Examensarbete

HÖGSKOLAN I SKÖVDE

FRAMTAGNING AV BYGGORDNINGAR FÖR RTS-SPEL MED HJÄLP AV SAMEVOLVERING

DEVELOPMENT OF BUILD ORDERS IN RTS-GAMES BY COEVOLUTION

Examensarbete inom huvudområdet Datalogi
Grundnivå 30 högskolepoäng
Vårtermin 2014

Per Näslund

Handledare: Daniel Sjölie
Examinator: Henrik Gustavsson

Sammanfattning

Det här arbetet undersöker hur konkurrenskraftiga byggordningar det går att generera genom att använda sig av en genetisk algoritm baserad på samevolution. För att realisera detta togs det fram en experimentmiljö genom att modifiera en utgåva av en open source spelmotor. För att kunna utvärdera hur konkurrenskraftiga byggordningarna som togs fram var; spelades ett antal matcher mot tre stycken vedertagna byggordningar.

Resultatet visade på att byggordningarna som togs fram inte kunde anses hålla måttet för konkurrenskraftighet då de till stor del blev besegrade av de vedertagna byggordningarna. På grund av tekniska begränsningar var algoritmen som genererade byggordningarna tvungen att ha låga parametervärden på populationsstorleken och maximalt antal generationer. Detta påverkade sannolikt byggordningarnas konkurrenskraftighet.

För framtida arbete kan en undersökning där spelmotorn är mer optimerad och klarar av att köra en samevolutionsalgoritm med högre parametervärden vara intressant. Tekniker som case-injection skulle också kunna undersökas.

Nyckelord: genetiska algoritmer, samevolution, RTS-spel, byggordningar

Innehållsförteckning

1 Introduktion

Den utmaning en spelprogrammerare ställs inför, när denne ska skapa en motståndar-A.I, är väldigt annorlunda beroende på vilken genre spelet i fråga tillhör. När det kommer till spel där spelarens skicklighet till stor del avgörs av hur bra reflexer och precision spelaren har – exempelvis en förstapersonsskjutare – är det inte speciellt svårt att skapa en motståndar- A.I som överträffar spelaren. Utmaningen ligger snarare i att skapa en A.I som utgör en lämplig svårighetsgrad för spelaren. När det istället handlar om en mer strategisk spelgenre är det snarare spelaren som i grunden har ett övertag i form av taktiska förmågor. Det är alltså betydligt svårare för en spelprogrammerare att skapa en motståndar-A.I som överträffar spelaren, än i fallet med en förstapersonsskjutare. Detta märks även ganska tydligt i många strategispel – både realtidsstrategispel och turordningsbaserade sådana – då motståndar-A.I:n ofta fuskar på de högre svårighetsgraderna, exempelvis i form av att samla resurser snabbare än spelaren (Ensemble Studios, 2002; Firaxis Games, 2010).

Inom realtidsstrategispel (RTS) är det speciellt utmanande att fram en A.I som har en konkurrenskraftig strategi. Spelare inom RTS måste bl.a. samla in resurser på ett effektivt sätt, ta kontroll över kartan, skapa en eller flera trupper med en bra sammansättning enheter, planera när byggnader ska konstrueras och välja när en attack med en eller flera trupper ska ske.

En del av A.I:n som måste bestämmas är hur dess byggordningar kommer att se ut. Ett problem med att ta fram byggordningar är att antalet möjliga sådana kan vara enormt. I spelet WarCraft II (1995) kostar exempelvis en *grunt* 600 guld att skapa, men för att kunna skapa en grunt krävs också att spelaren har byggt en barrack, som i sin tur kostar 700 guld och 400 trä. Det blir därför ett kombinatoriskt stort antal möjliga byggordningar när man ska välja vilka enheter man ska bygga och i vilken ordning de ska byggas (Ballinger and Louis, 2013a).

Oftast kan det vara tidskrävande att utveckla byggordningar om de ska skriptas manuellt av en programmerare. Det krävs bland annat domänkunskap för spelet och en hel del testning för att kunna ta fram en konkurrenskraftig strategi. Detta arbete undersöker ett alternativt tillvägagångssätt. Metoden arbetet använder sig av är genetiska algoritmer, baserade på samevolution, för att ta fram byggordningar inom RTS- spel. För att kunna utföra detta utvecklas en experimentmiljö genom att modifiera spelmotorn Stratagus (2007). Tillägget Wargus (2007) används för att importera speldatan från spelet WarCraft II (1995) och göra experimentmiljön till en klon av detta.

Sökrymden för algoritmen begränsas genom att införa en gentyp vid namn tillståndsgener och regler kring hur överkorsning får ske mellan två genomer. Tre stycken kartor används där det körs en samevolutionsalgoritm på vardera av de tre kartorna för att slutligen ta fram tre stycken byggordningar – en för varje karta. Byggordningarna utvärderas sedan genom att de får möta tre stycken kända byggordningar, som anses vara starka, i ett antal spelomgångar.

2 Bakgrund

2.1 RTS-Spel

Realtidsstrategispel är, som namnet antyder, spel med strategiska inslag som utspelar sig i realtid. Spelaren har en överblickande vy och observerar flertalet enheter och byggnader ovanifrån. Enheterna och byggnaderna hanteras av spelaren, som ger olika kommandon till dessa. Syftet med enheterna och byggnaderna är att de ska användas för att i någon grad komma närmare målet att vinna spelet mot motståndaren eller motståndarna. Detta mål är oftast att i någon grad förgöra motståndarens enheter och byggnader, men kan även vara alternativa mål. RTS-spel använder sig av s.k. fog of war, som innebär att en spelare inte kan se allting som motståndaren gör och vilka byggnader denne har byggt, såvida inte spelaren har någon enhet eller byggnad i närheten av motståndaren. Detta gör att RTS-spel kan ses som spel med icke-perfekt information.

Ett exempel på ett RTS-spel är spelet Age of Mythology (*Ensemble Studios*, 2002) som är tillgängligt på PC. I detta spel kontrollerar spelaren enheter och byggnader av en viss typ, som är beroende av vad spelaren har valt för folkslag innan spelomgången. Det finns tre stycken olika folkslag: greker, egypter och nordfolk. Dessa tre folkslag har alla olika typer av enheter och byggnader, som oftast fungerar på ett distinkt sätt i förhållande till varandra. När spelaren har valt folkslag väljer denne sedan en av tre mytologiska gudar, som hör ihop med folkslaget. Detta ger i sin tur folkslaget unika förmågor och möjligheter. Oavsett val av folkslag och mytologisk gud, fungerar spelet på ett grundläggande sätt. Enheter produceras från byggnader som byggs av enhetstypen bonde (i nordfolkets fall är det däremot infanteri som bygger byggnader). För att kunna konstruera en byggnad kan det finnas krav som måste uppfyllas – exempelvis att andra byggnader finns färdigkonstruerade. Byggnader kostar även resurser. Bönder används för att samla olika typer av resurser: trä, mat och guld. Resurserna spenderas när byggnader konstrueras eller enheter produceras. Det finns även en fjärde typ av resurs – favör. Denna representerar hur pass omtyckt spelarens folk är av de mytologiska gudarna. Favör samlas på olika sätt beroende på folkslag men är annars en resurs likt de andra som krävs för att bygga vissa byggnader eller producera vissa enheter. Enheter som produceras är oftast av militärisk typ och används för att för att förgöra motståndarens enheter och byggnader. Det går även att spendera resurser på att forska fram olika teknologier. Dessa ger fördelar åt spelarens olika enheter och byggnader, exempelvis att spelarens infanteri gör mer skada. Att forska fram en teknologi fungerar likt att producera en enhet, men tar oftast längre tid.

Spelet tar slut när det ena laget har förstört alla byggnader från det andra laget, eller när det ena laget ger upp. Det går dock att även vinna genom att det ena laget konstruerar ett s.k. wonder, vilket är en relativt dyr byggnad vars enda syfte är att laget som behåller byggnaden i 10 minuter vinner spelet, givet att byggnaden inte förstörs.

Många RTS-spel fungerar likt Age of Mythology (*Ensemble Studios*, 2002) med fokus kring resurssamlande, konstruerande av byggnader och forskning av teknologi. Då RTS-spel ofta kräver hög precision och hastighet för att hantera alla enheter på ett effektivt sätt, är den överväldigande majoriteten av spelen exklusivt tillgängliga för PC. Det finns dock undantag till detta; spelet Halo Wars (*Halo Wars*, 2009) är exklusivt till konsolen Xbox 360.

2.2 Byggordning

Den ordning som byggnader och enheter produceras i benämns vid termen byggordning (Churchill and Buro, 2011). Om en spelare följer byggordningen – d.v.s. utför de handlingar under spelets gång som krävs för att uppnå den produktion av enheter och byggnader byggordningen syftar på – uppnår spelaren ett visst resultat, oftast bestående av en sammansättning stridsenheter. För att byggordningarna ska fungera, måste de ta hänsyn till de olika kraven som varje byggnad och enhet har för att kunna produceras. Dessa krav är oftast i form av en kostnad, bestående av en viss mängd resurser, associerad med en byggnad eller enhet. Det är därför viktigt att byggordningar tar hänsyn till resurssamlandet som krävs för att spelaren ska ha råd med att fullfölja de olika stegen i byggordningen inom rimlig tid. Ett annat vanligt krav för en byggnad eller enhet är att en mängd tidigare byggnader ska finnas tillgängliga och/eller att en mängd uppgraderingar ska ha utförts. Detta måste naturligtvis också tas hänsyn till så att byggordningen är möjlig.

Målet med en byggordning är att utföra någon form av strategi, exempelvis s.k. rush – och timing–attacker(Weber and Mateas, 2009). En rush-attack är en strategi där spelaren försöker att göra en tidig attack genom att producera en trupp enheter väldigt snabbt i hopp om att motståndaren inte ska ha något bra försvar mot en så pass tidig attack. En timing-attack försöker att utföra en attack vid något speciellt nyckelmoment – exempelvis när en teknologi har forskats fram.

Byggordningar har i regel olika styrkor och svagheter, vilket får till följd att vissa byggordningar är bra mot vissa andra byggordningar och sämre mot andra. Det är sällan någon byggordning är strikt bättre än någon annan byggordning, vilket medför att det inte går att linjärt ordna alla byggordningar efter styrka.

Exempel på en byggordning kan ses i figur 1 nedan, denna är till RTS-spelet StarCraft 2 (*Blizzard Entertainment*, 2010) och har som mål att attackera motståndaren med en tidig attack som rasen Zerg. När en byggordning beskrivs i StarCraft 2 (*Blizzard Entertainment*, 2010) görs vissa antaganden för att inte beskriva onödiga detaljer. Ett sådant antagande är att spelaren kontinuerligt bygger arbetarenheter så fort denne har råd. Byggordningen beskrivs av en lista på olika handlingar som ska utföras av spelaren. Dessa handlingar är oftast antingen att konstruera en byggnad eller att producera en enhet. Till vänster om varje handling finns en siffra som beskriver när en viss handling ska utföras. Denna siffra beskriver populationen av enheter hos spelaren. Detta betyder alltså att så snart ett visst populationsantal har uppnåtts, ska spelaren utföra handlingen. Exempelvis betyder ”10 Spawning Pool” att byggnaden ”Spawning Pool” ska konstrueras när populationen har uppnått 10 till antal. Det finns även andra indikationer än populationsantalet för när en handling ska utföras, exempelvis ”@ 1:25” syftar på att spelets klocka ska visa tiden 1:25.

