

Examensarbete

Vägsökning i en virtuell värld

av

Anders Dahlberg

och

Carl Viktorsson

LiTH-IDA-Ex-02/111

2002-11-05

Handledare: Patrik Haslum

Examinator: Patrick Doherty

Abstract

This report presents a Pathfinding problem and an implementation for a specific situation, a three-dimensional virtual world. The modeled world is called Project Entropia and is currently under development by MindArk AB as basis for a Massively Multiplayer Online Role-playing Game.

The pathfinding problem consists of two parts. Firstly the abstraction of the world to a simpler search space and secondly the actual search through that space. The search is a subset of a general graph search problem that can include different demands on the generated path. In this example the two major limitations are processing power and a generated behavior in the computer controlled agents that seems natural to the players.

The subject of Pathfinding is well explored in artificial intelligence research. After presenting several alternative methods a modified A*-algorithm is chosen and implemented. Some alterations are explained and motivated while others are only mentioned as they represent a functional interface to the interpreting script engine that manages the game servers.

Lastly a comparison between the presented implementation and the previously used system is made with a follow up on the goals that was set when the work begun.

Sammanfattning

Rapporten presenterar ett vägsökningsproblem och en implementation för ett specifikt problem i en tredimensionell värld. Världen kallas Project Entropia och är utvecklad av MindArk AB som grunden för ett rollspel för tusentals simultana spelare över Internet.

Vägsökning är ett tvådelat problem, dels abstraktionen av världen till en sökrymd och dels själva sökningen genom denna. Sökningen blir ett generellt grafsökningsproblem som kan inkludera särskilda krav på lösningen. I detta exempel är det två största begränsningarna kravet på beräkningseffektivitet och att spelarna får ett naturligt intryck av de datorstyrda agenterna.

Inom artificiell intelligens är vägsökning ett väl utforskat område. Olika alternativ presenteras innan en modifierad A*-algoritm väljs och implementeras. Vissa modifikationer motiveras och förklaras i detalj, andra nämns bara eftersom de endast är ett funktionellt gränssnitt till den överliggande skriptmotor som driver spelvärlden. Till sist jämförs den presenterade lösningen med den som tidigare användes och med målen som ställdes upp i arbetets början.

Förord

Författarna till denna rapport studerar på programmet för Informationsteknologi vid Linköpings universitet. Rapporten är avslutningen på ett 2 x 20 poäng långt examensarbete som examineras av Institutionen för Datavetenskap. Arbetet är utfört på MindArk AB i Göteborg under perioden juni till november 2002. Frågeställningen inför arbetet var att undersöka och implementera ett vägsökningssystem för intelligenta agenter i en virtuell värld.

Författarna vill rikta ett tack till alla som hjälpt till med synpunkter, kritik, idéer och praktiska tips under arbetets gång. Speciellt vill vi nämna våra handledare; på MindArk AB Magnus Eriksson och på IDA Patrik Haslum. Dessutom har några personer i vår närhet på MindArk AB i särskilt hög grad bjudit på sin arbetstid och sitt tålamod: Per Svensson, Patrick Broddesson, David Malcus, Christian Holmquist, Martin Gunnarson och Alexander Hugestrand. Tack.

Innehåll

1	INLEDNING	1
1.1	MINDARK AB.....	1
1.2	PROJECT ENTROPIA.....	1
1.3	SYFTE	1
1.4	AVGRÄNSNINGAR	1
1.5	METOD.....	2
1.6	STRUKTUR	3
1.7	KÄLLKRITIK.....	3
2	FÖRUTSÄTTNINGAR.....	4
2.1	VÄGSÖKNING	4
2.2	SKRIPTFUNKTIONALITETEN.....	5
2.3	DEN BEFINTLIGA ARTIFICIELLA INTELLIGENSEN	5
3	TEORETISK REFERENS RAM	7
3.1	PLANERING	7
3.2	VÄGSÖKNING I DATORSPEL	8
3.3	ABSTRAHERING AV OMGIVNINGEN	9
3.4	SÖKNING.....	17
3.5	SPECIELLA IMPLEMENTATIONSTEKNIKER.....	28
4	IMPLEMENTATION	33
4.1	VÄRLDEN I PROJECT ENTROPIA	33
4.2	TESTNING.....	34
4.3	DESIGNBESLUT FÖR ABSTRAHERINGEN	34
4.4	SÖKNING GENOM ABSTRAKTIONEN	46
4.5	IMPLEMENTERINGENS GRÄNSSNITT.....	51
5	RESULTAT	53
5.1	VÄGSÖKARENS PRESTANDA	53
5.2	PRAKTISKA TESTER.....	54
5.3	SLUTSATSER	56
6	UTVECKLINGSMÖJLIGHETER.....	58
7	ORDLISTA.....	61
8	KÄLLFÖRTECKNING	63

BILAGA 1: ARBETSGÅNG

BILAGA 2: PSEUDOKOD

1 Inledning

Denna rapport är resultatet av ett examensarbete på civilingenjörsprogrammet för informationsteknologi vid Linköpings tekniska högskola och examineras av institutionen för datavetenskap, IDA. Författarna har en liknande profilering på sina utbildningar med en inriktning mot artificiell intelligens och säkra interaktiva system.

Arbetet som rapporten beskriver har utförts på MindArk AB i Göteborg. Företaget arbetar med att utveckla datorspelet Project Entropia som faller under kategorin MMORPG, Massively Multiplayer Online RolePlaying Game. Spelarna antar roller som personer i en virtuell värld och agerar med och mot varandra och andra datorstyrda agenter.

