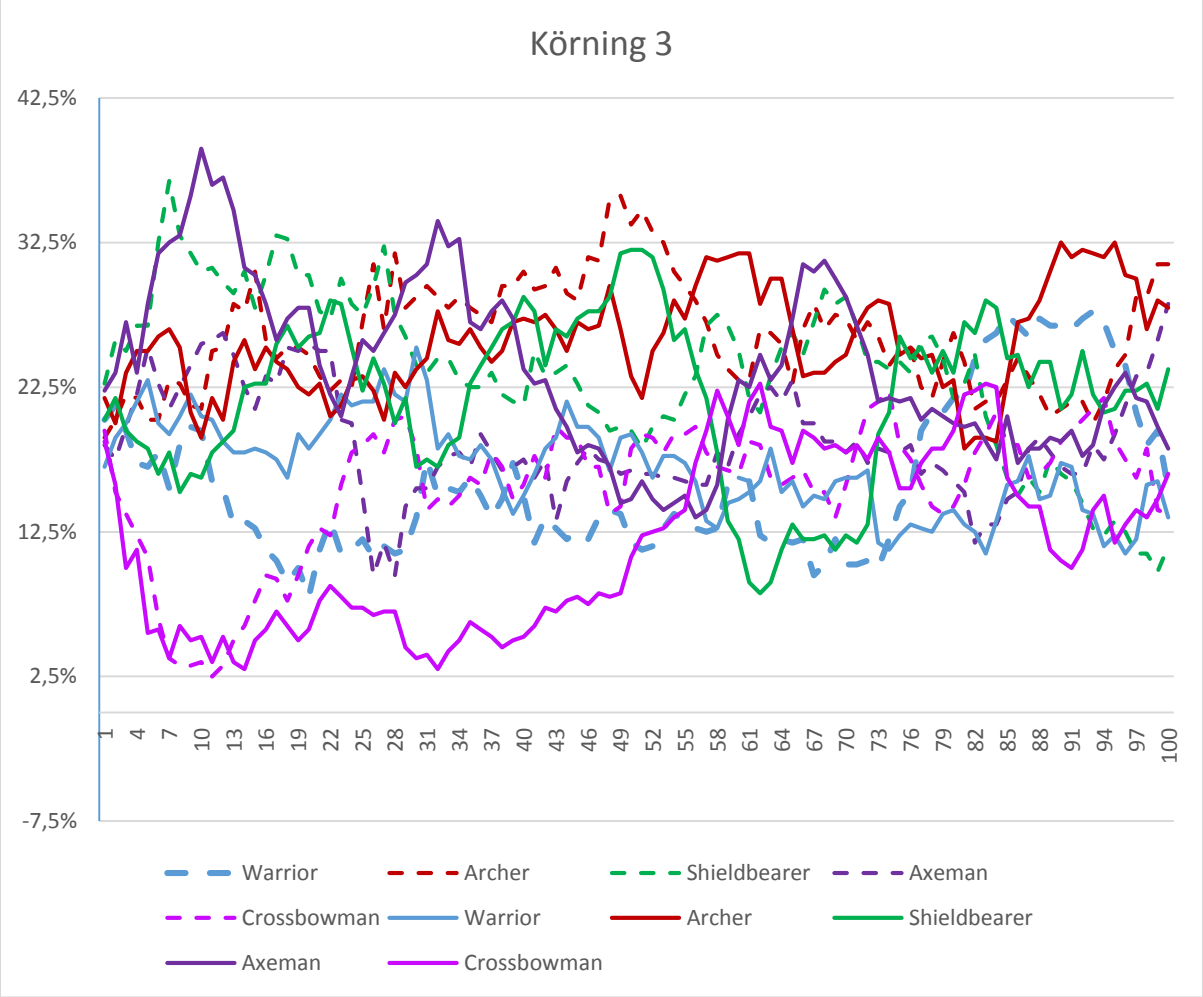
Enhetsegenskaper:

	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	180	200 pixlar.
Axeman.	**120**	50	7	8	75	Närkamp.
Crossbowman.	400	100	6	35	250	300 pixlar
Shieldbearer.	150	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Resultat:


Körning 1
42,5%
32,5%
22,5%
12,5%
2,5%
-7,5%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Körning 2
42,5%
32,5%
22,5%
12,5%
2,5%
-7,5%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 15 Resultatet ifrån den andra iterationen.

Den här gången är variationen hög, många olika enheter hamnar i hög användning i vissa punkter, vilket pekar på att ingen enhet är helt dominant. Dock så kan man se att enheten Archer visar sig vara en väldigt stark enhet, i alla tre körningarna. I körning ett är Archer använd mer än någon annan enhet, för bägge populationerna, under nästan hela körningen. I körning två är Archer och Warrior de mest använda enheterna. Det verkar också som att algoritmen lär sig att kontra Archer med Shieldbearer, eftersom befolkning 1, mot slutet av simulationen ser en ökning i antalet Archer-enheter, och befolkning två samtidigt ser en ökning i antalet Shieldbearers. Även i körning tre ser man detta mönster, runt generation 50, t.ex., är befolkning 2:s användning av Archer väldigt hög, samtidigt som befolkning 1:s användning av Shieldbearer är hög. Utöver detta så är Crossbowman underanvänd, linjen för denna enhet har en tendens att ligga på runt 12-15%.

I Hall of Fame:rna används Archer 36,28 och 29 % av gångerna. Crossbowman används 11.5 , 7 och 17 % av gångerna.

Slutsats:

Balansproblemen blir nu mindre och mindre uppenbara, men vi kan slutleda att enheten Archer är för stark, eftersom denna enhet är överrepresenterad både i graferna och i Hall of Fame. Enheten Crossbowman är för svag, då denna enhet är underrepresenterad i statistiken. I övrigt ser vi bra variation mellan de olika enheterna.

