Examensarbete

HÖGSKOLAN I SKÖVDE

GOAL-ORIENTED ACTION PLANNING

Utvärdering av A* och IDA*

Examensarbete inom huvudområdet Datalogi/Datavetenskap/Medier, estetik och berättande
Grundnivå 30 högskolepoäng
Vårtermin 2012

Fred Helmesjö

Handledare: Mikael Thieme
Examinator: Mikael Johannesson

Sammanfattning

Goal-Oriented Action Planning (GOAP) är en AI-arkitektur som tillämpar ett måldrivet beteende åt agenter i spel. Mål uppnås genom att planer med åtgärder genereras med hjälp av en sökalgoritm. Syftet med denna rapport är att undersöka hur två sökalgoritmer, A* och IDA*, presterar under planering i GOAP.

De experimenten som används är dels en miljö där agenter simuleras, samt ett test där planer genereras för samtliga implementerade mål utan rendering och simulering av agenter. Data som utvärderas är bl.a. planeringstiden, antal besökta noder under sökning och genererade planer.

Utvärderingen visar en tydlig fördel till A*, som i snitt är 38 % snabbare än IDA* vid planering av åtgärder i GOAP. Slutsatsen blir att A* är den algoritm att föredra om prestanda är det som eftertraktas men IDA* kan motiveras för dess egenskaper, så som lägre minneskomplexitet.

Nyckelord: GOAP, måldrivet beteende, planering av åtgärder, IDA*, A*

Innehållsförteckning

1 Introduktion

Artificiell intelligens har i många spelgenrer en avgörande roll för hur helheten av spelet uppfattas av spelare, och är därför ett viktigt forskningsområde. Det finns många traditionella tekniker för att simulera AI, som till viss del också har visat sig fungera väldigt bra för sina ändamål, men som mycket forskning i denna rapport visar (bl.a. Orkin, 2003) börjar dessa för många speltyper nå slutet på vad som är praktiskt hållbart vid utveckling och gör på grund av detta det också svårt att ge en tillräckligt realistisk upplevelse för spelare. Detta arbete fokuserar på en nyare teknik för simulering av AI som utnyttjar planering, nämligen GOAP (Orkin, 2003, 2004 & 2006; Millington & Funge, 2009), och tittar framför allt på två olika sätt att implementera just planeraren (sökalgoritmen). Bakgrunden (kapitel 2) koncentrerar på att ge läsaren bra förståelse av denna teknik inför det kommande, praktiska arbetet (kapitel 4) genom att övergripande men pedagogiskt gå igenom berörda områden. Upplägget är gjort på ett sätt som enkelt ska få läsaren att förstå motiveringarna bakom varje ny teknik som GOAP bygger på.

Hypotesen i detta arbete bygger på Millingtons m.fl. (2009) påstående om att sökalgoritmen IDA*, på grund av dess lägre minneskomplexitet samt inget krav på sorterad uthämtning av noder från lista, är bättre lämpad för GOAP (och t.o.m. kan ge bättre prestanda) än den mer vanligen använda sökalgoritmen A*.

För att testa detta har GOAP implementeras med de två olika sökalgoritmerna. Ett experiment i form av en testmiljö har skapats där agenter simuleras med respektive sökalgoritm för att sedan utvärdera dessa mot varandra. Utvärderingen har skett i form av olika prestandatest och analys av genererad data för att även upptäcka eventuella skillnader i planeringen. Detta arbete består alltså av två delsteg: Implementation (kapitel 4) och utvärdering av data (kapitel 5).

2 Bakgrund

I detta kapitel tas relevant bakgrundsinformation upp. Först ges en övergripande beskrivning av *Finit State Machine* (FSM, sv. Ändlig tillståndsmaskin) följt av lite mer detaljerad beskrivning av *Goal-Oriented Behavior* (GOB) och *Goal-Oriented Action Planning* (GOAP) med fokus på likheter och skillnader gentemot FSM. Därefter ges övergripande information om hur sökalgoritmen A* (A-star) fungerar i GOAP och hur detta möjliggör dynamisk problemlösning. För att övergå till det faktiska syftet med denna rapport ges också en kort inblick i Iterative Deepening A* (IDA*) och hur denna skiljer sig från A*, med för- och nackdelar diskuterade.

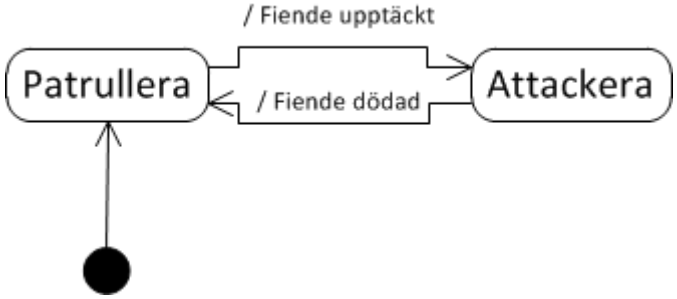
2.1 Artificiell Intelligens i spel

Allt eftersom spel blir mer realistiska inom områden som grafik och fysik medför detta också högre krav på den Artificiella Intelligensen (AIn), vilken ur en spelares perspektiv är minst lika viktig för att ge en positiv helhetsbild (Graepel, Herbrich & Gold, 2004). Den mer traditionella metoden, Finite State Machine (FSM), har länge varit förstavalet för spelutvecklare och har fram tills en tid tillbaka fungerat bra (Millington & Funge, 2009). Problemet med denna metod är dock att systemet lätt blir svårhanterat om komplexa och varierande beteenden hos Non-Playing Characters (NPCs. Sv. agenter) är önskvärt (Orkin, 2006). På grund av dess tätt kopplade struktur, d.v.s. mycket beroenden mellan merparten av klasserna, kan detta medföra att många delar av systemet måste besökas på nytt så det nya beteendet stödjs av alla möjliga tillstånd. Det är framför allt här GOAP bidrar med stor nytta (Orkin, 2006).

2.2 Finit State Machine

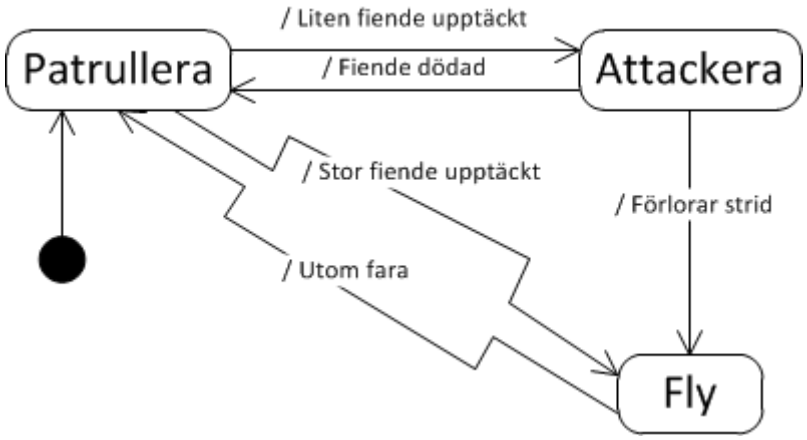
Ordet Finit State Machine (FSM, sv. ändlig tillståndsmaskin) kan tolkas olika beroende på sammanhanget (Millington m.fl., 2009), men gäller i denna rapport simulering av AI i spel. Det finns också många sätt att implementera en FSM på, men i denna rapport betraktas endast den enklare, hårdkodade varianten eftersom denna länge varit den vanligaste. Det finns varianter av FSMs som löser en del problem nämnda i detta kapitel, men det är utanför denna rapports omfattning.

Som Millington m.fl. (2009) beskriver är FSM den än i dag mest använda tekniken för AI i spel. En FSM är en metod att simulera AI genom att definiera ett ändligt antal möjliga tillstånd en agent kan befinna sig i (Figur 1).



Figur 1 Ett exempel på en enkel FSM med två tillstånd och en möjlig transaktion från vardera.

Som Figur 1 visar, vilket Millington m.fl. (2009) beskriver, så har varje tillstånd även en eller flera transaktioner (pilar i tillståndsschemat). Dessa transaktioner definierar en övergång från ett tillstånd till ett annat, vilket i praktiken innebär den kod som exekveras vid en övergång från ett tillstånd till ett annat. Generellt sett handlar det om två kodstycken (metoder); Exit (Utgång) och Entry (Ingång). Exit är, som namnet antyder, den kod som exekveras när ett tillstånd är på väg att lämnas, och Entry när ett tillstånd är på väg att antas. Ett exempel på detta kan tänkas vara när en agent går från tillståndet ”Patrullerar” till ”Attackerar” (Figur 2).



Figur 2 Ett exempel på en enkel FSM tre tillstånd och fem transaktioner

Det kan tänkas att villkoret för att en agent ska ändra det aktuella tillståndet från ”Patrullerar” till ”Attackerar” är att en fiende är inom en radie x. Då detta avstånd blir mindre än gränsvärdet x kommer en transaktion utlösas vilken kommer ta agenten till det nya önskade tillståndet. Innan övergången är färdig kommer dock dess utgång- och ingångskod att exekveras som ser till att på ett korrekt sätt avsluta det aktuella tillståndet ”Patrullerar”, och sedan förbereda tillståndet ”Attackerar”. I detta fall (Figur 1) kan det t.ex. handla om att agenten först slänger en cigarett den har i handen (avsluta ”Patrullera”), följt av att dra fram sitt vapen (förbereda ”Attackera”). Kortfattat kan en transaktion ses som den sekvens med åtgärder som sker när ett villkor är uppfyllt eller en särskild händelse sker och en agent ska byta tillstånd.

En FSM uppdateras vanligvis i varje frame (bilduppdatering) genom en metod *Update*. När Update exekveras ser den till att ta hand om aktuella transaktioner, eller om inga finns tillgängliga bara utför de åtgärder som specificerats för det aktuella tillståndet *(*Millington m.fl., *2009).* Ett exempel på detta kan vara att en agent som befinner sig i ”Patrullerar”-tillståndet ska förflytta sig inom en viss yta i en värld samtidigt som denna letar efter fiender.

Ett av de viktigaste motiven till att använda FSM är just att de i sitt grundutförande är relativt enkla att förstå och implementera (Millington m.fl., 2009). Som beskrevs ovan kan de dock snabbt bli komplexa och svåra att underhålla när antalet möjliga tillstånd ökar. Detta eftersom alla beslut måste hårdkodas för alla möjliga situationer (Orkin, 2006). En lösning på detta, som beskrivs av Millington m.fl. (2009), är att införa s.k. måldrivet beteende (Goal-oriented behavior).

2.3 Goal-Oriented Behavior

Finit state machines (FSMs, sv. Ändlig tillståndsmaskin), som beskrevs i föregående kapitel, är inom datorspel en teknik för att bestämma hur agenter i en spelvärld ska reagera på nya händelser och/eller om ett villkor uppfyllts, varpå den antar ett nytt, förutbestämt, tillstånd. En stor nackdel med denna teknik är enligt Orkin (2006) att det lätt uppstår ett tydligt mönster över hur agenter beter sig. Detta kan försämra känslan av realism för spelare eftersom dessa då kan förutsäga hur agenter kommer agera i olika situationer. Som både Orkin (2006) och Millington m.fl. (2009) dock nämner så går det att uppnå komplext beteende även med FSMs, men då till kostnaden av svårhanterad och tätt kopplad kod. En lösning på detta, som beskrivs av Millington m.fl. (2009), är att införa s.k. måldrivet beteende (Goal-oriented behavior). Måldrivet beteende är, enligt Millington m.fl. (2009), en teknik där agenter utför åtgärder (*Actions*) för att uppnå ett visst mål (*Goal*).

Ett mål (*Goal*), vilka det kan finnas flera hundra av, är ett tillstånd en agent strävar mot att uppnå genom att utföra dessa s.k. åtgärder. Ett exempel på mål kan vara att stilla hungern, återställa hälsan eller förgöra en fiende. Agenten kan ha obegränsat med mål aktiva, men det viktigaste prioriteras först. Prioriteringen är i praktiken bara en siffra som talar om hur viktigt det är att ”just detta” mål uppnås. Ett exempel kan vara att en agent börjar bli hungrig och måste äta. Får den samtidigt syn på en fiende, kan det tänkas att det nya målet ”döda fiende” är viktigare än att äta, vilket då kommer prioriteras därefter. Kortfattat kan det sägas att det finns två typer av mål, d.v.s. de som går att utföra fullständigt (t.ex. döda fiende) eller de som ständigt finns där (t.ex. hunger).

Åtgärder (*Actions*) är de handlingar en agent har tillgängliga för att lösa ett aktivt mål. Det finns olika sätt att representera vilka åtgärder som är tillgängliga, men det som illustreras i denna beskrivning är samma som Millington m.fl. (2009) beskriver, d.v.s. att låta objekt i världen generera åtgärder beroende på det aktuella tillståndet i spelet. Ett exempel på detta kan vara en agent som har ett aktuellt mål att sänka sin hunger, med andra ord att dess tillstånd är ”hungrig”. Objekt så som kylen kan då upptäcka detta och erbjuda en åtgärd ”hämta rå mat”. En åtgärd som denna lovar att på något sätt ta agenten ett steg närmare ett mål, men det kan krävas flera åtgärder för att faktiskt nå hela vägen fram till måltillståndet. För att fortsätta på exemplet där en agent är hungrig och en tillgänglig åtgärd är att hämta rå mat i kylen så kan denna åtgärd lova att uppfylla målet ”hungrig”, men kommer i sig inte minska den faktiska hungern. Istället blir en ny åtgärd tillgänglig av ugnen, ”laga mat”, som efter utförande i sin tur möjliggör en ny åtgärd ”plocka ut lagad mat”. Slutligen blir åtgärden ”ät mat” tillgänglig vilken är den som faktiskt sänker agentens hunger.

Åtgärder är definierade i förväg, men till skillnad mot FSMs så är ordningen de sker i eller när inte specificerat i systemet för varje tillstånd. Istället kan det med GOB ses som ett sätt att låta agenten själv välja bland tillgängliga åtgärder eller förbättringar (som i fallet med hunger) till det tillstånd den befinner sig i, för att så småningom nå ett måltillstånd. Att kunna både åtgärda eller förbättra ett tillstånd syftar i detta sammanhang på att ett mål inte alltid är något som måste lösas direkt (till skillnad mot t.ex. att döda en fiende), utan kan också vara att agenten t.ex. är hungrig eller trött. Dessa finns alltid som mål att motverka och ökar ständigt med tiden (agenter blir successivt mer hungriga och trötta). Detta behöver endast ”förbättras” när det når tillräckligt hög prioritet. Ett bra exempel på detta, beskrivet av Millington m.fl. (2009), är i The Sims-spelen där varje agent ständigt påverkas av omvärlden och ständigt ändrar sina attribut. Attribut kan vara saker som ”gladhet”, ”trötthet” m.fl. som

ständigt existerar, men inte alltid är lika viktiga att uppnå. En agent som är hungrig och vill äta kanske får ett nytt, mer prioriterat mål, som att släcka en eld.