1.1 MindArk AB

Bolaget grundades 1999 och har vuxit till att år 2002 omfatta ca 70 anställda; grafiker, programmerare och administrativ personal. MindArk AB är privat finansierat med ett fåtal större investerare och har sitt kontor i Göteborg. I dagsläget är MindArks verksamhet helt inriktad på utvecklingen av Project Entropia.

1.2 Project Entropia

Spelet Project Entropia utspelar sig på en planet kallad Calypso. Spelarna antar roller som utforskare och jagar monster och mineraler. Man kan ändra och förbättra egenskaper för sin karaktär under spelets gång. Den nyskapande idén är att verklig valuta kan växlas in mot krediter i spelet. Dessa används för sådant som spelarna inte får med sig i sin startutrustning och i framtiden ska olika betaltjänster vara tillgängliga. Spelarna kan tjäna pengar om de lyckas hitta stora mineralfyndigheter eller dödar farliga monster. MindArk AB:s inkomster är dock baserade på att spelarna i medel förbrukar mer krediter än de tjänar eftersom både programvaran och speltiden är kostnadsfri.

Inom spelet finns funktionalitet för social samverkan, förutom den normala textbaserade kommunikationen, som att starta egna privata grupperingar. Detta är ett steg mot visionen att Project Entropia ska uppfattas som en alternativ värld istället för endast ett nöje.

1.3 Syfte

Avsikten med denna rapport är att undersöka hur ett vägsökningssystem kan implementeras för agenter i en virtuell värld. I detta innefattas teoretiska analyser av befintliga metoder för sökning och abstraktion samt undersökningar av de olika systemens för- och nackdelar i ovanstående kontext. Resultatet av studien ska ge en rekommendation för hur vägsökning ska implementeras i Project Entropia och en praktisk implementation av en utvald lösning.

1.4 Avgränsningar

Arbetet är avgränsat till att omfatta studier av metoder som använts för vägsökning samt en implementation och utvärdering av någon metod som lämpar sig för Project Entropia.

1.4.1 Specifikation

Följande specifikation ingick i avtalet mellan författarna och MindArk AB då arbetet påbörjades. Den ska ses som riktlinjer för arbetet som sattes upp innan förstudien var genomförd.

SKALL utföras:

- Förstudier av MindArks system samt tidigare lösningar av vägsökningsproblemet och planering i AI-litteratur.
- Implementering av någon erkänd eller egenutvecklad algoritm (både abstraktion och sökalgoritm)
- Komplexitetsutvärdering av ovanstående samt praktiska tester.

BÖR utföras:

- Anpassning av ovanstående eller framtagande av en ny algoritm för inomhusmiljö inkl tester av densamma.
- Utökning av algoritmen m a p planering, tex med en lösning i flera steg och möjligheten att revidera planen vid oförutsedda händelser.

KAN utföras:

- Hänsynstagande till branta höjdskillnader.
- Tillägg av andra taktiska hänsyn, tex objekt att skydda, undvika eller bekämpa.

1.5 Metod

Bakom arbetet ligger såväl en teoretisk som en praktisk del. En teoretisk förankring finns i den förstudie av liknande implementationer och algoritmer som inledde arbetet.

Framförallt undersöktes sökalgoritmer och metoder för att optimera dessa efter varierande förutsättningar. Inriktningen mot sökalgoritmer grundades på den mer eller mindre vedertagna sanningen att sökning var det som tog tid vid vägsökning. Resultatet av förstudien utgör grunden för kapitel 3, Teoretisk referensram.

Då en teoretisk grund var lagd implementerades och testades några av de algoritmer som studerats fristående från varandra. Under denna testning framkom att sökningen i sig inte var det enskilt mest tidskrävande momentet utan att abstraheringen av agentens omgivning tog längre tid, om denna skulle göras dynamiskt under sökningens gång. Som en följd krävdes nya teoretiska studier för att, om möjligt, förbättra abstraheringen.

Efter att olika algoritmer testats och implementerats integrerades dessa med det befintliga systemet. En betydligt mer tidskrävande uppgift än planerat, där hantering av undantag och specialfall krävde mer tid än grundfunktionaliteten. Då vägsökningsalgoritmen var integrerad med Project Entropia utfördes iterativa tester av funktionalitet och prestanda. Framförallt fokuserades på att agenterna skulle få ett så naturligt rörelsemönster som möjligt samt att tidsåtgången för varje pass av vägsökningsalgoritmen skulle minimeras. Vid testningen användes verktygen Rational[®] Purify[™] för att hitta funktionella fel i koden och Intel[®] VTune[™] för att analysera vilka delar av koden som tog mest processorkraft i anspråk. Slutligen analyserades resultaten och arbetet sammanfattades i denna rapport.

1.6 Struktur

Rapporten är indelad i två huvudavsnitt. I det första ges en teoretisk grund till vägsökning. I det andra presenteras den algoritm som implementerats, med designbeslut och testresultat. Huvudavsnitten är i sin tur delade mellan abstraheringen av världen och sökningen genom abstraktionen. Till sist presenteras de resultat som arbetet gav.