Eftersom det är riskabelt att ändra på mer än en enhet i taget, på grund av man då inte säkert kan veta vilken ändring som påverkade resultatet (Adams & Dormans , 2012) så togs beslutet att endast ett av två balansproblem skulle försökas förbättras inför nästa iteration. Crossbowman-enhetens attackförsening minskades till 200, och Archer lämnades oförändrad, tills vidare.

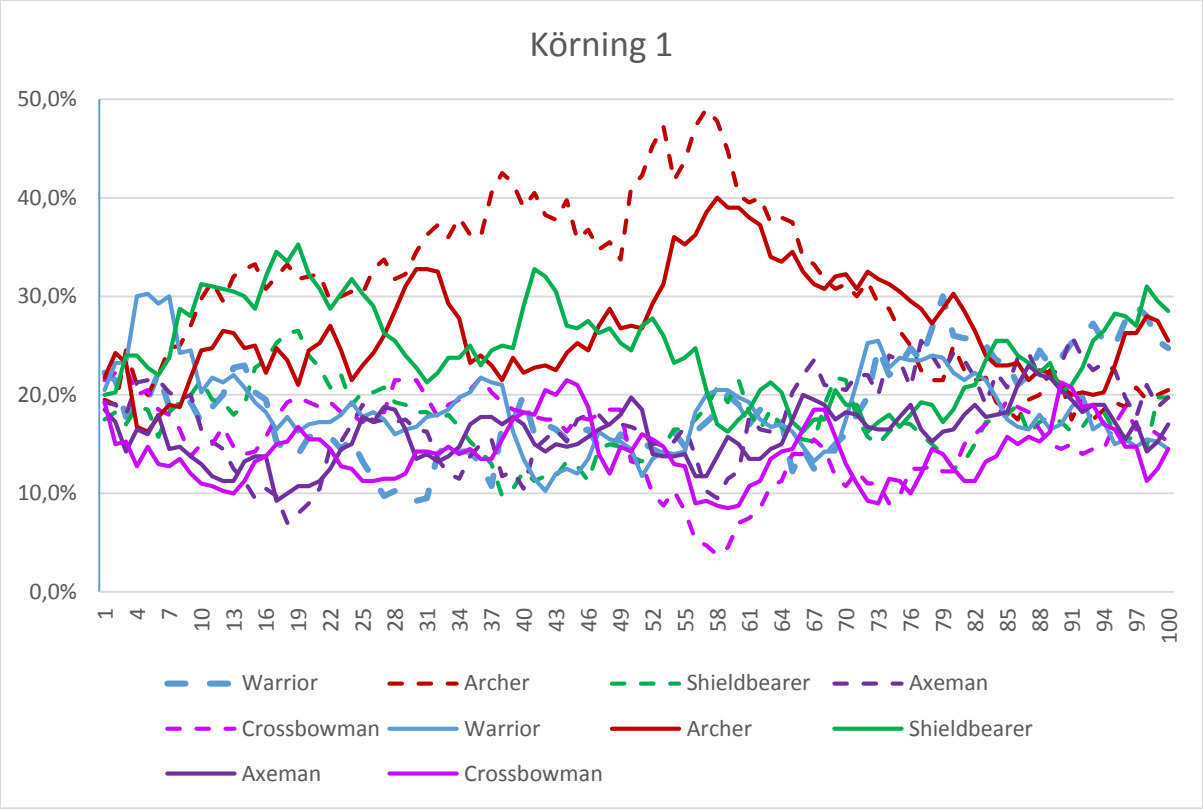
4.4.4 Iteration 3.

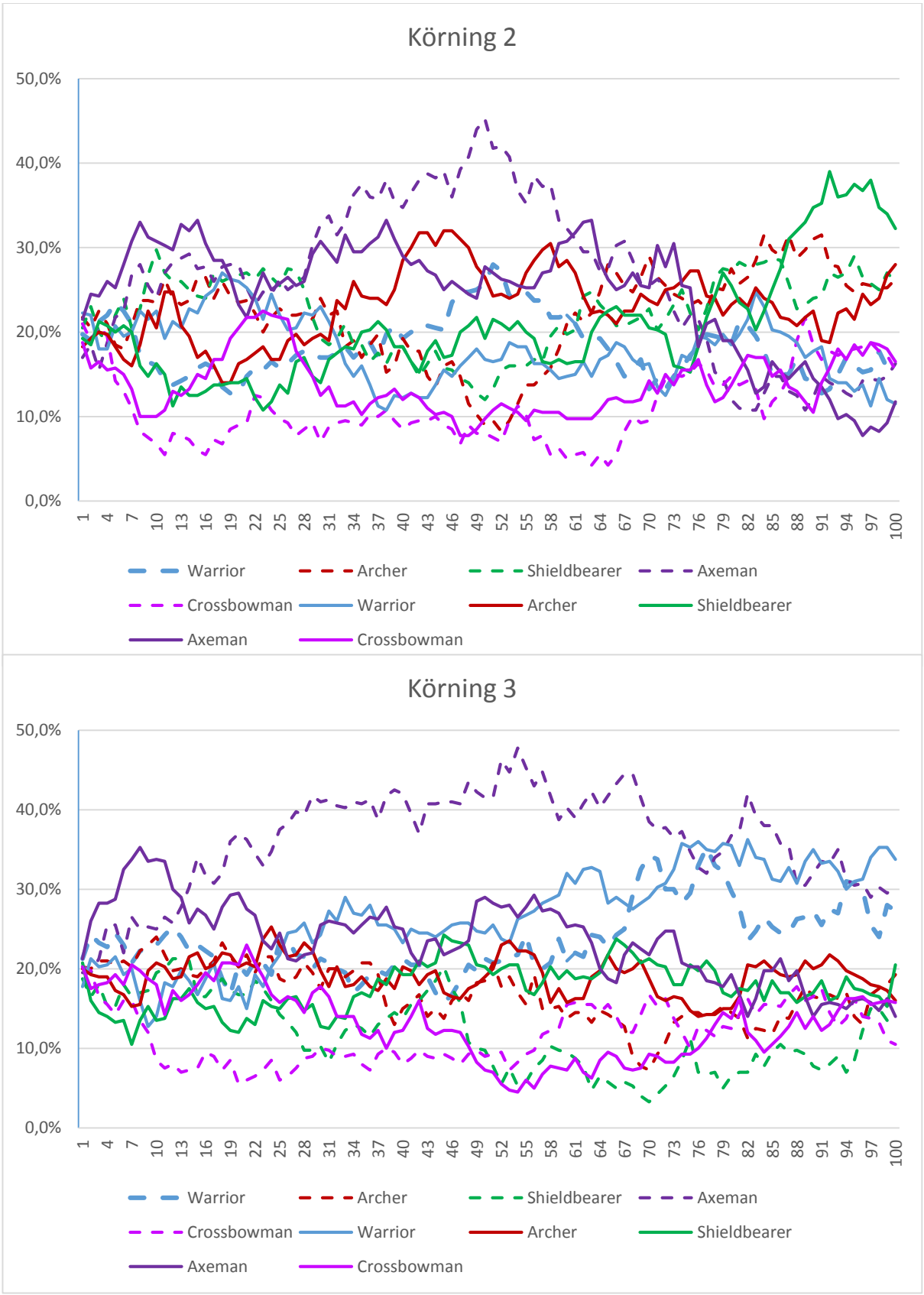
Efter att ha kört algoritmen visade det sig att denna ändring i Crossbowman-enheten inte räckte för att se en skillnad. Ännu en ändring gjordes därför, utöver den tidigare förändringen. Crossbowman-enhetens attackskada ökades till 40.

