Examensarbete

HÖGSKOLAN I SKÖVDE

ARTIFICIELL INTELLIGENS

Prestanda hos beteendeträd och Hierarchical Task Network.

ARTIFICIAL INTELLIGENCE

Performance of behavior trees and Hierarchical Task Network.

Examensarbete inom huvudområdet Datalogi
Grundnivå 30 högskolepoäng
Vårtermin 2014

Joel Juvél

Handledare: Daniel Sjölie
Examinator: Henrik Gustavsson

Sammanfattning

Detta arbete undersöker skillnader i tidseffektivitet mellan beteendeträd och Hierarchical Task Network. En enklare spelprototyp av typen top-down 2D shoot-em-up implementerades. Spelprototypen använder två typer av autonom motspelare så kallade botar. En bot för beteendeträd och en bot för Hierarchical Task Network.

Spelprototypen mäter körtiden för varje typ av bot i sex olika situationer. Varje situation svarar mot ett bestämt beteende hos boten. Ett beteende kan brytas ned i en samling uppgifter som boten kan utföra. Spelprototypen mäter körtiden för att bestämma ett enskilt beteende i en specifik situation. Resultaten från mätningarna tyder på att beteendeträd har bättre tidseffektivitet än Hierarchical Task Network.

Nyckelord: Artificiell intelligens, beteendeträd, Hierarchical Task Network, prestanda, spel, tidseffektivitet

Innehållsförteckning

1 Introduktion....1

2 Bakgrund....2

2.1 Beteendeträd....2

2.2 Planering....5

2.2.1 Goal-Oriented Action Planning (G.O.A.P)....5

2.2.2 Hierarchical Task Network (HTN)....5

3 Problemformulering....8

3.1 Metodbeskrivning....8

4 Implementation....10

4.1 Spelprototyp och beteenden....10

4.2 Botarkitektur....12

4.3 AI-kontroller....13

4.4 Pool-allokator....16

4.5 Testläge....16

4.6 Pilotstudie....18

5 Utvärdering....20

5.1 Mätmiljö....20

5.2 Resultat av körningar....20

5.3 Analys....22

5.4 Slutsatser....22

6 Avslutande diskussion....23

6.1 Sammanfattning....23

6.2 Diskussion....23

6.3 Framtida arbete....25

Referenser....26

1 Introduktion

Artificiell intelligens i datorspel har genom åren blivit allt mer komplex. Denna utmaning kräver tekniker som kan hantera den ökande komplexiteten som spelutvecklare ställs inför. Detta arbete fokuserar på två tekniker, beteendeträd och Hierarchical Task Network. Dessa tekniker har med fördel använts i flertalet spel. Båda tekniker har studerats var för sig i olika situationer. Det finns dock relativt sparsamt med information där dessa tekniker jämförs sida vid sida.

Bakgrunden i rapporten (kapitel 2) är tänkt att ge läsaren en inblick i framförallt teknikerna beteendeträd och Hierarchical Task Network. Även några relaterade alternativa tekniker tas upp, så som Goal-Oriented Action Planning.

En problemformulering presenteras i kapitel 3. Här ligger problemfokus på tidseffektiviteten hos teknikerna beteendeträd och Hierarchical Task Network, där hypotesen är att beteendeträd har bättre tidseffektivitet i vissa situationer. Motivationen till arbetet är att datorspel körs i realtid och därför ställer krav på tidseffektiviteten.

För att kunna mäta tidseffektiviteten implementerades en enklare spelprototyp som beskrivs i kapitel 4. Spelprototypen innehåller två typer av artificiella motspelare, så kallade botar. Den ena bottypen använder tekniken beteendeträd och den andra botttypen använder Hierarchical Task Network. Spelprototypen innehåller ett testläge där botarna utsätts för sex olika situationer. En situation är byggd för att resultera i ett specifikt beteende hos botarna. Ett beteende kan brytas ned till en plan som består av en samling uppgifter som boten kan utföra. Spelprototypen mäter körtiden som krävs för att formulera en plan. I kapitel 5 utvärderas mätdata från spelprototypen.

2 Bakgrund

Artificiell intelligens i datorspel har genom åren blivit allt mer komplex. Någon form av artificiell intelligens används ofta för att styra beteenden för artificiella motspelare. Allt eftersom spelmediet har utvecklats har även kraven på artificiell intelligens ökat markant. En artificiell motspelare, i ett datorspel, kan förväntas ta rimliga och logiska beslut. Den ökade komplexiteten på beteenden hos artificiella motspelare har blivit en allt större utmaning för spelindustrin (Isla, 2005; Orkin, 2006). Viktiga egenskaper för moderna tekniker är möjligheten att kunna återanvända och underhålla existerande lösningar. Men även tidseffektivitet är en viktig egenskap. Orkin (2006) beskriver hur komplexiteten vid användandet av finita tillståndsmaskiner var en motivation till utvecklingen av en alternativ teknik. Detta för att bättre kunna hantera riskerna vid förändringar i långt skridna projekt.

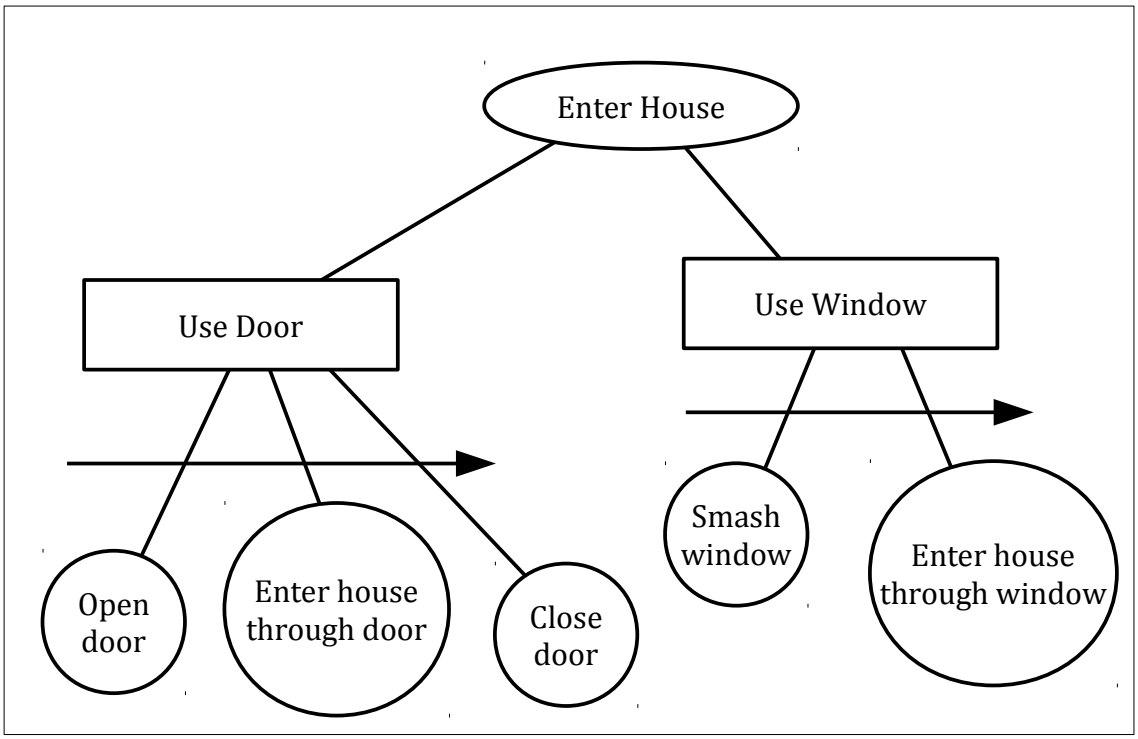
2.1 Beteendeträd

I datorspelet Halo 2 (Bungie Software, 2004) användes beteendeträd för att implementera artificiell intelligens (Isla, 2005). Viss forskning har utförts på beteendeträd. Bland annat har kombinationen av beteendeträd och genetiska algoritmer undersökts (Lim, Baumgarten & Colton, 2010; Perez, Nicolau, O'Neill & Brabazon, 2011). Beteendeträd har även använts för andra ändamål än datorspel. Bland annat har det forskats om användandet av beteendeträd för styrning av obemannade flygfarkoster (Ögren 2012; Klöckner 2014).

I Figur 1 nedan visas ett exempel på hur ett beteendeträd skulle kunna se ut. Ett beteendeträd är en form av riktad acyklisk graf som beskriver en hierarkisk tillståndsmaskin. Detta ska inte förväxlas med en vanlig tillståndsmaskin som endast befinner sig i ett tillstånd åt gången. En hierarkisk tillståndsmaskin kan befinna sig i flera tillstånd åt gången genom att ärva tillstånd. Om det aktiva tillståndet i Figur 1 vore *Open door*. Då är även tillstånden *Use door* och *Enter House* aktiva genom arv.

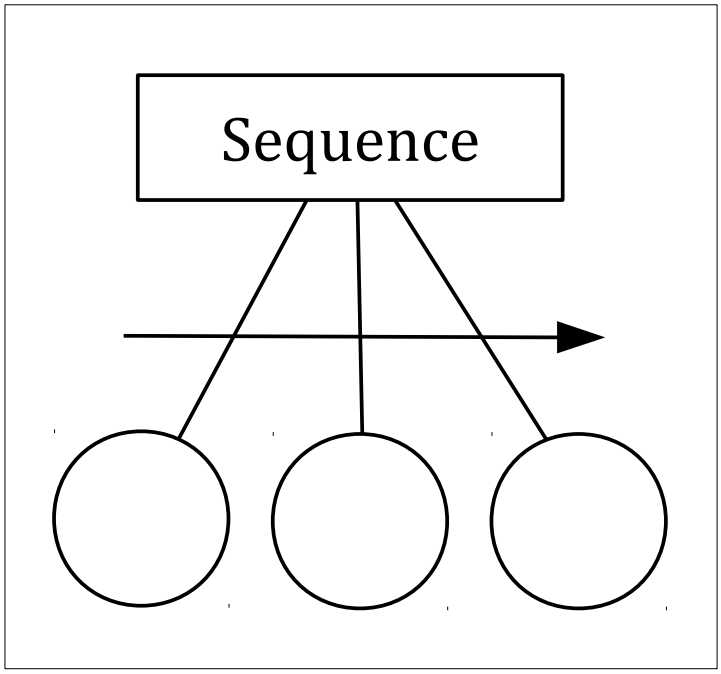
Ett beteendeträd byggs upp av olika typer av noder. En lövnod utför någon form av uppgift. En uppgift skulle till exempel kunna vara att motspelaren skall öppna en dörr. Det finns många typer av noder som kan användas för att bygga upp ett beteendeträd och det är enkelt att lägga till nya typer av noder. Ett exempel på detta är Johansson och Dell'Acqua (2012) som utökar beteendeträd med en ny typ av nod för att enklare kunna representera känslor hos artificiella motspelare.