Valet av åtgärder att utföra sker genom denna prioritering. Ju viktigare ett mål är att uppnå, desto mer kommer det ”pressa” agenten till att uppnå detta. I praktiken beskrivs det av ett heltal, som i fallet med hunger är enkelt att förstå då det ökar successivt med tiden, precis som i verkligheten. Med andra ord, ju hungrigare en agent är, desto högre prioriterat blir det att äta. När en agent väl ska välja vilket mål att försöka uppnå kan detta ske på lite olika sätt. En enkel variant som Millington m.fl. (2009) beskriver är att helt enkelt gå igenom de aktiva målen och först hitta det med högst prioritet, för att sedan ta den åtgärd som motverkar detta. Om en agent har målet ”hunger” prioriterat till 5 (d.v.s. ”5 enheter hungrig”) väljs den åtgärd som sänker detta värde mest. Detta kan ge ett beteende som inte alltid känns logiskt för en spelare, och något sorts resonemang bakom val av åtgärder hade möjligtvis varit mer lämpligt. Det är här Goal-Oriented Action Planning (GOAP) kommer in i bilden (Orkin, 2003).

2.4 Goal-Oriented Action Planning

Som visades i föregående kapitel, och som beskrivs av Millington m.fl. (2009), går det att med hjälp av måldrivet beteende (Goal-oriented behavior, GOB) få agenter mer flexibla i sitt målsökande. En nackdel var dock val av åtgärder. I exemplet i föregående kapitel valdes bara den för tillfället bästa åtgärden, utan att ta hänsyn till andra åtgärder och hur dessa hade påverkat framtida tillstånd. GOAP (goal-oriented action planning) är, som Orkin (2003 & 2006) förklarar, en teknik för beslutsfattande och har använts med stor framgång inom AI i spel, t.ex. i *F.E.A.R.* (Monolith productions, 2005). Genom att utnyttja en s.k. planerare, som genererar en sekvens med åtgärder att utföra, går det att få agenter att agera trovärdigt och logiskt. Till skillnad från GOB som denna bygger vidare på, är det här istället önskvärt att få den sekvens med åtgärder som på bästa/billigaste sätt uppnår ett mål. För att åstadkomma detta kommer två sökalgoritmer användas, närmare bestämt A* (A-star) och IDA* (Iterative deepening A-star). För att förstå GOAP är följande fyra begrepp viktiga; *Goal* (önskat tillstånd), *Worldstate* (aktuellt tillstånd), *Action* (åtgärd) och *Plan* (färdiga planen för att nå önskat tillstånd). Eftersom GOAP bygger vidare på GOB kan mål och åtgärder (*Goal* och *Actions*) ses på samma sätt i teorin, men skiljer sig implementationsmässigt för att stödjas av sökalgoritmerna.

Goal är ett tillstånd, precis som i GOB, vilket en agent vill uppfylla. Det är detta som används som mål för sökalgoritmen, vilken i detta arbete bygger på A* och IDA*.

Action är en möjlig åtgärd, också precis som i GOB, en agent kan utföra för att komma ett steg närmare målet. Varje åtgärd har ett *Precondition* (förhandsvillkor) för vad som måste vara uppfyllt för att denna åtgärd ska kunna utföras, samt en *Effect* (resultat) vilket är det resultat en åtgärd åstadkommer. Varje åtgärd representerar en övergång till ett nytt tillstånd. Varje åtgärd har även en kostnad associerad med sig. Denna kostnad kan t.ex. vara den tid det tar att utföra denna åtgärd (t.ex. tiden för animationen att dra fram ett vapen). En åtgärd kan även ha ytterligare krav vilka kallas *context preconditions* (sammanhangs förhandsvillkor). Detta är villkor som måste vara sanna, men som planeraren inte försöker uppfylla. Detta är ett mer komplicerat villkor, och hur detta fungerar är utanför detta arbetes tillämpningsområde.

Worldstate är det tillstånd agenten befinner sig i, vilken planeraren använder som utgångspunkt då den letar efter en sekvens av Actions som tar agenten från det aktuella tillståndet till det önskade tillståndet. I praktiken är *Worldstate* och *Goal* av samma typ, och kommer i implementationen därför representeras på samma sätt då de båda står för tillstånd en agent befinner eller kan befinna sig i. Under sökning skapar planeraren nya Worldstates allt eftersom åtgärder appliceras.

Plan är den sekvens av åtgärder som krävs för att uppfylla ett mål. Det är denna som kommer ta en agent från dess starttillstånd till ett önskat sluttillstånd (från *Worldstate* till *Goal*).

Som Orkin (2003) förklarar ersätter inte GOAP i sig behovet av en FSM, men förenklar den markant. Då all logik för övergångar mellan tillstånd sker i de möjliga åtgärderna (*Actions)* kan den FSM som används vara kortfattad. Så lite som två tillstånd kan krävas, nämligen; *Goto* och *Animate* (Orkin, 2006). *Goto* är det tillstånd då en agent rör sig till en angiven position i världen, och *Animate* när en animation ska spelas. Precis som Orkin (2006) beskriver så kan en FSM mycket väl ge liknande trovärdighet hos agenter som GOAP, men strukturen för en FSM blir snabbt väldigt komplex. Det är framförallt detta, dess löst kopplade struktur, som är den stora motiveringen att välja GOAP framför en renodlad FSM. Utöver att koden blir mer strukturerad, underhållbar och återanvändbar, får också agenterna ett mer varierande och mindre förutsägbart beteende. Agenter kan ibland lösa problem på sätt som under utveckling inte var förutsett (Orkin, 2004).

Den implementation av GOAP som (Orkin, 2003) beskriver fungerar på så sätt att en agent genererar en plan, direkt under körning, genom att förmedla ett mål som önskas uppnås till en s.k. *planerare*. Planeraren kommer i sin tur söka i en rymd med åtgärder efter en sekvens som tar agenten från dess starttillstånd (*Worldstate*) till ett måltillstånd (*Goal*). Detta kallas att *formulera* en plan. Precis som vid vägplanering kommer denna antingen returnera en sekvens med åtgärder om det existerar någon, annars en tom lista. Agenten kommer, om en existerar, följa denna plan tills den är färdig, ej längre är giltig eller tills ett annat mål blir mer relevant. I de två sistnämnda fallen kommer agenten avbryta och förmedla det nya målet och få, om möjligt, en ny plan genererad.

En stor fördel med GOAP är att den enkelt möjliggör dynamisk planering (Orkin, 2003). Eftersom agenter, direkt under körning, planerar de åtgärder att ta för att uppnå ett mål kan denna dynamiskt hitta alternativa lösningar på problem. Ett bra exempel som Orkin (2003) beskriver är när en agent har som mål att förgöra en fiende. En tänkbar sekvens med åtgärder hade kunnat vara att agenten drar sitt vapen och sedan skjuter mot fienden. I detta fall har däremot inte agenten något vapen, och måste därför först gå till en position i spelvärlden där det finns ett möjligt vapen och sedan plocka upp detta. Det kan också tänkas att vapnet är utan ammunition, och åter igen måste agenten ta sig till en plats där detta finns, för att till sista ladda sitt vapen och skjuta mot fienden. Allt detta upptäcks av planeraren som direkt under körning kan generera en, om möjligt, korrekt sekvens med åtgärder till agenten.

Att hantera situationer som den nämnd ovan är i GOAP en enkel process. Allt som behövs är att agenten, med hjälp av vetskap om dess omgivning och tillstånd, levererar sitt aktuella tillstånd och mål till planeraren som sedan ser till att hitta en lämplig lösning (om existerar). För att göra detta använder planeraren en sökalgoritm.

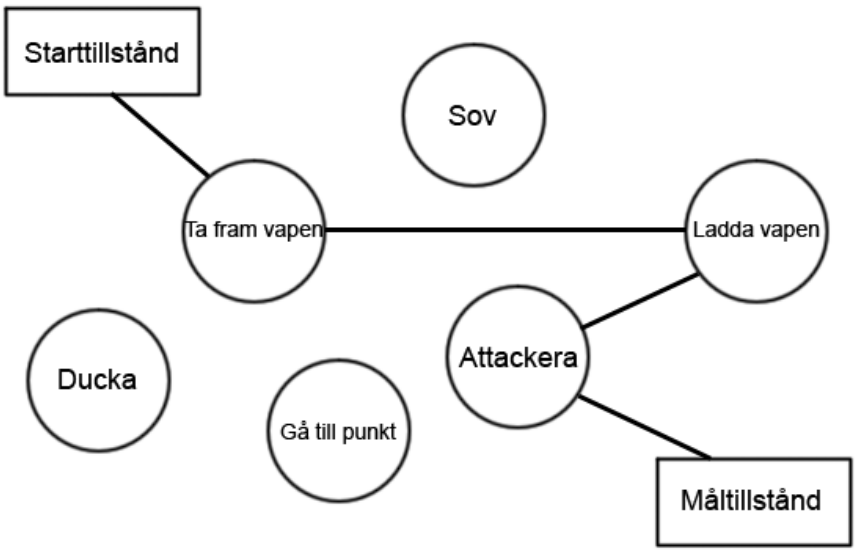
2.5 Planering med A*

Alla fördelar med GOAP som beskrivits ovan kan realiseras tack vare att en sökalgoritm används under körning för att dynamiskt hitta lösningar på problem. Sökalgoritmen som används i GOAP beskrivet av Orkin (2003, 2006) är A* (uttalat A-star).

A* är för de flesta utvecklare en välanvänd algoritm, dock vanligen för vägplanering (eng. *pathfinding*). Faktum är däremot att A* är en väldigt generell algoritm, och lämpar sig därför enligt Orkin (2003) ypperligt för att planera sekvenser av åtgärder i GOAP. Om implementerad på ett modulärt sätt som beskrivet av Higgins (2002) kan denna användas av både vägplaneraren och åtgärdsplaneraren (eng. *Action-planner*). Åtgärdsplaneraren behöver bara implementera sina egna klasser för A*s noder, kanter och mängd. I fallet med åtgärdsplaneraren används aktuellt tillstånd (eng. *Worldstate*) samt sluttillstånd (eng. *Goal*) som start- och slutnoder respektive, och åtgärder (eng. *Actions*) som kanter (Orkin, 2003) (Figur 3). För varje åtgärd som appliceras på det aktuella tillståndet nås en ny nod i mängden. Mängden kan representeras på flera sätt, men för att sökningen ska vara så snabb som möjligt måste planeraren enkelt kunna få en lista med nya åtgärder som är möjliga att använda för det aktuella tillståndet. Vid vägplanering, med mängden representerad som t.ex. ett rutnät (eng. Grid), är detta en enkel och snabb uppgift. Med hjälp av indexering går det att direkt komma åt närliggande noder (rutor). Allt som behövs är några korta tester för att se så en nod t.ex. är godkänd att röra sig över (eng. *Walkable*) m.fl. Detta går inte riktigt att lösa på samma sätt för GOAP.

På grund av hur strukturen ser ut i GOAP så är där inga explicita kopplingar mellan åtgärder (kanter) och mål (noder) eller åtgärder och åtgärder, men ett sätt att representera detta är symboliskt (Orkin, 2004). Symboliskt syftar i detta sammanhang på flaggor (enum i C++ för detta arbete) som indikerar om något är uppfyllt eller ej. På detta sätt går det att hitta giltiga åtgärder genom att se om dess effekt löser ett ouppfyllt mål i förhållande till måltillståndet. Mängden som representerar alla tillgängliga åtgärder kan sedan returnera alla de åtgärder som uppfyller hela eller delar av måltillståndet. I exemplet i föregående avsnitt där en agent vill förgöra en fiende kan ”Förgöra fiende” i måltillståndet ses som en symbol vilken är satt till *sann* (eng. true), men eftersom det aktuella tillståndet (WorldState) har samma symbol satt till *falsk* (eng. false), är det detta planeraren vill uppfylla. Detta används sedan för att returnera en åtgärd så som t.ex. ”Attackera” eftersom denna har effekten att den ändrar just symbolen ”Förgör fiende” till sann. I de fall där det finns fler än en åtgärd kan prioritering användas för att ta det som enklast uppnår ett mål. Så om agenten har ett vapen används detta framför att t.ex. slå med händerna. Har agenten inget vapen kommer denna åtgärd vid senare planering inte kunna utföras på grund av att dess förhandsvillkor inte kan uppfyllas, och planeraren kommer i så fall återgå till ett tidigare alternativ med vetskapen att inget vapen finns tillgängligt, nämligen att slå med händerna (vilket alltid kan anses vara möjligt).

Figur 3 visar ett exempel på hur planeraren söker sig fram via tillgängliga kanter (åtgärder, representerade som cirklar) för att från startnoden nå slutnoden (starttillståndet och måltillståndet, representerade som rektanglar). För varje åtgärd som appliceras kommer planeraren till ett nytt tillstånd, d.v.s. en ny nod i mängden. Målet i figur 3 är att förgöra en fiende.