Figur 1 En byggordning från RTS-spelet StarCraft 2 (*Blizzard Entertainment*, 2010). Att populationen enheter minskar beror på att en arbetare försvinner vid byggnadskonstruktion.

Vad för byggordning som bäst lämpar sig att användas av en spelare, beror dels på vad spelaren misstänker att motståndaren själv kommer att ha och dels på topologin av kartan man spelar på. Exempelvis gynnar kartor med kortare avstånd mellan spelare rush-attacker mer (Weber and Mateas, 2009).

Enligt Ovesson (2012) går byggordningar att dela in i proaktiva och reaktiva sådana. En byggordning som är ute efter att diktera spelet, utan någon vidare hänsyn till handlingar utförda av motspelaren, anses vara åt det proaktiva hållet. Ett bra exempel på en byggordning åt det proaktiva hållet är en rush-strategi. En byggordning av en mer reaktiv karaktär tar istället mer hänsyn till vad motståndaren gör och försöker därefter anpassa sig till att skapa enheter och byggnader som är bra mot det motståndaren har gjort. Generellt brukar byggordningar bli mer och mer reaktiva under spelets gång, eftersom mer och mer information om motspelaren avslöjas, vilket oftast är fördelaktigt att reagera på. Exempelvis kanske en spelare i spelet Age of Mythology (*Ensemble Studios*, 2002) väljer en byggordning som syftar på att realisera en ekonomisk strategi. Denna byggordning kommer då antagligen innebära att spelaren tidigt i spelomgången konstruerar byggnader och producerar arbetare som ger upphov till en stark ekonomi längre fram i spelomgången. Den reaktiva delen blir sedan när spelaren ska producera stridsenheter och anpassar denna produktion efter vad dennes uppfattning av vad motspelaren har för stridsenheter är.

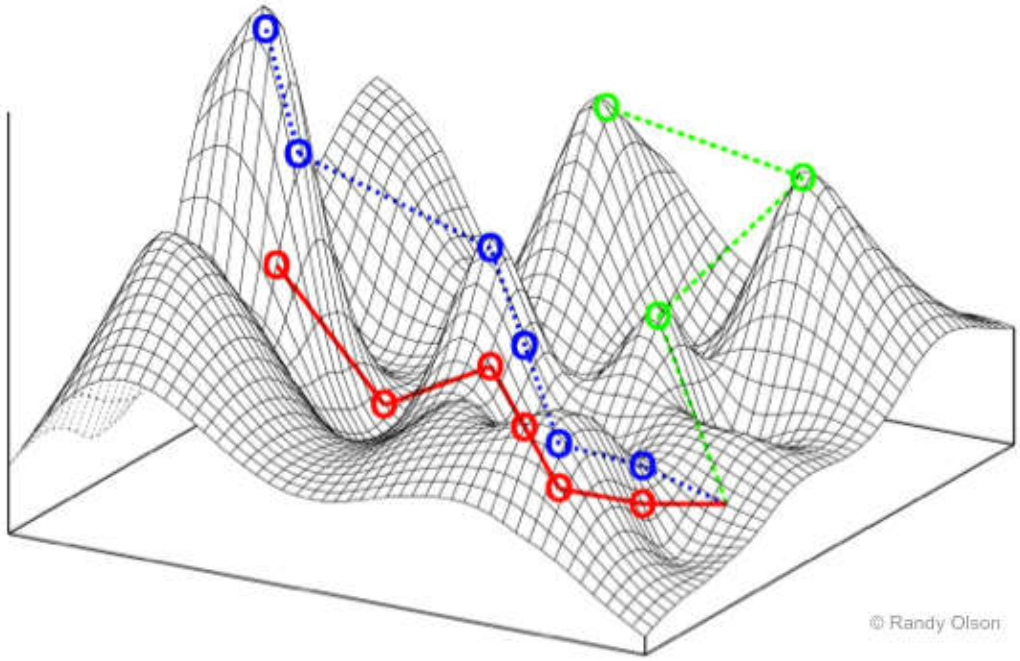
2.3 Genetiska Algoritmer

Genetiska algoritmer (GA) tar inspiration från Charles Darwins teori om naturligt urval(Bäck, 1996). Denna teori bygger på att individer tillhörande en population har gener som i sin tur ger individen olika egenskaper. Individerna med de olika egenskaperna

försöker sedan att anpassa sig till miljön för att överleva och reproducera i den. De individer med de egenskaper som är bäst anpassade för överlevnad kommer vara de individer som reproducerar sig i störst utsträckning. Vid reproduktion mellan två individer kommer gener från de båda ”föräldrarna” att i viss utsträckning överföras till deras avkommor. Resultatet av naturligt urval blir att de egenskaper som är bäst anpassade till miljön blir de vanligast förekommande bland populationen (Bäck, 1996).

Inom datavetenskap är GA en sökheuristik som är tänkt att fungera likt naturligt urval. En kandidatlösning benämns ofta som individ, kromosom eller genom och en mängd kandidatlösningar en population. Varje individ består av ett antal gener, som kan ses som olika parametervärden för att formulera fram en lösning till problemet. Chansen att en individ ska reproducera sig med en annan individ är olika beroende på dess fitnessvärde. Ett fitnessvärde tilldelas till varje individ genom en fitnessfunktion, som bedömer hur pass bra individen eller kandidatlösningen löser problemet i fråga (Goldberg, 1989). Om kandidatlösningen är bra på att lösa problemet tilldelas den ett högt fitnessvärde och vice versa. Oftast är fitnessvärdet en siffra mellan 0 och 1. När två stycken individer har valts ut för reproduktion eller överkorsning, kommer två stycken nya individer att skapas med vissa gener från de båda föräldrarna, beroende på vilken metod av överkorsning som används. Dessa nya individer som skapas utgör populationen för nästa generation. Allteftersom algoritmen körs kommer fler och fler generationer att skapas. Vanligtvis brukar algoritmen sluta köras när ett valt fitnessvärde har uppnåtts på en individ - alternativt slutar algoritmen efter ett visst antal generationer.

Ett vanligt förekommande koncept inom GA är fitnesslandskap, vilket är en representation av rymden av alla möjliga kandidatlösningar och deras fitnessvärde (Mitchell, 1996). Representationen tar form av ett diagram med $l + 1$ axlar, där l är längden på kromosomen. Varje kromosom representeras som en punkt i l-dimensioner och fitnessvärdet skrivs ut längst med den $l+1$:e axeln. Exempelvis, om en kromosom har längden 2, är diagrammet i tre dimensioner och varje kromosom är en punkt i ett plan. Fitnessvärdet till varje punkt skrivs ut längst en tredje axel, ortogonalt med planet. Hur ett sådant diagram kan se ut kan ses i figuren nedan.



Figur 2 En representation av ett fitnesslandskap.

Ett fitnesslandskap skapar toppar, som antingen är ett lokalt optimum eller globalt optimum. Ett lokalt optimum är inte nödvändigtvis det maximala fitnessvärdet som kan uppstå, men varje förflyttning ifrån denna tillför en minskning i fitnessvärdet (Mitchell, 1996).

GA används inom en stor mängd områden, exempelvis inom ekonomi, maskinlärning och optimeringsproblem (Mitchell, 1996).

2.3.1 Samevolution

En variant av genetiska algoritmer använder sig av samevolution. Processen är liknande den som sker för en traditionell GA. Den stora skillnaden är hur varje individ i populationen utvärderas för sin fitness. Individerna interagerar med varandra och resultatet av varje interaktion visar på en belöningsstruktur som används för att vägleda evolutionen mot än mer anpassade beteenden för individen (Ficici, 2004). Det här kan leda till att individerna - i sin process för att utnyttja belöningsmöjligheterna hos de andra individerna - i sin tur skapar nya möjligheter att själv bli utnyttjad av de andra individerna (Ficici, 2004). Den stora fördelen med att utvärdera individerna på det här sättet, jämfört med att använda separata fitnessfunktioner, är att man endast behöver en minimal förståelse för sökrymden hos problemet (Angeline and Pollack, 1993). Varje individ interagerar med varje annan individ i populationen; alltså kommer det att utföras n(n-1)/2 interaktioner där n är antalet

individer. Ett exempel på hur användning av samevolution kan ske, är att 10 stycken A.I-spelare möts i ett symmetriskt spel för två spelare. Varje A.I-spelare har en egen strategi för att vinna spelet och spelar en gång mot varje annan A.I-spelare. Detta medför att 45 stycken spelomgångar körs i varje generation. Efter att varje spelomgång har körts, utvärderas det hur det gick för respektive A.I-spelare genom att tilldela ett fitnessvärde till de båda – exempelvis positivt fitnessvärde vid vinst, negativt vid förlust.

2.4 Relaterat arbete

2.4.1 Ponsen (2004)

Ett arbete som är relaterat till detta är ett som gjordes av Ponsen (2004) och som behandlar dynamisk skriptning i syfte att skapa en A.I som anpassar sig under körning. Dynamisk skriptning är en teknik som innebär att A.I:n hämtar instruktioner till vad den ska göra från en uppsättning regler. Val av regel sker slumpmässigt med vikter som ökar sannolikheten att en specifik regel blir utvald. Efter varje spelomgång uppdateras vikterna med en uppdateringsfunktion som sätter nytt värde på vikterna. Ponsen (2004) använder sig av en uppdateringsfunktion som tar hänsyn till den fitness en specifik regel hade under spelomgången, samt vad hela skriptet överlag hade för fitness. Fitnessfunktionen beräknar fitnessvärdet genom att ta hänsyn till den poäng A.I-spelaren hade, som är beroende av hur många byggnader som konstruerades, hur många enheter som förstördes etc.

Ponsen(2004) kommer fram till att dynamisk skriptning inte ger helt önskvärda resultat mot s.k. optimerade strategier, som utgörs av de två strategierna Soldier Rush och Knight Rush. Den senare är en mer långsam strategi och den tidigare en snabbare. För att förbättra dessa mediokra resultat utgörs den senare delen av Ponsens arbete utnyttjandet av Evolutionära Algoritmer för att förbättra A.I:n. Här används Offline Inlärning, dvs. att den evolutionära algoritmen körs innan körningen av spelet.