Grundfunktionen för alla noder i ett beteendeträd är att de kan evalueras. När en nod evalueras kan evalueringen antingen lyckas eller misslyckas. Tre vanliga typer av noder i ett beteendeträd är selector, sequence och condition (Johansson, et al. 2012; Champandard, 2007). Johansson, et al. (2012) menar att det inte finns någon formell definition av beteendeträd och väljer därför att utgå från den vanligt förekommande definitionen som Champandard (2007) använder. Det är även denna definition av beteendeträd som detta arbete kommer att utgå från.



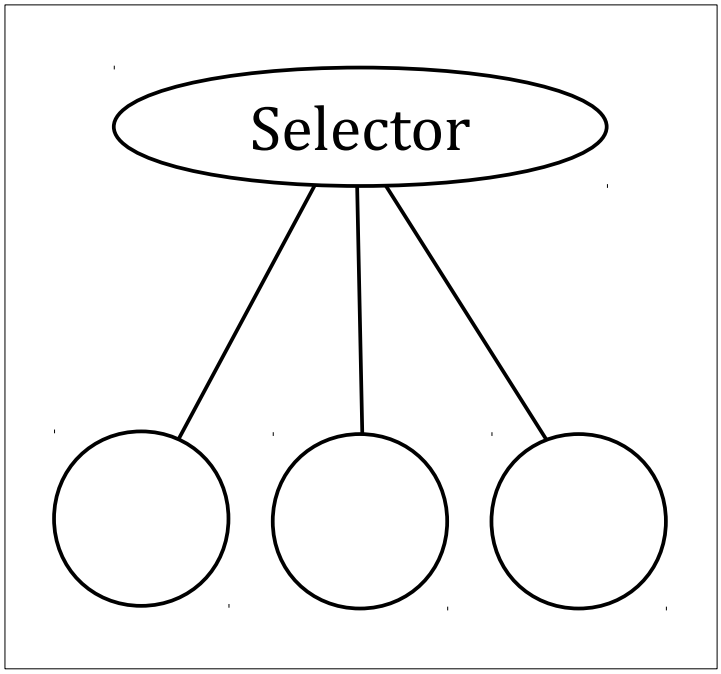
Figur 1 Exempel på ett beteendeträd

I Figur 2 nedan visas en sequencenod. En sequencenod evaluerar barnnoder från vänster till höger. Om en barnnod misslyckas med sin evaluering avbryts fortsatt evaluering och sequencenoden misslyckas med sin evaluering. Om alla barnnoder lyckas med sin evaluering så lyckas även sequencenoden med sin evaluering. Evalueringsresultatet från en sequencenod kan således jämföras vid en boolesk konjunktion mellan resultaten från barnnoderna.



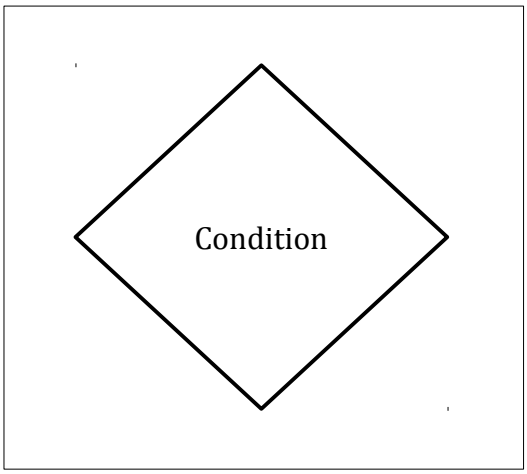
Figur 2 Sequencenod

I Figur 3 nedan visas en selectornod. En selectornod evaluerar barnnoder från vänster till höger. Om en barnnod misslyckas med sin evaluering fortsätter selectornoden med att evaluera nästa barnnod och så vidare. Om alla barnnoder misslyckades, då misslyckades även selectornoden. Om någon barnnod skulle lyckas med sin evaluering så avbryter selectornoden vidare evaluering av barnnoder och lyckas med sin evaluering. Evalueringsresultatet från selectornoden kan jämföras vid en boolesk disjunktion mellan resultaten från barnnoderna.



Figur 3 Selectornod

I Figur 4 nedan visas en condition-nod. En condition-nod kör någon form av villkorsfunktion som avgör om evalueringen lyckas eller misslyckas. En condition-nod kan användas för styra evalueringen av beteendeträdet.



Figur 4 Condition-nod

Beteendeträd kan användas för att uppnå olika former av *mål*. Målet kan till exempel vara att träda in i ett hus i en spelvärld. En *plan* är en samling uppgifter som skall utföras för att uppnå ett specifikt mål. Enligt Perez, et al. (2011) är beteendeträd reaktiva i sin natur. Firby

(1987) menar att reaktiva system bygger om eller förändrar planer under körning beroende på förändrade förutsättningar. En evaluering av ett beteendeträd utgår alltid från världens nuvarande tillstånd. Där världens tillstånd kan innehålla information om spelvärden. Till exempel om en dörr är öppen eller stängd.

2.2 Planering

Planering är ett traditionellt område inom artificiell intelligens. Planerare har använts med framgång på olika typer av problem från lagerlogistik till militäroperationer och datorspel. En planerare ges någon form av mål. Från detta mål formuleras sedan en plan för att uppnå detta mål. En klassisk planerare är STRIPS (Fikes & Nilsson, 1971). STRIPS är en förkortning av *Stanford Research Institute Problem Solver*. STRIPS används för att hitta en plan som kan transformera en världsmodell från nuvarande tillstånd till ett måltillstånd. Det önskade måltillståndet uppfyller någon form av villkor. En världsmodell representeras av en samling formler (*well-formed fomulas*). För att förändra världsmodellen används operatorer. En operator kräver ett giltigt förhandstillstånd. Vidare har en operator en effekt som förändrar världsmodellen. Algoritmen STRIPS försöker hitta en sekvens av operatorer som kan transformera världens nuvarande tillstånd till ett måltillstånd där det sökta målvillkoret är uppfyllt. Denna sekvens av operatorer kallas för en plan. STRIPS har använts som grund för bland annat Goal-Oriented Action Planning (Orkin, 2003, 2006).

2.2.1 Goal-Oriented Action Planning (G.O.A.P)

Goal-Oriented Action Planning (Orkin, 2003, 2006) är en planerare som använts i bland annat datorspelet F.E.A.R. (Monolith Productions, 2006). Det finns även exempel där Goal-Oriented Action Planning har applicerats på problemdomäner utanför datorspel, till exempel ruttplanering för missiler (Doris & Silvia, 2007).

Goal-Oriented Action Planning är en förenklad STRIPS baserad planerare. I Goal-Oriented Action Planning utförs olika typer av handlingar. Detta kan jämföras med operatorerna i STRIPS. En handling kräver ett giltigt förhandstillstånd och utför en effekt på världens tillstånd. Världens tillstånd är en samling variabler som beskriver olika aspekter av spelvärlden. Förhandstillståndet måste gälla för att handlingen skall kunna appliceras. En stor fördel med Goal-Oriented Action Planning är att mål och handlingar blir frikopplade från varandra.

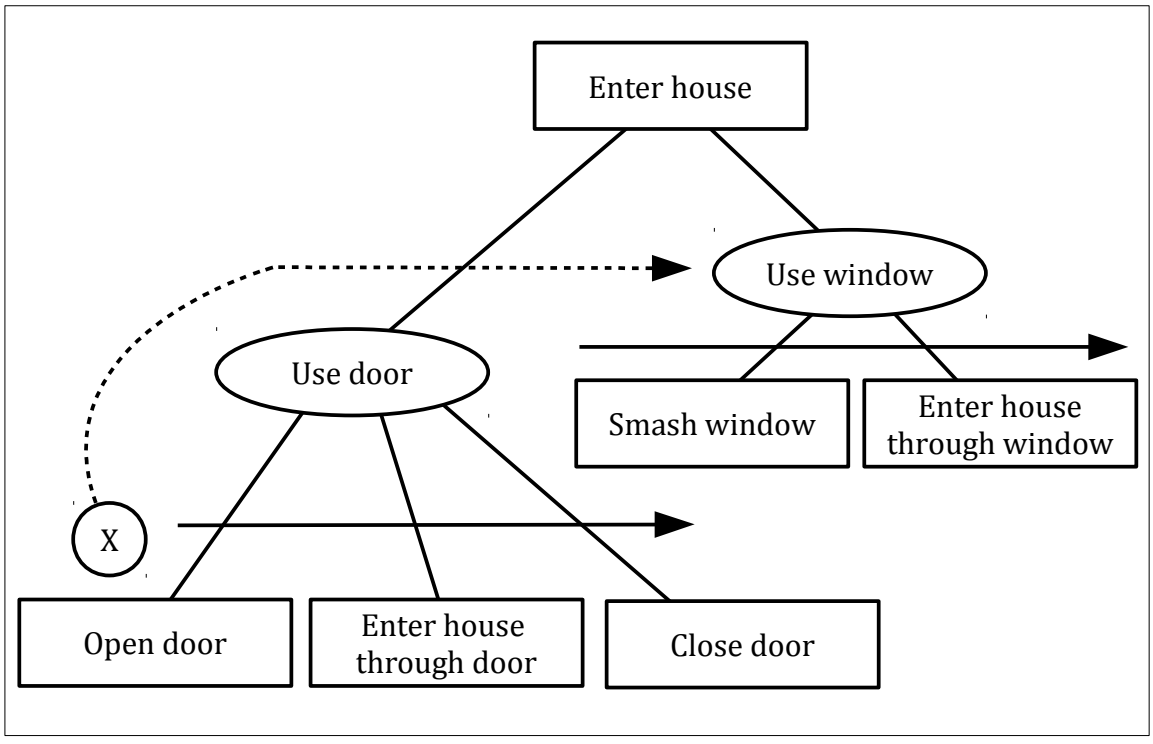
En viktig aspekt med Goal-Oriented Action Planning är tidseffektivitet. För att minimera planeringstiden används den generella sökalgoritmen A* (A-Star) (Hart, Nilsson & Raphael, 1968, 1972). A* algoritmen använder heuristik för att försöka hitta en väg, med låg kostnad, i en graf. I Goal-Oriented Action Planning används A* för att minska tiden som krävs för att söka bland möjliga handlingar. För att kunna söka bland handlingar ges varje enskild handling en kostnad. Kostnaden är ett vanligt flyttal. Detta möjliggör för den heuristiska funktionen att beräkna ett mått på den total kostnaden till måltillståndet. Här är en så låg kostnad som möjligt är att föredra. En intressant aspekt är att sökningen sker omvänt. Det vill säga från måltillståndet till nuvarande tillstånd. En nackdel med Goal-Oriented Action Planning är svårigheten att styra i vilken ordning handlingar får utföras.

2.2.2 Hierarchical Task Network (HTN)

Hierarchical Task Network (HTN) är ett samlingsnamn för planerare som använder hierarkiska nätverk av uppgifter för att fomulera en plan (Nau, Au, Ilghami, Kuter, Murdock,

Wu & Yaman, 2003). Enligt Champandard, Verweij och Straatman (2009) används en HTN-planerare i datorspelet Killzone 2 (Guerrilla Games, 2009).