Figur 3 Ett exempel på hur planeraren söker igenom en mängd

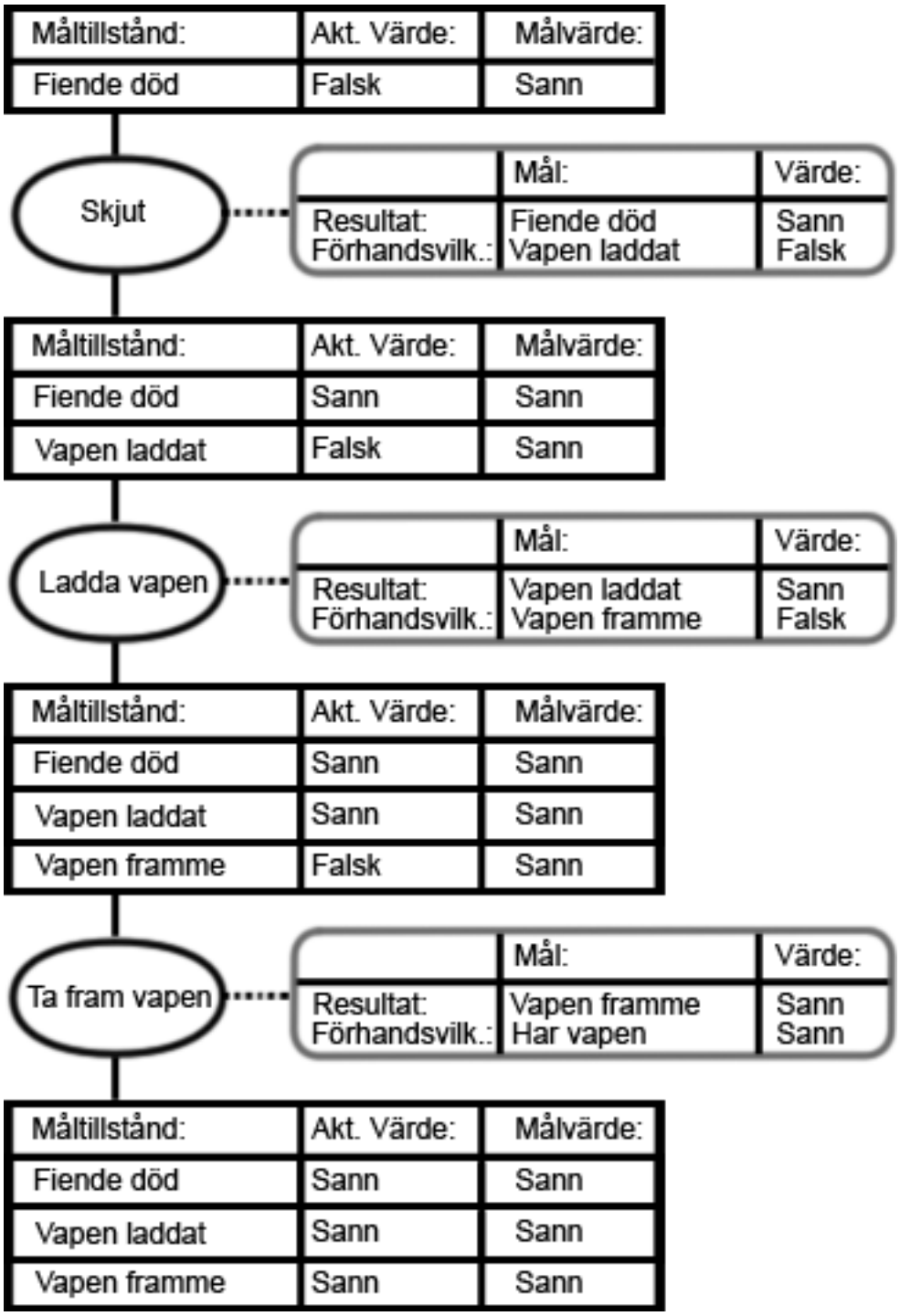
Vad som gör A* lämplig är det faktum att den utnyttjar heuristik (Millington m.fl., 2009). Detta är en metod att försöka få systemet till att göra smarta gissningar och på så sätt rikta sökningen åt lovande håll. I A* görs detta för t.ex. vägplanering (som är ett av de vanligaste användningsområdena) genom att beräkna en summa av kostnaden av den hittills hittade vägen och den uppskattade återstående vägen till målet, där den sistnämnda är det heuristiska värdet (Millington m.fl., 2009). Denna kostnad kallas för *total-estimated cost* (totala beräknade kostnaden). För att kunna utnyttja detta i GOAP visar Orkin (2003) hur ”den hittills hittade vägen” kan beskrivas som summan av kostnaderna av de åtgärder (*Actions*) som krävs för att komma till det aktuella tillståndet, och ”den återstående vägen” som summan av de ej uppfyllda målen i sluttillståndet. I figur 4 blir detta då summan av antalet mål ej uppfyllda, d.v.s. de målen satta till falskt (skillnaden mellan det aktuella tillståndet och måltillståndet). En *väg* är den sekvens av kanter (åtgärder i GOAP) som planeraren har sökt sig fram igenom, och syftar inte nödvändigtvis på en *väg* som i vägplanering.

Under sökning lagrar A* noder i två listor, nämligen *Open*- (öppna) och *Closed* (stängda)-listan (Millingon m.fl., 2009). Den öppna listan huserar de noder som har blivit besökta men ännu inte har bearbetats, medan den stängda listan håller de noder som har blivit bearbetade. Bearbetning av en nod sker en gång varje iteration, och det är här nodens närliggande noder (noder möjliga att nå från den aktuella noden. Eng. adjacent nodes) hämtas genom dess tillgängliga kopplingar (åtgärder i GOAP), vilka alla får den totala beräknade kostnaden uträknad för att sedan placeras i den öppna listan. Dessa noder är nu besökta, men har i sin tur själva inte fått sina grannar undersökta och kommer, beroende på deras ”totala beräknade kostnad”, bli bearbetade i senare iterationer. Detta eftersom den nod som bearbetas under nästkommande iteration är den nod med lägst beräknad totalkostnad i den öppna listan, då det är denna nod som verkar mest lovande att ingå i den färdiga lösningen. När en nod har bearbetats placeras denna i den stängda listan. Denna stängda lista används i praktiken endast för att indikera att en nod har bearbetats, vilket vid implementation oftast görs av en flagga som noden själv lagrar. På grund av detta är den ”stängda listan” endast till för att förklara principen bakom A*, och behövs egentligen inte

när algoritmen implementeras. Till skillnad mot den stängda listan krävs dock en öppen lista, eftersom denna håller reda på vilken nod att bearbeta under nästa iteration.

För att sammanfatta: När en sekvens ska planeras börjar planeraren med att bearbeta den första och enda noden i listan, nämligen startnoden (detta är utgångspunkten). Denna nod (aktuella noden) bearbetas sedan, som beskrivits ovan, genom att hämta alla dess noder möjliga att nå via kopplingarna (d.v.s. noder möjliga att nå från denna nod genom dess kanter), och beräknar deras kostnader samt placerar dessa i den öppna listan. Den aktuella noden ”placeras” sedan i den stängda listan, d.v.s. en flagga/indikator i noden sätts till ”I den stängda listan”. På samma sätt indikeras också en nod om den är ”I den öppna listan” eller ”Ej besökt”. Dessa flaggor används för att bl.a. snabbt avgöra hur en nod ska få heuristiken beräknad och är mer en implementationsspecifik egenskap för att optimera algoritmen. En alternativ lösning hade varit att även ha en stängd lista och inga flaggor, och under varje iteration söka igenom både den öppna och stängda listan för att på så sätt avgöra om en nod redan ligger i någon av dessa. Detta kräver dock extra sökning i respektive lista, vilket mycket uppenbart hade försämrat prestandan ordentligt.

En viktig egenskap i A* för GOAP beskriven av Orkin (2003) är att han låter den söka regressivt, d.v.s. baklänges från mål till start. Motiveringen bakom detta är att en sökning som sker på detta sätt steg för steg tar den lösning som uppfyller det aktuella måltillståndet, istället för att leta sig fram via delsteg som i efterhand inte gick att utföra fullt ut. Orkin (2003) beskriver ett scenario där en obeväpnad agent vill förgöra en fiende med hjälp av ett monterat lasergevär, som i sin tur kräver ström för att fungera. Vid en vanlig framåtsökning kommer planeraren först anse att agenten ska springa fram till det monterade lasergeväret med åtgärden *GotoPoint* följt av att den ska använda det med åtgärden *AttackMounted*. Villkoret för att *AttackMounted* ska kunna utföras visare sig dock vara att en generator måste vara aktiverad vilket den inte visar sig vara. Det kommer sedan krävas en uttömmande sökning (exhaustive brute force) för att komma fram med en giltig plan som först tar spelaren till generatorn och startar denna, och sedan till lasergeväret för att förgöra fienden. En s.k. regressiv sökning kommer vara mer intuitiv, enligt Orkin (2003), eftersom denna genom att gå baklänges först kommer börja vid sluttillståndet och direkt leta efter åtgärder som uppfyller kraven för att utföra *AttackMounted*. Detta kommer leda till att en plan genereras som direkt ser till att agenten först aktiverar generatorn, sedan använder lasergeväret. Ett exempel på hur en regressiv sökning i GOAP kan ske illustreras i figur 4. Rektanglar representerar måltillståndet, och figur 4 visar hur detta ökar efterhand som nya åtgärder adderas som uppfyller ett av målen. Detta eftersom åtgärder i sig har s.k. *preconditions* (förhandsvillkor) som i sin tur också måste uppfyllas. Elipserna representerar dessa åtgärder som görs, och de avrundade rektanglarna visar dess *effect* (resultat, de mål som uppfylls) och förhandsvillkor.



Figur 4 Ett exempel på hur regressiv sökning i GOAP kan ske.

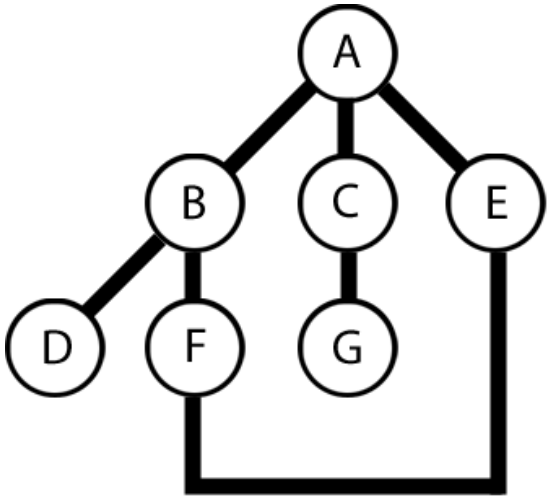
En stor nackdel med standarimplementationen av A* tillsammans med GOAP är att den tar för givet att där alltid finns minst en väg till det önskade målet, och kommer att söka tills den hittar denna eller terminera när alla möjliga kombinationer är testade (Millington m.fl. 2009). I fallet med vägplanering är detta inget problem, då det oftast finns en giltig väg eller åtminstone en begränsad sökmängd. Vid planering av åtgärder i GOAP är det däremot mer vanligt att en plan inte går att utföra, men att det samtidigt ständigt finns nya möjliga åtgärder att utvärdera. Då sökningen, precis som i fallet med vägplanering, i värsta fall kan pågå tills alla möjliga kombinationer har utvärderats finns risken att denna fastnar i en s.k. ”evighetsloop”. Detta kan ofta undvikas i mindre spel, där agenter inte har fler än ett dussin

olika möjliga åtgärder att utföra, men allt eftersom dessa blir fler ökar antalet möjliga sekvenser av åtgärder lavinartat. Då A* använder s.k. *Open*- (öppen) och *Closed* (stängd)-listor (inte nödvändigtvis den stängda listan som beskrevs tidigare) för att hålla reda på tidigare besökta och utvärderade noder kan detta få som påföljd att minnet blir översvämmat. Ett enligt Millington m.fl. (2009) enkelt sätt att lösa detta är att införa ett max-djup för sökningen, d.v.s. en gräns för hur många åtgärder en sekvens får innehålla och på så sätt undvika en ”evighetsloop”.

En nackdel som återstår är att den dynamiska planering som GOAP möjliggör också medför att sökningarna kan bli frekventa. Detta tillsammans med ovannämnda minnesproblem skapar stor stress på minnet. Eftersom minnesåtkomst är en dyr process (Manegold, Boncz & Kersten, 2000; R. Mahapatra, N m.fl., 1999; Carvalho, C., 2002) kan det finnas bl.a. prestanda att vinna genom att utnyttja en algoritm med lägre minneskomplexitet, och en kandidat för detta är IDA* (Millington m.fl., 2009).

2.6 Iterative Deepening A*

Iterative deepening A* är, trots sitt namn, väldigt olik A*. Som Millington m.fl. (2009) beskriver så håller den inte reda på någon öppen eller stängd lista. Istället jobbar den iterativt med begränsade djupet-först sökningar, tillsammans med ett extra *cut-off value* (avbrytningsvärde).



Figur 5 Illustration av ett möjligt sökträd för en djupet-först sökning.

Genom att begränsa djupet-först sökningarna kan s.k. oändligt djup eller cykler undvikas (Russell & Norvig, 2003), vilket kan ses i figur 5. En vanlig djupet-först sökning hade i detta träd hamnat i en evighetsloop, eftersom sökningen via nod F hade tagit den vidare till nod E, vilken i sin tur har en koppling tillbaka till rotnoden som sedan tar sökningen tillbaka till F o.s.v. Med begränsat djup kan detta undvikas eftersom sökningen avbryts när en nod befinner sig på samma djup som max-djupet, även om ytterligare noder hade kunnat nås från den aktuella noden. Ett problem som uppstår med denna lösning är om detta begränsade djup är lägre än målnoden, eftersom denna då inte kan nås. Detta går att lösa med iterativ begränsad djupet-först sökning (eng. Iterative deepening depth-first search, eller IDDFS). Genom att istället utföra begränsad djupet-först-sökning under flera iterationer, med ett max-djup som inkrementeras för varje ny iteration, går det att på så sätt hitta en målnod på

det lägsta möjliga djupet. Fördelen med IDDFS är att den inte lagrar data från föregående iterationer utan utför ny djupet-först sökning för varje iteration, men med ökat max-djup. Detta ger därför minneskomplexitet i klass med vanlig djupet-först sökning: O(b*d), där b är förgreningsfaktorn i sökträdet, och d är djupet där målnoden hittades (Russell m.fl., 2003). På grund av de upprepade sökningarna av noder som redan har besökts tidigare kan det tänkas att detta medför mycket extra arbete. Jämfört med bredden-först sökning kommer detta skapa extra beräkningar, men skillnaden är inte så stor som det först kan tänkas. Tvärtom så har inte dessa upprepa sökningar så stor inverkan, eftersom sökträd med samma (eller nästan samma) förgreningsfaktor håller flest noder på högsta djupet (Russell m.fl., 2003). Detta medför att de noder som besöks flest gånger (de på lägre djup) ändå är mycket färre till antalet än de på det högsta djupet, vilka endast besöks en gång. Mer exakt så kommer de noder på första djupet (rotnoden) besökas d gånger (där d är det lägsta djupet för att nå målnoden), andra djupet d-1 gånger, tredje djupet d-2 gånger o.s.v. fram till max-djupet (där målnoden hittas) som endast besöks en gång. Detta medför att IDDFS får ungefär samma tidskomplexitet som bredden-först sökning, nämligen: O(b^d), där b är förgreningsfaktorn i sökträdet, och d är djupet där målnoden hittades.

IDA* bygger ut detta koncept ytterligare genom att införa heuristik till IDDFS. Den stora skillnaden mellan dessa två är att IDA* använder ett avbrytningsvärde, vilket är den totala beräknade kostnaden, och inte endast max-djupet för att veta när en sökning ska avbrytas (Russell m.fl., 2003). Detta avbrytningsvärdet sätts under varje iteration till den lägsta totala beräknade kostnad en nod uppnådde som är större än det avbrytningsvärde i föregående iteration.

Vid sökning börjar IDA* med att sätta detta avbrytningsvärde till den heuristiska kostnaden från startnoden till målnoden (eftersom kostnaden vid startnoden är 0). Därefter utförs en begränsad djupet-först sökning. Om sökningen når målnoden avslutas den och planen returneras. Annars sätts avbrytningsvärdet till den lägsta totala beräknade kostnaden större än föregående avbrytningsvärde och en ny djupet-först sökning påbörjas med ett högre max-djup.

För att undvika att samma mängd med noder besöks återkommande går det att använda ett s.k. *transposition table*. Exakt hur dessa fungerar är utanför denna rapports tillämpningsområde, men i denna implementation är det i grund och botten en enkel hashtabell (Millington m.fl., 2009). För GOAP håller denna tabell tidigare tillstånd vilka uppnåtts genom en sekvens av åtgärder, där en hashnyckel skapad från nodens tillstånd som nyckel. Dessa tillstånd kontrolleras vid varje djup av en djupet-först söking, och med hjälp av denna tabell går det att snabbt kolla upp om ett tillstånd tidigare har uppnåtts (d.v.s. om en sekvens med åtgärder redan åstadkommit detta tillstånd). Om så är fallet lagras det tillstånd som krävde lägst antal åtgärder för att uppnås. Är istället det tillstånd den aktuella sökningen uppnått sämre än det som redan existerar i tabellen kan denna sökväg helt kasseras. Genom att endast ha ett värde associerat med en hashnyckel går det även att uppnå konstant tidskomplexitet vid uppslagning (O(1)).