Generna i en genom är indelade i fyra olika typer: *build* (konstruktion), *research* (forskning/teknologi), *economy* (skapande av arbetare) och *combat* (stridsaktiviteter). Dessa olika typer motsvarar olika aktiviteter A.I:n kan utföra och en gen tillhörande någon av typerna representeras genom att första bokstaven anges, samt ett antal parametrar beroende på typ. Exempelvis anges bokstaven B följt av parametern 2 för att konstruera en barrack.

Generna delas även in i olika *states* (tillstånd). Varje genom har alltid som lägst state 1 aktiverat och avancerar till ett annat genom att exekvera minst 1 gen tillhörande ett annat state. Varje state ger olika alternativ gällande exempelvis vilka olika typer av trupper som kan skapas. Fördelen med att markera ut states i genomen är att det gör det möjligt att utföra *state crossover* som beskrivs nedan.

Varje evolutionär algoritm använder sig av ett antal genetiska operatorer. En vanligt förekommande sådan är överkorsning, vilket i detta arbete av Ponsen (2004) förekommer i form av *state crossover*. Denna version av överkorsning fungerar på det sätt att det först säkerställs att vardera föräldern har minst tre stycken gemensamma aktiverade states. Sedan kopieras alla gener mellan två stycken av dessa gemensamma states till barnet. Det säkerställs även att gener från båda föräldrarna kopieras.

2.4.2 Ballinger och Louis (2013)

Arbetet som Ballinger och Louis (2013a) gjorde gick ut på att undersöka hur robusta byggordningar framtagna av samevolution kunde bli. Detta utvärderades genom att de fick möta byggordningar framtagna av en GA, tre hårdkodade strategier samt en mänsklig motståndare.

Samevolutionsalgoritmen använde sig av ett s.k. teach set som är en mängd kromosomer varje individ i populationen får spela mot vid varje generation. Detta teach set bestod av 4 kromosomer från ”hall of fame” och 4 kromosomer från ”shared sampling”. Det förstnämnda är en lista på kromosomer som har presterat bra sedan tidigare. Vid varje generation lägger man till den kromosomen med högst fitness i ”hall of fame”. Ballinger och Louis (2013a) menar att detta förhindrar att individerna i den nuvarande generationen inte glömmer hur man besegrar individer från äldre generationer. Shared sampling fungerar på det sättet att kromosomer som har besegrat många medlemmar tillhörande teach set:et väljs ut. Dock ger shared sampling mindre vikt åt medlemmar som andra kromosomer, som tidigare har valts ut av shared sampling, redan har besegrat.

Resultatet av undersökningen var att byggordningar framtagna av samevolution kunde besegra byggordningar framtagna av en GA och två stycken av de hårdkodade strategierna. Den mänskliga spelaren tyckte också att det var den svåraste motståndaren att spela mot.

3 Problemformulering

Inom spelutveckling kan det vara ganska resurskrävande att skapa bra byggordningar genom skriptning för en A.I till ett RTS-spel. Om spelet är helt nytt kan det vara svårt att veta vilka strategier som är bra att använda. Byggordningar brukar oftast skapas av spelare själva efter spelet har släppts. Det kan därför vara relevant att se på alternativa metoder till att utveckla byggordningar annat än genom skriptning. En sådan metod är att använda sig av genetiska algoritmer och samevolution för att ta fram byggordningar. Det är även denna metod som examensarbetet fokuserar på. En fördel med att använda sig av genetiska algoritmer för att ta fram byggordningar är att det inte är lika resurskrävande som att skripta för hand. Med resurskrävande menas att det generellt sett inte tar upp lika mycket utvecklingstid att skriva en genetisk algoritm som att skripta en byggordning. I det här fallet beror utvecklingstiden också mycket på om samevolution används i den genetiska algoritmen eller inte. Om inte samevolution används måste den genetiska algoritmen ha något att tränas mot – antagligen en vedertagen, stark byggordning. Eftersom detta i sin tur kräver att man har tillgång till minst en sådan byggordning, är detta tillvägagångssätt inte speciellt tidsbesparande om det är ett nytt spel man utvecklar A.I till, eftersom man först behöver skripta byggordningar ändå. Det är därför arbetet fokuserar på genetiska algoritmer som bygger på samevolution. Då samevolution innebär att individernas fitnessvärde beror på hur de ”presterar” mot varandra, behövs ingen färdig byggordning att tävla mot, om varje individ representerar en byggordning.

Examensarbetets syfte var att utvärdera om genetiska algoritmer och samevolution är ett lovande tillvägagångssätt till att utveckla byggordningar för ett RTS-spel. För att det ska kunna anses vara ett lovande tillvägagångssätt måste byggordningarna som tas fram vara någorlunda starka och balanserade jämfört med byggordningar som redan anses vara starka och balanserade. En frågeställning kan därför formuleras till följande: Går det att ta fram konkurrenskraftiga byggordningar med hjälp av genetiska algoritmer och samevolution?

För att svara på frågeställningen blev valet att göra en praktisk lösning i form av en undersökning i en experimentmiljö. Ett alternativ till detta hade varit att göra en litteraturstudie. Beslutet att inte använda detta alternativ var på grund utav anledningar som även Wångsell (2013) tog upp; som att den litteratur som fanns på området sällan undersöker byggordningarnas kvalitet. I fallet med det här arbetet fanns det dessutom ännu mindre litteratur där samevolution användes för att ta fram byggordningar.

3.1 Experimentmiljö

Experimentmiljön måste möjliggöra en implementering av den genetiska algoritmen. Med detta menas att experimentmiljön måste ha stöd för att kunna utföra de olika delarna hos en genetisk algoritm; exempelvis stöd för populationsgenerering. Förutom stöd för implementering ska det även finnas stöd för att utvärdera en byggordning.

Alternativen för att få en lämplig experimentmiljö är att antingen utveckla en egen RTS-motor eller att använda en befintlig Open Source-motor. Fördelen med att utveckla en egen motor är att det kan vara enklare att anpassa den på just det sättet man vill, eftersom man inte behöver lära sig hur motorn är uppbyggd och sätta sig in i koden. Nackdelen med att

utveckla en egen motor är naturligtvis att det kan vara väldigt tidskrävande. I synnerhet om man inte har någon speciellt hög erfarenhet av att utveckla spelmotorer till RTS-spel.

3.2 Utvärderingsmetod

En färdig byggordning måste kunna utvärderas när den har tagits fram. Vad detta ställer för egentliga krav på spelmotorn är helt beroende på vilken metod för utvärdering som har valts. Om tester, bestående av spelomgångar mot en A.I med förskriptad byggordning, ska exekveras är det viktigt att en spelomgång inte tar för lång tid. En stor provstorlek ska med fördel användas för att undvika statistiska fel.

Målet med att utvärdera en byggordning är att få ett slags mått på hur pass stark den är. Ett sätt att utföra detta skulle kunna vara att låta en mänsklig spelare spela ett antal spelomgångar mot en A.I, användandes den byggordning i fråga som ska utvärderas. Det är här viktigt att det är just byggordningen som hamnar i fokus och inte någon annan aspekt av A.I-spelaren, exempelvis dess enhetskontroll. Själva utvärderingen kan vara att låta den mänskliga spelaren ge ett subjektivt omdöme på hur spelaren upplevde styrkan hos A.I-spelarens byggordning kombinerat med mer objektiv statistik, som om A.I-spelaren vann spelomgången och hur många byggnader den raserade. Nackdelen med en sådan undersökning är att man gärna vill ha testpersoner med olika färdigheter; som spelare med expertfärdigheter och något mer oerfarna spelare. Det är dessutom ganska resurskrävande att ha en spelare som sitter och spelar spelet om denne ska spela på flera olika kartor, mot flera olika byggordningar och om man vill ha en betydande provstorlek.

Ett andra alternativ är att använda sig av A.I-motståndare istället för mänskliga spelare. A.I-motståndarna använder sig av beprövade byggordningar som anses vara starka strategier. Man kör sedan ett antal spelomgångar där de framtagna byggordningarna spelas mot de beprövade. När spelomgångarna är körda ser man efter i vinststatistiken hur många som vanns av den framtagna byggordning man utvärderar. Denna metod används till viss del inom forskning med spelmotorn Stratagus (The Stratagus Team, 2007) och tillägget Wargus (Aha et al., 2005; Ponsen, 2004; The Wargus Team, 2007; Wångsell, 2013). En fördel med detta tillvägagångssätt är att det enkelt går att testa med ett stort antal spelomgångar och med flera olika konfigurationer, exempelvis olika kartor.

3.3 Metodbeskrivning

Detta arbete har använt sig av RTS-motorn Stratagus med tillbyggnaden Wargus, som är en klon av RTS-spelet Warcraft II (Blizzard Entertainment, 1995). Anledningen till detta beslut var framförallt för att det är en motor som tidigare forskning har använt sig av och anser fungerar bra (Aha et al., 2005). Ponsen (2004) tar upp anledningar som att det är en sofistikerad RTS-motor som är enkel att ändra och utöka.