1.7 Källkritik

Informationen som rapporten bygger på har framförallt hämtats från vedertagna källor, exempelvis litteratur och artiklar från tidningar och konferenser. Eftersom vägsökning är viktig i datorspelsbranschen som utvecklas mycket snabbt finns inte alltid information tillgänglig genom de klassiska distributionskanalerna. I dessa fall har ofta information hämtats från Internet istället. Då källor från Internet används är det viktigt att kontrollera ursprung och publiceringstidpunkt. I stort sett allt material som hämtats från Internet kommer från stora erkända webbplatser med inriktning mot datorspel, tex Gamasutra.com och Gamedev.net. Internetbaserade sökmotorer har även använts för att hitta relevanta artiklar.

Under arbetets gång har många problem diskuterats med anställda på MindArk AB och via e-post med handledaren på Linköpings universitet. Dessa informella samtal har ofta bidragit till att en viss lösning valts. Dessutom har praktiska och teoretiska begrepp klargjorts. Informella samtal ska inte ensamma ligga till grund för beslut som presenteras i rapporten och målsättningen är att alltid ge fakta som motiverar ett beslut.

1.8 Målgrupp

Rapporten har ett brett teoretiskt spektrum där delar som kan uppfattas som grundläggande kunskaper presenteras tillsammans med mer avancerade aspekter. Det medför att läsare helt utan kännedom om datastrukturer, algoritmanalys och artificiell intelligens kan uppfatta den som svårgenomtränglig medan personer med en datavetenskaplig bakgrund kan uppfatta delar av bakgrunden som trivial. Om läshastigheten anpassas enligt detta bör alla kunna tillgodogöra sig resultatet av arbetet.

2 Förutsättningar

Nedan presenteras de förutsättningar som låg till grund för arbetet. Framförallt beskrivs vilka krav Project Entropia ställer på vägsökning och hur agenterna vägsökte innan arbetet påbörjades.

2.1 Vägsökning

Arbetet som rapporten beskriver rör styrandet av de datorstyrda agenterna och specifikt problemet att med en för ändamålet lämplig algoritm finna en väg mellan två punkter i den virtuella världen, något som vanligtvis benämns vägsökning (eng. *pathfinding*) [12]. Omgivningen kan till exempel representeras av en karta, en matris, en graf eller bilder från en kamera. Vanligtvis är målet inte enbart att finna en väg utan den ska dessutom uppfylla olika bivillkor, kanske ska ett så stort område som möjligt avsökas eller så ska den sökta vägen vara så kort som möjligt i något avseende [4]. Vanliga tillämpningar är autonoma fordon som ska undvika kollisioner då det färdas eller datorsimuleringar, exempelvis i datorspel eller i datornät. Problemet är akademiskt välutforskat, särskilt då det gäller sökmetoder som utgör en betydande del av lösningen.

Om agenterna har tillgång till en fullständig beskrivning av världen runt sig är det möjligt att utan andra krav alltid finna en optimal väg. Faktorer som gör detta praktiskt svårt är de begränsningar på tids- och minneskomplexitet som ställs av datorn som ska utföra beräkningen i så nära realtid som möjligt samt av de mänskliga spelarna som inte vill se agenten ta tankepauser när den letar sig fram. Dessutom ska agenten röra sig naturligt efter den genererade vägen. En jämförelse med det inom artificiell intelligens klassiska Turingtestet ger att den perfekta vägsökaren inte ska gå att skilja från en agent kontrollerad av en människa [35].

2.1.1 Vägsökning i Project Entropia

Det finns några faktorer som ställer särskilda krav på vägsökningen i Project Entropia. För det första finns realtidskrav, användarna förväntar sig omedelbar respons från NPC:erna i världen. NPC står för *Non Player Character* och är i datorspelsammanhang datorstyrda agenter som de mänskliga spelarna kan interagera med. Eftersom alla beräkningar som rör förflyttning av NPC:er sker på en central server och inte går att distribuera till användarna ställs höga krav på en så liten beräkningstid som möjligt. Det innebär att optimeringar efter förutsättningarna i Project Entropia blir viktiga för att minimera serverbelastningen.

För det andra finns egenskaper som är direkt kopplade till logiken i spelet. Slutmålet, tex en spelare som NPC:n vill springa till, är ofta rörligt. Under exekveringen av en färdigberäknad färdväg kan alltså målets position förändras. Enkla algoritmer har inget stöd för detta utan kräver ett helt nytt pass av beräkningar. Istället kan delar av den gamla planen utnyttjas för att skapa den nya, så kallad omplanering (eng. *replanning*). Förutom att mål ändras dynamiskt är det tänkt att vissa delar av världen kan vara stokastiska. Med det menas att NPC:n inte på förhand kan få en exakt bild av sin omgivning. Skogar genereras med hjälp av en täthetsfaktor på träden som sedan placeras ut slumpmässigt då agenten kommer tillräckligt nära. Om dessa träd är hindrande för rörliga objekt så förändras NPC:ns bild av omgivningen under exekveringen och särskilda algoritmer, så kallade dynamiska sökningar, måste användas för att få en effektiv lösning.

En annan faktor som är speciell för Project Entropia är storleken på världen. Målet med Project Entropia är en värld med en dimension på miljontals kvadratmeter där tusentals NPC:er rör sig samtidigt. Dessutom är deras beteende och miljön de rör sig i varierande. Vissa NPC:er är fredliga försäljare som endast ska röra sig inomhus i sin butik medan andra är aggressiva robotar som kan förflytta sig över stora ytor i en utomhusmiljö. I dagsläget finns ingen möjlighet att separera de olika miljöerna utan en algoritm måste vara så generell att den fungerar för samtliga scenarion.