Enhetsegenskaper:

	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	180	200 pixlar.
Axeman.	120	50	7	8	75	Närkamp.
Crossbowman.	400	100	6	**40**	**200**	300 pixlar
Shieldbearer.	150	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Resultat:


Körning 1
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 16 Körningarna för Iteration 3.

Denna iteration lades med i slutrapporten eftersom den visar hur hög variation algoritmen ger när spelets balans är i detta stadie. Archer är dominant i körning 1. Axeman dominant i körning 2 och, till mindre del, i korning 3, men Warrior används också mkt i slutet av 3. Vi verkar ha nått en punkt där olika enheter kan vara dominanta under olika körningar. Crossbowman är fortfarande kraftigt underanvänd, användningen av denna enhet faller ibland under 10 %, och är nästan aldrig över 20 %.

Hall of Fame för de tre körningarna har också väldigt skilda sammansättningar ifrån varandra. I första körningens Hall of Fame är Archer dominant, med 36 % användning. I den tredje körningen är Axeman dominant (36.5%), men i körning två är bägge enheterna nästan lika välanvända (Archer, 25 %, Axeman 29.5%).

Slutsats:

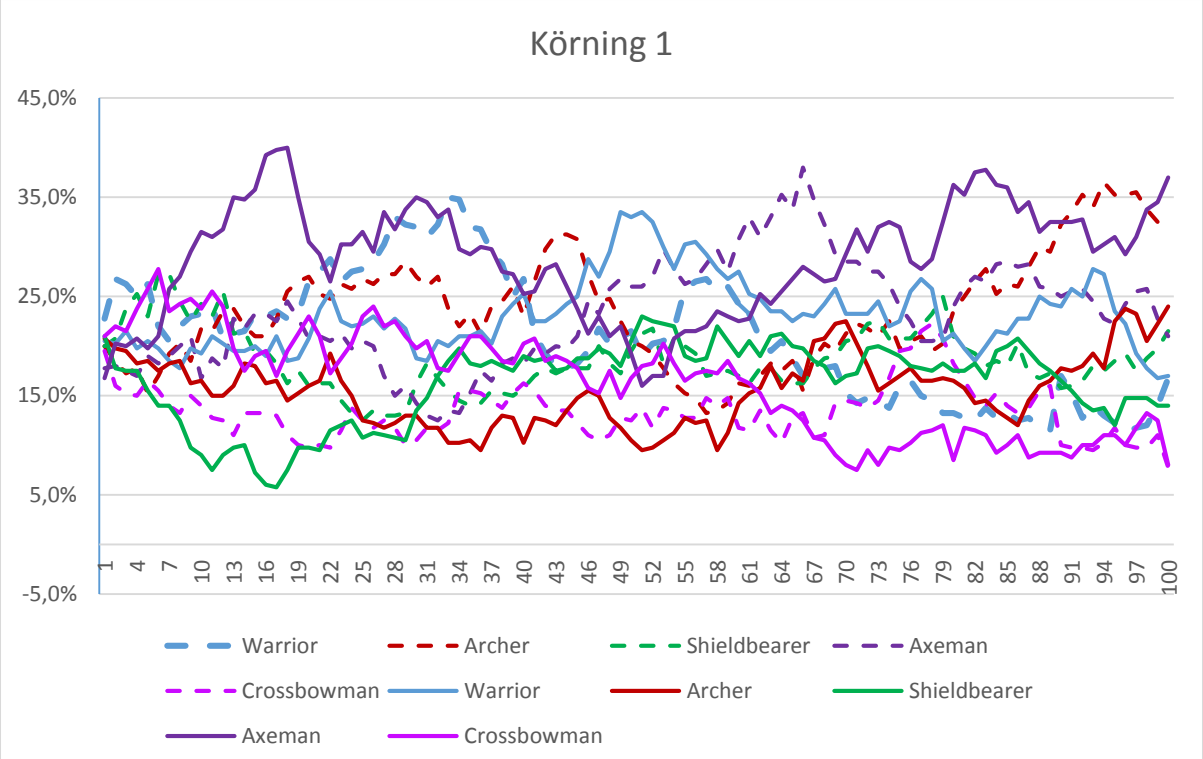
Det är inte enkelt att dra en slutsats med så varierande resultat, både i och mellan de olika körningarna. Det enda de tre körningarna har gemensamt är att Crossbowman fortfarande är underanvänd. Det verkar enklast att fortsätta balansera Crossbowman, och se hur detta påverkar simulationen. Crossbowmans kostnad minskas till 350 pengaenheter.