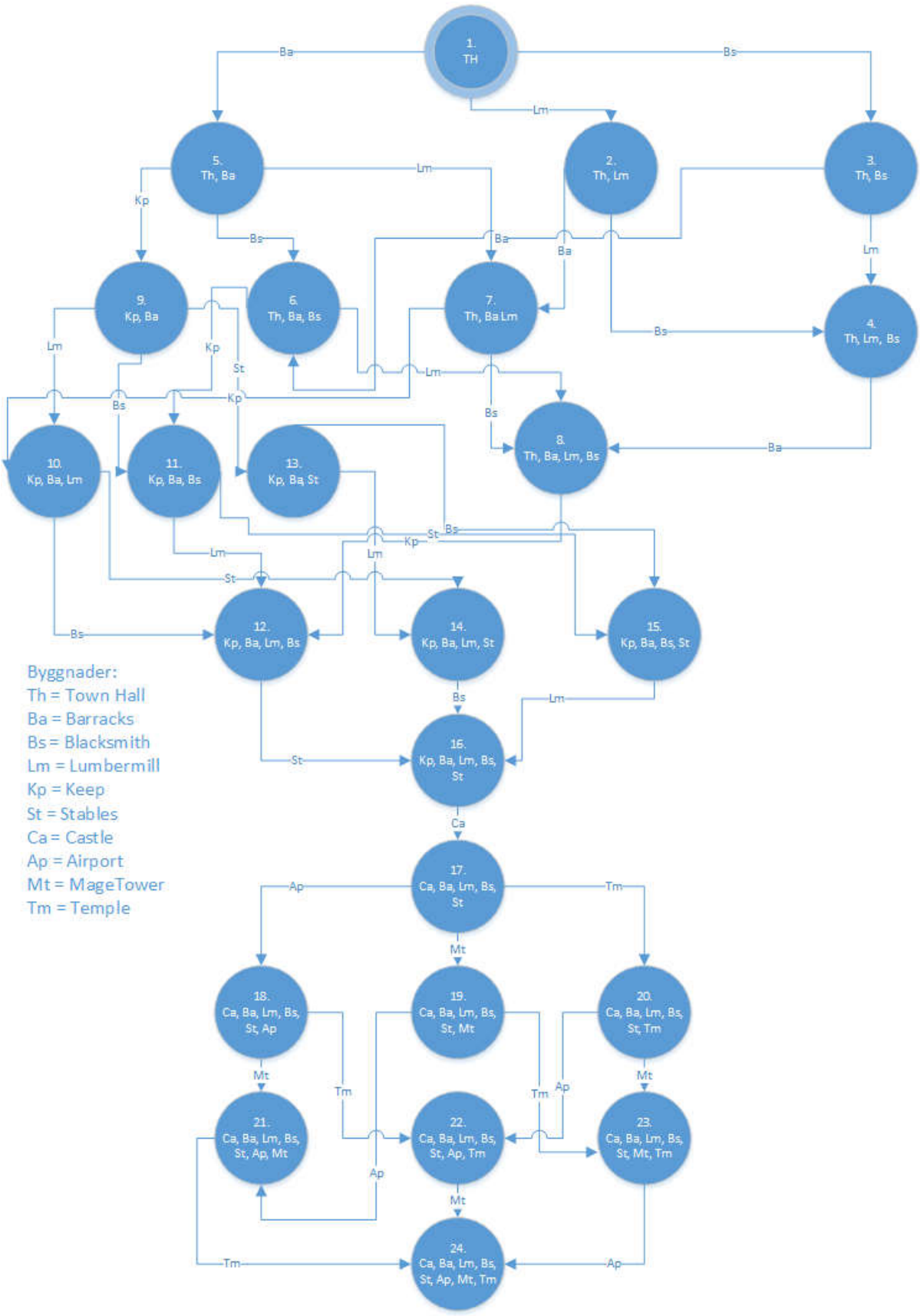
4 Implementation

För att på enklast möjliga sätt kompilera källkoden till Stratagus med alla dess tillhörande bibliotek valdes programmeringsmiljön Microsoft Visual C++ (MVC++). Det finns tillgängligt en färdig projektfil till MVC++ att hämta tillsammans med källkoden, vilket underlättade arbetet.

Det första som behövde göras var att möjliggöra att två stycken A.I-spelare kunde spela mot varandra utan att någon mänsklig spelare behövde delta i spelomgången. En lösning till detta enligt Wångsell (2013) är att ta bort den mänskliga spelarens enheter. Detta gick att åstadkomma genom att först anropa spelarstrukturens clear()-funktion. Sedan måste även ändringar i filen som specificerar banans utseende göras, eftersom den alltid placerar ut en arbetarenhet när den har laddats. Slutligen måste funktionen som ser efter vilken spelare som har vunnit ändras så att den inte tar med den mänskliga spelaren i beräkningen. Resultatet av dessa åtgärder blev att det var tre spelare med i spelomgången, en mänsklig och två A.I-styrda. Den mänskliga spelaren kunde däremot endast övervaka spelomgången och påverkade den inte på något sätt. Detta kan liknas vid ett slags åskådarläge som kan finnas i andra spel.

Den evolutionära algoritmen som implementerades behövde spelas igenom ett antal spelomgångar med två stycken A.I-spelare. Dessa spelomgångar snabbades upp betydligt, för att algoritmen inte skulle ta för lång tid att köra igenom. Detta gick att åstadkomma genom att öka antalet spelcykler per sekund. Genom att stänga av gränsen för hur många spelcykler som maximalt kunde ske per sekund, samt att stänga av grafikutritningen, nåddes detta mål.

Det första som bör bestämmas för att realisera en samevolveringsprocess är hur gener och en genom representeras i kod. Vanligt är att en genom representeras av siffror, där varje siffra representerar en gen. Inspiration från Wångsell (2013) togs för att bestämma hur gener ska representeras. Varje gen representeras av en siffra och har antingen typen typgen eller värdegen. Två gener – en från vardera typ – beskriver tillsammans en specifik byggnad, uppgradering eller trupp som ska skapas. Detta var hur genrepresentationen var till en början, men senare under arbetets gång blev det av praktiska skäl lämpligt att införa en ytterligare gentyp: tillståndsgener. Dessa gener har som funktion att förklara i vilket tillstånd en viss byggnad/uppgradering/trupp befinner sig i. En spelare befinner sig alltid i ett av 24 stycken tillstånd beroende på vad för typ av byggnader som har konstruerats. Ett visst tillstånd indikerar vad för typ av trupper och byggnader som kan skapas. I figuren nedan visas ett diagram över alla tillstånd i spelet och vilka byggnader som behöver konstrueras för att gå från ett tillstånd till ett annat.

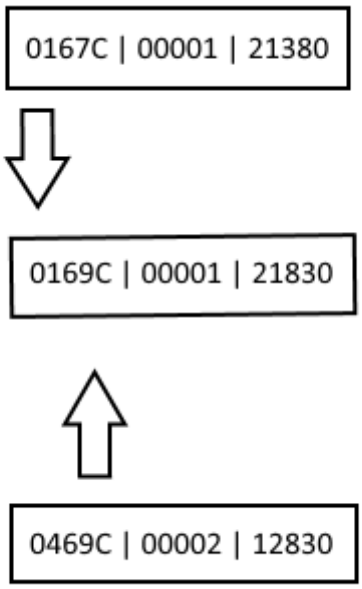


Figur 3 Diagram över alla möjliga tillstånd i spelet.

Exempelvis går det att se i figuren att spelaren i tillstånd 2 har möjlighet att bygga antingen en Ba (Baracks) eller BS (Blacksmith). Byggs den förstnämnda hamnar spelaren i tillstånd 7 och om det sistnämnda byggs hamnar spelaren i tillstånd 4.

Syftet med att hålla kolla på vilket tillstånd spelaren befinner sig i är att förhindra att korrupta genomer (byggordningar) genereras. Exempel på en korrupt byggordning är en som kräver att en barracks byggs till följd av ett stable, eftersom ett stable kräver att det finns ett keep. Om det noteras i genomen i vilket tillstånd byggordningen befinner sig i vid varje ny byggnad/trupp/uppgradering, kan generering av byggordningen begränsas till att bara generera fram byggordningar som inte blir korrupta.

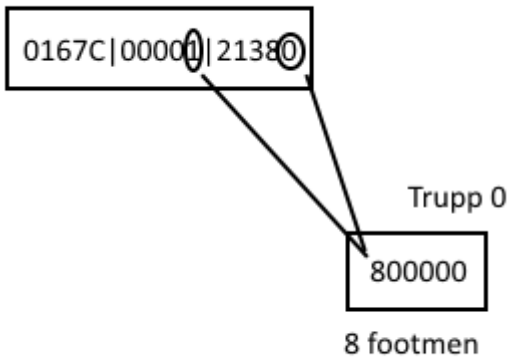
Det är inte bara vid generering av genomen som tillståndsgenerna kommer till användning, utan även när det sker en överkorsning. Om överkorsning görs enligt exempelvis tvåpunktskorsningsmetoden och det inte sätts några begränsningar på vilka punkter som får väljas ut, kan överkorsningen resultera i att korrupta genomer skapas. En metod för att förhindra korrupta genomer används, och är lik den även Ponsen (2004) använder sig av. Principen är att överkorsningsfunktionen tar reda på hur många gemensamma tillstånd de båda genomerna har och om det är minst 3 stycken gemensamma kan en överkorsning ske. Från någon av föräldrarna väljs en sträng av gener ut som startar i ett av de gemensamma tillstånden och slutar i nästa gemensamma tillstånd. Ett exempel på två stycken genomer som överkorsas visas i figuren nedan.



Figur 4 Överkorsning av två stycken genomer

I figuren ser man hur två stycken byggordningar, den översta och den understa, överkorsas och skapar byggordningen i mitten. De fem första generna representerar tillståndsgenerna, de fem mittersta representerar typgenerna och de fem sista värdegenerna.

En utmaning med genrepresentationen var hur man skulle representera trupper. Det ska gå att representera både offensiva och defensiva trupper innehållandes ett varierande antal olika enheter. Detta är svårt att representera med enbart en gen. Lösningen till detta är lik den Wångsell (2013) använder sig av och går ut på att länka en värdegen med en särskild trupp, som har slumpats fram. Truppen som värdegenen är länkad till representeras i sin tur av 7 siffror, där siffrans index visar vilken typ av enhet det är och siffrans värde hur många av den enheten som finns i truppen. Ett exempel på hur en trupp kan representeras visas i figuren nedan.

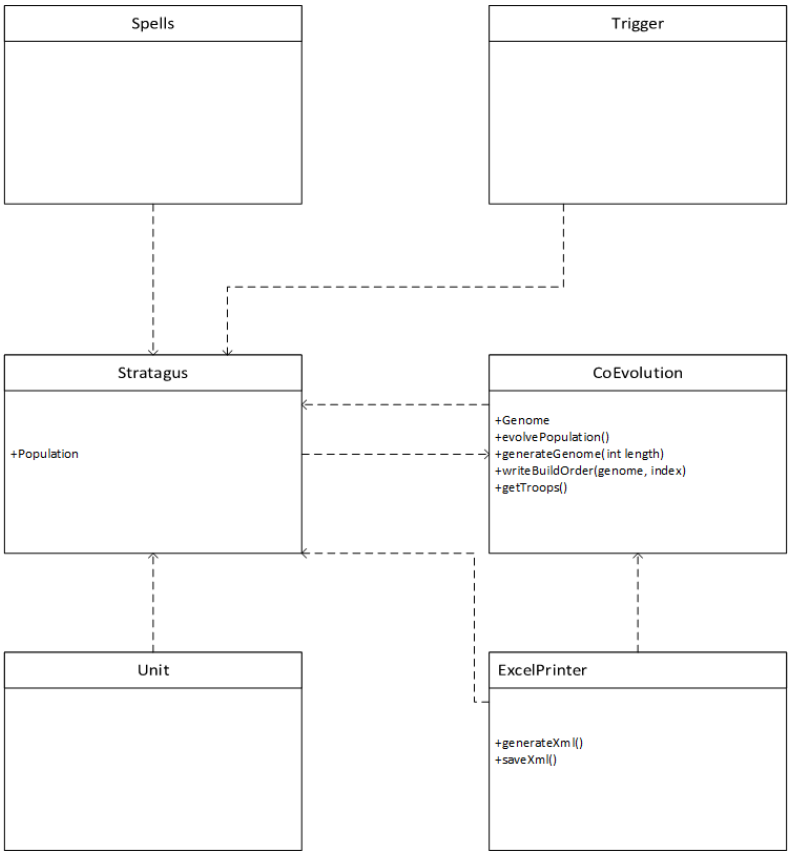


Figur 5 Representation av en trupp

I figuren går det att se att typgenen 1, som benämner att det är en offensiv trupp, hör samman med värdegenen 0. Denna länkar i sin tur till trupp med index 0. Den sju siffror långa koden ger en trupp bestående av åtta stycken footmen.