Detta arbete kommer fokusera närmare på HTN-planeraren SHOP (Nau, Cao, Lotem & Muñoz-Avila, 1999). SHOP är en relativt enkelt algoritm. Det finns två typer av uppgifter i SHOP. Sammansatta uppgifter och primära uppgifter. En sammansatt uppgift löses med hjälp av en metod. Det kan finns flera metoder för att lösa en enskild uppgift. Varje metod kräver ett giltigt förhandstillstånd i världen för att kunna appliceras. Metoden delar sedan upp den sammansatta uppgiften i flera mindre uppgifter. En primär uppgift är en uppgift som inte kan delas upp i mindre bitar. Den kan då appliceras som en operator som förändrar världens tillstånd. Om en situation uppstår där en metod misslyckas med att hitta en metod för att dela upp en sammansatt underuppgift, försöker istället algoritmen hitta en alternativ metod för att dela upp uppgiften. I Figur 5 nedan visas ett exempel på hur ett Hierarchical Task Network skulle kunna se ut. I detta exempel är *Enter house* en sammansatt uppgift. Två metoder, *Use door* och *Use window,* kan appliceras i detta fallet. Vidare illustreras, med ett kryss och en streckad pil, vad som händer när en metod misslyckas med uppdelningen. I detta fallet misslyckades metoden *Use Door* med att dela upp den sammansatta uppgiften *Open Door*. Detta leder till att metoden *Use Door* inte längre kan användas och algoritmen försöker istället att använda nästa metod som i detta fallet är *Use Window*.



Figur 5 Ett exempel på ett Hierarchical Task Network (HTN).

En stor skillnad mellan STRIPS baserade planerare och planerare baserade på HTN är kunskapen om problemdomänen som finns i HTN. Eftersom ett HTN byggs för hand har någon brutit ned problemen i mindre delproblem och specificerat en inbördes ordning mellan uppgifter. I STRIPS och G.O.A.P har varje operator/handling ett förhandstillstånd och en effekt. Med STRIPS blir det problematiskt om ordningen som handlingar kan utföras i spelar roll. Detta kan till viss del kontrolleras i G.O.A.P genom balansering av kostnader. I SHOP elimineras problemet med ordning helt och hållet genom användandet av metoder.

Uppdelningen av uppgifter sker alltid från vänster till höger. Den resulterande planen i SHOP är alltid totalt ordnad.

3 Problemformulering

Syftet med detta arbete är att utvärdera implementationen av två olika tekniker för artificiell intelligens. Dessa två tekniker är beteendeträd och Hierarchical Task Network. Frågeställningen som arbete försöker besvara är hur tidseffektiviteten skiljer sig åt mellan implementationerna och hypotesen är att beteendeträd har bättre tidseffektivitet än Hierarchical Task Network i problemdomäner med flera möjliga val, där något val misslyckas.

Ett exempel på en problemdomän för en autonom spelare i ett dator spel kan vara att *träda in i ett hus med två ingångar*. Då kan ett val vara att välja en av två möjliga ingångar. Hypotesen grundar sig i att beteendeträd är reaktiva i sin natur (Perez, et al. 2011) och därför utgår från världens nuvarande tillstånd. I algoritmen SHOP (Nau, et al. 1999) måste varje förändring av världens tillstånd följas för att kunna applicera alternativa metoder utifall att någon metod skulle misslyckas. Vidare utför SHOP algoritmen en dynamisk plangenerering som expanderar sammansatta uppgifter i mindre deluppgifter. Expansionen kräver att algoritmen gör plats i den nuvarande planen för att kunna spara deluppgifterna.

Tidseffektivitet är relevant ur den synvinkeln att datorspel körs i realtid. Isla (2005) menar att uppemot 50 olika beteenden användes i beteendeträdet för grundbeteenden i datorspelet Halo 2 (Bungie Software, 2004). Champandard, et al. (2009) nämner att planeraren i Killzone 2 (Guerrilla Games, 2009) genererade ungefär 500 planer per sekund, i flerspelarläget, med 14 aktiva artificiella motspelare. Vidare hade varje artificiell motspelare 138 möjliga beteenden.

Om varje artificiell motspelare i ett datorspel har en stor mängd komplexa beteenden, samt att det finns många aktiva artificiella motspelare, då ställer detta krav på tidseffektiviteten. Beteendeträd och Hierarchical Task Network är två tekniker som använts datorspel. Det verkar dock inte finnas mycket information som jämför dem. En jämförelse i tidseffektivitet är ett relevant mått då det går att dra slutsatser som att till exempel en algoritm är tre gånger snabbare än en annan algoritm i en specifik situation. Detta kan tillsammans med ytterligare faktorer användas av spelutvecklare för att göra en avvägning om en algoritm lämplig att användas i någon form av datorspel.

3.1 Metodbeskrivning

Metoden som skall användas för att utvärdera de två teknikerna beteendeträd och Hierarchical Task Network är experiment. Användning av intervjustudier, enkätundersökningar eller andra kvalitativa metoder utesluts då dessa inte är lämpliga för att mäta mycket korta tidsintervall. Eftersom grundproblemet handlar om kontrollerad tidmätning är automatiserade tester mer lämpade för att utföra uppgiften.

För att utföra experimenten behöver en spelprototyp byggas som implementerar de två teknikerna som detta arbete fokuserar på. Denna spelprototyp kommer vara av typen top-down 2D shoot-em-up. Spelprototypen kommer innehålla två typer av autonoma spelare. Så kallade botar. De två typerna av botar kommer att ha likvärdiga beteenden. Vilka regler som gäller för spelprototypen och vilka faktiska beteenden som en bot använder beskrivs i kapitel 4. Ena typen av bot kommer att använda beteendeträd för att styra botens olika beteenden. Den andra typen av bot kommer istället att använda Hierarchical Task Network.

För att undersöka botarnas beteende kommer specifika situationstester att göras. Dessa specifika situationstester utgår från en samling fördefinierade situationer. Varje typ av bot ställs inför dessa situationer ett bestämt antal gånger och botens val av beteende noteras. Därefter kan val av beteende i de olika situationerna jämföras med förväntat utfall. Det kan dock vara svårt att testa alla möjliga övergångar mellan beteenden på grund av den stora mängden kombinationer som kan uppstå. Därför kommer endast en begränsad mängd situationer att testas. Valet av situationer beskrivs i kapitel 4.

Tidseffektiviteten kommer utvärderas genom att först mäta tiden som de olika algoritmerna spenderar för att bestämma nästa beteende. Därefter kommer resultaten jämföras. Mätningen kommer att göras med specifika situationstester, enligt den modell som nämndes ovan. Här noteras exekveringstiden som krävs för att välja ett specifikt beteende. Den uppmätta exekveringstiden kan sedan jämföras mellan de två typerna av botar. En styrka med mätning av tidseffektivitet är att det kan ge en indikation på hur väl algoritmerna presterar i praktiken.

Mätning av tidseffektivitet har tidigare utförts av Blum och Furst (1997), för att jämföra tidseffektiviteten hos en samling planeringsalgoritmer, där exekveringstid uppmättes för varje mål planeringsalgoritmerna klarade av att lösa. Denna typ av mätning kan direkt kopplas till metoden i detta arbete. Botarna kommer i varje situationstest att generera ett beteende som skall utföras för att uppnå ett specifikt mål i en problemdomän. Detta beteenden blir en *plan* som bryts ned i en samling uppgifter som skall utföras. Tiden som krävs för att formulera planen är det som mäts och senare kommer att jämföras.

För att säkerställa att en genererad plan är korrekt kommer den valideras. En plan valideras genom att säkerställa att alla uppgifter i planen överensstämmer med förväntat utfall. Blum och Furst (1997) utför planvalidering i sitt arbete för att säkerställa att en genererad plan verkligen är korrekt. Detta steg är nödvändigt för att med säkerhet kunna avgöra att exekveringstiden mäts för rätt plan.

Det kan var svårt att avgöra hur algoritmerna presterar i värsta fall utan att göra speciella testfall. Ett värsta fall skulle kunna vara att algoritmerna behöver undersöka en större mängd beteenden innan ett giltigt beteende hittas som uppfyller ett sökt mål. Värsta fallet i en implementation beror dock mycket på problemdomänen. Vilka faktiska beteenden som finns att välja bland.

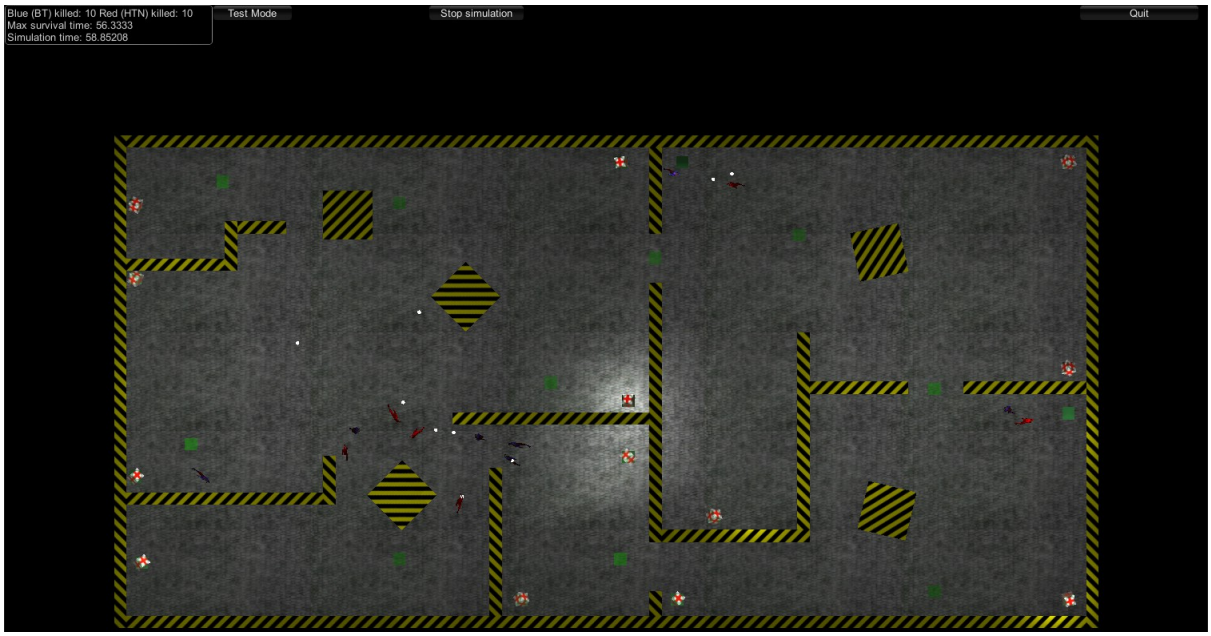
Ett alternativ till implementation av teknikerna hade varit att använda algoritmanalys. Algoritmanalys ger en uppfattning av algoritmernas tidskomplexitet. Tidskomplexiteten är oberoende av hårdvara. Men med endast tidskomplexiteten är det dock svårt att säga hur väl algoritmen fungerar i realtid, eftersom inget exakt mått ges på exekveringstiden. Därför är implementation ett bättre val då datorspel körs i realtid.