De krav som förutsättningarna i Project Entropia gav vid handen kan sammanfattas i att:

- Det nya vägsökningssystemet ska klara av den belastning som NPC:erna utövar idag.
- Algoritmen för vägsökning ska kunna integreras med den befintliga artificiella intelligensen.
- Samma algoritm ska kunna användas i hela Project Entropia.
- Exekveringstiden ska vara så snabb och minnesåtgången så låg att algoritmen kan köras på samma server som spelmotorn.

2.2 Skriptfunktionaliteten

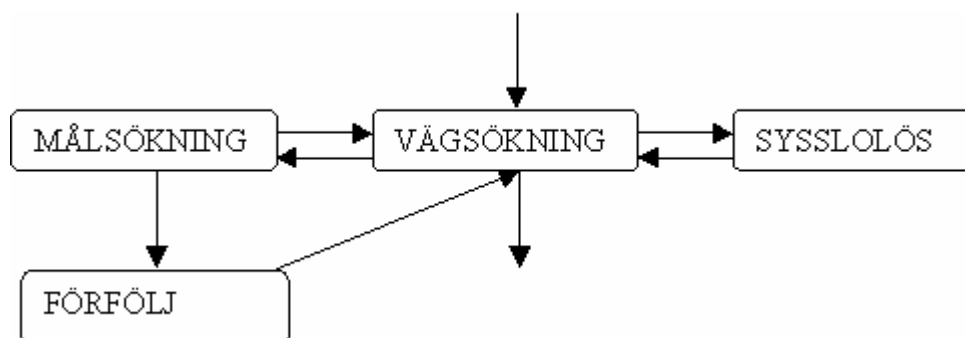
Motorn som driver spelvärlden är utvecklad i C++ men den interpreterar ett annat språk som är egenutvecklat av MindArk AB och kallat BOLD. Tanken med BOLD är att erbjuda ett gränssnitt till de som arbetar med design av världen som ska vara enklare att använda än C++ och inte förutsätta förkunskaper inom programmering. BOLD anropar i sin tur C++-funktioner som praktiskt hanterar kommunikation, grafik och dylikt.

När arbetet påbörjades så fanns det redan en grundläggande trelagers artificiell intelligens implementerad i BOLD för styrningen av spelets NPC:er. Den bestod av en tillståndsmaskin för strategiska och taktiska beslut samt en stimuli-respons-styrning för undvikande av hinder. Om det gick att låta det befintliga systemet hämta ut delmål ett och ett längs med en förplanerad bana istället för att alltid gå rakt mot slutmålet så kunde det lämnas intakt. Funktionaliteten hos det gamla kollisionsundvikandet och beslutsfattandet var dessutom eftersträvansvärt att behålla eftersom det var väl testat.

2.3 Den befintliga artificiella intelligensen

Beslutsfattandet hos NPC:n var i det ursprungliga systemet en tillståndsmodell med två lager samt stimuli-respons-styrning för kollisionsundvikande. De tillstånd som var intressanta för vägsökaren var i det övre, strategiska, lagret ANFALL, FLY och VILA. Alla tre tillstånden kan aktivera en navigering. När en NPC är i VILA kan den bestämma sig för att gå till ett slumpmässigt valt mål för att ge ett naturligare intryck hos de mänskliga spelare som står och tittar på. I tillstånden ANFALL och FLY vill NPC:n röra sig mot respektive bort från målet. Hur NPC:n växlar mellan dessa tillstånd är inte intressant för vägsökningen. Det räcker att veta att det övre lagret bestämmer det övergripande målet, antingen ett rörligt objekt eller en absolut position.

Det underliggande, taktiska, lagret innehåller tillstånd som mer konkret styr förflyttning och beteende. De som berör vägsökaren är: MÅLSÖKNING, FÖRFÖLJ, VÄGSÖKNING och SYSSLOLÖS.



Figur 1 Del av det taktiska lagrets tillstånd.

SYSSLOLÖS är det taktiska vilotillståndet som NPC:n står i när den inte har fått någon uppgift från det överliggande lagret.

Läget VÄGSÖKNING hanterar de situationer där det saknas en färdig målkoordinat, antingen för att en ny uppgift just har kommit in från det övre lagret eller för att NPC:n uppnått ett delmål. Om det finns fler mål längs med färdvägen hämtas det ut ett nytt och tillståndet ställs om till MÅLSÖKNING.

I MÅLSÖKNING försöker NPC:n färdas i en rät linje mot den aktuella delmålskoordinaten. Under färden är tre känselspröt aktiva som undviker oförutsedda hinder. Dessa kan vara andra rörliga objekt eller fasta objekt som inte vägsökningen lyckats undvika pga yttre påverkan på NPC:n eller dess rörelsetröghet.

Tillståndet FÖRFÖLJ aktiveras när avståndet till ett rörligt mål understiger ett gränsvärde. FÖRFÖLJ använder ingen planering utan styr rakt mot målet. Det besparar beräkningskraft när NPC:n och spelarna rör sig runt varandra under en närstrid.

När ett slutligt mål uppnåtts rapporteras det upp till det strategiska lagret. Där ställs det taktiska beteendet om till vad syftet var med navigeringen, tex ATTACK, HÅLSNING eller ett fortsatt FLY-beteende.

Det är inte intressant, för vägsökningens funktionalitet, att beskriva övriga delar av tillståndssystemet eller hur det strategiska lagret väljer beteende och mål.