När det gäller valet av selektionsmetod valdes den som enligt Ponsen (2004) är bäst lämpad för denna typ av problem; nämligen turneringsselektion. Denna selektionsmetod ger upphov till ett elitistiskt urval där individer med hög fitness till stor del är de som väljs ut för överkorsning. Selektionen fungerar på det sättet att det slumpvist väljs ut en viss mängd av populationen och sedan väljs den individ med bäst fitness i den delmängden ut för överkorsning.

All implementation som gjordes i stratagus-motorns kod illustreras i nedanstående diagram i figur 6.



Figur 6 Klassdiagram. Variablen Population i Stratagus är global och används av samtliga övriga klasser/filer.

Det lades till två stycken klasser. Klassen som erbjuder funktionaliteten för att implementera en samevolveringsprocess heter CoEvolution. Denna skapades med syfte att kunna utföra de huvudsakliga delarna av en samevolutionsalgoritm. Den tillhandahölls även en struct med namnet ”Genome”, som användes för att representera en genom. Den andra klassen som skapades var ExcelPrinter, som användes för att skriva ut relevant statistik till excelfiler. ”Spells”, ”Trigger”, ”Stratagus” och ”Unit” fanns i stratagus-motorn från början och är egentligen inte klasser utan filer som innehåller funktioner, konstanter och definitioner. Detta är för att stratagus-motorn är huvudsakligen skriven i C och därmed har den struktur som ett program skrivet i C tenderar att ha. I dessa filer lades det till kod. Variablen ”Population” i filen ”Stratagus” är en global array som håller den nuvarande populationen av genomer.

Samevolutionsalgoritmen startar sin exekvering från funktionen StartMap i filen Stratagus.cpp, vilket är funktionen som anropas när en ny spelomgång ska köras igång. Med hjälp av CoEvolution-klassen genereras först en initial population genomer. Efter detta påbörjas den evolutionära algoritmen som körs ett bestämt antal generationer. I stora drag är händelseförloppet i algoritmen att populationens alla genomer möter varandra i en spelomgång och får ett fitnessvärde tilldelat. Innan varje spelomgång väljs det ut två byggordningar (genomer) från populationen som ska utvärderas genom att låta två AIs spela

mot varandra med respektive byggordning. Efter varje spelomgång tilldelas de två byggordningarna som deltog ett fitnessvärde baserat på hur de båda presterade. Den totala summan fitness beräknas genom tabellen nedan.

Händelse	**Fitness**
Vinst	+4000
Förlust	-4000
Byggnad förstörd	+200
Enhet dödad	+50

Figur 7 Klassdiagram. Variablen Population i Stratagus är global och används av samtliga övriga klasser/filer.

Om en genom exempelvis vinner matchen och förstör 5 byggnader och 10 enheter får denne 4000 + 5*200 + 10*50 = 5500 i fitness. Valet av värdena i Fitnesstabellen baserades delvis på vad Wångsell (2013) hade använt, men är annars ganska godtyckliga.

När varje genom i den nuvarande populationen har fått ett fitnessvärde tilldelat ska populationen modifieras med hjälp av genetiska operationer, som exempelvis överkorsning, för att bilda nästa generation av genomer. Detta görs fortfarande från funktionen StartMap i filen Stratagus.cpp med hjälp av CoEvolutionen-klassens funktion ”evolvePopulation()”. Denna funktion använder sig av turneringsselektion och statecrossover, som beskrevs tidigare i detta avsnitt, för att skapa nästa generation. När nästa generation är skapad återupprepas algoritmen ett önskat antal gånger.

Generering av en genom tar hänsyn till de 24 olika tillstånden beskrivna tidigare för att inte generera fram korrupta genomer. Exempelvis kommer det inte att genereras fram att A.I:n ska konstruera ett Gryphon Aviary som första byggnad. Algoritmen ser istället efter i vilket tillstånd byggordningen befinner sig i och vilka möjliga byggnader, uppgraderingar och enheter det går att skapa och väljer slumpvist mellan dessa.

En byggordning i Stratagus beskrivs med ett lua-script. Funktioner talar om vilka byggnader och enheter som behövs och i vilken ordning de ska skapas. I byggordningen innefattar även när A.I:n ska attackera och med vilka trupper. Utöver vad byggordningen beskriver sköts A.I:n automatiskt av stratagus-motorn. Detta innefattar saker som resurssamlande och byggnation av farmar för att möjliggöra en större population. Även striderna sköts av A.I:n förutom just när den ska attackera och med vilka trupper.

Det finns i spelet en enhet som medvetet uteslöts vid genereringen av byggordningar – enheten torn. Detta beror på att torn är ett specialfall vad det gäller enheter. Det är en fast stationerad stridsenhet, som bara kan attackera i en viss area. En mänsklig spelare kan oftast enkelt undvika att ta skada av torn, genom att helt enkelt gå med sina trupper utanför tornets area. Det är dessutom hög risk att torn leder till att A.I:n får ett väldigt defensivt och passivt beteende där den endast försöker försvara (Wångsell, 2013).

Då byggordningar beskrivs med lua-script, skapades en funktion för att översätta en genom till en lua-fil. På detta sätt integreras de byggordningar som skapas med samevolutionsprocessen med de övriga skriptade byggordningarna. Denna funktion finns i klassen CoEvolution och tar emot en genom som den sedan läser av och skapar en lua-fil ifrån. Ett exempel på hur ett sådant lua-script kan se ut för en byggordning som har genererats finns i figuren nedan.

```
local test3_funcs = {
    function() return AiNeed(AiCityCenter()) end,
    function() return AiWait(AiCityCenter()) end,
    function() return AiSet(AiWorker(), 30) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBarracks()) end,
    function() return AiWait(AiBarracks()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiForce(0, {AiShooter(), 9}) end,
    function() return AiForce(0, {AiCatapult(), 2}) end,
    function() return AiForceRole(0, "defend") end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiForce(1, {AiShooter(), 8}) end,
    function() return AiForce(1, {AiCatapult(), 8}) end,
    function() return AiForceRole(1, "attack") end,
    function() return AiForce(2, {AiSoldier(), 5}) end,
    function() return AiForceRole(2, "attack") end,
    function() return AiTest3Endloop() end,
}
```

Figur 8 Lua-script av en genererad byggordning.

Figuren visar den mest relevanta delen av skriptet, som beskriver kärnan i byggordningen; dvs. vilka enheter som ska skapas, byggnader som ska byggas etc. Koden beskriver en samling funktioner, som kommer att exekveras i en sekventiell ordning. I nedanstående figur beskrivs de funktioner som används för att implementera en byggordning.

AiSleep(frames)	Låter A.I:n vänta ett antal bildrutor.
AiNeed(unit-type)	Används för att tala om för A.I:n att en byggnad behövs byggas.
AiWait(type)	Används för att vänta tills en byggnad är klar.
AiForce(force, unit-type-1, count-1, … ,unit-type-N, count-N)	Används för att skapa en trupp med ett givet index och givna enheter.
AiForceRole(force, role)	Anger om en trupp är offensiv eller defensiv.
AiUpgradeTo(unit-type)	Används för att uppgradera en byggnad.
AiResearch(upgrade)	Startar en uppgradering.

Figur 9 Tabell över funktioner som används för att representera en byggordning i lua-script.

Byggordningen i figur 7 inleds alltså med att A.I:n konstruerar ett ”CityCenter”, vilket är huvudbyggnaden som behövs för att bl.a. skapa arbetare. Sedan väntar A.I:n på att byggnaden ska bli klar innan den fortsätter i nästa steg i byggordningen och bygger en ”Blacksmith”. Som det går att se i figur 7 är mönstret vad det gäller att konstruera byggnader så att A.I:n alltid väntar på att byggnaden i fråga ska byggas klart. Detta infördes av säkerhetsskäl för att undvika situationer då A.I:n försöker göra något den inte har tillåtelse att göra för att en viss byggnad inte finns ännu.

I slutet av mängden funktioner i figur 7 finns en funktion vid namnet ”AiTest3EndLoop()”. Denna startar en loop av funktioner som fortsätter att anropas tills spelets slut. Vid det här arbetets generering av byggordningar infördes loopen i figur 9 nedan som standard för alla genererade byggordningar.

```
- local end_loop_funcs = {
  function() return AiForce(5, {AiSoldier(), 10}) end,
  function() return AiWaitForce(5) end,
  function() return AiAttackWithForce(5) end,
  function() ai_test3_end_loop_func[player] = 0; return false end,
  }
```

Figur 10 Loop som fortsätter att exekveras till spelets slut.

Denna slutloop skapar en trupp bestående av 10 ”Soldiers” och attackerar med denna. Funktionerna ”AiWaitForce(force)” och ”AiAttackWithForce(force)” används här för att attackera med truppen.

4.1 Utvärderingmetod

Under körningen av samevolutionsalgoritmen användes en population på 6 individer och samevolveringen fortsatte tills dess 10 generationer uppnåddes. Anledningen till valet av dessa värden var begränsningar i spelmotorn, nämligen tiden det tog för en spelomgång mellan två A.I-motståndare att avslutas. Detta tog i snitt ca 3 minuter på banan Mysterious-dragon-isle. Om alla individer i en population på 6 stycken ska möta varandra i en spelomgång ska 15 stycken spelomgångar köras. Detta innebar alltså att en generation tog 15 * 3 = 45 minuter. Om populationen skulle varit ännu större – exempelvis 10 individer – skulle 45 spelomgångar behövt köras igenom, vilket alltså skulle tagit 45 * 3 = 135 minuter. Då det är 10 generationer, skulle hela körningen ha tagit 135 * 10 = 1350 minuter. Ska samevolutionsalgoritmen köras på flera kartor är detta en ohållbar körtid. Pga. dessa begränsningar blev tyvärr ett högre värde på parametrarna ohållbart.

Följande tre kartor användes för att utvärdera byggordningarna:

- Mysterious-dragon-isle – En stor och öppen bana.
- Death-in-the-middle – En bana av medelstorlek där mitten av banan består av en öppen yta.
- Nowhere-to-run – En liten bana där spelarna omringas av träd vid deras startpositioner. Detta medför att träden måste huggas ner för att spelarna ska kunna komma i kontakt med varandra.

Begränsningarna i spelmotorn gjorde det ohållbart att välja för många banor, eftersom det skulle ta för lång tid att köra samevolveringen på alla. Därför valdes tre banor ut med syftet att vara så olika varandra som möjligt för att främja diversifiering av strategier. Resultatet blev en liten bana, en medelstor bana och en stor bana.