4 Implementation

I detta kapitel presenteras en enklare spelprototyp vars syfte är att lösa problemet som presenterades i kapitel 3. Spelprototypen som implementerats har två lägen, ett simuleringsläge och ett testläge. I simuleringsläget spelar två lag med botar mot varandra. I testläget ställs varje typ av bot inför en samling fördefinierade situationer.

4.1 Spelprototyp och beteenden

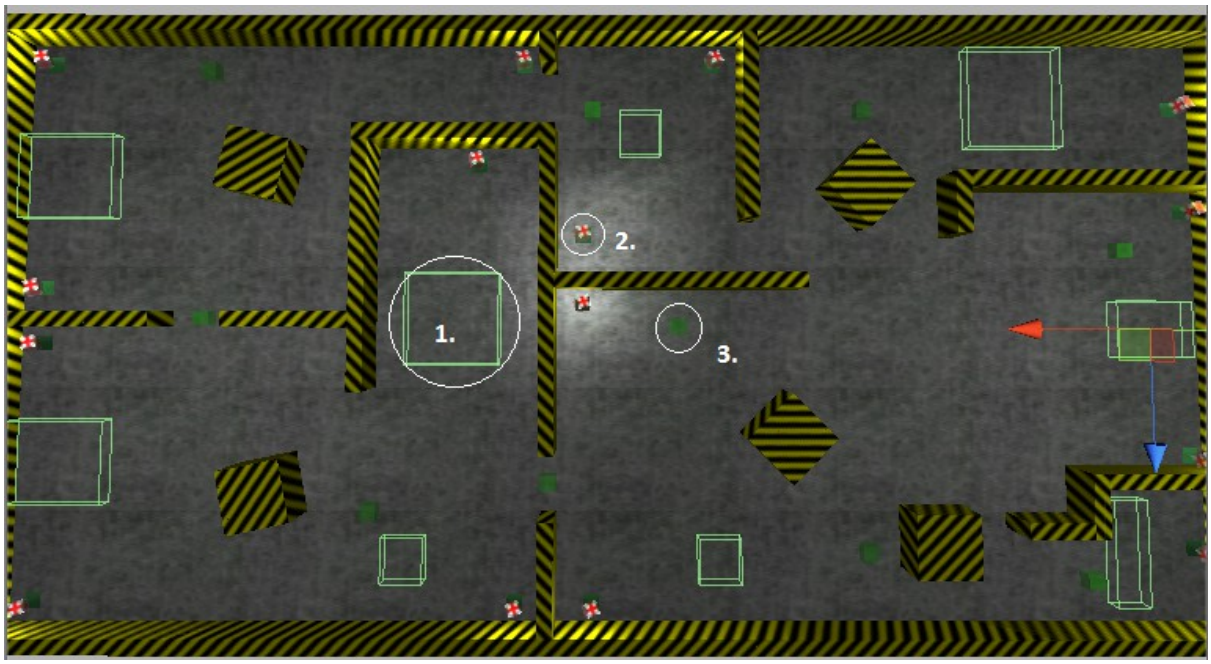
Spelprototypen med de två valda teknikerna som nämndes i kapitel 3 implementerades i spelmotorn Unity (Unity Technologies, 2014). Programmeringsspråket C# användes till implementationen. I spelet finns det två lag med botar, röd och blå. Botarna i det röda laget använder tekniken Hierarchical Task Network för att välja vilka beteenden som skall utföras. I det blå laget använder botarna istället beteendeträd för att välja beteenden. Det går att välja hur många botar det skall vara i varje lag innan ett nytt spel startas. Ett exempel på ett pågående spel, med åtta botar i varje lag, visas i Figur 6 nedan. Spelprototypen håller ordning på hur många botar som dött under spelets gång i respektive lag, samt hur länge spelsimuleringen har körts. I detta exempel har simuleringen körts i ungefär 59 sekunder och 10 botar i varje lag har dött. Det finns knappar för att avbryta simuleringen och för att avbryta programmet. Samt finns en knapp för att byta till testläget. Information om pågående spel visas i övre vänstra hörnet i figuren. När simuleringen avslutas sparas relevant information om spelets status i en textfil, en så kallad loggfil.



Figur 6 Pågående spel i spelprototypen. Det röda laget mot det blå laget.

I Figur 7 nedan är olika typer av intressanta objekt markerade på spelplanen. Dessa är inringade och numrerade ett till tre. Nummer ett är en startzon. Det finns totalt åtta olika startzoner. Dessa är osynliga när spelet körs. Alla spelare som laddas in måste starta i en startzon. Nummer två är en hälsodepå med en aktiv hälsopåfyllare. Om en bot ställer sig på en hälsopåfyllare kommer denna att konsumeras och ge boten ett antal extra hälsopoäng. En hälsopåfyllare försvinner från spelplanen då den blivit konsumerad. Om hälsopåfyllaren i en

hälsodepå blivit konsumerad kommer en ny hälsopåfyllare att laddas in i depån efter en viss tidsfördröjning. Nummer tre är en taktisk punkt. Taktiska punkter används av botarna för att navigera på spelplanen. Till exempel besöks taktiska punkter för att söka efter motståndare.



Figur 7 Olika objekt på spelplanen. Nr 1 är en startzon. Nr 2 är hälsodepå. Nr 3 är en taktisk punkt.

När en bot börjar spelet väljs en slumpmässig startzon, samt en koordinat i startzonen som blir botens startpunkt. Vid början av ett spel har varje bot 100 hälsopoäng och 100 vapenpoäng. En bot har två möjliga sätt att attackera motspelare, r*ange shot* och *burst shot*. Alla attacker sker med hjälp av projektiler och konsumerar vapenpoäng. Om antalet vapenpoäng någonsin blir noll måste boten ladda om sitt vapen för att kunna använda sina attacker igen. En omladdning ger 100 nya vapenpoäng och tar en viss tid att genomföra. Om en bot blir träffad av en motståndares projektil kostar detta hälsopoäng. Hur många hälsopoäng boten förlorar beror på vilken typ av attack som motståndaren använde. Om antalet hälsopoäng blir mindre eller lika med noll betraktas boten som död och måste omedelbart lämna spelet. När en bot dör laddar spelet in en ny bot, som tillhör samma lag som den döda boten, i en slumpmässigt vald startzon.

Varje bot har totalt sex olika beteenden. Dessa beteenden kan delas in tre olika beteendegrupper, *preserve*, *attack* och *scout*. I Tabell 1 nedan finns alla sex beteenden indelade i beteendegrupper. Raderna i tabellen är indelade efter fallande prioritet, uppifrån och ned. Till exempel har *Low Health* högre prioritet än *Dodge bullet*. Vidare är beteendegrupperna rangordnade efter fallande prioritet, vänster till höger. *Preserve* har högst prioritet och syftar till att hålla boten vid liv. *Attack* används för att bekämpa botar i det motsatta laget. Lägst prioritet har *scout* som gör att boten besöker taktiska punkter på spelplanen.

Tabell 1 Beteenden för botar

Preserve	**Attack**	**Scout**
Low health	Burst attack	Scout
Dodge bullet	Range attack	
	Chase target	

Beteendet *low health* kan aktiveras när botens hälsopoäng är mindre än ett bestämt gränsvärde. Då försöker boten hitta en hälsodepå som den sedan besöker. För att beteendet *dodge bullet* skall aktiveras måste en fiendeprojektil vara på kollisionskurs med boten. Samt gäller att beteendet inte varit aktivt de senaste 500 millisekunderna. Om beteendet aktiveras kommer boten försöka röra sig bakåt och åt ena sidan under en kortare tid. Sidan boten rör sig åt bestäms slumpmässigt. Detta beteende ger boten en chans att undvika projektiler.

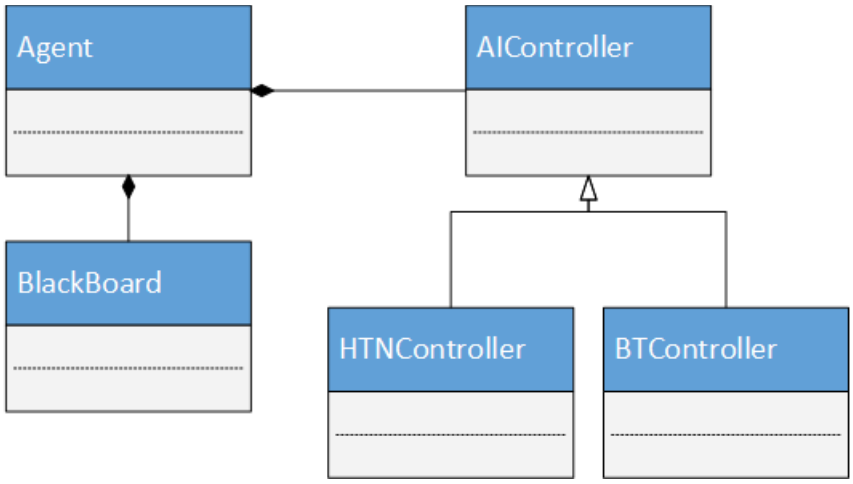
För att beteendet *burst attack* skall kunna aktiveras måste boten ha tillräckligt många vapenpoäng. Vidare gäller att en fiendebot måste befinna sig inom skotthåll. Om dessa villkor är uppfyllda skjuter boten tre projektiler mot fiendeboten. Avståndet från boten till fiendeboten avgör om fiendeboten befinner sig inom skotthåll. Beteendet *range attack* liknar *burst attack* men förbrukar färre vapenpoäng. Vidare har *range attack* en längre räckvidd än *burst attack* men utdelar lägre skada vid en träff. Samt avfyras endast en projektil. Beteendet *chase target* kan aktiveras om den fiendebot som senast var i fokus försvinner ur botens synfält eller om en fiendebot finns i fokus. Om ett väggobjekt befinner sig mellan en bot och en fiendebot räknas detta som att fiendeboten inte syns. Om beteendet *chase target* aktiveras kommer boten besöka fiendebotens position såvida det finns en fiendebot i fokus. Om det inte finns en fiendebot i fokus kommer boten istället att besöka senaste kända positionen för den fiendebot som tidigare var i fokus. Detta beteende gör att en bot kan jaga en fiendebot även om den tappar bort den under kortare perioder.

Det sista beteendet, *scout,* kräver inget förhandstillstånd för att kunna aktiveras. När *scout* aktiveras kommer boten besöka alla taktiska punkter på spelplanen. Målet med detta är att förhoppningsvis stöta på en fiendebot som kan bli nuvarande fokus för boten.

4.2 Botarkitektur

Botarna i spelprototypen har en gemensam grundarkitektur som används för att utföra uppgifter. Dessa uppgifter kan till exempel vara att avfyra en projektil, besöka en taktisk punkt eller avgöra om en fiendebot skall vara i fokus. För att styra vilka uppgifter en bot skall utföra används en AI-kontroller. En AI-kontroller genererar, varje tidssteg, en plan som består av ett antal uppgifter som skall utföras. Därefter utförs dessa uppgifter i en sekvens. För att representera olika tillstånd hos en bot användes en datastruktur som gavs namnet *black board*. Klöckner (2014) använder en liknande datastruktur för att förmedla världens tillstånd till beteendeträdet. Datastrukturen *black board* är en samling variabler som beskriver olika tillstånd. Dessa tillstånd kan till exempel vara att boten har en fiende i fokus eller att den kanske har tappat bort en fiende. Botens *black board* används av AI-kontrollern som beslutsunderlag. I Figur 8 nedan visas ett klassdiagram för botarna. Klassen *Agent* hanterar botens grundfunktioner. Det finns två typer av AI-kontroller, *HTNController* och

BTController. Dessa använder internt Hierarchical Task Network respektive beteendeträd för att planera vilka uppgifter som skall utföras. En ny plan genereras och utförs varje tidssteg. Innan en ny plan genereras uppdateras *black board* med relevant data om botens interna tillstånd.