Genom att i IDA* inte lagrar någon stängd- eller öppenlista kan minneskomplexiteten jämfört med A* minskas ordentligt, samt inget krav på sorterad uthämtning av noder (A* kräver att nästa nod som expanderas är den billigaste, se kapitel 2.5). Som beskrevs ovan så krävs dock iterativa sökningar för att åstadkomma detta, vilket har nackdelen att tidigare redan utforskade noder kan behöva besökas flera gånger. Med hjälp av ett s.k. *transposition*

table kan detta till viss del motverkas, och en implementation som den beskriven här är enligt Millington m.fl. (2009) tack vare detta mer lämpad för GOAP än A*, och kan i vissa fall även vara snabbare.

3 Problemformulering

Allt eftersom spel blir mer realistiska inom områden som grafik och fysik medför detta också högre krav på den Artificiella Intelligensen (AIn), vilken ur en spelares perspektiv är minst lika viktig för att ge en positiv helhetsbild (Graepel, Herbrich & Gold, 2004). På grund av detta är det viktigt att hela tiden sträva efter att utveckla bra metoder för skapandet av AI, både med avseende på utvecklingstid samt realism hos AIn. Detta var för Orkin (2003) det stora motivet bakom att införa goal-oriented action planning (GOAP) i spel, en metod som låter agenter själva ”resonera” fram lösningar genom att planera sina åtgärder under körning.

Precis som med utvecklingen av grafik och fysik medför också mer avancerad AI generellt sett högre belastning på systemet det exekveras på. Detta gäller även GOAP, som jämfört med en typiskt hårdkodad Finit State Machine (FSM) utnyttjar en sökalgoritm för att uppnå nya tillstånd och genererar sekvenser med åtgärder direkt under körning. Sökalgoritmen som vanligen används för GOAP är A* (A-star) (Orkin, 2003), vilken har den stora nackdelen att den utnyttjar stora mängder minne under sökning, samt kräver en sorterad lista (Millington m.fl., 2009). Då just sortering samt minnesåtkomst är en förhållandevis dyra processer (Manegold m.fl., 2000) vid kritiska beräkningar så som realtidssökning är det önskvärt att minska detta, utan att låta tidskomplexiteten för algoritmen lida för mycket. Enligt Millington m.fl. (2009) är därför IDA* en bättre lämpad algoritm. Trots åldern på referensen Manegold m.fl. (2000), har detta påstående visat sig gälla sedan 1980-talet över en period på mer än tjugo år (R. Mahapatra, N m.fl., 1999; Carvalho, C., 2002). Dessa visar hur processorns prestanda ökat med cirka 60 % per år samtidigt som accesstiden till minnet endast ökat med knappt 10 % per år.

Syftet med detta arbete är att utvärdera hur väl två olika sökalgoritmer lämpar sig för GOAP med avseende på prestanda. Arbetet tittar på hur en standard-implementation av A* står sig prestandamässigt mot IDA* för GOAP, där den sistnämnda enligt Millington m.fl. (2009) är bättre lämpad på grund av dess egenskaper. Detta arbete kommer inte titta på direkt minnesanvändning då detta hade krävt ingående analys av hur en processor kommunicerar med systemminnet och hur cacheminnet utnyttjas, vilket hade försenat arbetet.

Det första delmålet, vilket vidare är uppdelat i två sub-delmål, är implementationen. Detta består av att först utveckla AI kapabel till att fatta beslut med hjälp av GOAP vilken använder A*. Agenterna ska kunna utnyttja denna för att röra sig i en virtuell spelvärld i 3D och interagera med eventuella faror, t.ex. attackera eller fly. Nästa sub-delmål är att implementera IDA* istället för A*, applicerbar på samma virtuella 3D-värld som A*.

Det andra delmålet är att testa implementationen för A* och IDA* respektive genom att skapa experiment. För att detta experiment ska bli så rättvist som möjligt måste samma antal agenter simuleras för respektive implementation, samt att 3D-världen som dessa rör sig i måste vara identisk i båda testen (för testen av båda algoritmerna). Data ska även lagras för att kunna se om implementationerna förändrar agenternas beteende.

Hypotes: Genom att använda IDA* som enligt Millington m.fl. (2009) minskar minnesanvändningen, vilken är en stor flaskhals i förhållande till processorn (Manegold m.fl. 2000) samt inte kräver en sorterad lista, kommer prestandan för GOAP-implementationen öka.

3.1 Metodbeskrivning

För att möjliggöra testning av hypotesen kommer en implementation av GOAP göras, och experiment skapas för att testa denna.

För att uppnå det första delmålet kommer först och främst A* implementeras, med motiveringen att det är en väletablerad och väldokumenterad algoritm. Genom att göra detta på ett modulärt sätt (Higgins, 2002) och sedan testa algoritmen för vägplanering går det att på så sätt få upp en design som gör det enkelt att i efterhand implementera fler sökalgoritmer. Valet att först testa algoritmen för vägplanering baseras på den visuella återkopplingen vägplanering kan ge under sökning, t.ex. om ett rutnät (eng. grid) används för att representera sökgrafen så kan enskilda rutor markeras med olika färger beroende på var sökningen befinner sig. När en grunddesign av sökmotorn är implementerad kommer IDA* implementeras, och eventuella buggar som uppstår bör då endast bero på den nya koden och är därför lättare att hitta och lösa. Därefter implementeras GOAP, vilket medför att designval kan göras som förenklar planeringen av åtgärder.

Implementationen kommer därför bestå av två steg; implementera grunderna för A* samt IDA*, och därefter GOAP, och på så sätt tvinga att algoritmerna implementeras på ett modulärt sätt. Detta gör också att GOAP enkelt kan skifta mellan de båda. Genom detta upplägg, d.v.s. att ha resterande delar av GOAP implementerade på samma sätt för båda algoritmerna, kan ett rättvist experiment genomföras.

För att möjliggöra testning av hypotesen, och därmed uppnå delsteg två kommer ett experiment skapas vilket kommer bestå i att skapa en virtuell testbana. Denna testbana kommer husera totalt tjugo stycken agenter. Dessa agenter kommer samtliga styras av GOAP vilken under testkörning använder samma sökalgoritm för alla. Den data som ska lagras kommer bestå av alla åtgärder med respektive plan, tiden det tog för vardera planering samt hur många noder som expanderats under denna sökning. Genom att lagra dessa data går det att delvis uppnå det andra delmålet, d.v.s. att upptäcka eventuella avvikelser mellan de olika sökalgoritmerna.

Det finns många tidigare arbeten som tittar på mer avancerade sätt för att mäta prestanda hos en sökalgoritm (Gaschig, 1979; Korf, Reid & Edelkamp, 2001), men då just detta arbete fokuserar på spel är den generella prestandan (processortid spenderad) mer intressant. För att mäta prestandan kommer därför vanliga tidsstämplar utnyttjas för att mäta exakt hur mycket processortid som gått åt till att sökningen med respektive algoritm. Även profileraren i Microsoft Visual Studio 2010 Ultimate kommer användas för att hitta de metoder som kostar mest processortid för vardera algoritm. En annan möjlig väg hade varit att använda försökspersoner, och låta dessa betrakta simuleringarna för de båda algoritmerna. Detta känns dock väldigt svårarbetat, eftersom skillnaderna mellan algoritmerna troligen blir väldigt små och är antagligen inget en person kan upptäcka endast via betraktning.

3.1.1 Tidsstämplar & Visual Studio Profiler

Tidsstämplar är ett enkelt men effektivt sätt att testa hur mycket processortid som går åt för olika uppgifter. Praktiskt sett handlar detta bara om att placera ut tidsräknare med hjälp av start- och stopmetoder. Dessa placeras runt t.ex. ett funktionsanrop, och kommer med en precision på millisekunder kunna tala om ungefär hur lång tid den tar att utföra. För att testa den genomsnittliga tiden det tar för respektive algoritm att utföra en sökning kommer en

tidsstämpel användas som lagrar denna tid vid varje sökning. Detta värde kommer sedan användas för att skapa diagram för vardera algoritm, så de enkelt kan jämföras.

Visual Studio Profiler är ett verktyg för att hitta och mäta resursanvändning av individuella delar i ett system. Detta verktyg kan användas för att enkelt lokalisera s.k. flaskhalsar i systemet, och kommer i det här arbetet användas för att få en övergripande bild av resursåtgången för sökningen och möjligvis för att visa oönskade skillnader mellan testerna för respektive algoritm.

3.1.2 Analys av lagrad data

Den data som lagras under körning kommer vid avslut sparas i Microsoft Excel-dokument för bra överblick och smidig överföring till diagram. För varje mål skapas ett nytt excel-dokument med all data för båda algoritmerna. Som beskrevs ovan så kommer dessa data vara de åtgärder som planerats, d.v.s. vilka sekvenser med åtgärder som krävs för att uppnå ett mål, söktiden samt antal expanderade noder. Genom att analysera dessa, och jämföra mellan de två algoritmerna, går det att på så sätt tydligt (för denna implementationen) se vilken algoritm som är den effektivare, samt om åtgärder för samma mål skiljer sig mellan algoritmerna.

4 Implementation

Detta kapitel kommer beskriva hur programmet designades och implementerades. Upplägget baseras på den ordning komponenter implementerades i under utvecklingen av programmet. Detta skedde stegvis genom att först implementera en sökmotor kapabel att skifta mellan A* och IDA* under körning (vilka under utvecklingen testades för vägplanering) följt av implementationen av GOAP. Inledningsvis kommer dock först en genomgång av tidigare arbeten som implementationen baseras på.

Flera designbeslut gjordes under implementationen, och dessa kommer beskrivas i detalj.

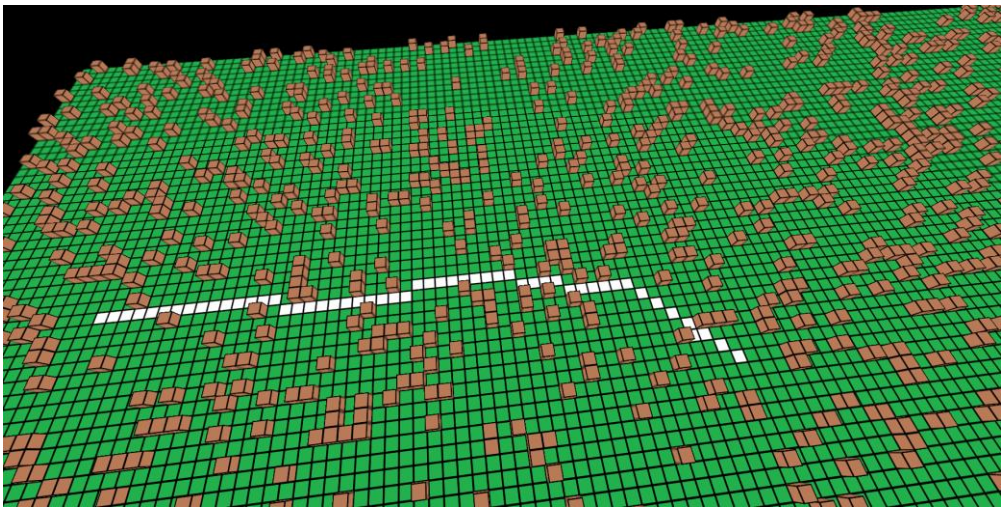
4.1 Förstudie

Då F.E.A.R. var det första spelet att utnyttja planering för att uppnå mål, samt att källkoden finns tillgänglig på internet (Monolith Productions, 2005), så kändes det naturligt att använda denna som referens under utvecklingen av implementationen. Jeff Orkin (Orkin, 2012), ensam AI-programmerare på Monolith Productions under utvecklingen av bl.a. F.E.A.R., har även skrivit ett flertal artiklar i ämnet (Orkin, 2003, 2004 & 2006) som underlättade processen. Utöver dessa fanns det tyvärr inte mycket konkret information om just implementeringen av GOAP, så en del ordväxling via mail med Jeff gav svar på frågor och problem som uppstod.

Implementeringen av sökalgoritmerna A* och IDA* är utvecklade framförallt utefter beskrivningarna givna av Millington m.fl. (2009) samt Higgins (2002a & 2002b), men en del designbeslut gjordes för att anpassa dem bättre för den aktuella problemdomänen. Exakt hur algoritmerna är implementerade kommer inte tas upp då de beskrivs väl i tidigare delar av rapporten (kapitel 2.5 & 2.6), men viktiga detaljer om hur dessa används för önskad sökrymd beskrivs i detalj i kapitel 4.2 och 4.3.3.

4.2 Sökmotorn

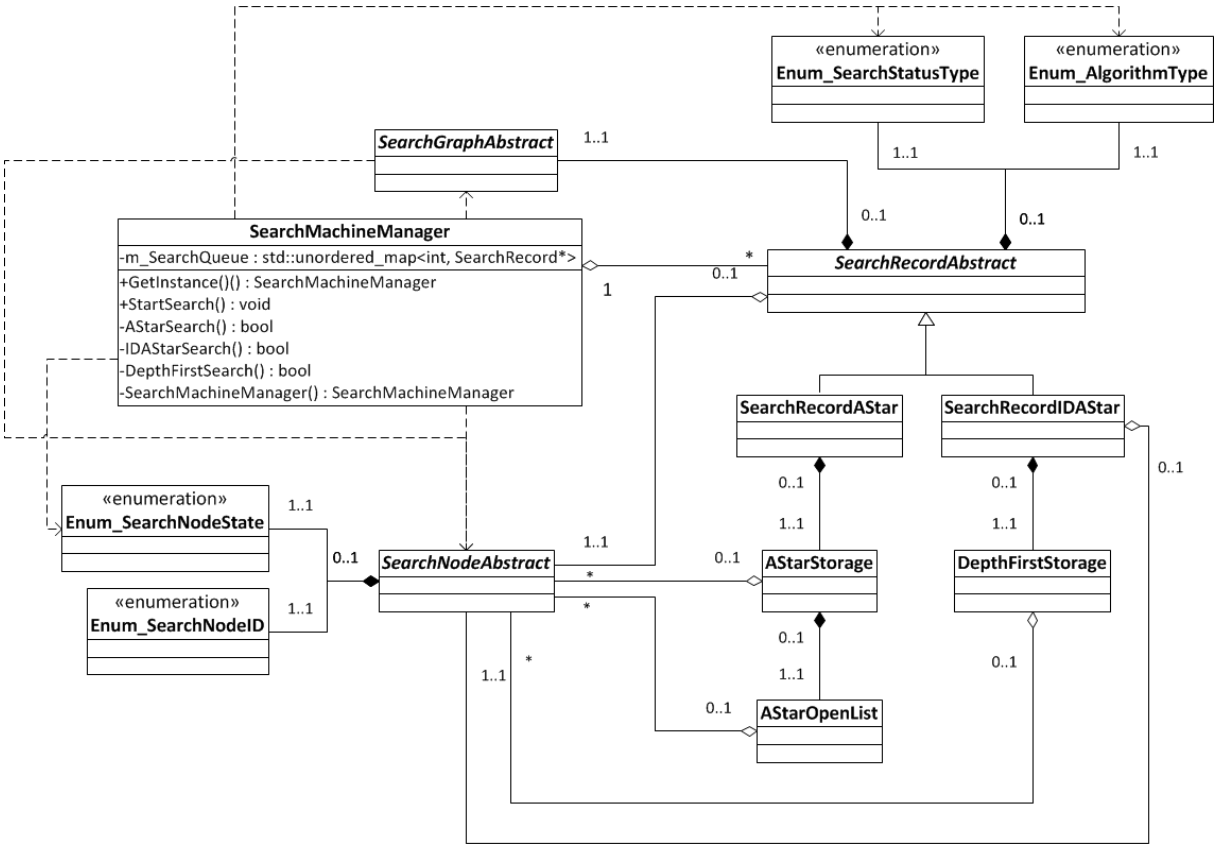
Sökmotorn är en vital del och implementerades därför först under detta projekt. För att underlätta testningen gjordes den kapabel att välja mellan algoritmerna A* och IDA* under körning av programmet (ej under en pågående sökning). En klass för vägplanering skapades också som både kunde användas vid testning av algoritmerna samt för att möjliggöra agenter som kan navigera sig i världen. Detta delkapitel ger först en överblick av designen bakom sökmotorn, samt beskriver de enskilda abstrakta klasser som används. Hur väghanteraren fungerar är utanför denna rapports tillämpningsområde, men exempel på hur en graf kan representeras finns med som referens.