Efter att en byggordning för varje bana har tagits fram via samevolution, ska dessa tre byggordningars styrka utvärderas. Detta sker, som tidigare nämnt, genom att den framtagna byggordningen får möta tre stycken olika beprövade byggordningar, som anses vara starka. De beprövade byggordningarna är följande:

- Land-Based-Attack (LBA) – En balanserad strategi som går ut på att bygga offensiva och defensiva trupper. Detta anses vara den svagaste av de tre byggordningarna (Wångsell, 2013).
- Soldier-Rush (SR) – En rush-strategi som bygger marktrupper tidigt i spelomgången i syfte att utföra en tidig attack. Som alla typer av rush-strategier, fungerar den bäst på mindre banor med kortare avstånd mellan spelarna.
- Knight-Rush (KR) – En strategi som går ut på att göra en relativt sen men stark attack med starka enheter. Denna strategi anses vara den starkaste av de tre

byggordningarna (Wångsell, 2013). Den är starkast på större banor med långt av stånd mellan spelarna.

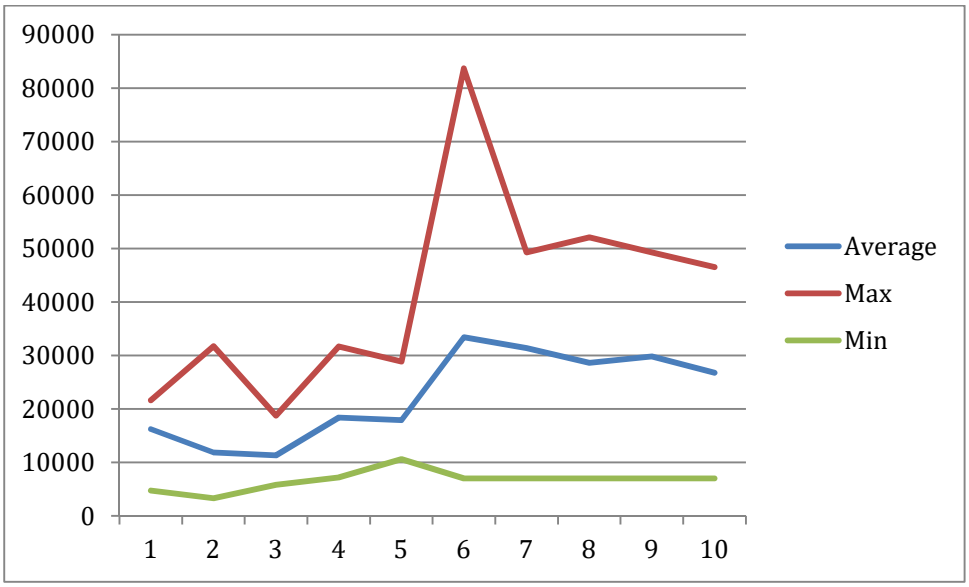
När en framtagen byggordning har spelat igenom sina spelomgångar mot de tre ovan nämnda skripten, bedöms den på hur stark den är efter hur många spelomgångar den har vunnit mot skripten. Det skulle även gå att göra en mer detaljerad bedömning – exempelvis att man mäter hur övertygande vinsten var. Den enklare bedömningen valdes för att hålla utvärderingen på en inte alltför komplex nivå.

5 Utvärdering

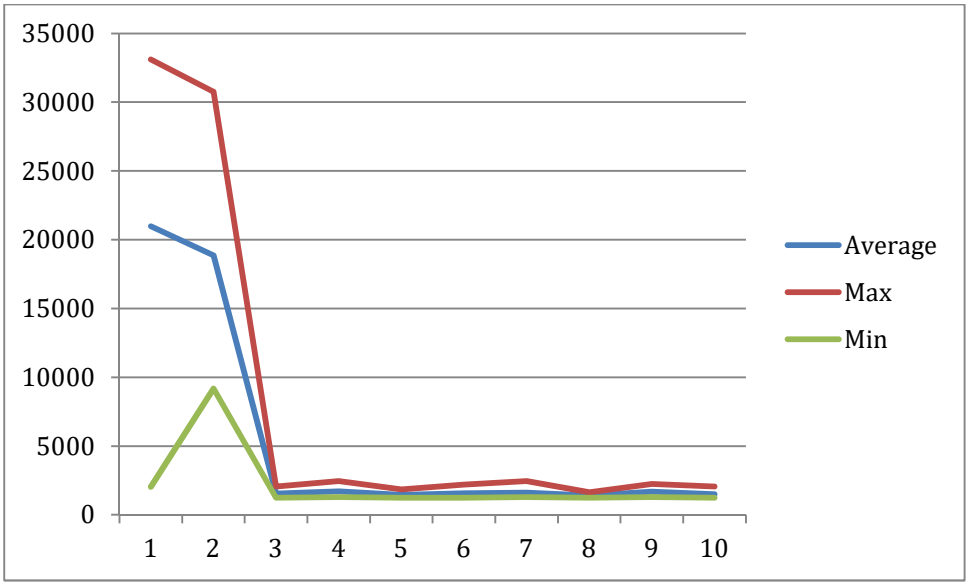
I den här delen presenteras resultaten från undersökningen med en efterföljande analys. Först presenteras tre stycken grafer över hur populationens fitnessvärde utvecklades när samevolutionsalgoritmen kördes på de tre kartorna. Efter detta visas den lua-kod som de tre byggordningarna som togs fram består av. Slutligen visas den vinstatistik som uppkom för de framtagna byggordningarna mot de tre vedertagna byggordningarna.

5.1 Resultat

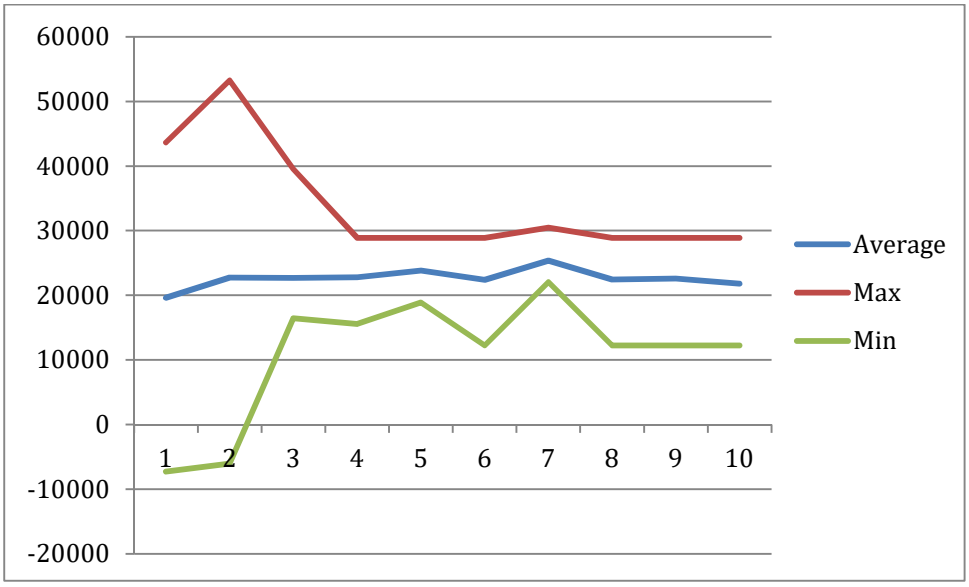
De tre graferna nedan visar hur populationens genomsnittliga fitness för varje karta förändrades under samevolveringsprocessen.



Figur 11 Fitnessgraf för Mysterious-dragon-isle



Figur 12 Fitnessgraf för Death-in-the-middle



Figur 13 Fitnessgraf för Nowhere-to-run

Som det går att se i figur 11, 12 och 13 ovan; har graferna för varje karta väldigt olika mönster. Detta är en konsekvens av fitnessberäkningen vid samevolution. Eftersom att fitnessvärdet baseras på hur pass väl genomerna presterar mot varandra, inte bara hur de löser ett specifikt problem, är inte nödvändigtvis trenden en genomsnittlig ökning av fitness vid varje generation. Det kan istället bli som i exempelvis figur 12, att det genomsnittliga fitnessvärdet sänks.

Följande tre byggordningar i figurerna nedan genererades fram för respektive karta.

```
local test6_funcs = {
    function() return AiSleep(AiGetSleepCycles()) end,
    function() return AiNeed(AiCityCenter()) end,
    function() return AiWait(AiCityCenter()) end,
    function() return AiSet(AiWorker(), 30) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBarracks()) end,
    function() return AiWait(AiBarracks()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiForce(0, {AiShooter(), 9}) end,
    function() return AiForce(0, {AiCatapult(), 2}) end,
    function() return AiForceRole(0, "defend") end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiForce(1, {AiShooter(), 8}) end,
    function() return AiForce(1, {AiCatapult(), 8}) end,
    function() return AiForceRole(1, "attack") end,
    function() return AiForce(2, {AiSoldier(), 5}) end,
    function() return AiForceRole(2, "attack") end,
    function() return AiTest6Endloop() end,
}
```

Figur 14 Byggordning för Mysterious-dragon-isle

Byggordningen för figur 14 ovan gör en del onödiga val, som att bygga 5 stycken blacksmiths (smedjor). Det behövs inte mer än 1 blacksmith för att utnyttja dess egenskaper. Den gör lite senare 1 defensiv styrka bestående av enheter som attackerar på distans, samt två offensiva styrkor som framförallt består av distansenheter men även några grundläggande marktrupper (”AiSoldier”) som attackerar i närstrid.

```
local test5_funcs = {
    function() return AiSleep(AiGetSleepCycles()) end,
    function() return AiNeed(AiCityCenter()) end,
    function() return AiWait(AiCityCenter()) end,
    function() return AiSet(AiWorker(), 30) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBarracks()) end,
    function() return AiWait(AiBarracks()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiForce(0, {AiSoldier(), 5}) end,
    function() return AiForce(0, {AiCatapult(), 3}) end,
    function() return AiForceRole(0, "defend") end,
    function() return AiForce(1, {AiSoldier(), 5}) end,
    function() return AiForce(1, {AiCatapult(), 8}) end,
    function() return AiForceRole(1, "defend") end,
    function() return AiForce(2, {AiSoldier(), 6}) end,
    function() return AiForce(2, {AiShooter(), 6}) end,
    function() return AiForce(2, {AiCatapult(), 4}) end,
    function() return AiForceRole(2, "defend") end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiForce(3, {AiShooter(), 2}) end,
    function() return AiForce(3, {AiCatapult(), 6}) end,
    function() return AiForceRole(3, "attack") end,
    function() return AiForce(4, {AiSoldier(), 8}) end,
    function() return AiForce(4, {AiShooter(), 4}) end,
    function() return AiForce(4, {AiCatapult(), 9}) end,
    function() return AiForceRole(4, "defend") end,
    function() return AiForce(5, {AiSoldier(), 4}) end,
    function() return AiForce(5, {AiCatapult(), 4}) end,
    function() return AiForceRole(5, "attack") end,
    function() return AiForce(6, {AiSoldier(), 3}) end,
    function() return AiForce(6, {AiShooter(), 3}) end,
    function() return AiForce(6, {AiCatapult(), 3}) end,
    function() return AiForceRole(6, "defend") end,
    function() return AiForce(7, {AiSoldier(), 2}) end,
    function() return AiForce(7, {AiShooter(), 5}) end,
    function() return AiForceRole(7, "defend") end,
    function() return AiForce(8, {AiSoldier(), 2}) end,
    function() return AiForce(8, {AiCatapult(), 5}) end,
    function() return AiForceRole(8, "defend") end,
    function() return AiTest5Endloop() end,
}
```

Figur 15 Byggordning för Death-in-the-middle

För banan Death-in-the-middle skapades byggordningen i figur 15 ovan. Denna byggordning gör fler styrkor jämfört med den i figur 14. Den gör även styrkorna tidigare. Däremot gör även denna byggordning några överflödiga byggnader – fyra lumbermills (sågverk) och två blacksmiths. Detta är dock mindre redundans än byggordningen i figur 14. De nio styrkor som skapas består av samma typer av enheter som för den tidigare byggordningen och sju av dem nio är defensiva och två är offensiva.