3 Teoretisk referensram

I detta kapitel presenteras den litteratur som använts som teoretisk bakgrund för den praktiska implementationen. Om något begrepp är oklart för läsaren, antingen för att ämnet är obekant eller för att den svenska översättningen har ersatt en vanligare engelsk term, så finns en ordlista i slutet av rapporten.

Vårt vägsökningsproblem kan i stora drag sammanfattas i två delproblem, att transformera omgivningen till data som kan tolkas av agenten och att söka den bästa vägen i denna. Dessa utgör tillsammans ett planeringsproblem, alltså är vägsökning en typ av planering.

3.1 Planering

Planering handlar om att förändra tillstånd genom handling, på ett förutsägbart sätt. Planering är ett betydligt mer generellt problem än vägsökning. Eftersom planering inte endast handlar om förflyttning, som vägsökning, utan om kombinationer av alla för agenten möjliga handlingar blir tillämpningarna obegränsade. Yang tar upp två kriterier som är avgörande för en planerare; effektivitet och kvalitet [13]. Effektivitet är viktigt eftersom antalet kombinationer av tillstånd och handlingar ofta är mycket stort, vilket medför att en naiv planerare måste kontrollera miljontals sekvenser av handlingar innan en korrekt plan hittas. Kvalitet innebär att planen är så optimal som möjligt. Vad som menas med optimal definieras ofta av omgivningen, det kan vara maximal vinst, minimalt antal handlingar för att uppfylla målet eller något annat.

Vidare menar Yang att en planerare måste utnyttja någon intelligent metod för att vara effektiv och ge lösningar med hög kvalitet. Exempel på sådana intelligenta metoder är söndra-och-härskas (eng. *divide-and-conquer*) och hierarkisk abstraktion. Söndra-och-härskas innebär att ett komplext problem delas upp i flera mindre delproblem som löses var för sig. Nya aspekter som måste behandlas är hur problem delas upp och hur dellösningarna sätts samman för att bilda den kompletta planen. Hierarkisk abstraktion använder förenklade, eller mer abstrakta, problem. Lösningen till ett abstrakt problem används som vägledning då det ursprungliga mer komplexa problemet ska lösas. På så sätt behöver förhoppningsvis färre handlingar undersökas. [13]

Då forskning kring planering som en del i artificiell intelligens inleddes uppstod snabbt två huvudmetoder representerade av två, numera klassiska, rapporter om QA3 respektive STRIPS. QA3 bygger på en representation där handlingar och tillstånd axiomatiseras med logiska uttryck. Genom att ställs upp logiska teorem och falsifiera eller verifiera dessa skapas en plan. STRIPS å andra sidan genererar en plan genom att söka bland mängden av möjliga tillstånd och handlingar tills en sekvens av handlingar som leder från start- till måltillståndet hittas. Dessa metoder brukar benämnas logik- respektive sökbaserad planering. Idag är sökbaserad planering dominerande. [15]

Schemaläggning

Ett område besläktat och lätt att förväxla med planering är schemaläggning. Skillnaden ligger i att i ett schemaläggningsproblem finns en färdig omgivning där någon målfunktion ska minimeras (eller maximeras) under olika bivillkor, tex kombinatorisk optimering där en kortaste väg ska hittas i en graf eller nätverk [14]. Yang menar att för ett schemaläggningsproblem måste en struktur för lösningen finnas (tex en graf med

kostnader) och effekterna av handlingar måste vara kända medan ett planeringsproblem även innefattar generering av strukturen och resonemang om följderna av handlingar [13].

3.2 Vägsökning i datorspel

Det första valet vid utvecklingen av datorstyrda agenter är vilken övergripande metod för styrning som ska användas. Den beräkningskraft som vägsökning kräver kan ofta undvikas med hjälp av enkla regelbaserade system. De kan klara lokal kollisionshantering, undvikande av hinder och andra navigeringsbegränsningar. I de fall då omgivningen ställer för stora krav på ett stimuli-respons-system, exempelvis vid navigering i en labyrinth, orsakar det istället gärna låsta lägen. Regelbaserad navigering är begränsad av sin brist på information om omgivningen vilket resulterar i suboptimala lösningar eller fallor agenten inte kan komma ur, lokala maxima. [4]

Ofta har agenten heller inget konkret geografiskt mål utan istället ett syfte med navigeringen som tex att undfly en annan spelare eller att behålla honom inom synhåll. I dessa situationer duger inte en statisk planerare utan planen måste kunna revideras under utförandet. En populär lösning är att dela upp problemet i flera lösningsnivåer. Ett strategiskt lager tar hand om det övergripande syftet med navigeringen. Ett taktiskt utvärderar omgivningen och tar ett beslut som en utförare hanterar.

Omgivningens dynamik

Björn Reese et al [4] föreslår en kategorisering av olika former av dynamik.