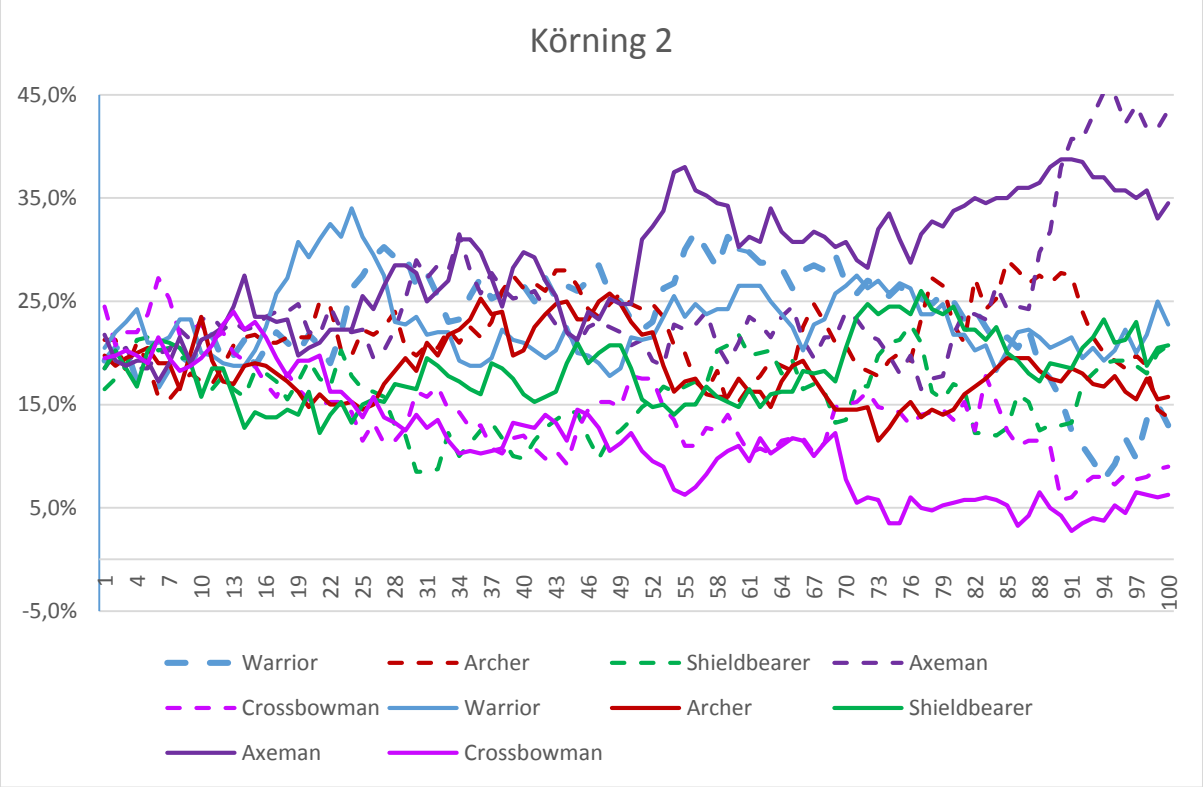
4.4.5 Iteration 4.

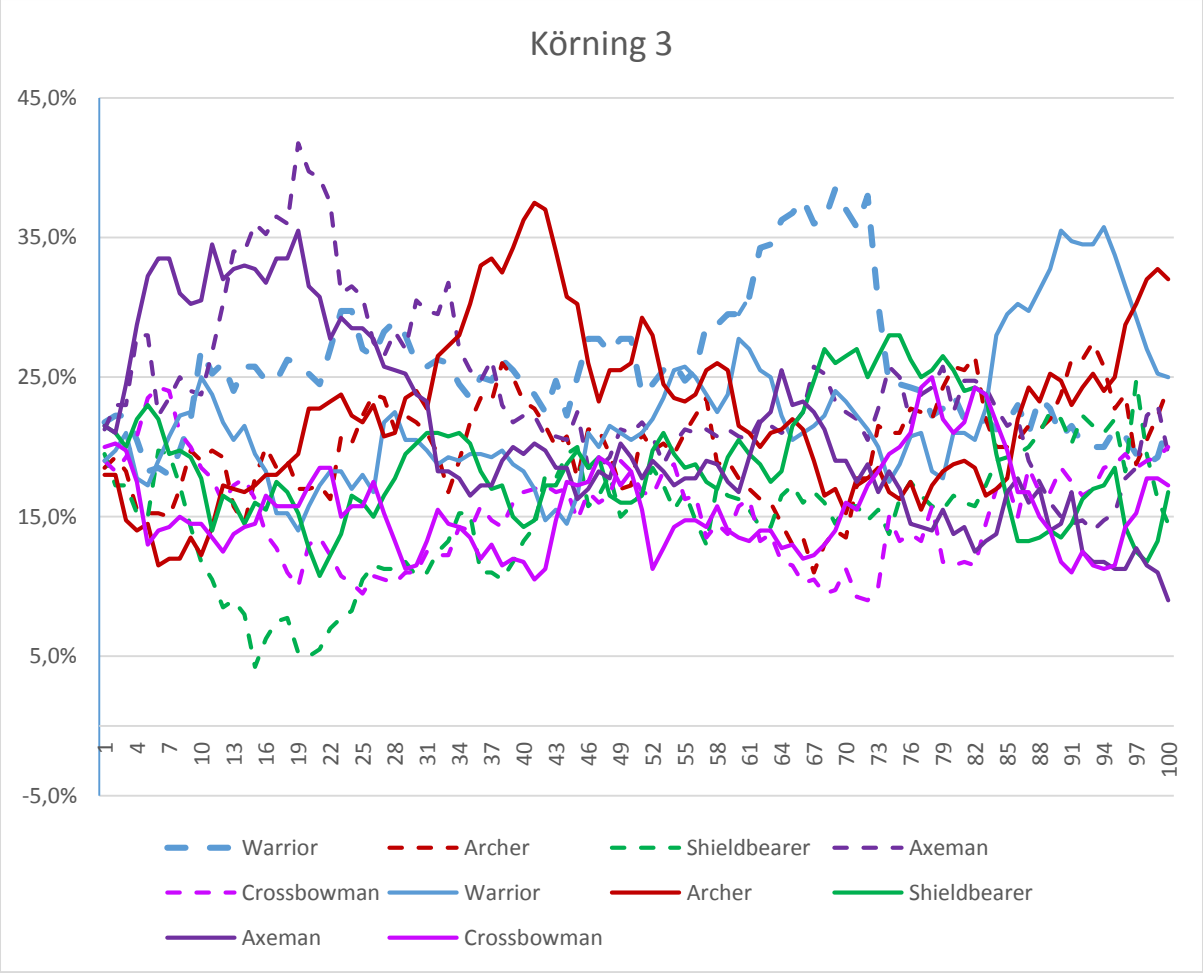
Enhetsegenskaper:

	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	180	200 pixlar.
Axeman.	120	50	7	8	75	Närkamp.
Crossbowman.	**350**	100	6	40	200	300 pixlar
Shieldbearer.	150	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Resultat:


Körning 1
45,0%
35,0%
25,0%
15,0%
5,0%
-5,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Körning 2
45,0%
35,0%
25,0%
15,0%
5,0%
-5,0%
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 17 Resultat av körningarna i iteration 5.

Axeman, Warrior och, till mindre del, Archer används flitigt I alla tre körningarna. Crossbowman och Shieldbearer blir relativt oanvända. I körningarnas Hall of Fame:ar överanvänds Axeman (40 %, 28 % och 27.5%), medan enheten Warrior överanvänds något (23 %, 23 % och 29 %).

Slutsats:

Eftersom Crossbowman är menad för att kontra Shieldbearer så bör en förstärkning av enheten Shieldbearer leda till en ökad användning av bägge dessa enheter. Eftersom Shieldbearer ska ha låg attackskada så kommer inte denna egenskap att ändras, och eftersom meningen är att en Crossbowman fortfarande ska kunna slå ut en Shieldbearer med endast ett skott så kan inte Shieldbearers hälsopoäng ökas. Istället fick enheten en prissänkning, från 150 pengaenheter till 120.

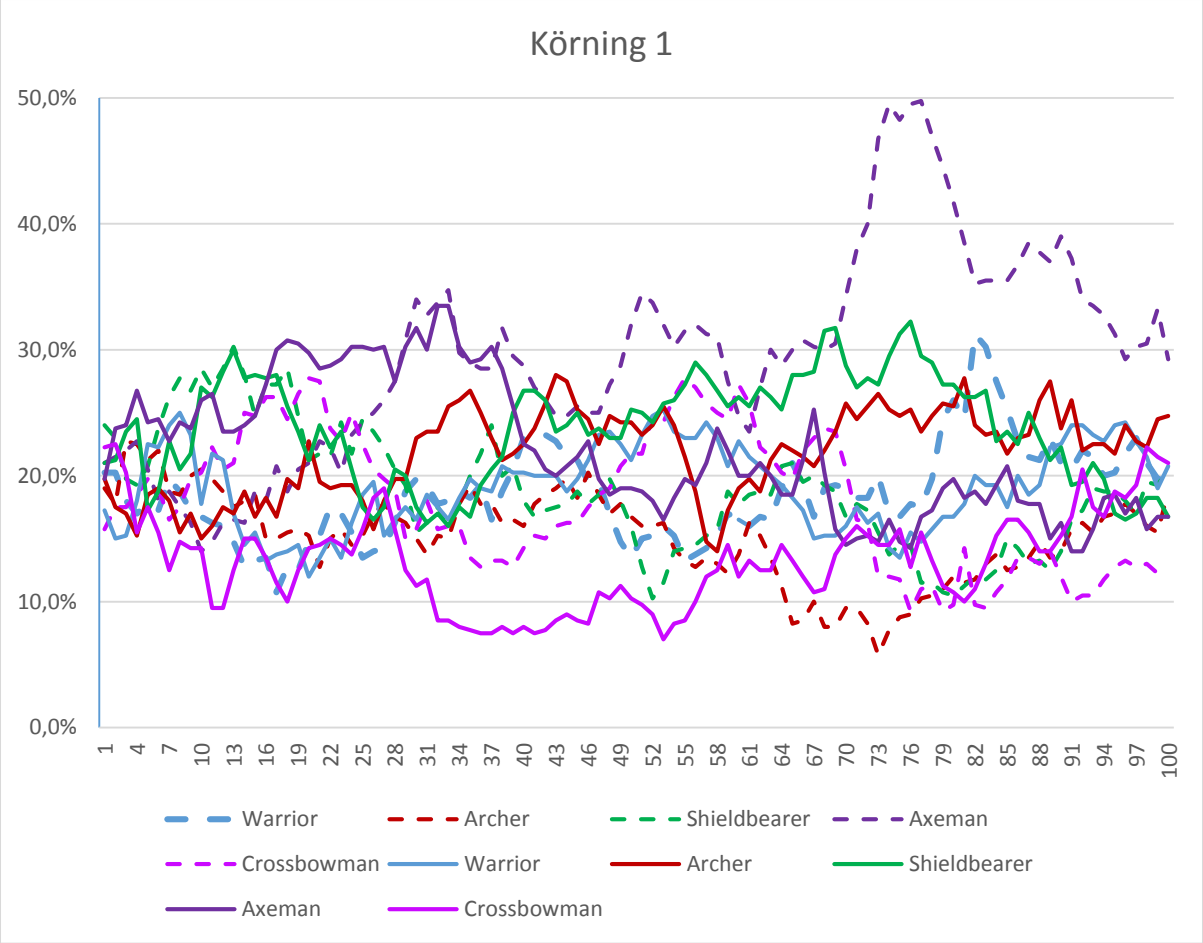
4.4.6 Iteration 5 – Slutgiltig konfiguration.