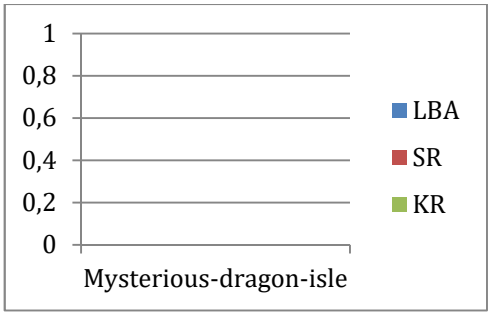
```
local test1_funcs = {
    function() return AiSleep(AiGetSleepCycles()) end,
    function() return AiNeed(AiCityCenter()) end,
    function() return AiWait(AiCityCenter()) end,
    function() return AiSet(AiWorker(), 30) end,
    function() return AiNeed(AiBarracks()) end,
    function() return AiWait(AiBarracks()) end,
    function() return AiForce(0, {AiSoldier(), 3}) end,
    function() return AiForceRole(0, "attack") end,
    function() return AiForce(1, {AiSoldier(), 3}) end,
    function() return AiForceRole(1, "attack") end,
    function() return AiNeed(AiLumberMill()) end,
    function() return AiWait(AiLumberMill()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiUpgradeTo(AiBetterCityCenter()) end,
    function() return AiWait(AiBetterCityCenter()) end,
    function() return AiNeed(AiStables()) end,
    function() return AiWait(AiStables()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiStables()) end,
    function() return AiWait(AiStables()) end,
    function() return AiNeed(AiStables()) end,
    function() return AiWait(AiStables()) end,
    function() return AiNeed(AiStables()) end,
    function() return AiWait(AiStables()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiNeed(AiBarracks()) end,
    function() return AiWait(AiBarracks()) end,
    function() return AiUpgradeTo(AiBestCityCenter()) end,
    function() return AiWait(AiBetterCityCenter()) end,
    function() return AiNeed(AiAirport()) end,
    function() return AiWait(AiAirport()) end,
    function() return AiNeed(AiStables()) end,
    function() return AiWait(AiStables()) end,
    function() return AiNeed(AiTemple()) end,
    function() return AiWait(AiTemple()) end,
    function() return AiNeed(AiAirport()) end,
    function() return AiWait(AiAirport()) end,
    function() return AiNeed(AiBlacksmith()) end,
    function() return AiWait(AiBlacksmith()) end,
    function() return AiTest1Endloop() end,
}
```

Figur 16 Byggordning för Nowhere-to-run

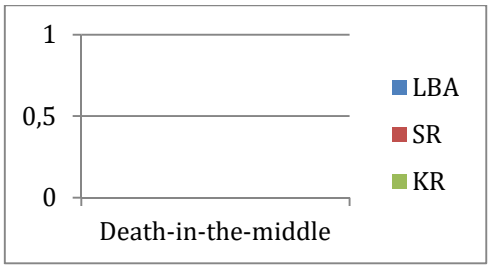
Den tredje och sista byggordningen som skapades var för banan Nowhere-to-run och visas i figur 16 ovan. Denna byggordning skiljer sig rätt markant från de två föregående, eftersom den gör en tidig attack. Denna attack består av två stycken offensiva styrkor med marktrupper som slåss på närstrid. Dessa styrkor skapas direkt efter den första baracken har byggts, vilket i detta fall är den första byggnad som byggs efter den obligatoriska huvudbyggnaden. Detta gör att hela byggordningen kan beskrivas som en rush-taktik. Resten av byggordningen består enbart av byggnader, som aldrig utnyttjas av A.I:n.

5.1.1 Vinststatistik

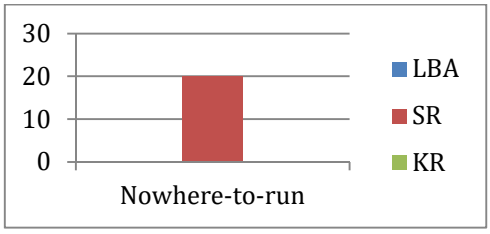
Nedan följer tre stycken stapeldiagram som visar hur vardera av de tre framtagna byggordningarna presterade i 20 spelomgångar mot de tre byggordningarna LBA, SR och KR. Staplarna visar hur många vunna spelomgångar den framtagna byggordningen hade. När ingen stapel visas innebär det att den inte lyckades vinna någon spelomgång.



Figur 17 Vinststatistik för Mysterious-dragon-isle



Figur 18 Vinststatistik för Death-in-the-middle



Figur 19 Vinststatistik för Nowhere-to-run

5.2 Analys

Resultaten visar på att ingen av de tre framtagna byggordningarna presterade särskilt bra mot någon av de beprövade strategierna LBA, SR och KR. Det enda undantaget är på banan

Nowhere-to-run, där den framtagna byggordningen vann alla sina spelade spelomgångar mot SR. Den spelade även lika alla sina spelomgångar mot LBA, vilket inte syns i statistiken då den endast visar vunna spelomgångar.

Anledningen till att de två första byggordningarna inte presterade särskilt bra på sina kartor är framförallt för att strategierna inte är speciellt optimerade - som tidigare nämnt byggs en del onödiga byggnader. Den tredje byggordningen har däremot en mycket mer optimerad start, då den bygger minimalt antal byggnader som krävs för att kunna skapa en tidig trupp av enkla marktrupper. Detta räcker för att spela lika mot LBA och att besegra SR, men inte för att vinna över den starkare KR.

5.3 Slutsatser

På grund av begränsningar hos spelmotorn kunde inte arbetet undersöka effekten av samevolution med större parametervärden. Det hade varit intressant att se om byggordningarna hade blivit mer optimerade om exempelvis en större population hade använts. Som resultatet blev nu, kan inte denna metod anses vara ett lovande tillvägagångssätt för att ta fram byggordningar. Den byggordning som var något konkurrenskraftig var den för banan Nowhere-to-run, men denna var så pass trivial att samevolveringsprocessen egentligen kan anses vara överflödig. En sådan strategi, dvs. en ”rush-strategi”, är väldigt enkelt uppbyggd och lätt att tänka ut och därför är det onödigt att använda en samevolveringsalgoritm för att ta fram en sådan.

6 Avslutande diskussion

6.1 Sammanfattning

Syftet med arbetet var att besvara frågeställningen: Går det att ta fram konkurrenskraftiga byggordningar med hjälp av genetiska algoritmer och samevolution? För att svara på denna fråga behövdes någon metod för att kunna utvärdera om en framtagen byggordning är konkurrenskraftig eller inte. Metoden som användes för detta var att låta en framtagen byggordning möta tre stycken redan beprövade byggordningar i ett antal spelomgångar. Det som även behövdes för att svara på frågeställningen var ett sätt att utföra samevolveringsprocessen för att kunna ta fram de byggordningar som ska utvärderas. En experimentmiljö skapades för att möjliggöra framtagningen respektive utvärderingen av byggordningar. Denna experimentmiljö var en modifierad version av spelmotorn Stratagus, som användes med tillägget Wargus för att efterlikna RTS-spelet WarCraft II. Tre stycken olika kartor användes i experimentet. För varje karta togs det fram en byggordning. Slutligen blev det alltså tre stycken byggordningar som skulle utvärderas. Utvärderingen skedde genom att låta de tre framtagna byggordningarna möta tre stycken beprövade byggordningar på den karta de togs fram på.

Resultatet blev att två av de tre framtagna byggordningarna förlorade alla sina spelomgångar och alltså uppnådde maximalt dåligt resultat. Den tredje byggordningen uppnådde något bättre resultat, då den vann sina spelomgångar mot en av byggordningarna och spelade lika mot en annan. Slutsatsen blev att samevolvering, i detta fall, inte kan anses vara ett lovande tillvägagångssätt för att ta fram konkurrenskraftiga byggordningar.

6.2 Diskussion

Att ta fram olika strategier i form av byggordningar genom att använda sig av genetiska algoritmer som använder samevolution, är en teknik som författaren anser skulle finna stort användningsutrymme vid utveckling av nya RTS-spel. Det behövs ingen direkt domänkunskap, utan denna teknik skulle kunna användas till ett helt nyutvecklad RTS-spel där ingen egentligen vet vad för strategier som är framgångsrika. En framtagen byggordning skulle också kunna användas som inspiration till att manuellt ta fram byggordningar, istället för att ta fram dem från scratch.

Resultatet från arbetet var inte så lovande, då byggordningarna inte höll måttet avseende konkurrenskraftighet. Resultatet är dock baserat på parametervärden hos algoritmen som är tvivelaktiga. Det är en förhållandevis låg populationsmängd som använts; endast en population på 6 individer. Antalet generationer som kördes igenom var 10 stycken, vilket också kan anses vara ett lågt antal. Andra arbeten som (Ponsen, 2004) och (Wångsell, 2013) använde sig av en population på 50 respektive 20 individer. Anledningen till valet av det låga värdet på parametrar var som tidigare nämnt att en spelomgång tog för lång tid att spela igenom för att man skulle hinna med ett större antal. Värt att nämna är också att körningstiden för en GA som använder sig av samevolution, växer exponentiellt med populationen, eftersom varje individ ska möta varje annan individ. Detta gör att det är enklare att använda en större population om man använder sig av en GA som inte bygger på samevolution.

Då parametervärdena hos algoritmen var förhållandevis låga, kan man inte med säkerhet dra slutsatsen att den här tekniken inte kan rekommenderas, även om resultatet tyder på det. Vad hade hänt om algoritmen hade kört i 200 generationer med en population på 20 individer? Är de låga parametervärdena den huvudsakliga anledningen till att byggordningarna inte blev konkurrenskraftiga eller beror det på tekniken i sig? Något som däremot går att konstatera utifrån statistiken är att byggordningarna som togs fram blev väldigt olika. Framförallt går det att se, om man jämför de tre fitnessgraferna i kapitel 5.1, att evolutionen skiljde sig mycket åt i de tre körningarna. Det här är inte helt oförutsägbart, eftersom arbetets metod var att slumpa fram en initial population och låta individerna möta varandra. Det fanns ingenting som styrde evolutionen i en viss riktning, förutom att viss belöning skedde om byggnader och enheter förstördes. För att man skulle ha fått en någorlunda bättre kontroll på utvecklingen av populationen, hade man med fördel kunnat använda sig av exempelvis Case-injection (Ballinger and Louis, 2013b), där man lägger in färdiga mänskliga strategier som individerna från träna mot utöver varandra. Avsaknaden av ett sådant riktmärke att förhålla sig mot, innebar att det sannolikt inte enbart berodde på de låga parametervärdena att de framtagna byggordningarna inte höll måttet; utan även på tekniken i sig. Vidare är fitnessvärdet vid samevolution subjektivt, eftersom det bara representerar hur bra individen är i förhållande till andra individer i populationen. Detta säger inte speciellt mycket jämfört med traditionell GA, där fitnessvärdet är objektivt, då det reflekterar hur väl individerna löser ett externt problem.