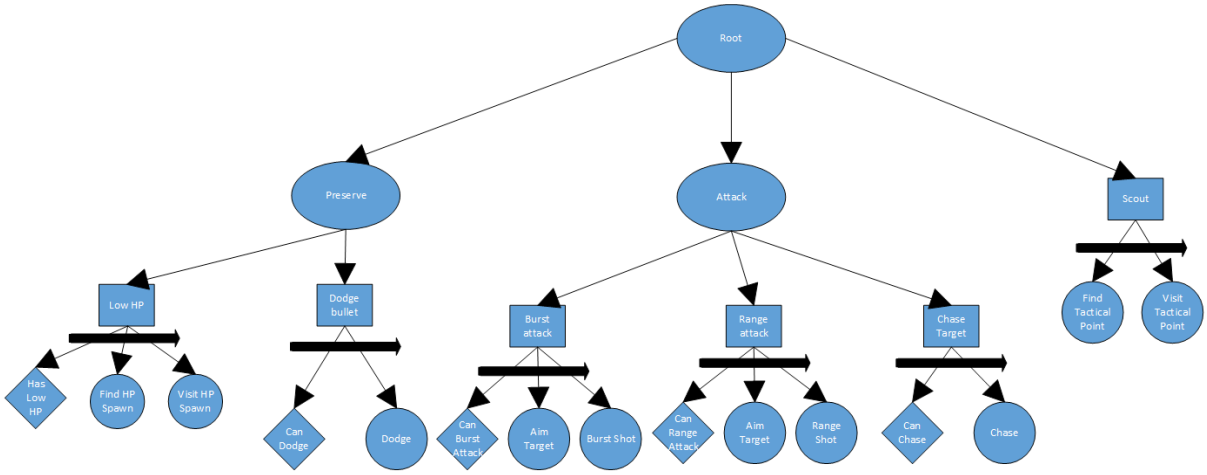
Figur 8 Klassdiagram för botarna

4.3 AI-kontroller

Boten är designad så att valet av AI-kontroller sker när boten skapas. Det finns tre möjliga val. Det första valet är att inte använda någon AI-kontroller. Då kommer boten inte utföra några aktiva uppgifter. Endast automatiska passiva uppgifter kommer att utföras. Till exempel kan en bot utan AI-kontroller fortfarande ta skada och eventuellt dö. Ytterligare val av AI-kontroller är *BTController* och *HTNController*. *BTController och HTNController* har som mål att generera en plan i varje given situation. Denna plan ska innehålla samma uppgifter oavsett om boten använder *BTController* eller *HTNController*. Som utgångspunkt används de beteenden och prioriteringar som beskrevs i kapitel 4.1.

Implementationen av *BTController* använder beteendeträd baserade på arbetet av Champandard (2012). För att förenkla tidmätning har funktionaliteten ändrats en aning. Istället för att utföra uppgifter under evalueringen av trädet sparas istället uppgifterna i en lista. Listan med uppgifter är den nuvarande planen. Uppgifterna utförs efter det att planen har genererats. Denna ändring gör att hela planeringsfasen kan mätas i en enda tagning. Nackdelen är att användningen av listan ger något sämre tidseffektivitet, eftersom algoritmen måste utföra ytterligare steg som normalt inte behöver utföras. Alternativet hade varit att mäta exekveringstiden för varje nod som evalueras, därefter summera tiderna. Detta hade dock blivit en mer komplicerad lösning och gett ett större utrymme för mätfel.

Implementationen tillhandahåller alla de noder som presenterades i bakgrunden i kapitel 2.1. En referens till botens *black board* skickas till varje nod som skall evalueras. Detta används av condition-noder för att avgöra om evalueringen av sekvenser skall fortsätta. I Figur 9 nedan visas det slutgiltiga beteendeträdet som används av klassen *BTController*. Beteendeträdet är byggt för att representera de beteenden som beskrevs i det tidigare kapitlet 4.1. För att avgöra om ett beteende kan aktiveras används botens *black board* i condition-noderna.

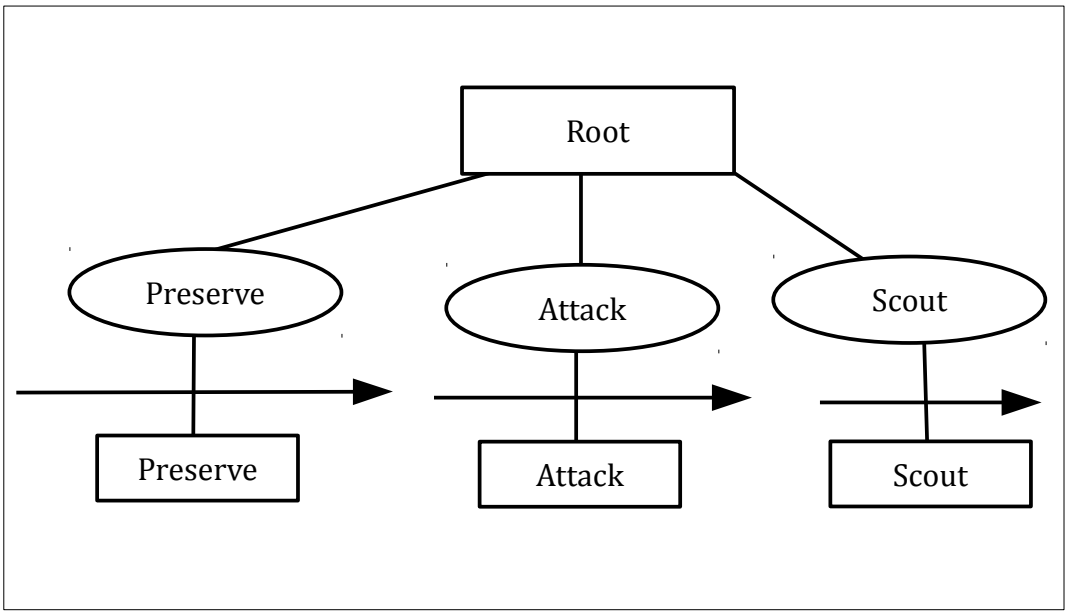


Figur 9 Botarnas fullständiga beteendeträd

Implementationen av *HTNController* använder en Hierarchical Task Network algoritm baserad på programbiblioteket Pyhop (Nau, 2013). Pyhop är en enkel HTN planerare baserad på SHOP algoritmen (Nau, 1999), skriven i programspråket Python. För att representera världens nuvarande tillstånd används botens *black board*. Detta skiljer sig från Pyhop som använder ett så kallat uppslagsverk för att representera världens tillstånd. Ett uppslagsverk översätter någon form av värde till annat värde. Ett uppslagsverk i C# är System.Collections.Generic.Dictionary. En annan väsentligt skillnad är användningen av generiskprogrammering för datatypen som identifierar metoder och operatorer. Detta skiljer sig från Pyhop implementationen som använder textsträngar för att identifiera metoder och operatorer. Klassen *HTNController* använder en enumdatatyp för att identifiera metoder och operatorer. Fördelen med en enumdatatyp är att dessa representeras som numeriska värden. Jämförelseoperationer med en enumdatatyp är normalt snabbare än motsvarande operationer med textsträngar.

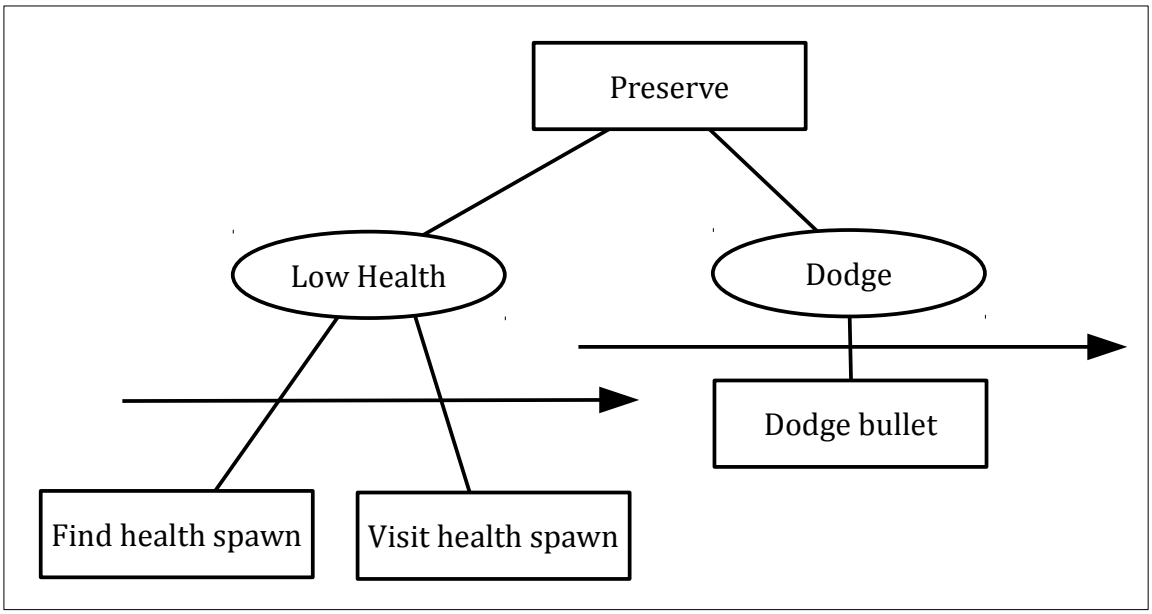
Metoderna är sparade i ett uppslagsverk över listor med metoder. Detta innebär att algoritmen kommer iterera igenom listan med metoder som kan lösa en specifik uppgift. Således kommer metoder väljas i den ordning de lades till.

HTNController implementerar de beteenden som beskrevs i det tidigare kapitlet 4.1. De operatorer som används förändrar inte världens tillstånd. Detta beror på att denna effekt inte är nödvändig för att avgöra vilket beteende som skall väljas. I Figur 10 nedan visas metoderna som kan lösa beteendet *root*. När *HTNController* genererar en plan kommer den alltid att försöka hitta en lösning på beteendet *root*. För att kunna generera en plan försöker algoritmen först att generera en plan med metoden *preserve*. Om detta misslyckas genereras istället en plan med metoden *attack*. Skulle även detta misslyckas genereras en plan med metoden *scout*. Metoden *scout* är konstruerad så att den alltid kan lyckas.