Figur 6 Ett exempel på en väg i rutnätet hittad av A*.

Figur 6 visar en färdig väg mellan två rutor på rutnätet som gjordes under testningen av sökalgoritmerna för vägplanering, och detta används för att möjliggöra navigering av agenterna i världen när de utför olika åtgärder (t.ex. hämta vapen, hämta ammunition etc.). Gröna rutor är s.k. ”walkable” (sv. gångbara), d.v.s. rutor som agenter kan röra sig över, och de beigea är blockerade (ej gångbara), vilket syns i exemplet där vägen går sidan om blockerade rutor. Världen genereras procedurellt vid uppstart, och under testningen av algoritmerna slumpades beigea/blockerade kuber ut för att se att A* och IDA* verkligen fungerade. Inför det riktiga experimentet kommer banorna skapas enligt samma villkor, så de inför varje testkörning är identiska.

Figur 7 visar en överblick av strukturen för sökmotorn, exklusive GOAP- och vägplanerings-specifika klasser.



Figur 7 Klassdiagram över sökmotorn.

Som syns i diagrammet så är det singleton-klassen SearchMachineManager som är gränssnittet utåt för de komponenter som önskar göra en sökning (därav listas dess medlemmar). För varje sök-begäran som göras (via StartSearch()-funktionen) så köas ett nytt sökregister, vilket är av någon av de två typerna SearchRecordAStar eller SearchRecordIDAStar beroende på vilken algoritm som ska köras. Dessa två ärver och lagras som den abstrakta basklassen SearchRecordAbstract för att abstrahera bort allt specifikt. Dessa register används för att möjliggöra ett kösystem så flera sökningar kan dela på exekveringstid om de överskrider en angiven tidsgräns per bilduppdatering, främst för att underlätta felsökning då övriga delar av systemet således också får tid att uppdateras. När en specifik sökning hämtas från kön och exekveras (vilken algoritm det gäller avgörs via en uppräkningstyp i sökregistret) så kan sökregistret säkert typ-omvandlas till rätt deriverad klass.

På samma sätt så vet sökmotorn endast om existensen av SearchNodeAbstract och SearchGraphAbstract, men under ytan ärvs dessa beroende på vilken sökrymd det gäller. För vägplanering, t.ex., så skapas en ny graf-klass SearchPathGraph som håller ett rutnät över de noder som besöks, samt korrekt implementation av de abstrakta metoder ärvda från SearchGraphAbstract. Noderna är i sin tur också specifika för vägplanering, vilket är klassen SearchPathNode som ärver ifrån SearchNodeAbstract. SearchPathNode lagrar en 3D-vektor som den position noden är associerad med, vilken extraheras när sökningen är klar. För varje ny nod som besöks allokeras och lagras denna i en hashtabell med en nyckel som specificeras via en ärvd metod. När en ny nod ska hämtas kontrolleras först om denna redan är allokerad och återanvänds i så fall, annars skapas en ny. Detta görs för att undvika onödig minnesallokering.

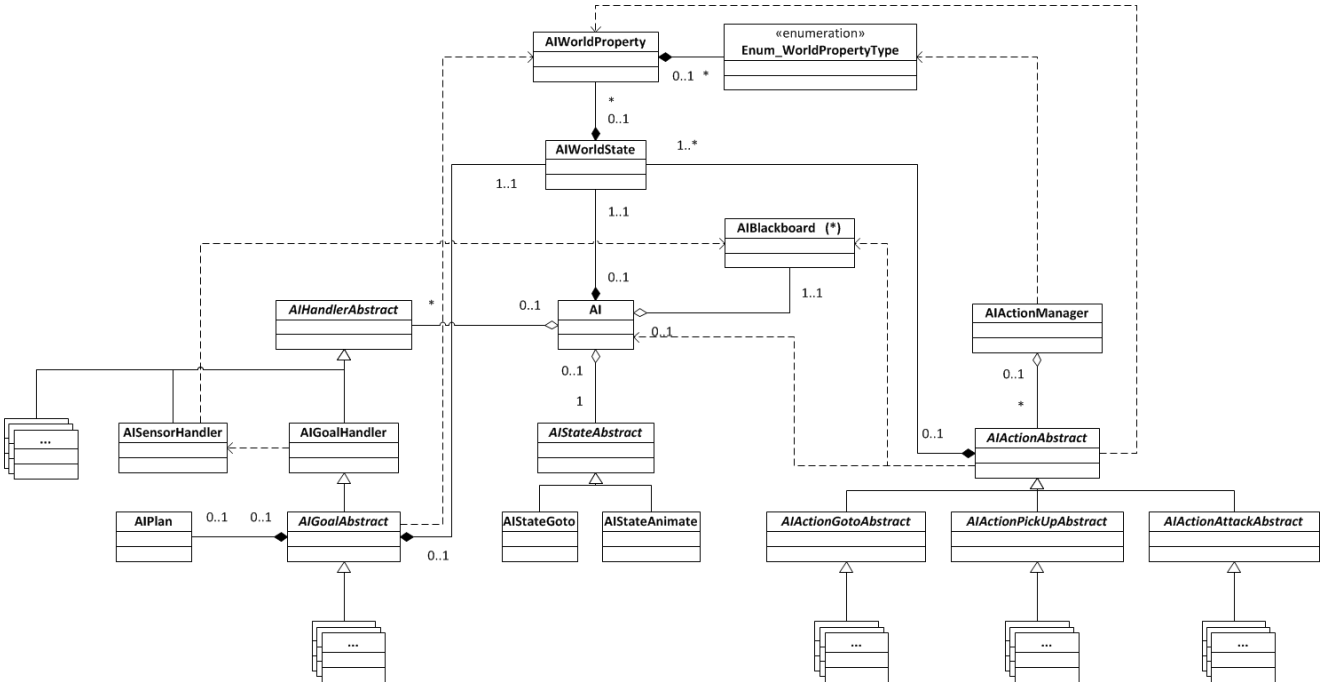
Detta är grundprincipen för strukturen bakom sökmotorn, vilken då tillåter sökning i godtyckliga sökrymder, med kravet att en graf- och nod-klass skapas som tillhandahåller specifik implementation av de metoder som specificeras av respektive basklass (t.ex. när tillgängliga kopplingar till en nod hämtas).

4.3 GOAP

4.3.1 Överblick AI

AI-arkitekturen i detta projekt bygger på den beskriven av Orkin (2003, 2004 & 2006) och består i denna implementation av följande komponenter:

- AI – Central klass för alla agenter
- AIWorldProperty – Representerar ett par med en tillståndstyp och associerat värde, t.ex. FiendeDöd = Sant.
- AIWorldState – En mängd med tillstånd (AIWorldProperty). Varje agent har ett AIWorldState som representerar det aktuella världstillståndet.
- AISensorHandler – Hanterare för alla sensorer.
- AISensorAbstract – Basklass för alla tillgängliga sensorer, t.ex. ”SeFiende och SeFöremål”.
- AIGoalHandler – Hanterare för alla mål.
- AIGoalAbstract – Basklass för alla tillgängliga mål, t.ex. ”DödaFiende” och ”Patrullera”.
- AIStateAbstract – Basklass för de tillgängliga tillstånden Goto och Animate som anges när en åtgärd aktiveras (inte att missuppfatta som världstillstånd).
- AIActionAbstract – Basklass för alla tillgängliga åtgärder, t.ex. Attackera, ladda vapen, gå till position osv.
- AIActionManager – Singleton-klass som håller alla tillgängliga åtgärder. Används under sökning för att hitta åtgärder som löser ett tillstånd som vill uppnås.
- AIBlackboard – Ett abstraktionslager mellan AI-klassen och olika delar av systemet för att minska beroenden. Internt fungerar den som en ”Getter- & Setter”-klass, där sensorer sätter data och t.ex. mål hämtar data (vid uträkning av målets relevans).



Figur 8 Klassdiagram över AI-arkitekturen.

Som syns i klassdiagrammet (figur 8) så centrerar arkitekturen kring klassen AI. Det är denna komponent som ”är” varje agent och lagrar bl.a. agentens ID, det aktuella världstillståndet, positionen, riktningen, tillståndet (gå eller animera), en s.k. blackboard, samt sköter diverse hanterare för t.ex. rörelse, mål, sensorer, ”förråd” (eng. inventory).

4.3.2 Huvudkomponenter

AIWorldProperty är, som beskrevs kort ovan, den klass som representerar ett enstaka tillstånd genom en symbol (Orkin, 2004), t.ex. ”Fiende död = Sant”. Detta implementeras med en nyckel av uppräkningstyp (typ av tillstånd, t.ex. ”Fiende död”) som paras med ett booleskt värde (sant eller falskt).

Följande symboler (nycklar) är implementerade i detta projekt:

```
enum Enum_AIWorldPropertyType{
            WSPK_InvalidKey = -1,
            WSPK_AtPositionRandom,
            WSPK_AtPositionCover,
            WSPK_AtPositionMedpack,
            WSPK_AtPositionAmmo,
            WSPK_AtPositionWeapon,
            WSPK_AtPositionWeaponMounted,
            WSPK_AtPositionTarget,
            WSPK_Healed,
            WSPK_Idling,
            WSPK_Dead,
            WSPK_TargetIsDead,
            WSPK_WeaponArmed,
            WSPK_WeaponMounted,
            WSPK_WeaponDrawn,
            WSPK_WeaponLoaded,
            WSPK_WeaponHasAmmo,
            WSPK_UsingObject,
```

```
		//Count always last
		WSPK_Count
};
```

WSPK står för ”Worldstate Property Key” och dessa uppräkningsvärden används sedan i klassen *AIWorldState* för att enkelt hitta önskat enskilt världstillståndspar.

AIWorldState, d.v.s. det ”kompletta världstillståndet”, är den klass som håller alla enskilda tillstånd för en agent. Detta görs med hjälp av *unordered_map*, en hashtabell tillgänglig i *Standard Template Library* (STL), ett bibliotek tillgängligt i C++. Att använda en hashtabell är till stor fördel framförallt under planeringen, då det snabbt går att kontrollera om ett tillstånd är samma i det aktuella världstillståndet och målets världstillstånd, och på så sätt avgöra vilka nya åtgärder som kan behövas. Unordered_map tillåter även endast ett element per hashnyckel, vilket medför att dubbletter av tillstånd aldrig kan existera.

AIActionAbstract är den basklass som deriveras av alla åtgärder. Varje åtgärd implementeras i kort av förhandsvillkor (eng. *preconditions*), resultat (eng. *Effect*), en aktiverings- och deaktiveringsfunktion, funktioner för att applicera förhandsvillkor och resultat på andra världstillstånd samt sammanhangs-förhandsvillkor (eng. *Context Preconditions*). Aktivering och deaktiveringen står för den logiken som sköter övergången till ett nytt tillstånd (Animate eller GoTo) samt uppdatering av data (t.ex. addera hälsa till agenten om detta plockades upp).

Följande är ett kodexempel på hur aktiveringen, deaktiveringen och förhandsvillkorskontrollen för åtgärden ”ReloadWeapon” ser ut:

```
void AIActionReloadWeapon::ActivateAction( AI* p_AI )
{
		//Set state to animate reload animation
		p_AI->SetState(new AIStateAnimate(p_AI, this, Animation_Reload));
}
void AIActionReloadWeapon::DeactivateAction( AI* p_AI )
{
		//Reload weapon
		p_AI->GetWeaponHandler().GetDrawnWeapon()->ReloadWeapon();
}
bool AIActionReloadWeapon::ValidateContextPreconditions(AI* p_AI, const bool&
p_IsPlanning)
{
		if(p_IsPlanning == true)
			return true;

		return p_AI->GetWeaponHandler().IsWeaponDrawn();
}
```

Förhandsvillkoret behöver i fallet med åtgärden ”ReloadWeapon” inte kontrolleras under sökning eftersom åtgärden för att få vapnet draget, ”DrawWeapon”, kan vara en del av planen och är då ännu inte uppfyllt för agenten.

AIActionManager, som namnet antyder, hanterar alla åtgärder (eng. *Actions*). Eftersom åtgärder är delade, behöver de endast allokeras en gång vardera, och kan sedan hämtas via denna klass. Detta görs via en hashtabell, eller *unordered_multimap* i STL. Skillnaden mellan *unordered_map* och *unordered_multimap* är att den senare tillåter flera element per hashnyckel. Detta används då åtgärder sätts in med dess resultat (d.v.s. det tillstånd som åtgärden uppfyller) som hashnyckel. Med en *unordered_multimap* går det således att ha flera åtgärder som uppfyller samma tillstånd placerade under samma hashnyckel, för att

under sökning snabbt komma åt alla dessa. Ett exempel på detta är åtgärderna ”AttackMelee” och ”AttackRanged”, som båda uppfyller världstillståndet ”TargetIsDead”.

AISensorHandler och *AIGoalHandler* är båda s.k. hanterare (eng. *Handlers*). Hanterare är komponenter som sköter underhållning av olika delar hos varje enskild agent, t.ex. rörelsehanterare, förrådshanterare (eng. *Inventory handler*) samt de två nämnda ovan.