Arbetet skulle gå att återupprepas genom att implementera en samevolveringsalgoritm i någon spelmotor som beter sig som en WarCraft II – klon. Algoritmen skulle använda samma parametrar som det här arbetet – dvs. en population på 6 och antal generationer på 10 – och inte använda sig av enheten torn. Däremot skulle resultatet inte nödvändigtvis vara detsamma, på grund av slumpfaktorn vid genereringen. Det skulle, om än osannolikt, vara möjligt att det redan vid den initiala populationen skulle finnas byggordningar som håller hög konkurrenskraft. De låga parametervärdena och avsaknaden av ett riktmärke, gör att det blir svårt att förutse hur de slutgiltiga byggordningarna skulle se ut vid återupprepade körningar av algoritmen (med riktmärke menas, som diskuterat tidigare i avsnittet, en färdig byggordning som är beprövat bra). Vidare kan påpekas att det går att använda ett annat RTS-spel än WarCraft II med samma samevolutionsalgoritm. Däremot behövs det i sådana fall tas fram tre andra beprövade strategier att utvärdera mot, istället för LBA, SR och KR som användes i det här arbetet (eftersom dem är specifika för WarCraft II).

Syftet med att ta fram starka byggordningar är trots allt för att dessa ska kunna användas för att skapa utmanande A.I-spelare för en mänsklig spelare. Det är därför relevant att diskutera hur en mänsklig spelare skulle prestera mot det här arbetets byggordningar. Slutsatsen att byggordningarna inte höll måttet var baserat på att de fick möta beprövat starka byggordningar, men hade en annan slutsats dragits om det var mänskliga spelare som fick spela mot de framtagna byggordningarna? Det går naturligtvis inte att med säkerhet förutse vad en sådan utvärdering hade gett för resultat, exempelvis hade de mänskliga spelarnas skicklighet spelat roll. Däremot går det att se att de byggordningar som togs fram i det här arbetet hade påtagliga brister, som onödigt byggande. En någorlunda van spelare av WarCraft II hade antagligen besegrat sådana ineffektiva byggordningar, men det går såklart inte att med säkerhet konstatera utan att ha testat.

Sökrymden hos samevolutionsalgoritmen i arbetet hade redan begränsats till att inte innefatta så kallade ”korrupta byggordningar”, dvs. byggordningar som inte är möjliga genomföra fullt ut. Det hade varit möjligt att begränsa sökrymden ytterligare för att eliminera att byggordningar innefatta massa onödigt byggande av överflödiga byggnader. Det här var som tidigare nämnt en påtaglig brist de framtagna byggordningarna hade. En begränsning av sökrymden på detta sätt hade gett mer optimerade byggordningar tidigare i evolutionsförloppet, vilket är önskvärt då man är begränsad av låga parametervärden.

Syftet med ett RTS-spel är att det ska vara underhållande för en mänsklig person att spela. Anledningen till att man vill ta fram en konkurrenskraftig byggordning, som det här arbetet har fokuserat på, är just för att det ska bidra till underhållningsvärdet i spelet. Det finns undersökningar som visar på att en av de huvudsakliga anledningar som gör att folk spelar spel är för att hamna i det s.k. flow-tillståndet (Murphy, n.d.). Om färdigheten hos en spelare är hög men utmaningen låg ger detta en känsla av leda (Csikszentmihalyi, 2009). För att hamna i flow måste utmaningen då ökas. Ett enkelt sätt att öka utmaningen i fallet med RTS-spel, är att ge A.I:n fördelar, som att den samlar resurser snabbare än spelaren. Detta gör däremot att A.I:n får fördelar som den mänsklige spelaren inte har, vilket kan upplevas som orättvist. Därför är det bättre alternativet att istället ta fram en A.I med en strategi som är utmanande att spela mot. Då syftet med arbetet var just att ta fram strategier som är konkurrenskraftiga, dvs. utmanande att spela mot, kan arbetet kopplas till ett försök att bidra till underhållningsvärdet för RTS-spel. Tekniken var även tänkt att vara ett mindre resurskrävande alternativ jämfört med att manuellt skripta byggordningarna. Detta skulle i sin tur bidra till spelbranschen genom att minska kostnaderna för spelutveckling.

Arbetet är inte bara till nytta för spelbranschen. Samevolutionsalgoritmer har ett brett användningsområde och kan användas till att ta fram lösningar på en mängd olika problem. Det här arbetet har mer specifikt använt sig av competitive coevolution, eftersom individer blir belönade till bekostnad av andra individer. Just detta tillvägagångssätt har använts inom finans för att exempelvis ta fram optimala strategier för en leverantör inom elbranschen på en elmarknad (Tiguercha et al., 2013). Finansmarknader kan liknas vid ett spel där olika aktörer har olika strategier som tävlar mot varandra. Det är därför inte helt långsökt att göra en liknande undersökning där individernas strategier är av finansiell karaktär, snarare än strategier för RTS-spel.

6.3 Framtida arbete

För framtiden skulle arbetet framförallt dra nytta av att man optimerade spelmotorn så att en spelomgång tar väsentligt mindre tid än vad den gjorde. Detta medför att man kan testa med högre parametervärden på samevolutionen än vad som gjordes nu, vilket skulle kunna ge en bättre bild av hur konkurrenskraftiga byggordningar det går att ta fram med detta tillvägagångssätt.

Ett alternativ är att genomföra undersökningen i en annan spelmotor. Ballinger och Louis (2011) använde sig av den egenutvecklade spelmotorn WaterCraft (2011) i sitt arbete med samevolution. Denna spelmotor är tänkt att vara en kopia av spelet StarCraft (Blizzard Entertainment, 1998) och inte av WarCraft II (Blizzard Entertainment, 1995). Framförallt designades spelmotorn med hänsyn tagen att den skulle användas för evolutionära beräkningar (Ballinger och Louis, 2013a). En annan spelmotor som skulle kunna lämpa sig

bra är ORTS, vilken är utvecklad med ändamålet att kunna göra A.I- forskning med (Buro, 2005).

Förutom att ha möjlighet att köra samevolutionsalgoritmen med högre parametervärden, är det, utifrån det här arbetets resultat, en bra idé om man inte enbart använder sig av den egna populationen att träna sig mot. Samevolveringen som sker blir väldigt oförutsägbar. Fitnessvärdet är, som tidigare diskuterat, subjektivt och ett högt fitnessvärde säger inte så mycket. En bättre idé är förslagsvis att man likt Ballinger och Louis (2013b) delar upp individerna i en population och ett ”teachset”. Det förstnämnda är likt det här arbetet den population av individer som ska evolveras och slutligen ska användas för lösningen. Teachset är en mängd individer som populationen tränar mot. I det här arbetet var population och teachset identiska, dvs. de individer populationen tränade mot var populationen själv. En bra idé är att exempelvis använda sig av något Ballinger och Louis (2013b) benämnde Case-injection, som i korta drag innebär att ett antal mänskliga strategier, som är beprövat bra mot samevolverade strategier, läggs in i det teachset som varje generations individer får möta. På detta sätt hoppas man kunna få samevolutionen att ta fram strategier som kan besegra dessa mänskliga strategier.

Det skulle även gå att använda sig av en liknande samevolutionsalgoritm för att göra ett arbete utanför spelbranschen – exempelvis inom finans som tidigare nämnts. Genrepresentationen hade i sådant fall behövt ändras om för att passa just de spelregler som skulle råda under ett sådant arbete.

Referenser

Aha, D.W., Molineaux, M., Ponsen, M., 2005. Learning to win: Case-based plan selection in a real-time strategy game, in: Case-Based Reasoning Research and Development. Springer, pp. 5–20.

Angeline, P.J., Pollack, J.B., 1993. Competitive Environments Evolve Better Solutions for Complex Tasks., in: ICGA. pp. 264–270.

Bäck, T., 1996. Evolutionary Algorithms in Theory and Practice.

Ballinger, C., Louis, S., 2013a. Robustness of coevolved strategies in a real-time strategy game, in: Evolutionary Computation (CEC), 2013 IEEE Congress on. IEEE, pp. 1379–1386.

Ballinger, C., Louis, S., 2013b. Finding robust strategies to defeat specific opponents using case-injected coevolution, in: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. IEEE, pp. 1–8.

Ballinger och Louis, 2011. WaterCraft.

Blizzard Entertainment, 2010. StarCraft II.

Blizzard Entertainment, 1998. StarCraft.

Blizzard Entertainment, 1995. WarCraft II.

Buro, M., 2005. ORTS - a free software rts game engine.

Churchill, D., Buro, M., 2011. Build Order Optimization in StarCraft., in: AIIDE.

Csikszentmihalyi, M., 2009. Flow: The psychology of optimal experience, Nachdr. ed, Harper Perennial Modern Classics. Harper [and] Row, New York.

Ensemble Studios, 2002. Age of Mythology.

Ficici, S.G., 2004. Solution concepts in coevolutionary algorithms. Citeseer.

Firaxis Games, 2010. Civilization V.

Goldberg, D.E., 1989. Genetic algorithms in Search, Optimization, and Machine learning, 1 edition. ed. Addison-Wesley Professional.

Halo Wars, 2009. . Ensemble Studios.

Mitchell, M., 1996. An introduction to genetic algorithms, Complex adaptive systems. MIT Press, Cambridge, Mass.

Murphy, C., n.d. 5.6 Why Games Work and the Science of Learning Why Games Work and the Science of Learning.

Ponsen, M., 2004. Improving adaptive game AI with evolutionary learning. Citeseer.

The Stratagus Team, 2007. Stratagus.

The Wargus Team, 2007. Wargus.

Tiguercha, A., Ladjici, A.A., Boudour, M., 2013. Suppliers' optimal biding strategies in day-ahead electricity market using competitive coevolutionary algorithms, in: Systems and Control (ICSC), 2013 3rd International Conference on. IEEE, pp. 821–826.

Wångsell, J., 2013. Användning av genetiska algoritmer för framtagning och utvärdering av byggordningar i RTS-spel.

Weber, B.G., Mateas, M., 2009. Case-Based Reasoning for Build Order in Real-Time Strategy Games., in: AIIDE.