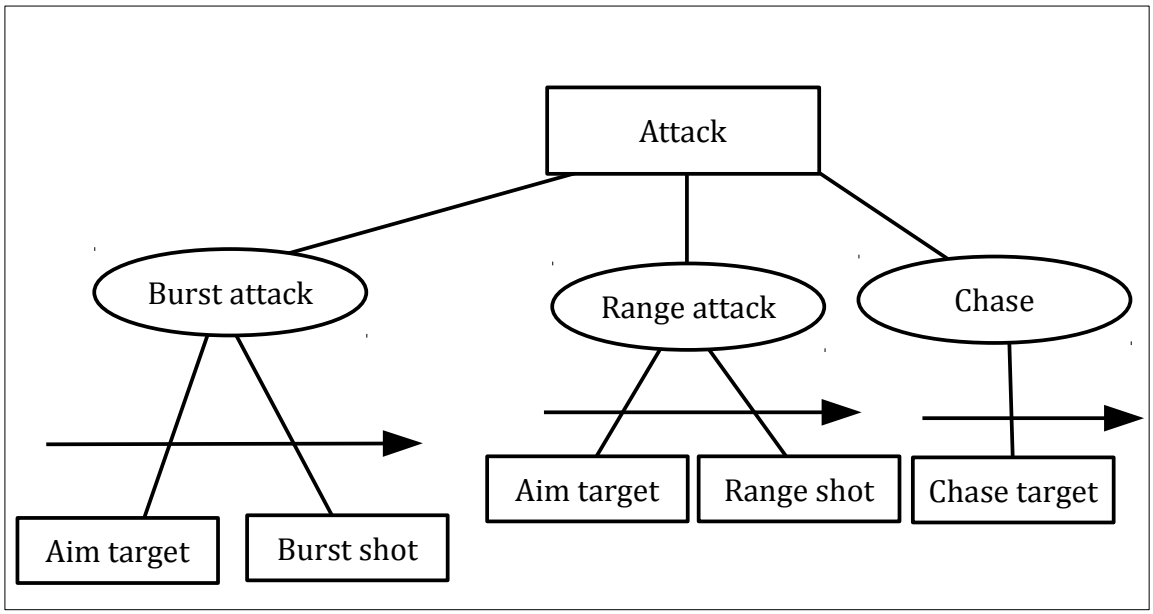
Figur 10 Root beteendet

I Figur 11 nedan visas beteendet *preserve* och metoderna som kan hantera detta beteende. Metoden *dodge* har endast en operator. Dock kräver definitionen av SHOP algoritmen att en metod används för att kunna applicera operatorer.



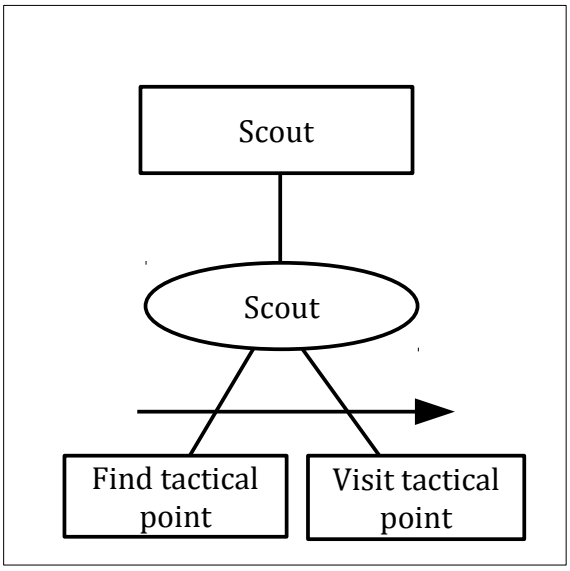
Figur 11 Preserve beteendet

I Figur 12 nedan visas beteendet *attack* och de metoder som kan användas.



Figur 12 Attack beteendet

I Figur 13 nedan visas beteendet med lägst prioritet, *scout*.



Figur 13 Scout beteendet

4.4 Pool-allokator

Ett problem identifierades under implementationen av Hierarchical Task Network. Problemet var att implementationen använde många minnesallokeringar. Detta förvärrar tidseffektiviteten något. Problemet åtgärdades genom att förallokera en samling listor som algoritmen sedan använder. Den ursprungliga implementationen allokerade dessa under körning av algoritmen. Förändringen gav bättre tidseffektivitet.

4.5 Testläge

Testläget i spelprototypen aktiveras genom att trycka på knappen *Test Mode*. I testläget kan varje typ av bot automatiskt testas i sex fördefinierade situationer. Efter att användaren

trycker på knappen *Start tests* kommer de automatiska testen att utföras ett åt gången. Information om utförandet av individuella tester kommer att skrivas till en loggfil som skapades när testläget aktiverades.

Situationstesterna är byggda så att de skall aktivera ett speciellt beteende hos boten. Varje enskilt test använder samma basklass. Basklassen hanterar de generella delar som varje test använder. Först körs specifik kod för varje test som bygger upp en viss situation. En sådan situation kan till exempel vara projektiler utplacerade på bestämda platser på spelplanen. Sedan kommer varje typ av bot att testas i situationen. Först laddas boten in på en bestämd plats på spelplanen. Därefter utförs testet. Sedan tas boten bort från spelplanen. Samma procedur utförs med nästa bot. Detta upprepas tills alla tester har utförts.

Innan ett enskilt test kan utföras görs en planvalidering. Detta innebär att botens AI-kontroller genererar lika många planer som det finns uppgifter i det förväntade utfallet. Dessa planer kommer att testas mot det förväntade utfallet där varje uppgift en och endast en gång kommer att bytas ut mot en felaktig uppgift. Detta valideringstest försöker upptäcka fel i valideringen av det förväntade utfallet. I Tabell 2 nedan visas ett exempel på ett förväntat utfall i en situation, samt de två felaktiga planer som kommer att användas i valideringstestet. Varje kolumn i tabellen representerar en plan.

Tabell 2 Planvalidering

Förväntad plan	**Felaktig plan 1**	**Felaktig plan 2**
FINDTACTICALPOINT	NONE	FINDTACTICALPOINT
VISITTACTICALPOINT	VISITTACTICALPOINT	NONE

Om en felaktig plan skulle valideras som korrekt kommer detta generera ett felmeddelande i loggfilen. Efter valideringssteget kommer mätsteget. I mätsteget genereras ett stort antal planer samtidigt som tiden för att generera varje plan uppmäts. Därefter beräknas medelvärdet för att generera en plan. Alla mätpunkter sparas också i en separat loggfil. Tidmätningen sker med hjälp av den inbyggda C# klassen StopWatch. Implementationen av StopWatch använder en högprecisionstimer. Varje plan som genereras under tidmätningssteget valideras efter tidmätningen för att säkerställa att det är en korrekt plan. Om den genererade planen inte är korrekt kommer detta resultera i ett felmeddelande i loggfilen. Om ingenting gick fel under testet kommer tidmedelvärdet att sparas i loggfilen.

Programspråket C# har en automatisk skräpsamlare, en så kallad *garbage collector*. Syftet med en *garbage collector* är att förenkla minneshanteringen i ett program genom att minne inte behöver frigöras manuellt. Istället kommer *garbage collector* att frigöra minne automatiskt när objekt inte längre används. Dock innebär detta avsaknad av kontroll över tidpunkten när detta minne frigörs. Detta kan ställa till problem med tidmätning eftersom det kan hända att *garbage collector* körs när tidmätningen sker. För att försöka minimera detta problem körs den inbyggda funktionen GC.Collect() vid var femte mätning. Totalt utförs 1000 mätningar. Funktion GC.Collect() tvingar *garbage collector* att starta och minskar risken att den startar under själva tidmätningen. Vidare förändras prioriteten för den nuvarande exekverande tråden från normal till hög prioritet. På samma sätt ökas prioriteten för processen som den exekverande tråden tillhör. Detta gör att operativsystemet

ger den nuvarande exekverande tråden mer körtid. Detta minskar risken för att andra processer körs under själva tidmätningen.

4.6 Pilotstudie

En pilotstudie genomfördes för att avgöra att spelprototypen fungerar och att mätdata kan användas för att svara på problemformuleringen i kapitel 3. Först testkördes spelprototypen i simuleringsläget med 8 botar i varje lag. Efter 80 sekunder avbröts simuleringen. Kördata från loggfilen visas i Tabell 3 nedan. Detta resultat visar att spelprototypens simuleringsläge fungerar och i denna körning har lagen dessutom ett mycket jämt resultat. Om resultatet varit mycket ojämnt hade detta kunna tyda på att botarna har skillnader i beteende. Dock kan resultatet till viss del bero på slump. Huvudsyftet med simuleringsläget är att avgöra om botarna fungerar korrekt.

Tabell 3 Loggdata från körning i simuleringsläge

-------STATS-------
Blue(BT) killed: 11
Red(HTN) killed: 11
Top survival time: 80.31321s
Simulation time: 80.34666s

Vidare kördes också spelprototypen i testläget. I Tabell 4 nedan visas loggdata från körningen i testläget. Testen verkar ha utförts korrekt utan några rapporterade fel. Den rapporterade tidsåtgången verkar ligga i linje med det förväntade utfallet. Det vill säga att Hierarchical Task Network är långsammare än beteendeträd. Denna typ av data bör kunna användas för att avgöra hur stora skillnader det är i planeringstid för algoritmerna i olika situationer.

Tabell 4 Loggdata från körning i testläge

---------- START TESTS ----------
Test 1 (Scout behavior) STARTED
Desired plan: FINDTACTICALPOINT VISITTACTICALPOINT
Behavior tree average time 0.0012284 milliseconds.
Hierarchical task network average time 0.00658279999999998 milliseconds.

Test 2 (Dodge behavior) STARTED
Desired plan: DODGEBULLET
Behavior tree average time 0.000542099999999996 milliseconds.
Hierarchical task network average time 0.00200400000000001 milliseconds.

Test 3 (Low HP behavior) STARTED
Desired plan: FINDHEALTHSPAWN VISITHEALTHSPAWN
Behavior tree average time 0.000605699999999993 milliseconds.
Hierarchical task network average time 0.00408149999999999 milliseconds.

Test 4 (Burst attack behavior) STARTED
Desired plan: AIMTARGET BURSTSHOT
Behavior tree average time 0.00117739999999999 milliseconds.
Hierarchical task network average time 0.00430619999999999 milliseconds.

Test 5 (Range attack behavior) STARTED
Desired plan: AIMTARGET RANGEDSHOT
Behavior tree average time 0.00151880000000001 milliseconds.
Hierarchical task network average time 0.00452929999999998 milliseconds.

Test 6 (Chase target behavior) STARTED
Desired plan: CHASETARGET
Behavior tree average time 0.0010367 milliseconds.
Hierarchical task network average time 0.00276739999999998 milliseconds.

5 Utvärdering

I detta kapitel utvärderas tidseffektiviteten hos algoritmerna detta arbete fokuserar på. Detta kapitel består av fyra delar. Den första delen presenterar mätmiljön som använts vid alla mätningar. Den andra delen presenterar resultaten av de mätningar som utförts. Den tredje delen innehåller en analys av resultaten från den andra delen. Den fjärde och sista delen innehåller slutsatser.