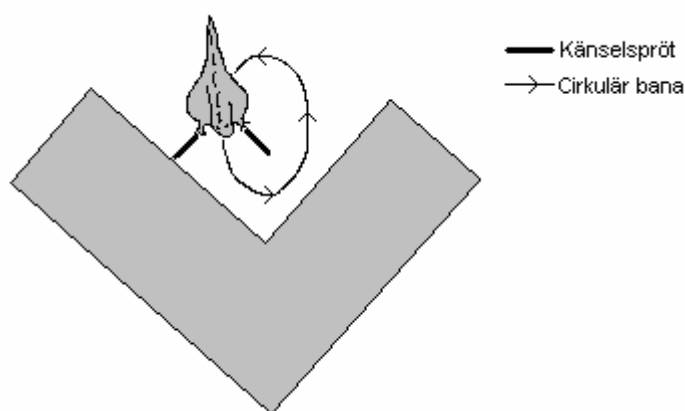
- Statisk omgivning. Navigering är relativt lätt eftersom allt är förutsägbart, problemet är deterministiskt.
- Rörliga spelare och statiska hinder. Närvaron av andra agenter måste hanteras dynamiskt eftersom deras färdvägar är oförutsägbara även om deras momentana hastighet och riktning är kända. Det är ofta möjligt att ignorera andra spelare under planerandet och sedan hantera dem lokalt under färdvägen.
- Rörliga hinder. Med rörliga hinder är det inte möjligt att planera en fullständig färdväg i förväg. Att använda sig av fullständig omplanering är beräkningsmässigt tungt men alternativ som D* kan användas [5].
- Försvinnande hinder. Om hindren inte bara kan röra sig utan också dyka upp och försvinna försvåras planerandet ytterligare eftersom det inte finns någon kontinuitet i tiden att utnyttja.
- Manipulering. Om agenten har möjligheten att påverka sin omgivning under färden uppstår fler problem för planerandet. Agentens påverkan kan förstöra, skapa eller flytta hinder.

Vilken grad av dynamik problemlösaren ska klara är avgörande för vilken algoritm som är lämplig. Den vanligaste algoritmen för vägsökning, A*, hanterar i sitt grundutförande endast statiska miljöer och måste om miljön förändras göra en ny fullständig planering. En dynamisk variant av A*, D*, klarar av miljöer där problemlösaren inte är fullt informerad någorlunda bra men utnyttjar ändå inte kunskaper som kan finnas om hur miljön förändras. Metoder som klarar av att ta hänsyn till förändringar i en tidsdimension går beräkningsmässigt bortom de tidsgränser man kan vänta sig i ett realtidsdatorspel.

Omgivningens geometri

Björn Reese et al [4] fortsätter med att nämna olika aspekter som spelar in på hur problemlösaren kan hantera terrängen.

- Former. Agenter och hinder kan representeras som punkter eller som geometriska former som rätblock eller cirklar. Dessa är lättare att hantera än godtyckliga former. Rotation av objekten är särskilt beräkningskrävande om man inte har använt en lämplig abstraktion.
- Partitionering. Vanligen används en grafbaserad sökalgoritm för att hitta en väg. Grafen skapas genom en lämplig sönderdelning av sökområdet. Storleken hos grafen och kopplingsgraden mellan noderna påverkar algoritmens prestanda. Det är en avvägning mellan en enkel sökning i en karta som är komplex att skapa eller vice versa.
- Konkavitet. Regelbaserade navigeringstekniker som utgår från att alla hinder ska undvikas istället för att finna en väg mellan dem får problem med vissa former på hinder, tex ickekonvexa polygoner där vinklarna mellan två ytor får vara $< 180^\circ$ (se Figur 2).
- Särskilda strukturer. Terrängen kan innehålla föremål som försvårar partitioneringen av kartan, tex portaler mellan två punkter eller loopar. Det kan vara nödvändigt att hantera dessa i en särskild struktur, tex med fast utlagda delmål som måste passeras mellan två punkter.
- Varierande terräng. Terrängen behöver inte vara binär, dvs passerbar eller ej. Olika terräng kan behöva få olika vikter som måste hanteras av sökalgoritmen.



Figur 2 Illustration av hur en reaktiv agent kan fastna i ickekonvexa polygoner.

3.3 Abstrahering av omgivningen

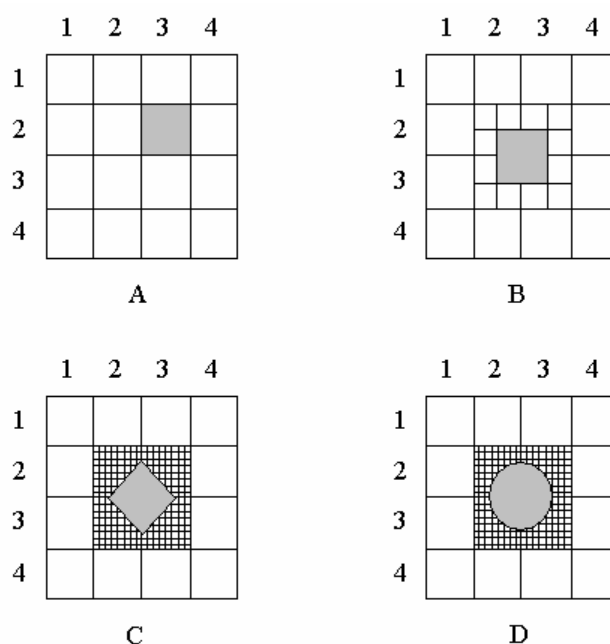
På något sätt finns agentens omgivning tillgänglig i systemet, antingen förinläst eller avläsbar med sensorer. Dessvärre kan tillgänglig data skilja sig radikalt från den representation som är lämpligast för vägsökningen. Därför behöver ofta agentens omgivning abstraheras till ett format som passar vägsökningsalgoritmen bättre. I de fall då abstraheringen dessutom kan minska antalet noder som behöver avsökas optimeras algoritmen ytterligare [38].