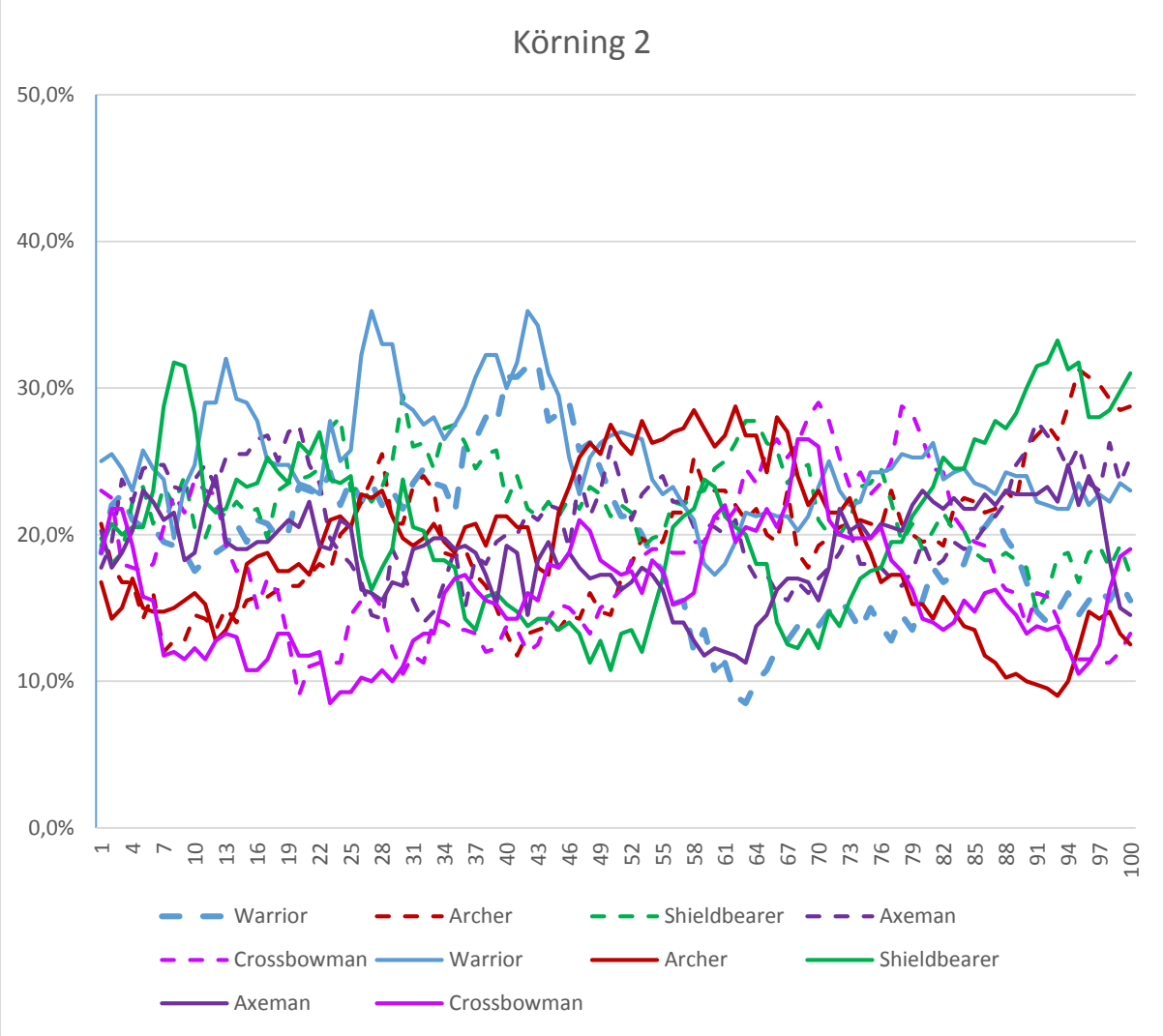
Enhetsegenskaper:

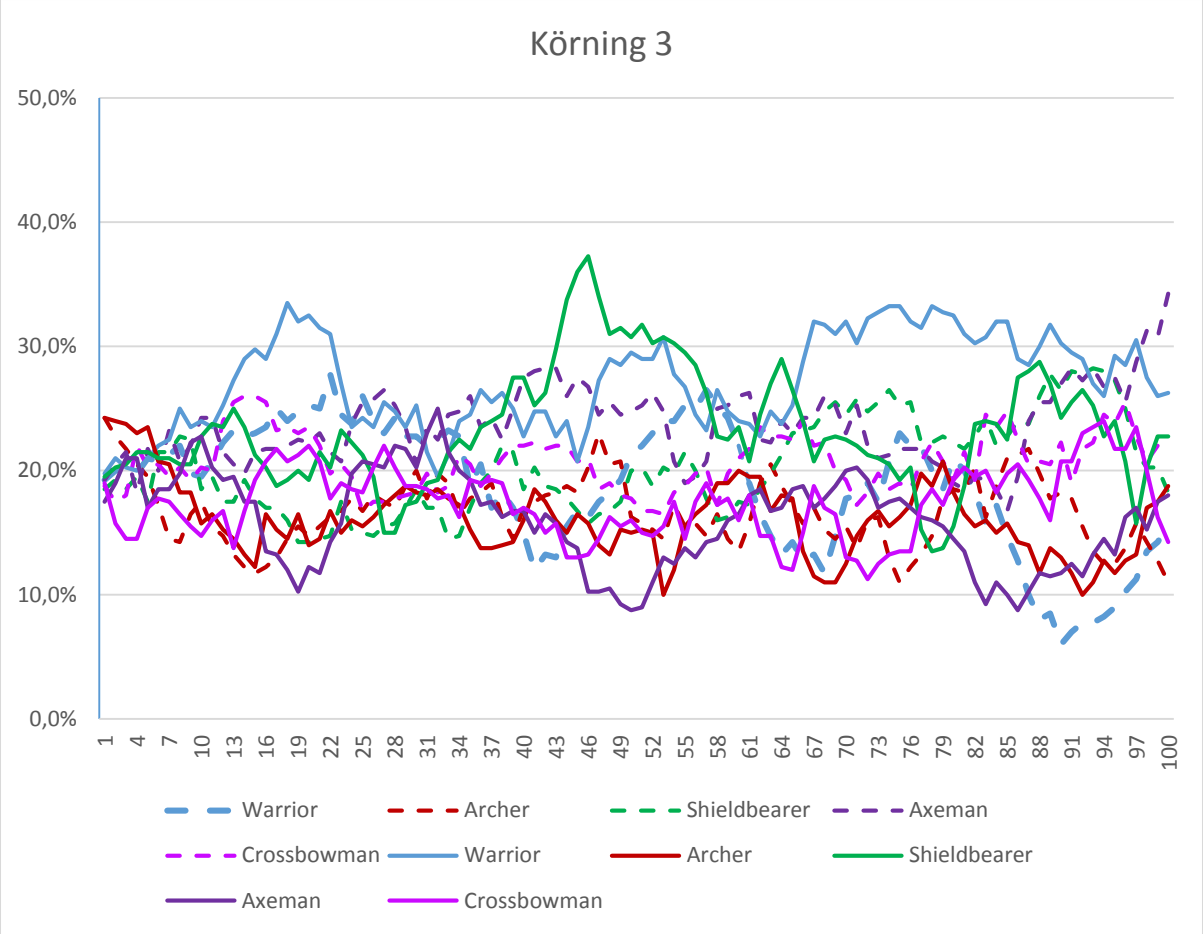
	Kostnad.	Produktionstid.	HP.	DMG.	”Attackdelay”.	Räckvidd.
Archer.	150	50	6	5	180	200 pixlar.
Axeman.	120	50	7	8	75	Närkamp.

Crossbowman.	350	100	6	40	200	300 pixlar
Shieldbearer.	**120**	50	40	2	140	Närkamp.
Warrior.	100	50	12	6	120	Närkamp.

Resultat:


Körning 1
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman


Körning 2
50,0%
40,0%
30,0%
20,0%
10,0%
0,0%
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
Warrior
Archer
Shieldbearer
Axeman
Crossbowman
Warrior
Archer
Shieldbearer
Axeman
Crossbowman



Figur 18 De tre körningarna för iteration 5.

I körning 1 använder sig befolkning ett av många Axeman, och det verkar som om befolkning 2 anpassar sig genom att bygga flera Archers och Crossbowmen. I körning två ser vi hög användning av alla enheter under olika perioder och i körning tre likaså. Crossbowman används nu betydligt oftare än innan, och detta verkar ha skett utan att förstärkningen av Shieldbearer har gjort denna för dominant. Det finns punkter där Shieldbearer används ofta och det finns punkter där den används mindre ofta, ungefär som andra enheter.

Slutsats:

Det blir luddigare och luddigare ju mer arbete som läggs ner på balanseringen, vi verkar ha nått ett läge där alla enheter periodvis används ofta, och periodvis används mer sällan. Det verkar inte som att någon enhet är grovt överrepresenterad, eller grovt underrepresenterad, så det är svårt att lista ut vad nästa steg skulle vara. Man skulle kanske kunna stärka Crossbowman igen, men inte ens denna enhet är uppenbart underrepresenterad.

Avslutande diskussion

4.5 Sammanfattning

Målet var att balansera ett strategispel med hjälp av en evolutionär algoritm, för att undersöka hur evolutionära algoritmer kan användas under balansprocessen för ett datorspel. Algoritmen använde sig av rouletthjulsselektion, envägsöverkorsning, samevolution mellan två befolkningar, och elitism. Spelet i fråga är ett simpelt strategispel som utvecklades specifikt för detta arbete.

Vid utvärderingen genererades ett antal grafer och textfiler med information om hur ofta de olika enheterna i spelet användes under algoritmens förlopp och denna information användes för att ta beslut om hur spelbalansen kunde förbättras.

Slutsatsen är att genom att koppla ett strategispel till en evolutionär algoritm och analysera tusentals evolverade spelstrategier har spelet till viss del balanserats. Spelet är nu rättvist till en sådan grad att inga uppenbara balansproblem längre kvarstår. Däremot är det tveksamt om denna metod effektivt kan användas för att finjustera balansen ytterligare, eftersom det blir svårare att tolka informationen när spelets balans blir bättre och bättre.

4.6 Diskussion

Eftersom ”bra spelbalans” är väldigt subjektivt så kommer det alltid att finnas olika sätt att tolka resultaten ifrån en sådan här undersökning. Det finns dock en del mer objektiva brister som påverkar resultatets trovärdighet.