AISensorHandler, som namnet antyder, sköter uppdatering av alla de sensorer som en agent har tillgängliga. En sensor implementerad är bl.a. ”SeeEnemy” (sv. SeFiende). Under uppdatering kommer denna sensor kontrollera om det finns några fiender i närheten, och i så fall spara en av dem som ”CurrentTarget” (sv. AktuellMåltavla). Detta görs i agentens s.k. *AIBlackboard* (vilken kan ses som en anslagstavla).

AIGoalHandler sköter uppdateringen av alla mål som finns tillgängliga för tillhörande agent. Uppdatering av mål sker dock inte på samma sätt som för sensorer, utan endast det mål som är aktiverat uppdateras och fortsätter på dess plan (eller planerar ny om ingen plan existerar än). För att avgöra vilket mål som ska aktiveras beräknas ett s.k. ”relevans-värde” till varje mål. Detta relevansvärde beräknas ofta utefter vad en särskild sensor detekterat, t.ex. om en fiende är sedd. Detta medför att flera av de implementerade målen kräver att en särskild sensor existerar för den aktuella agenten, vilken ser till att uppdatera rätt data på ”anslagstavlan” som målet sedan använder för att räkna ut sin egen relevans.

Följande är ett kodexempel på hur relevansen för målet ”KillEnemy” (sv. DödaFiende) beräknas:

```
void AIGoalKillEnemy::CalculateRelevance()
{
            if(m_SkipNextTime == true){
                        m_SkipNextTime = false;
                        return;
            }
            //If there is a valid target (who is not dead!), this goal is highly
relevant
            if(m_AI->GetBlackboard()->HasValidTarget() && m_AI->GetBlackboard()-
>GetCurrectTarget()->GetDamageController().IsDestroyed() == false)
                        m_GoalRelevance = 1.1f;
            else
                        m_GoalRelevance = 0.f;
}
```

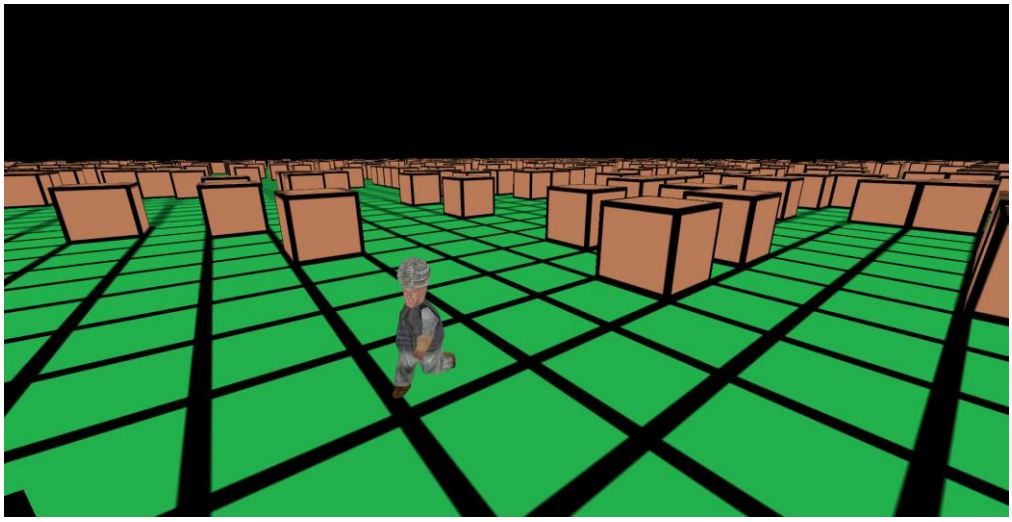
Detta kodstycke kollar om en fiende har registrerats på anslagstavlan (eng. *Blackboard*) av sensorn ”SeFiende”, samt om fienden i fråga inte redan är död. Relevansvärdet är manuellt balanserat för att föredra vissa mål framför andra om flera är aktiveringsklara samtidigt. Genom att relevansvärdena är, som syns i koden ovan, fasta värden (0 eller 1.1 i detta fallet) gör detta denna prioritering lätt att kontrollera.

4.3.3 Planera åtgärder

För att koppla ihop allt detta med sökmotorn så skapas en ny graf, AISearchStateGraph, och nod, AISearchStateNode, som deriveras ifrån AISearchGraphAbstract och AISearchNodeAbstract respektive. Grafen implementerar sedan bl.a. metoden GetNeighbors, som har till uppgift att returnera tillgängliga grannar till en nod som sökmotorn skickar med som argument.

När åtgärder planeras så håller varje tillståndsnod ett aktuellt tillstånd och ett måltillstånd. Det aktuella tillståndet är vid första noden, där sökningen börjar, en kopia på agentens

aktuella tillstånd, och måltillståndet är det tillstånd ett mål kräver för att uppnås. För att hitta eventuella grannar till en tillståndsnod så jämförs dess aktuella tillstånd och måltillståndet för att avgöra vilka symboler (AIWorldProperty) som ej är uppfylla, och i så fall, via AIActionManager, hittar möjliga åtgärder som uppnår detta. För varje ny åtgärd som hittas så skapas en ny nod där det aktuella tillståndet nu är föregående tillstånd, plus de symboler som åtgärden lovar att uppfylla. Det nya målet för denna nod blir nu föregående mål, plus de förhandsvillkor som åtgärden kräver för att kunna utföras. Ett exempel är åtgärden ”AttackRanged”, som lovar att uppfylla symbolen ”EnemyDead”. För att åtgärden ska kunna utföras krävs det dock att symbolerna ”WeaponDrawn” och ”WeaponLoaded” är sanna. När dessa noder utvärderas utförs samma process, men med respektive aktuella tillstånd och måltillstånd.



Figur 9 Ett exempel på en agent som utför en åtgärd (gå till vapen).

Figur 9 visar ett exempel på en agent som följer en aktuell plan för att uppnå målet ”fiende död”, vilken här har genererats med hjälp av IDA*. Planen i detta fall är precis som i figur 4, med skillnaden att agenten inte hade något vapen tillgängligt och hittade därför ytterligare en åtgärd ”ta upp vapen”, som i sin tur krävde att agenten befinner sig vid en position där ett vapen existerar. Detta löses därefter genom åtgärden ”hämta vapen”, vilken inte har några förhandsvillkor.

4.4 Experimentmiljö

Som tidigare beskrevs i metodbeskrivningen (kapitel 3.1) så är experimentmiljön uppbyggd i en 3D-värld. Detta görs med hjälp av grafikmotorn Irrlicht (2012), som renderar ut kuber i varje ruta på rutnätet vilket i sin tur används till vägplaneringen (Figur 6). Under tidigare delar av utvecklingen var beslutet att endast simulera i 2D för att på så sätt kunna fokusera mer tid på implementationen av GOAP, men ganska tidigt blev det uppenbart att en 2D-simulering inte är lika naturlig att relatera till dagens AAA-spel och framförallt övertygande nog för att motivera GOAP. En mer krävande grafikmotor möjliggör även testning där GOAP delar exekveringstid med ett mer krävande system, och därför föll valet på att använda 3D-motorn Irrlicht. Detta var både på gott och ont. Det var under senare delar lättare att se fördelarna med GOAP, då den visuella responsen av vilka åtgärder som utfördes blev mer tydlig. Däremot så ökade även utvecklingstiden, och det färdiga programmet blev vid deadline inte lika långt kommet som det först hoppats på. Detta medförde att antalet mål och åtgärder blev färre än först önskat, men förhindrar inte testningen.

Testningen består av X antal agenter (antalet beror på vilket test som körs) som är uppdelade i två lag, med lika många agenter i vardera. De har alla ha samma mål så som ”döda fiende”, ”ta skydd”, ”patrullera” m.fl. samt alla nödvändiga åtgärder så det fungerar likt ett vanligt ”first person-shooter”-spel (bortsett från att ”spelaren” endast kommer styra en flygande kamera). Den data som lagras är sekvenser med åtgärder (s.k. planer) tillsammans med det mål som planen genererats åt, samt ett medelvärde för planeringstiden vilket används som prestandavärde (totala planeringstiden/antalet planer genererade). För att verkligen stresstesta sökalgoritmerna kördes även ett test utan rendering och simulering för att inte begränsas av grafikmotorn eller andra resurskrävande komponenter, samt slumpen när agenter patrullerar.

Det första testet fokuserar på skillnader i de genererade planerna och därav främst med ett lägre antal agenter (<10st) för att se vilka åtgärder som föredras, men också ett med högre antal (>80st) för att hitta maxgränsen för denna implementation. Ett tredje test består av att planer genereras för alla tillgängliga mål i flera iterationer, utan att ”riktiga” agenter faktiskt simuleras. Anledningen till att inte enbart testa GOAP som i det tredje testet (d.v.s. utan rendering och simulering) är för att också se hur effektivt det kan vara i ett tänkt spel, där exekveringstiden är delad med andra komponenter. Det är därför också rimligt att även ha med statistik över hur många GOAP-agenter som max är möjligt att simulera samtidigt i ett tänkt spel (med den aktuella implementationen som referens). Detta eftersom valet att använda GOAP i ett spelprojekt ofta bero på spelets genre. T.ex. om det kan vara värt att överväga för storskaliga krigsspel med fler än hundra agenter som måste simuleras samtidigt.

5 Utvärdering

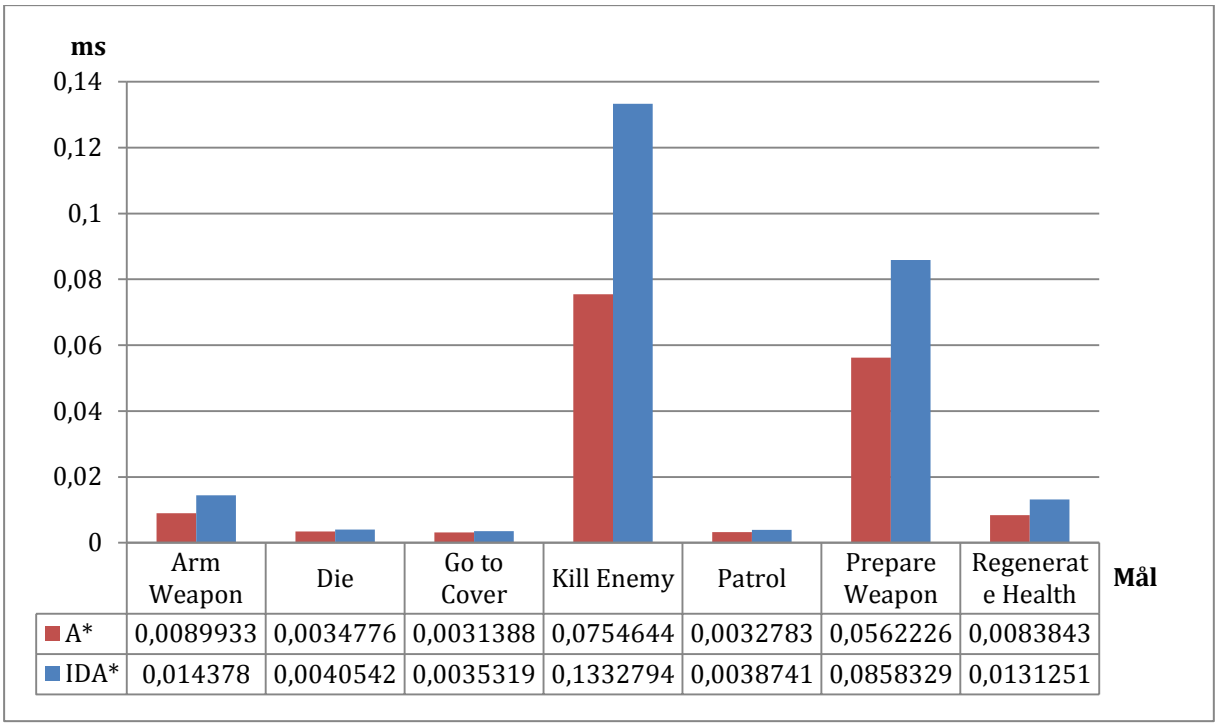
Syftet med detta arbete är att jämföra effektiviteten mellan två olika sökalgoritmer vid planering av åtgärder i GOAP (Goal-Oriented Action Planning). Detta kapitel kommer presentera de resultat som samlats under utförandet av experimentet, samt analyser kring hur dessa uppstod. Upplägget baseras på de delmoment som beskrevs i Problemformuleringen (kapitel 3).

För att uppnå delmål 2 (kapitel 3) utvärderas resultaten genom statistikdata med hjälp av diagram skapade i Microsoft Excel 2010. För att åstadkomma ett rättvist resultat kördes samtliga sökningar för de implementerade målen i tiotusen (10000) iterationer var, där ett medelvärde av planeringstid, antal besökta noder samt planeringstid genom antal besökta noder (planeringstid / besökta noder) för varje mål och algoritm sparades (figur 10, 11 & 12). De tidsstämplar som användes isolerades även till respektive sökalgoritm för att inte påverkas av andra komponenter (t.ex. kösystemet, kapitel 4.2). Samtliga mål planerades utifrån ett tomt världstillstånd (t.ex. inget vapen tillgängligt osv.). Figur 13 visar resultat efter test för max antal agenter simulerade åt gången för vardera algoritm. Även Profileraren i Visual Studio 2010 användes för att lokalisera fördelning av exekveringstiden för vardera algoritm (figur 14 & 15). För att upptäcka eventuella skillnader i de genererade planerna sparades även mål med genererade åtgärder i en lista (figur 16).

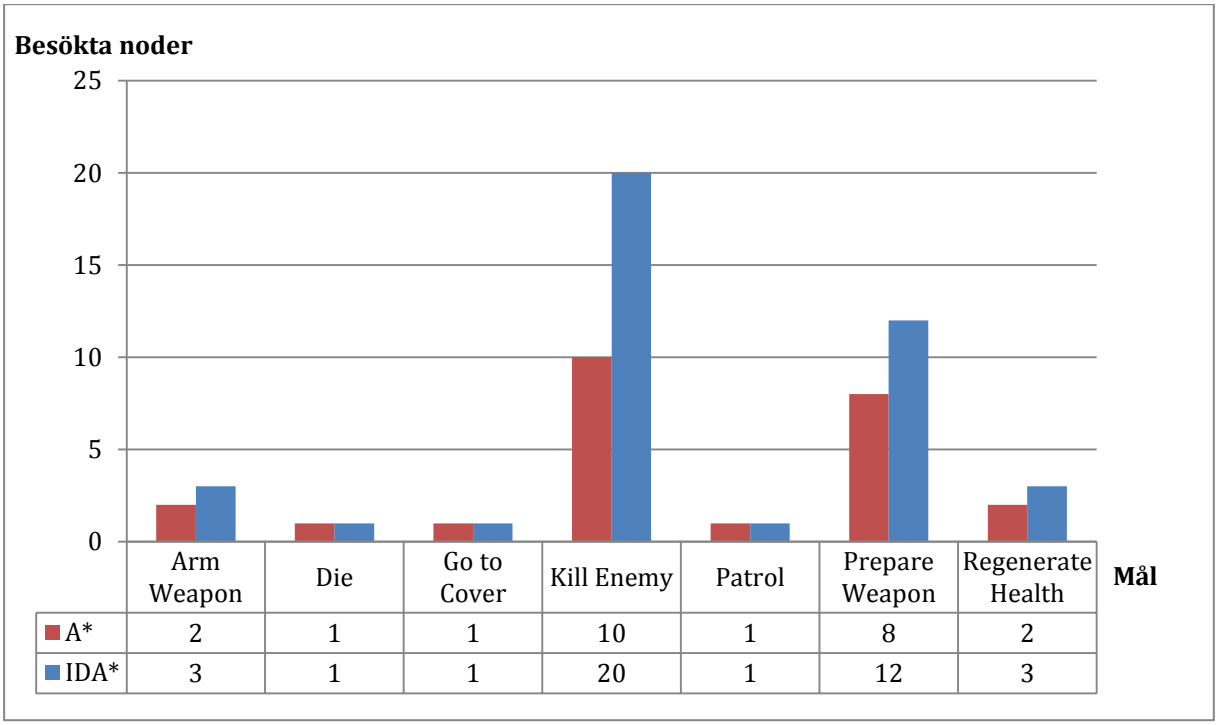
För att undvika avvikelser i testkörningarna var samtliga test automatiserade och exekverades direkt vid uppstart, för att sedan avsluta hela programmet då testet blev färdigt.