I en tredimensionell värld är objekten vanligtvis lagrade som begränsningsvolym [34]. En begränsningsvolym är ett tredimensionellt skal med geometriska egenskaper såsom

position, storlek och rotation. Agentens mål är att undvika dessa när den förflyttar sig från startpunkten till målpunkten. I en tvådimensionell värld motsvaras begränsningsvolymerna av begränsningsareor. Dessa två representationer är tänkbara format på agentens bild av världen. I Project Entropia är världen tredimensionell och därför ligger tyngdpunkten i det följande kapitlet på omgivningar som i huvudsak byggs upp av begränsningsvolymer. Andra tänkbara format är bilder eller videosekvenser tagna med en kamera. I det fallet krävs ofta avancerad bildbehandling för att extrahera värdefull information ur bilderna som kameran ger, något som ligger utanför denna rapports omfattning. Mer abstrakt kan även världen representeras av en tillståndsgraf, exempelvis med en uppsättning handlingar som agenten kan utföra och eventuellt information om en handlings följder.

Vägsökning i en värld som utgår från grafrepresentationer kräver vanligtvis inte lika omfattande abstrahering som i en värld uppbyggd av geometriska objekt, eftersom många sökalgoritmer är direkt anpassade till grafsökning. Två- och tredimensionella världar är i grunden likartade eftersom en tvådimensionell värld är en sorts förenkling av en tredimensionell.

Ett första konstaterande är att formen hos begränsningsvolymerna påverkar upplösningen som krävs i vägsökningssystemet [4]. Typen av volymer är avgörande för vilken abstraktion som är lämplig. Är positionen för objekt bestämd i diskreta steg behövs inte en lika hög upplösning som om den är godtycklig. Ytterligare komplikationer kan uppstå om objekten är roterade eller kan ha andra geometriska former än rätblock, tex klot. Detta illustreras i Figur 3.



Figur 3 I bild A är volymen av bestämd storlek och tillåts endast på vissa positioner. Bild B visar att upplösningen måste ökas då volymerna kan placeras friare. I C och D krävs en godtyckligt hög upplösning för att de mer avancerade volymerna, i dessa fall behöver antagligen abstraktionen bygga på andra former är rutnät.

Utöver de nämnda begränsningsvolymerna kan det finnas olika typer av terräng som påverkar vägsökningen; berg, vatten, vägar, skogar etc som alla har olika karaktäristiska egenskaper. Dessa kan ha godtycklig form och dessutom olika parametrar, exempelvis

lutning för ett berg eller täthet i en skog, som i sig påverkar framkomligheten på ett annat sätt än de rena begränsningsvolymerna.

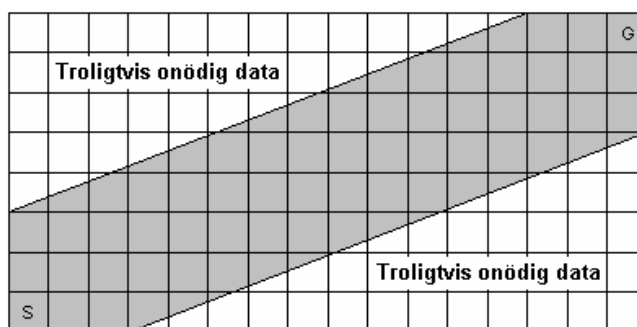
En tredimensionell värld är ofta på förhand ordnad i något system för att minska beräkningstiden vid tex kollisiondetektering och rendering. Om ett sådant system existerar kan det vara lämpligt att använda vid abstraheringen av omgivningen. I en tredimensionell värld är det vanligt att använda sig av celler för att dela in världen i mindre områden, framförallt så underlättas renderingen av detta. Varje cell har en mängd objekt som rumsligt befinner sig i cellen, diverse grafiska parametrar som ljuskällor och transformationer, samt en lista med portaler. En portal är en koppling från en cell till en annan.

Representationen av världen har stor betydelse för den slutgiltiga vägsökningsalgoritmens effektivitet och dessutom ger en bra representation ofta mer naturliga rörelser hos agenterna [17]. Historiskt sett har två olika idéer funnits om hur världen kan abstraheras för en vägsökare: Uppdelning i celler eller skelettisering (eng. *skeletonization*) [17, 22]. Förutom den övergripande abstraheringstekniken finns en rad underliggande frågor såsom vilken datastruktur som ska användas för lagring och vilka följderna för den slutgiltiga vägen blir.

3.3.1 Cellrepresentationer

En enkel cellrepresentation är att använda ett tvådimensionellt rutnät där cellerna tilldelas ett värde efter hur svåra de är att passera (med en bit per cell blir cellen antingen blockerad eller passerbar). Varje cell i nätet representerar ett område i världen som agenten rör sig i. Denna representation kan enkelt utökas till tre dimensioner om agenten ska kunna röra sig på andra ytor än i markplanet. I de flesta rapporter om vägsökning utgår man från ett rutnät som implementeras med en tvådimensionell array [4, 9, 11 m fl]. Den är enkel att förstå för programmeraren och det går snabbt att adressera ett element för sökalgoritmen.

En nackdel med arrayer är att de är ineffektiva då naturlig terräng ska abstraheras [16]. Det beror på att i naturlig terräng är likartade objekt ofta grupperade i områden. Om ett sådant område ska sparas i en matris kommer många närliggande celler innehålla samma information vilket innebär ett onödigt stort minnesutnyttjande [20]. Många intilliggande celler med samma värde innebär också en onödigt stor sökrymd [38]. Dessutom är ofta stora delar av rutnätet outnyttjat av sökalgoritmen då det ligger alldeles för långt från vägen mellan start och slutpunkt (se Figur 4). En annan nackdel med rutnät är att den funna vägen tenderar att bli hackig eftersom vägen alltid går i diskreta steg och riktningar [38].