5.1 Mätmiljö

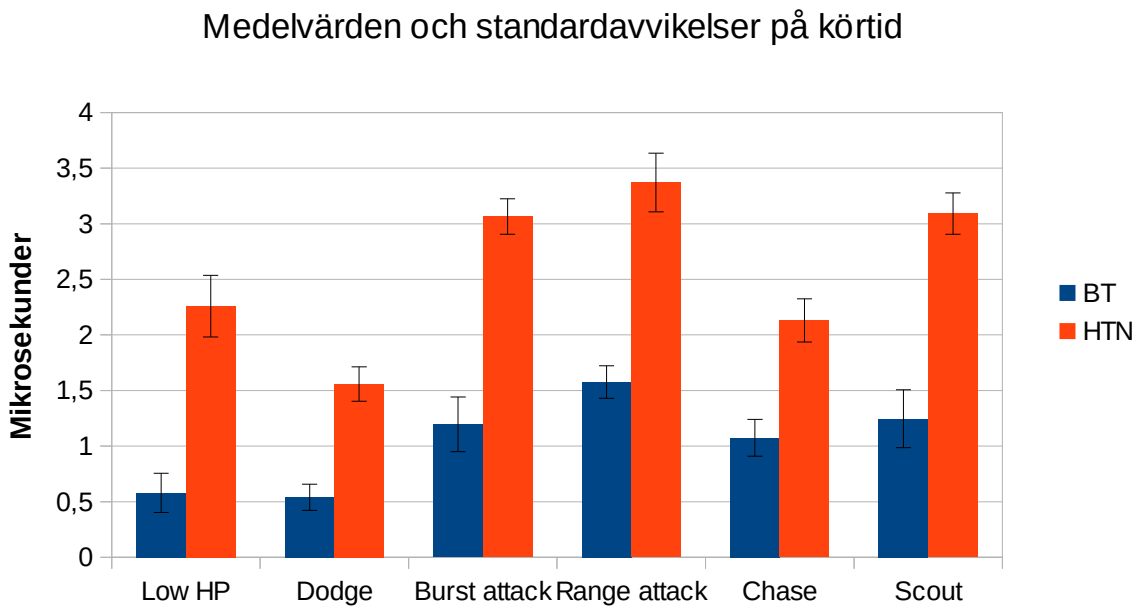
Alla mätningar har utförts i en och samma mätmiljö. Karakteristiken för mätmiljön finns i Tabell 5 nedan. Mätningarna har utförts direkt efter att spelprototypen startats.

Tabell 5 Mätmiljö

Spelmotor	Unity 4.3.4f1
Operativsystem	Windows 7 Home Premium
Processor	Intel Core i7-2630QM 2,00 GHz
Ram	8 GB DDR3
Grafikkort	Nvidia GTX 560M

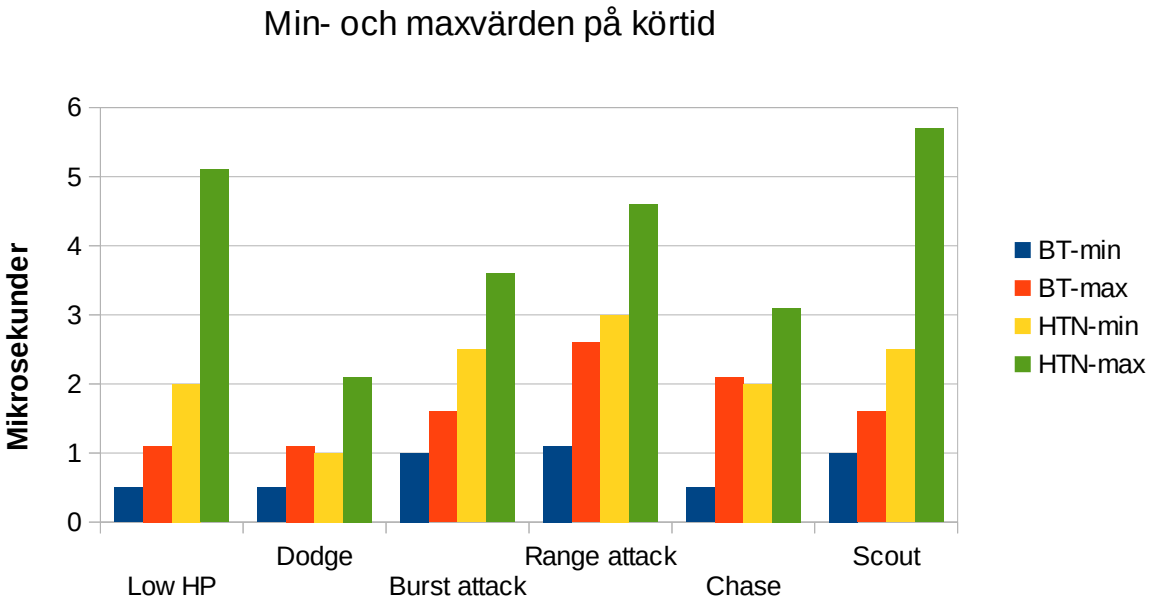
5.2 Resultat av körningar

Tidseffektiviteten uppmättes genom att köra spelprototypen på samma sätt som i pilottestet i kapitel 4. Spelprototypen använder två typer av autonoma botar. En bot för vardera teknik som detta arbete fokuserar på. Spelprototypen utsätter varje bot för sex olika situationer. Botarna utsätts för varje enskild situation 1000 gånger. Varje situation är konstruerad för att resultera i ett bestämt beteende hos boten. Ett beteende kan brytas ned till en plan som är en samling uppgifter som boten kan utföra. Vid varje körning uppmäts körtiden som krävs för bestämma ett beteende. I Figur 14 nedan visas ett stapeldiagram med medelvärden och standardavvikelser på körtider för varje beteendet. De blå staplarna representerar körtider uppmätta för tekniken beteendeträd. De röda staplarna representerar körtider uppmätta för tekniken Hierarchical Task Network. Alla beteenden i diagrammet är ordnade från vänster till höger i den ordning de kan väljas. De beräknade standardavvikelserna visar att mätningarna har relativ liten variation kring respektive medelvärde. Den högsta variationen är ungefär 0.28 mikrosekunder och den lägsta ungefär 0.12 mikrosekunder.



Figur 14 Medelvärden på körtid med algoritmerna beteendeträd och Hierarchical Task Network, i sex olika situationer. Varje algoritm kördes 1000 gånger i varje enskild situation. Standardavvikelsen för varje medelvärde är markerad i diagrammet.

I Figur 15 nedan visas min- och maxvärden på alla körningar. Ett minvärde är när algoritmen presterar som bäst i mätserien. Ett maxvärde är när algoritmen presterar som sämst i mätserien. Det framgår av diagrammet att minvärdet för beteendeträd alltid är lägre än minvärdet för Hierarchical Task Network inom samma beteendekategori.



Figur 15 Min- och maxvärden på körtid med teknikerna beteendeträd och Hierarchical Task Network, i sex olika situationer.

5.3 Analys

Diagrammet i Figur 14 visar att standardavvikelsen är relativt liten för varje medelvärde. Detta innebär att enskilda mätvärden har liten variation omkring respektive medelvärde. Vidare överlappar aldrig standardavvikelsen för vare sig beteendeträd eller hierachical task network inom samma beteendekategori. Det tyder på att mätvärdena inte varierar speciellt mycket inom samma område. Det framgår tydligt från diagrammet att beteendeträd presterar bättre än hierachical task network i alla beteendekategorier. Alla beteenden förutom *dodge* och *chase,* består av två uppgifter. Istället består *dodge* och *chase* av endast en uppgift vardera. Diagrammet visar att beteendet d*odge* har lägre körtid än beteendet l*ow hp*. Detsamma gäller för beteendet *chase* som har lägre körtid än beteendet *range attack*.

Skillnaden i körtid för varje beteende varierar. Den största skillnaden har beteendet *low hp*. Här har Hierarchical Task Network 3,9 gånger längre körtid än beteendeträd. Den minsta skillnaden finns i beteendet *chase*. Här har Hierarchical Task Network ungefär 2 gånger längre körtid än beteendeträd.

Genom att jämföra minvärden i Figur 15, var beteendekategori för sig, framgår det att det finns ett avstånd mellan beteendeträd och Hierarchical Task Network. Beteendeträd har lägre minvärden än Hierarchical Task Network, inom samma beteendekategori, i alla sex beteendekategorier. Detta stärker resultatet ytterligare. Vad som orsakar storleken på maxvärden är svårt att avgöra. Det skulle kunna bero på faktorer som till exempel skräpsamlaren eller cachemissar hos processorn. Värt att notera är att dessa extremvärden inte är frekventa eftersom standardavvikelsen är relativt liten i alla mätningar.

5.4 Slutsatser

Mätningarna tyder på att beteendeträd har bättre tidseffektivitet än Hierarchical Task Network. Analysen visar att Hierarchical Task Network har ungefär 2 till 3,9 gånger längre körtidsmedelvärde, i alla undersökta situationer. Hypotesen var att beteendeträd har bättre tidseffektivitet i situationer med flera val. Detta stämmer eftersom beteendeträd har bättre tidseffektivitet i alla undersökta situationer. Dessutom är skillnaden i tidseffektivitet mellan algoritmerna påtaglig. Vidare är antalet extremvärden i mätdatan relativt få eftersom standardavvikelsen är liten. Detta stärker resultatets trovärdighet.

6 Avslutande diskussion

Detta kapitel sammanfattar arbetet från frågeställning till slutresultat. Vidare förs en diskussion kring arbete som sedan avslutas med tankar kring framtida fortsatt arbete.

6.1 Sammanfattning

Arbetet har utgått från en frågeställning om vilka skillnader det finns i tidseffektivitet mellan beteendeträd och Hierarchical Task Network. Hypotesen var att beteendeträd har bättre tidseffektivitet då det finns flera val. För att kunna undersöka skillnader i tidseffektivitet skapades en enklare spelprototyp. Spelprototypen använder två typer av autonoma motspelare, så kallade botar. Den ena bottypen använder beteendeträd för att bestämma beteenden. Den andra bottypen använder istället Hierarchical Task Network, i detta fallet en variant av algoritmen SHOP (Nau, et al. 1999), för att bestämma beteenden. Ett beteende kan representeras med hjälp av en plan. En plan är en samling uppgifter som boten kan utföra.

Spelprototypen innehåller två lägen. Ett simuleringsläge där botarna kan tävla mot varandra. Syftet med detta läge är att upptäcka potentiella avvikelser i botarnas beteende. Det andra läget är ett testläge. I testläget utsätts varje typ av bot för sex olika situationer. I varje situation måste varje enskild bot, 1000 gånger, generera en plan. Varje genererad plan valideras mot ett förväntat utfall. Om det faktiska och det förväntade utfallet inte stämmer överens, kommer ett felmeddelande genereras. Vidare mäter spelprototypen körtiden som krävs för att generera varje enskild plan i testläget.

Mätningar som utförts av spelprototypen ligger till grund för utvärderingen som presenteras i kapitel 5. Resultatet visar att beteendeträd har bättre tidseffektivitet i alla situationer som undersökts. I det bästa uppmätta fallet har Hierarchical Task Network 2 gånger längre körtid än beteendeträd. I det värsta uppmätta fallet har Hierarchical Task Network ungefär 4 gånger längre körtid.