5.1 Resultat

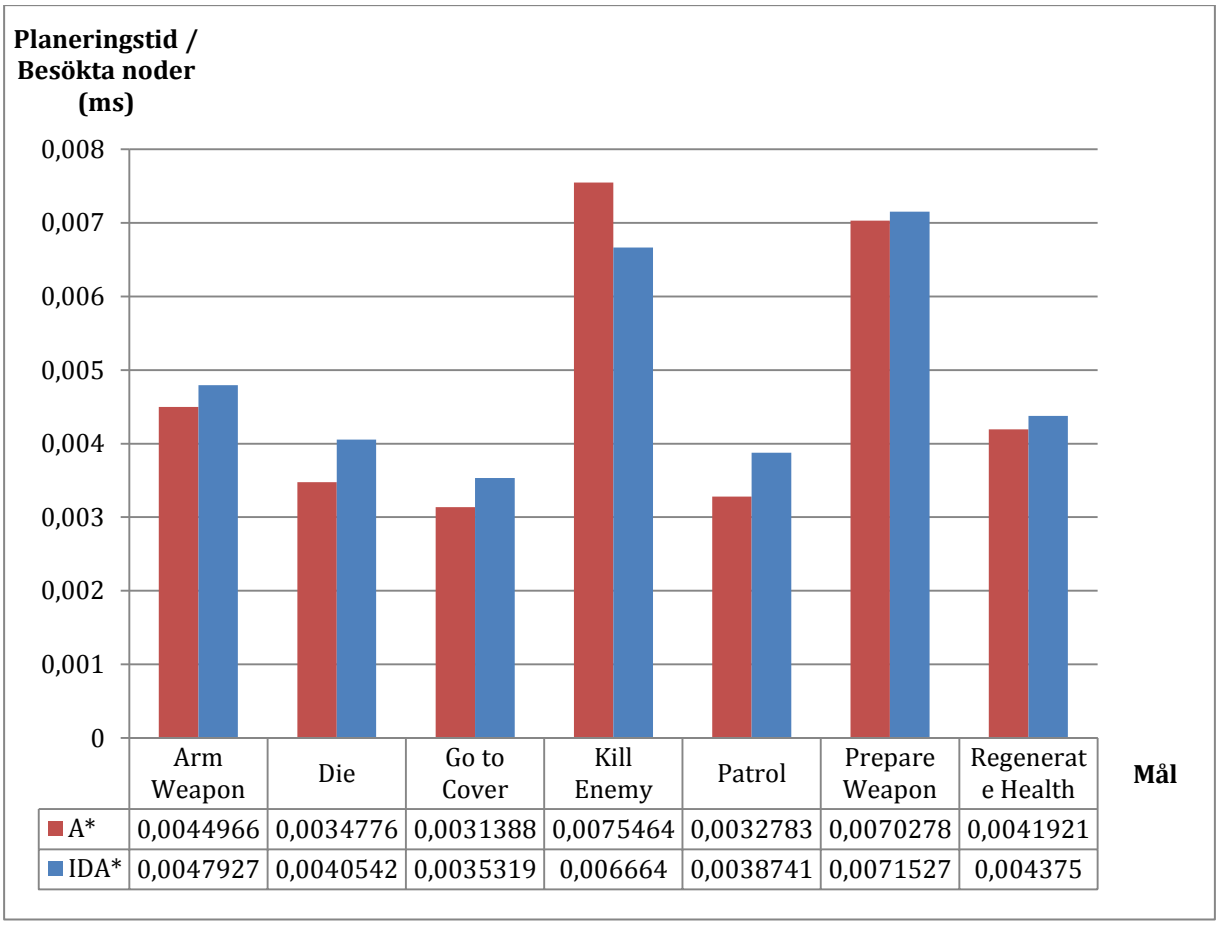
Nedan presenteras först de övergripande resultaten, följt av en sammanställning och analys av respektive figur.



Figur 10 Planeringstiden för respektive mål och algoritm i snitt över tiotusen iterationer. Lägre tid är bättre.

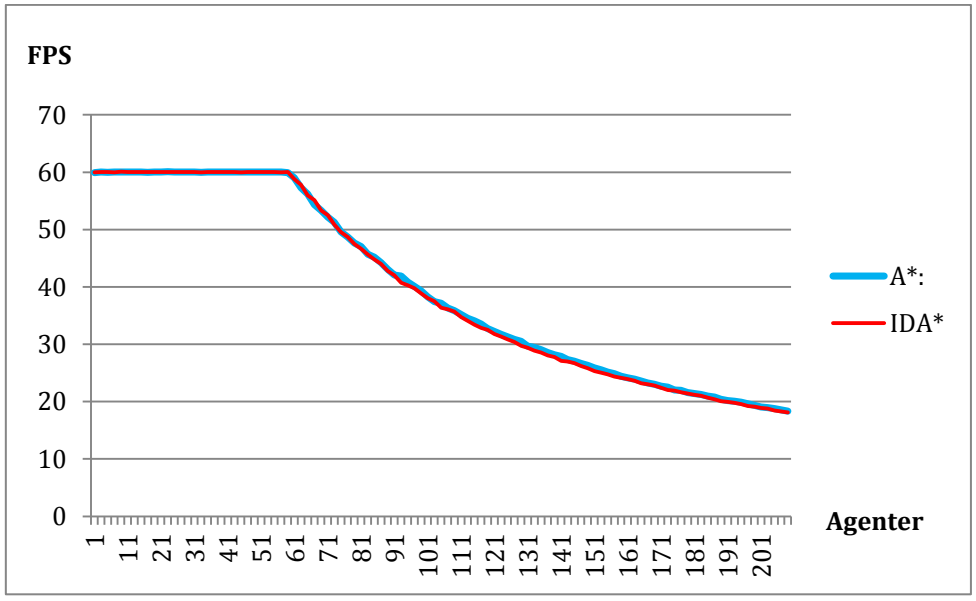


Figur 11 Antalet besökta noder för vardera mål och algoritm.

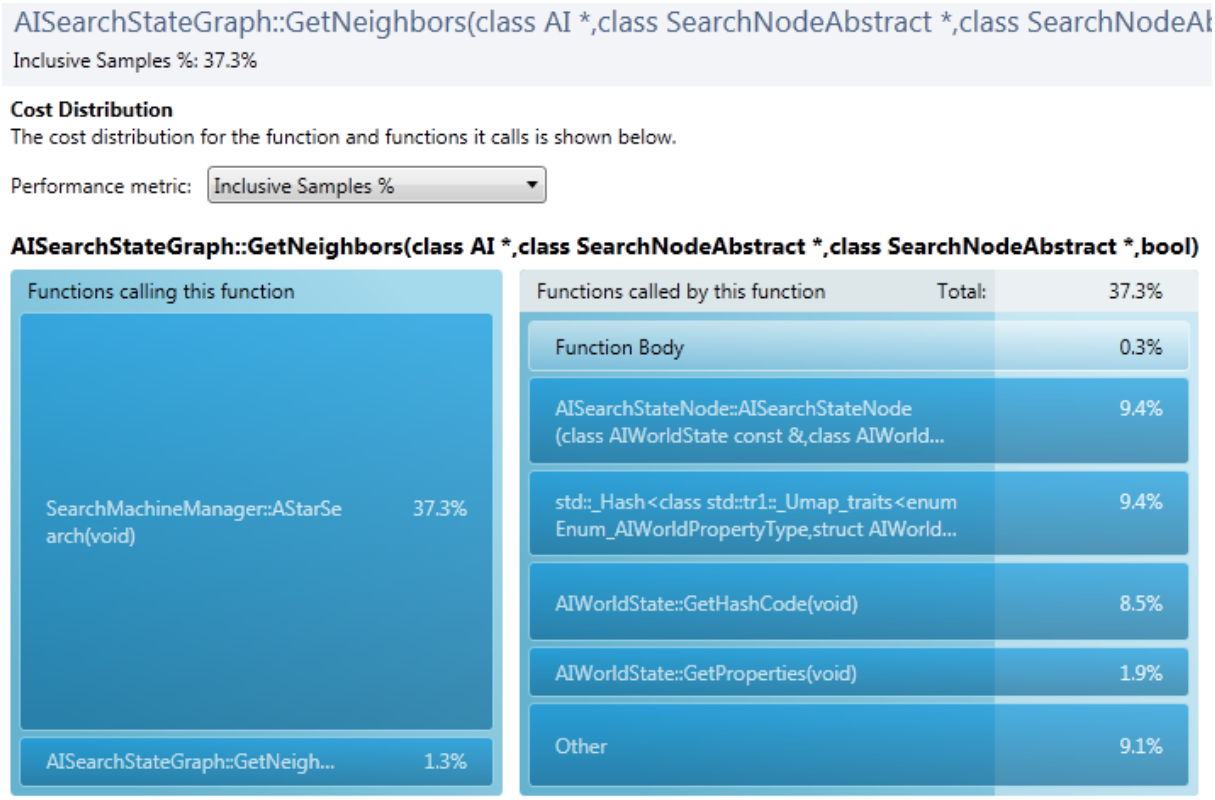


	Arm Weapon	Die	Go to Cover	Kill Enemy	Patrol	Prepare Weapon	Regenerate Health
A*	0,0044966	0,0034776	0,0031388	0,0075464	0,0032783	0,0070278	0,0041921
IDA*	0,0047927	0,0040542	0,0035319	0,006664	0,0038741	0,0071527	0,004375

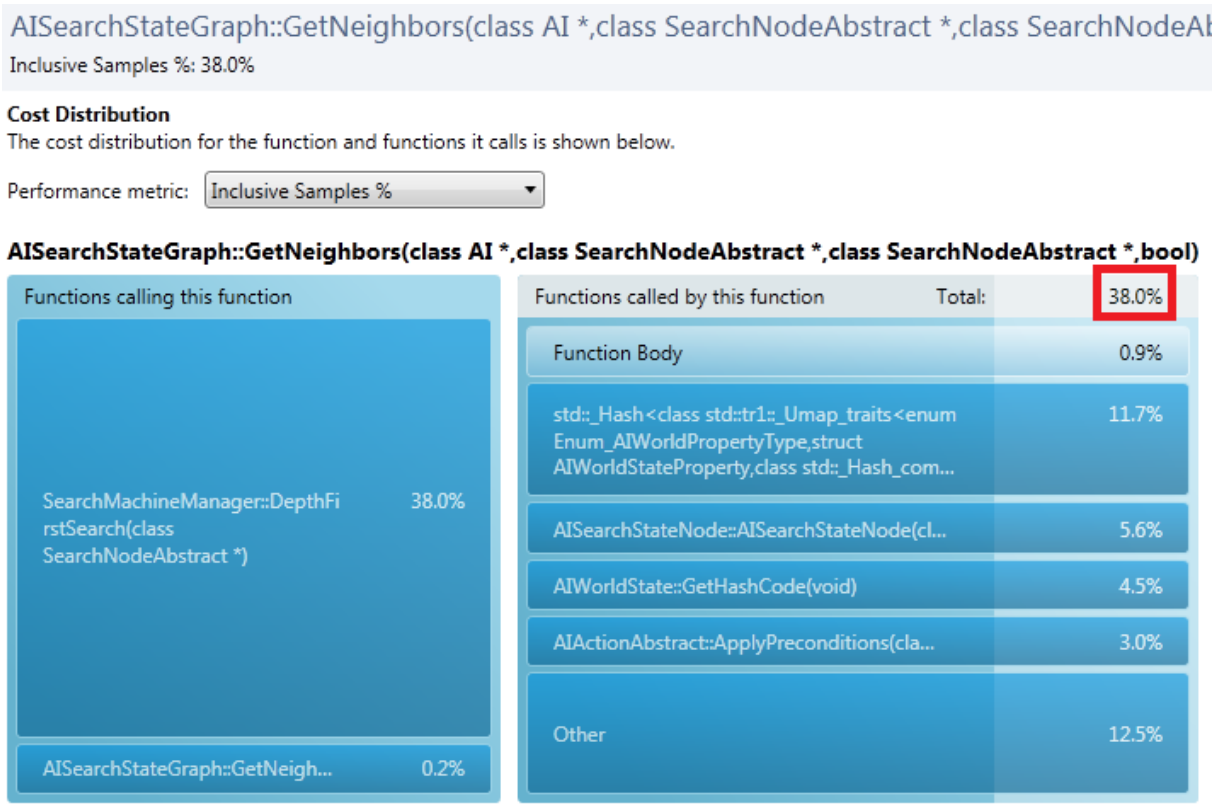
Figur 12 Planeringstiden per besökt nod för vardera mål och algoritm.



Figur 13 Antal bilduppdateringar per sekund per antal agenter simulerade.



Figur 14 Sammanfattning skapad av profileraren i Visual Studio 2010 för A*.



Figur 15 Sammanfattning skapad av profileraren i Visual Studio 2010 för IDA*.