Det mest uppenbara är att informationen ifrån körningarna kan tolkas väldigt olika ifrån person till person. Om har och göra med ett extremfall så är det lätt att se vilken enhet som bör justeras, men när spelets balans blir bättre så blir det svårt att analysera graferna, eftersom användandet av de olika enheterna då börjar variera väldigt mycket, och det blir svårare att upptäcka balansproblem. Det hade kanske varit en bra idé att parallellt med graferna använda någons sorts heatmap i stil med vad Leigh m.fl. (2008) använde.

Byggordningarna som genererades var alltid 20 enheter stora, oavsett om det fanns pengar för att bygga alla 20 enheter, eller inte. Detta innebar att många enheter aldrig byggdes, vilket leder till två brister.

För det första så bidrar inte de obyggda enheterna till en individs fitness, men dessa enheter har fortfarande lika hög chans att föras vidare till individens barn som de som verkligen byggdes. Detta minskar algoritmens effektivitet, eftersom en del värdelös information (oanvända enheter) för över ifrån generation till generation, på bekostnad av enheter som faktiskt används. För det andra så räknades även dessa ”värdelösa” enheter in när graferna och resten av informationen som utvärderingen är baserad på genererades.

En annan brist, jämfört med Leigh m.fl. (2008) är att i denna metod användes Hall of Fame endast för att undersöka strategierna som fått högst poäng, medan man i deras arbete testade befolkningarna mot individer ifrån Hall of Fame, så att dessa lösningar hade en aktiv roll under evolveringen.

Spelet som har skapats har vissa brister jämfört med konventionella strategispel. Enheterna kan inte ”svärma”, på grund av att alla enheter endast rör sig i en rak linje. D.v.s. flera närkampsenheter kan inte attackera samma fiende, utan måste vänta på sin tur. Många billiga enheter skulle kanske ha större chans att vina mot ett fåtal dyra, om de kunde omringa dessa och därför ha flera enheter som delar ut skada samtidigt. Dessa förenklingar infördes för att förenkla AI:n till spelet, så att inga komponenter än den evolutionära algoritmen behövdes för att en dator ska kunna spela spelet. Ponsen m.fl. (2006) har dock visat att liknande evolutionära algoritmen kan användas i mer komplicerade spel, förutsatt att de används i samband andra former av AI.

Eftersom algoritmen kan simulera tusentals matcher på mindre än tjugo minuter så verkar det troligt att metoder liknande denna med fördel kan användas i samband med mänsklig speltestning för stora spelföretag. Detta bör kunna korta ner tiden det tar att balansera ett spel, och öka kvalitén på spelen i fråga, genom att använda den stora mängden data som genereras av algoritmen för generella balansjusteringar. Därmed behöver man inte förlita sig helt på mänsklig speltestning, som är dyrare och mer tidkrävande (Leigh, Shonfeld & Louis (2008)).

4.7 Framtida arbete

I nuläget utför ändringar i enhetsbalansen manuellt genom att ändra i källkoden, men det skulle vara möjligt att låta algoritmen automatiskt justera t.ex. enheternas pris baserat på hur ofta de används. På så vis kan enheter som används ofta höjas i pris tills det deras användning ligger på samma nivå som andra enheter, och enheter som används sällan kan sänkas i pris till de används i samma utsträckning som andra enheter. Det kan också bli enklare att analysera simulationerna om fler grafiska hjälpmedel används i samband med graferna, t.ex. heatmaps.

En annan prioritet är att förbättra algoritmen så att den tar enheternas pris i åtanke när den genererar lösningar (så man slipper oanvända enheter i slutet på varje byggordning), och göra så att lösningarna i Hall of Fame fortsätter vara inblandade i evolutionsprocessen.

Referenser

Adams, E. & Dormans, J. (2012) *Game Mechanics: Advanced Game Design. 1st edition.* Berkeley, CA: New Riders Games.

Buckland, M. (2002) *AI Techniques for Game Programming. 1st edition.* Cincinnati, OH: Premier Press.

Črepinšek, M., Liu, S-H. & Mernik, M. (2013) Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR) Volume 45 Issue* 3 *Article No. 35. doi: 10.1145/2480741.2480752.*

Hassan A, S. & Rafie, M. (2010) A Survey of Game Theory Using Evolutionary Algorithms. *2010 International Symposium in Information Technology (ITSim). Volume 3 s. 1319-1325. doi: 10.1109/ITSIM.2010.5561648*

Jaffe, A., Miller, A., Andersen, E., Liu, Y., Karlin, A., & Popović, Z.(2012) Evaluating Competitive Game Balance with Restricted Play. *Proceedings, The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. s. 26-31.*

Leigh, R., Schonfeld, J. & J. Louis, S. (2008) Using Coevolution to Understand and Validate Game Balance in Continuous Games. *Proceedings of the 10th annual conference on Genetic and evolutionary computation.* s. 1563-1570. doi: *10.1145/1389095.1389394.*

Obtiko, M. (1998) Crossover and Mutation. Tillgänglig på Internet: http://www.obitko.com/tutorials/genetic-algorithms [Hämtad februari 11 2013]

Ponsen, M., Muños-Avilla, H., Spronk, P. & W. Aha, D. (2006) Automatically Generating Game Tactics through Evolutionary Learning. *AI Magazine Volume 27 Number 3. doi: http://dx.doi.org/10.1609/aimag.v27i3.1894.*

The Wargus Team(2002-2011) *Wargus* (Version 1.0) [Datorprogram]. The Wargus Team. Tillgänglig på Internet: http://wargus.sourceforge.net/index.shtml [Hämtat februari 20, 2013]

Westwood Studios(1995) *Command & Conquer* (Version 1.0) [Datorprogram]. Virgin Interactive.

Appendix A - Designdokument etc.

[Appendix ska fungera som referenslistan - dvs det ska finnas referenser till den från texten. Appendix ska inte vara numrerade utan ska namnges med: Appendix A, Appendix B osv. De ska vara sidnumrerade (I, II, III ...) men de ska inte finnas med i innehållsförteckningen. Varje nytt appendix ska börja på toppen av sidan.]