6.2 Diskussion

Artificiell intelligens har många användningsområden. I spelvärlden används ofta någon form av artificiell intelligens för att bestämma beteenden hos artificiella motspelare. I takt med att spel blir allt mer komplexa ställer detta stora krav på artificiell intelligens. De tekniker som används bör vara robusta och kunna hantera de krav som ställs på dem. Vidare bör dessa tekniker ha tillräckligt god tidseffektivitet för att hantera alla sina uppgifter. Ett allt större område för spelindustrin är olika former av handburna spel. Detta kan till exempel vara olika former av mobilspel. Ett spel till en handburen plattform har helt andra förutsättningar än ett spel till en stationär dator. Mängden processkraft är betydligt lägre. Men det finns fler aspekter, som till exempel batteritid. Tidseffektiviteten kan till exempel kopplas till batteritiden för en mobil, eftersom batteritiden bland annat beror på användningen av processorn. Vidare kan tidseffektiviteten vara kopplad till hur responsivt ett program kan vara. Om tidseffektiviteten är dålig kanske programmet har långa svarstider. En spelare kanske förväntar sig att en artificiell motspelare ska kunna fatta snabba beslut.

Det kan argumenteras för att arbetets metod är relevant ur flera aspekter. För det första undersöker den algoritmernas tidseffektivitet i en spelrelaterad domän, i flera olika situationer. I varje situation genererar algoritmerna planer som valideras mot ett förväntat utfall. Detta för att säkerställa att algoritmerna verkligen löser rätt planeringsproblem. Vidare uppmäts planeringstiden i varje situation. Blum och Furst (1997) använder en liknande metod där de jämför tidseffektiviteten för en samling planeringsalgoritmer. Samt utförs också planvalidering för att säkerställa att de planer som genereras verkligen löser planeringsproblemen i deras problemdomäner. En skillnad är dock att de problemdomäner Blum och Furst (1997) använder resulterar i betydligt längre uppmätta körtider. I deras fall ger detta en tydlig skillnad i körtid mellan algoritmerna.

Eftersom studien i detta arbete endast är utförd på relativt få beteenden med ganska få uppgifter är det svårt att säga om resultatet skulle bli markant annorlunda om mängden beteenden och uppgifter ökas. Därför går det inte med säkerhet säga att resultatet skulle bli det samma för alla typer av problem.

Det kan dock argumenteras för att ett resultat med korta körtider kanske är mer representativt för algoritmer som skall användas i situationer som kräver extremt korta körtider. Planeringsalgoritmer med bra tidseffektivitet kanske vore intressant för olika former av automatiska styrsystem, till exempel i bilar eller flygplan. Ett system med kort svarstid är antagligen att föredra om det till exempel används till att varna för eller undvika farliga situationer. Ytterligare ett exempel skulle kunna vara obemannade fordon på andra planeter. Ett sådant fordon kanske har ett automatiskt styrsystem med kort svarstid, som snabbt kan undvika eller hantera faror som uppkommer.

Det skulle dock vara intressant att undersöka hur algoritmerna skiljer sig åt i planeringstid i problemdomäner där planeringstiderna är mycket längre. En sådan problemdomän skulle kunna vara olika former av simuleringar som kanske inte kräver realtidsplanering. Till exempel ett system som simulerar trafikrörelser för att sedan kunna planera optimala rutter för exempelvis räddningstjänst eller godstrafik. En så kort körtid som möjligt är kanske att föredra även i detta fallet. Men körtiden kanske inte behöver vara så kort som i ett styrsystem som måste kunna utföra och planera uppgifter i realtid.

Resultatets trovärdighet skulle kunna ökas genom att genomföra studien på andra plattformar och operativsystem. Till exempel mobiltelefoner. Ett problem med användningen av C# är den automatiska minneshanteringen. Det är helt enkelt svårt att styra när den automatiska skräpsamlaren körs. Detta kan påverka körtiden negativt. Men det är ett intressant problem eftersom många spel utvecklas för system som använder C#. Spelmotorn Unity (Unity Technologies, 2014) är ett sådant system. Detta gör skräpsamlaren till en realitet som måste kunna hanteras.

Arbetets implementation av beteendeträd är baserad på definitionen gjord av Champandard (2007) som är en av de vanligaste. Det är dock möjligt att det kan finnas avvikelser eller brister i implementationen som gjordes för det här arbetet, som försämrar tidseffektiviteten. Implementationen av Hierarchical Task Network är baserad på programbiblioteket Pyhop (Nau, 2013). Pyhop är utvecklat i programspråket Python. Det är möjligt implementationen som gjordes för det här arbete, i programspråket C#, innehåller avvikelser eller brister som försämrar tidseffektiviteten. Det är också möjligt att det kan finnas optimeringar för implementationerna av beteendeträd och Hierarchical Task Network som skulle kunna

förbättra tidseffektiviteten. En sådan optimering gjordes till exempel med pool-allokatorn som beskrivs i kapitel 4.4.

Ett intressant användningsområde för artificiell intelligens är obemannade flygfarkoster. Både Ögren (2012) och Klöckner (2014) undersöker användningen av beteendeträd för styrning av just obemannade flygfarkoster. Obemannade flygfarkoster skulle kunna ha både civil och militär användning. Detta arbete kommer antagligen inte vara någon avgörande faktor till beslut om användning av någon specifik teknik. Dock skulle det kunna vara en etisk fråga om resultatet användes för militära ändamål. Det motsatta kan också vara tänkvärt. Användning i civila applikationer, till exempel obemannade flygfarkoster som upptäcker gräsbränder eller oljespill i havet. Kortare körtid skulle möjligtvis kunna kopplas till lägre strömförbrukning och eventuellt frigöra processorkraft för andra ändamål.

Något som detta arbete inte undersöker är flexibiliteten hos implementationerna av beteendeträd och Hierarchical Task Network. Flexibilitet skulle kunna vara till exempel underhållbarhet eller skalbarhet. Det är mycket möjligt att någon algoritm är bättre än den andra i vissa problemdomäner med avseende på flexibilitet.

6.3 Framtida arbete

Det finns flera aspekter som skulle kunna undersökas för bägge tekniker. Till exempel fler beteenden och djupare beteendenivåer med fler uppgifter. Detta skulle kunna ge indikationer på om tidseffektiviteten förvärras snabbare för någon av teknikerna. En annan aspekt som skulle kunna undersökas är hur tidseffektiviteten förändras vid reaktiv planering. Med en reaktiv planering skulle beteendeträd kunna avbryta en plan innan den slutförts. Hierarchical Task Network skulle fortfarande behöva generera hela planen även om den avbryts. Detta skulle möjligtvis förändra tidseffektiviteten. Framtida arbeten skulle även kunna undersöka minnesförbrukningen hos algoritmerna. Detta skulle möjligen kunna vara intressant för till exempel inbyggda system med lite primärminne.

En annan aspekt som detta arbete inte undersöker är skalbarhet och underhållbarhet. Hur förändras skalbarhet och underhållbarhet när mängden beteenden ökar för respektive teknik. Denna aspekt vore intressant att undersöka eftersom moderna spel kan innehålla en stor mängd beteenden.

På lång sikt vore det intressant att undersöka hur väl algoritmerna fungerar i andra problemdomäner än datorspel. Till exempel styrsystem för olika former av robotar eller drönare.

Referenser

Blum, A. L. & Furst, M. L. (1997) Fast planning through planning graph analysis. *Artificial intelligence*, 90(1), s. 281-300.

Bungie Software (2004) *Halo 2* [Datorprogram]. Bungie Software.

Champandard, A. (2007) Behavior trees for next-gen game AI. *Proceedings of Game Developers Conference,* Lyon, Frankrike 2007.

Champandard, A. (2012) Understanding the Second-Generation of Behavior Trees. Tillgänglig på internet: http://aigamedev.com/insider/tutorial/second-generation-bt. [Hämtad: 14.04.06]

Champandard, A., Verweij, T. & Straatman, R. (2009) The AI for Killzone 2's multiplayer bots. *Proceedings of Game Developers Conference,* Paris, Frankrike 2007.

Doris, K. & Silvia, D. (2007) Improved Missile Route Planning and Targeting using Game-Based Computational Intelligence. *Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Security and Defense Applications (CISDA 2007),* Honolulu, Hawaii, April 1-5, s. 63-68.

Firby, R. J. (1987) An investigation into reactive planning in complex domains. *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, USA, s. 202-206.

Guerrilla Games (2009) *Killzone 2* [Datorprogram]. Guerrilla Games.

Hart, P., Nilsson, N. & Raphael, B. (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Science and Cybernetics, SSC-4(2): s. 100-107.*

Hart, P., Nilsson, N. & Raphael, B. (1972) Correction to 'A Formal Basis for the Heuristic Determination of Minimum-Cost Paths'. *SIGART Newsletter*. 37, s 28-29.

Isla, D. (2005) Handling Complexity in the Halo 2 AI. *Proceedings of the Game Developer Conference,* San Francisco, California, USA 2005.

Johansson, A. & Dell'Acqua, P. (2012) Emotional Behavior Trees. *IEEE Conference on Computational Intelligence and Games (CIG),* s. 355-362.

Klöckner, A. (2014) The Modelica BehaviorTrees Library: Mission Planning in Continuous-Time for Unmanned Aircraft. *Proceedings of the 10th International Modelica Conference,* Lund, Sverige 2014.

Lim, C., Baumgarten, R. & Colton, S. (2010) Evolving Behaviour Trees for the Commercial Game DEFCON. *Applications of Evolutionary Computation,* 6024, s. 100-110.

Monolith Productions (2006) *F.E.A.R.* [Datorprogram]. Monolith Productions.

Nau, D., Cao, Y., Lotem, A. & Muñoz-Avila, H. (1999) SHOP: Simple Hierarchical Ordered Planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* s. 968-973.

Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J., Wu, D. & Yaman, F. (2003) SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20, s. 379-404.

Nau, D. (2013) *Pyhop* (Version: 1.2.2) [Programbibliotek]. Tillgänglig på internet: https://bitbucket.org/dananau/pyhop. [Hämtad 14.04.07]

Fikes, R. & Nilsson, N. (1971) STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence,* 2 (3/4) : s. 189-208.

Orkin, J. (2003) Applying Goal-Oriented Action Planning to Games. I: S. Rabin (red.). *AI Game Programming Wisdom 2* (s. 217-228). Charles River Media.

Orkin, J. (2006) Three States and a Plan: The A.I. of F.E.A.R. *Proceedings of the Game Developer Conference,* San Jose, California, USA 2006.

Perez, D., Nicolau, M., O'Neill, M. & Brabazon, A. (2011) Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution. *Applications of Evolutionary Computation,* 6624, s. 123-132.

Unity Technologies (2014) *Unity* (Version: 4.3.4f1) [Datorprogram]. Unity Technologies. Tillgänglig på internet: http://unity3d.com.

Ögren, P. (2012) Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. *AIAA conference on Guidance, Navigation and Control,* Minneapolis, USA 2012.