A*	**Goal:**	Arm Weapon
	Plan:	*->Go to Weapon ->Pick Up Weapon*
IDA*	**Goal:**	Arm Weapon
	Plan:	*->Go to Weapon ->Pick Up Weapon*
	Goal:	Die
	Plan:	*->Die*
	Goal:	Die
	Plan:	*->Die*
	Goal:	Go to Cover
	Plan:	*->Go to Target*
	Goal:	Go to Cover
	Plan:	*->Go to Target*
	Goal:	Kill Enemy
	Plan:	*->Go to Weapon ->Pick Up Weapon ->Go to Ammo ->Draw Weapon ->Pick Up Ammo ->Reload Weapon ->Attack Ranged*
	Goal:	Kill Enemy
	Plan:	*->Go to Weapon ->Pick Up Weapon ->Go to Ammo ->Draw Weapon ->Pick Up Ammo ->Reload Weapon ->Attack Ranged*
	Goal:	Patrol
	Plan:	*->Go to Random*
	Goal:	Patrol
	Plan:	*->Go to Random*
	Goal:	Prepare Weapon
	Plan:	*->Go to Weapon ->Pick Up Weapon ->Go to Ammo ->Draw Weapon ->Pick Up Ammo ->Reload Weapon*
	Goal:	Prepare Weapon
	Plan:	*->Go to Weapon ->Pick Up Weapon ->Go to Ammo ->Draw Weapon ->Pick Up Ammo ->Reload Weapon*
	Goal:	Regenerate Health
	Plan:	*->Go to Medpack ->Pick Up Medpack*
	Goal:	Regenerate Health
	Plan:	*->Go to Medpack ->Pick Up Medpack*

Figur 16 Lista över alla genererade planer för respektive mål och algoritm.

5.2 Analys

Figur 10, vilket är det mest konkreta och explicita resultatet, visar hur planering av åtgärder i GOAP med IDA* hamnar en bit efter A* sett till ren exekveringstid. Den procentuella skillnaden ger att A* i snitt endast tar 62 % så lång tid att planera med jämfört med IDA*. Vid en vidare titt på figur 11, syns antalet noder som besökts (d.v.s. hämtats med funktionen *GetNeighbors*, kapitel 4.3.3) under planering. Denna är väldigt snarlik figur 10 av en anledning, då den stora nackdelen med IDA* är att den kan behöva återbesöker samma noder flera gånger. Detta löses till viss del genom att utnyttja en hashtabell, där besökta noder lagras med en nyckel genererad från dess världstillstånd (kapitel 2.6), men problemet återstår fortfarande att varje nod måste hämtas för att sedan kontrollera vilka som inte redan har besökts. Dock, som syns i figur 12, är det väldigt jämt mellan IDA* och A* sett till hur snabbt noder utvärderas för respektive algoritm, där IDA* för åtminstone ett av målen (*KillEnemy*) lyckas utprestera A*. Trots att A* tar övertaget i de nämnda resultaten är en viktig punkt att faktiskt testa i ett riktigt speltest. Figur 13 visar ett test där två agenter skapas var tionde sekund, och ett snittvärde från antalet bilduppdateringar per sekund (FPS) lagras i förhållande till antalet agenter simulerade. Som syns i diagrammet är det väldigt små, nästan obetydliga skillnader, mellan de två algoritmerna. För A* blir snitt-FPS:en 40.0 jämfört med IDA* som får en snitt-FPS på 39.8. Detta är en obetydlig skillnad, och kan förklaras genom de små skillnader de faktiskt handlar om mellan A* och IDA* i figur 10. Detta ger slutsatsen att GOAP har en förhållandevis liten påverkan på prestandan under riktiga förhållanden, och att förändringen i y-led snarare beror på renderaren och andra krävande uppgifter (t.ex. beräkning av agenters synfält). Att FPS:en ligger stadigt vid 60 och sedan sjunker drastiskt beror på att den är låst till denna maxgräns.

GetNeighbors är, enligt resultaten av profileraren (figur 14 & 15), den funktion som exekveras mest under planering, för båda algoritmerna. Detta är inte nödvändigtvis helt exakta resultat, då profileraren kortfattat endast gör stickprov under tiden programmet körs, men det är ett bra fingermått. *Inclusive Samples* säger, i procent, hur mycket av exekveringstiden som skett i den listade funktionen och alla dess inre funktioner. Som syns för IDA* jämfört med A* så ligger mer exekvering på den hashtabell som används i grafen för att hålla koll på tidigare noder (används endast i grafen, kapitel 4.2, och ska inte förväxlas med den hashtabell som används specifikt för IDA*). Detta beror på de upprepade sökningar som sker, där flera noder kommer besökas ett flertal gånger.

Figur 16 visar de färdiga planerna för vardera mål och algoritm, och som synes är resultaten identiska mellan de båda algoritmerna. Detta beror troligen till stor del på det begränsade antalet åtgärder som vid testtillfället fanns implementerade. Vid sökning i en större sökrymd, så som vid vägplaneringen vilket användes för att testa A* och IDA* under implementeringen, fanns där små men tydliga skillnader i de genererade vägarna (där båda var lika kostsamma).

6 Slutsatser

I detta kapitel presenteras en sammanfattning av implementationen (kapitel 4) samt en sammanfattning av de resultat som samlats under testningen (kapitel 5). Dessa diskuteras sedan i ett större sammanhang, t.ex. hur väl de skulle fungera i ett kommersiellt spel samt hur detta arbete kan utnyttjas för vidarearbetning av andra utvecklare.

6.1 Resultatsammanfattning

Implementeringen av GOAP (kapitel 4, vilket motsvarar det första delmålet), samt de testmiljöer som skapades för utvärderingen var en stadig men långsam process. Som syns i kapitel 4 (Implementationen) består dess struktur av ett flertal komponenter, varierande till storleken, som alla sedan kopplades samman. Detta gjorde att utvecklingstiden blev längre än väntat, och således blev sökrymden för respektive algoritm mindre än förutspått. Detta förhindrade dock inte testningen, men resultatet bör därav snarare ses som en antydan mer än ett konkret bevis över vilken algoritm som är den ”bästa”. A*, förutom att i detta test vara den snabbare, har även fördelen att vara en mer använd och därav mer dokumenterad algoritm. IDA* är en förhållandevis ny algoritm, så implementeringen av denna krävde mer tid till efterforskning, vilket är en viktig detalj att ha i åtanke om tidigare erfarenhet med denna saknas vid implementering.

Det andra delmålet, testningen, bestod i kort av tre delar: Prestandatest, olikhet i sökresultat samt analys av fördelning av exekveringstid. För att på ett rättvist sätt fastställa vilken algoritm som var den snabbare kördes planering för vardera mål i tiotusen iterationer (utan simulering av agenter), där ett snittvärde skrevs ut till ett Excel-dokument. Därifrån skapades sedan de diagram som syns i kapitel 5.1. Även ett prestandatest där agenter simulerades kördes, detta för att hitta en maxgräns av antal agenter för den aktuella implementationen. Målet var att hitta den algoritm som klarar flest antal agenter över 30 FPS (30 bilduppdateringar per sekund). Detta test visade att i verkliga spelsammanhang är skillnaden mer eller mindre obetydlig, då GOAP (i denna implementation) är en förhållandevis resurssnål arkitektur för båda algoritmerna. Detta beror delvis på det låga antalet åtgärder som fanns implementerade under testning, och hur en ökning av tillgängliga åtgärder påverkar resultaten är ett intressant område att vidareutveckla detta arbete på. För de båda algoritmerna uppgick antalet samtidigt simulerade agenter till ca 130 st. innan bilduppdateringsfrekvensen sjönk under 30.

Totalt visar utvärderingen (kapitel 5) en liten men tydlig fördel till A*, som i samtliga fall var den algoritm som i snitt snabbast hittade rätt plan för respektive mål. Över lag handlar det om en procentuell skillnad på 38 %, d.v.s. att A* i snitt (sett över samtliga mål) endast tog 62 % på sig av den tiden IDA* krävde. Vid test av max antal möjliga agenter simulerade samtidigt blev det dock uppenbart att skillnaderna är för små för att upptäcka i spelsammanhang (för denna implementation).

6.2 Diskussion

Strävan efter mer realistisk AI ökar drastiskt för att leva upp till de förväntningar spelare har då grafik och fysik förbättras i snabb takt. GOAP-arkitekturen visade sig vara ett möjligt steg i rätt riktning efter succén med F.E.A.R. (Monolith productions, 2005), men en stor nackdel är dess dynamiska planering som medför större belastning på processorn. Detta arbete bidrar till spelindustrin genom att tillhandahålla resultat som visar skillnader mellan en välanvänd algoritm A* och en minneseffektiv variant av A*, nämligen IDA*. Trots att IDA* i detta arbete visade sig kräva längre exekveringstid för planering i GOAP än A* kan denna fortfarande tänkas vara ett alternativ tack vare dess lägre minneskomplexitet (Millington m.fl., 2009), t.ex. för spel på handhållna enheter som smartphones som, på grund av deras storlek, har ett mer begränsat minne än datorer.

Ett användningsområde annat än spel som detta arbete kan tänkas bidra mycket till är t.ex. robotik, som ständigt kräver mer avancerad AI vilken samtidigt ska vara kapabel till att exekvera i realtid. Då det inte är en helt orimlig tanke att förutsäga att robotar blir vanligt förekommande i våra och framtida generationers liv kan det tänkas att det är extra viktigt med en implementation lik GOAP, där agenten (roboten) dynamiskt kan lösa problem på bästa/effektivaste sätt.

Att implementera GOAP för ett kommersiellt spel är en tung uppgift, och blev i det här arbetet anledningen till det lägre antalet implementerade mål och åtgärder som hann bli klara innan deadline. Då den största arbetstiden ligger i att implementera basstrukturen för GOAP (kapitel 4.3) kan detta arbete komma väl till hands för andra utvecklare som funderar på att använda GOAP i sina spel. Det ger en välstrukturerad och tydlig genomgång om vad GOAP innebär, vilka huvudkomponenter som krävs och vad som förväntas av den sökmotor som används, samt referenser till artiklar för vidare läsning.

Att skillnaderna vid testning av max antal agenter var obetydliga är ett intressant resultat, som visar att fastän IDA* möjligtvis inte är den snabbaste algoritmen, trots detta kan föredras för andra möjliga egenskaper, t.ex. att den undviker oändliga cykler (Russell & Norvig, 2003) eller har lägre minneskomplexitet än A* (Millington m.fl., 2009).

Etiska aspekter anses inte relevant för detta arbete, och kommer inte diskuteras.

6.3 Framtida arbete

Som nämnts tidigare så blev detta arbete lidande av en för kort utvecklingstid, och ytterligare tester med ett högre antal mål och åtgärder, uppåt hundratalet, hade varit optimalt för att verkligen urskilja vilka för och nackdelar de båda algoritmerna har för användning i GOAP. För att testa detta i riktiga sammanhang krävs dock en full implementation av varje mål och åtgärd, så som förhandsvillkor, kostnader för att balansera sökningar o.s.v. Att utveckla en komplett GOAP-arkitektur med över hundra mål och åtgärder kan anses som en onödig uppgift för att verkligen testa GOAP (eftersom man då redan har den fullt implementerad), men om GOAP i framtiden visar sig vara den givna arkitekturen för AI kan en sådan forskning tänkas bli väl eftertraktad.

Valet att utnyttja en 3D-renderare (Irrlicht, 2012) motiverades av dess relevans till dagens AAA-spel, för att se hur GOAP presterar då exekveringstiden är delad med andra, tyngre, komponenter. Ett vidare experiment med samma implementation, men begränsad till 2D,

hade kunnat vara av intresse. Då genom att endast fokusera på GOAP-prestandan, och ha övriga komponenter så resurssnåla som möjligt (t.ex. renderaren och vektor-matematiken).

Precis som för andra implementationer finns det här även rum för förbättringar. Av de resultat från profileraren i Visual Studio 2010 ges en fingervisning om var eventuell optimering kan ske, vilket är till stor nytta vid vidareutveckling av denna implementation. Trots att denna implementation visade sig vara förhållandevis resurssnål är det högst troligt att utbyggnad med fler mål och åtgärder förändrar detta då sökrymden blir större. Även mindre förbättringar är viktiga i de metoder som exekveras upprepade gånger under samma uppdatering (eng. *frame*).

Som beskrivs i diskussionsdelen är denna implementation högst återanvändbar, och detta arbete kan anses som en god källa för andra utvecklare vilka har som mål att implementera GOAP i ett eget utvecklat spel. Grundstrukturen för GOAP är, tack vare dess löst kopplade struktur, väl applicerbar på andra projekt med endast mindre modifikationer (förutsatt att även sökmotorn efterliknas).

Referenser

AIGameDev. (2008) *An Overview of the AI Architecture Inside the F.E.A.R. SDK*. AIGameDev.com.
Tillgänglig på Internet: http://aigamedev.com/open/article/fear-sdk/ [Hämtad 12.04.01].

Carvalho, C. (2002). The gap between processor and memory speeds. Proc. of IEEE International Conference on Control and Automation.
Tillgänglig på Internet:
http://gec.di.uminho.pt/discip/minf/ac0102/1000Gap_Proc-Mem_Speed.pdf
[Hämtad 12.02.06].

Gaschig, J. (1979). Performance Measurement and Analysis of Certain Search Algorithms.
Tillgänglig på Internet:
http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA286363
[Hämtad 12.02.05].

Graepel, T., Herbrich, R. & Gold, J. (2004) LEARNING TO FIGHT. Microsoft Research Ltd. 7 J J Thomson Avenue, CB3 0FB Cambridge, UK.

Higgins, D. (2002a) Generic A* Pathfinding. S. Rabin (red.). *AI Game Programming Wisdom* (s. 114-121). Charles River Media.

Higgins, D. (2002b) How to Achieve Lightning-Fast A*. S. Rabin (red.). *AI Game Programming Wisdom* (s. 133-145). Charles River Media.

Irrlicht (2012) Irrlicht (Version: 1.7.3) [Datorprogram]. Nikolaus Gebhardt et al. Tillgänglig på Internet: http://www.lith.com/Games/F-E-A-R- [Hämtad 12.02.10].

Korf, R.E., Reid, M., Edelkamp, S. (2001). Time complexity of iterative-deepening-A∗. Artificial Intelligence 129, 199-218.

Laird, J. E. & van Lent, M. (2001) Human-Level AI's Killer Application Interactive Computer Games. *AI Magazine,* 22(2), 15-26.

Mangegold, S. A., Boncz, P, L., Kersten, M. & M.P. Atkinson (red.) (2000) Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9, 231–246.

Millington, I. & Funge, J. (2009) *Artificial Intelligence for Games* (2:a upplagan). Morgan Kaufmann.

Monolith Productions (2005) *F.E.A.R.* (Version: 1.08) [Datorprogram]. Monolith Productions.
Tillgänglig på Internet: http://www.lith.com/Games/F-E-A-R- [Hämtad 12.02.10].

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

Orkin, J. (2004) Symbolic Representation of Game World State: Toward Real-Time Planning in Games. Presenterad vid *Game Developers Conference*, San Jose, California 24-27 mars, 2004.

Orkin, J. (2006) Three states and a plan: the AI of FEAR. Presenterad vid *Game Developers Conference*, San Jose, California 20-24 mars, 2006.

Orkin, J. (2012).
Tillgänglig på Internet: http://web.media.mit.edu/~jorkin/ [Hämtad 12.04.01].

R. Mahapatra, N & Venkatrao, B. (1999) The Processor-Memory bottleneck: Problems and Solutions. *Crossroads – Computer Architecture*, 5.

Russell, S. & Norvig, P. (2003) *Artificial Intelligence: A Modern Approach* (3:e upplagan). Upper Saddle River.