Figur 4 Det gråfärgade området markerar den information som sökningen troligtvis utnyttjar.

För att undvika den onödiga minnesåtgången kan man tänka sig andra tvådimensionella strukturer än rena rutnät. Pottinger föreslår en karta av godtyckliga polygoner i en artikel om terränganalys, ungefär som ett pussel av polygoner [18]. Godtycklig i detta sammanhang avser storlek och form hos polygonen. Då kan en polygon bestå av en, eller flera sammanhängande, begränsningsvolym eller ett område med en typ av terräng i världen. En snarlik beskrivning finns i [17] men då med betoning på skelettisering. Andra idéer finns där polygonerna begränsas till trianglar eller andra geometriska former [38].

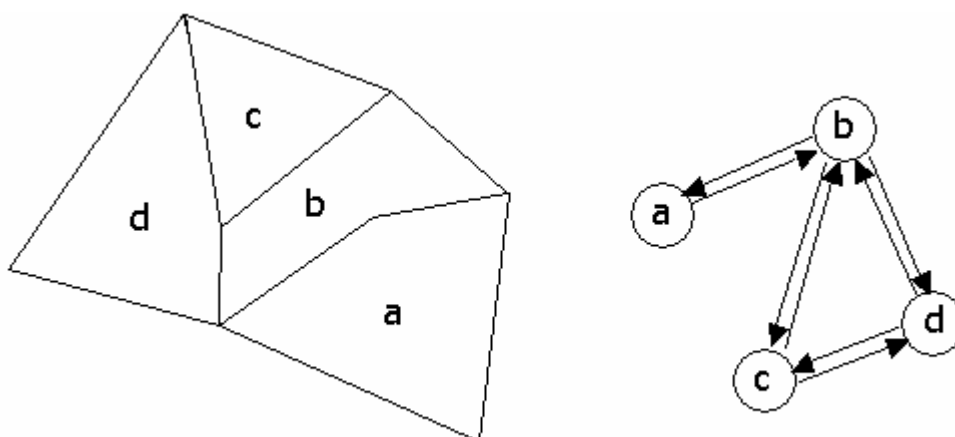
Egentligen är också ett rutnät en mängd polygoner med tilläggsvillkoret att dessa är kvadratiska med sidor av samma längd. På samma sätt som ett rutnät kan utökas till att representera tre dimensioner motsvaras polygoner av volymer i tre dimensioner. Problemet med strukturer där områdena inte definieras av absoluta regler är att hitta formen på varje område. En variabel form kan även innebära att vägen får konstiga svängar. Att hitta den minsta mängd polygoner med samma parametrar som täcker ett givet område utifrån en given mängd polygoner är, med avseende på tiden, $O(n^3 \log n)$ men det finns andra algoritmer som garanterar en nästan optimal mängd på linjär tid [17]. För att kunna skapa polygoner måste den data som håller informationen om världen vara tillgänglig i något format som enkelt kan göras om till polygoner.

I vissa tredimensionella miljöer är begränsningsvolym och terräng på förhand formade som polygoner, i dessa fall kan det vara fördelaktigt att utnyttja den givna representationen [38]. En sådan metod presenteras i [39] och kallas *Navigation Mesh*. Polygonerna, som i detta fall är trianglar, projiceras på markytan och fungerar sedan som ett sorts rutnät fast bestående av trianglar. Fördelen är att cellerna kan vara av variabel storlek och därmed mycket god precision. Den främsta nackdelen är att merkostanden i polygonhanteringen blir för stor för realtidssystem. [39]

En idé om hur terräng ska kunna delas in i meningsfulla områden som skulle kunna motsvara polygoner i en representation ges i [21]. Metoden kallas *Qualitative Spatial Reasoning* och innebär att världen delas upp i regioner beroende på fysiska egenskaper och uppgiftsberoende parametrar. Uppdelningen underlättas av att egenskaper och parametrar diskretiseras till enstaka brytvärden. Man föreslår tex egenskaper som lutning och parametrar som områden där agenten är synlig för andra enheter. Tyvärr presenteras inga algoritmer eller komplexitetsanalyser och det är lätt att anta att många parametrar kräver mycket beräkningstid. Om utgångspunkten istället är geometriska objekt (volym eller areor) kan det vara möjligt att använda dessa direkt för att skapa en graf av sammansatta volymer eller areor. [21]

Både strukturen i ett rutnät och i en mängd godtyckliga polygoner kan ses som en graf. En (oriktad) graf är en icke-tom mängd av noder och en mängd av (oriktade) bågar [14]. I ett rutnät har varje nod ett konstant antal bågar (vanligtvis fyra eller åtta) medan i en graf där noderna är godtyckliga polygoner kan varje nod ha lika många bågar som motsvarande polygon har sidor (se Figur 5). Det fyller alltså ingen funktion att implementera ett rutnät som en graf eftersom antalet bågar per nod är konstant. En graf kan vara ett lämpligt steg i en abstraktion av en värld som representeras av polygoner eller lätt kan transformeras till polygoner. En graf implementeras exempelvis av en länkad lista eller med en array [27]. För sökning är grafer väl lämpade, problemet är att generera cellerna som grafen bygger på. Ytterligare en nackdel med en graf är svårigheten att hitta en nod som motsvarar ett visst område i världen. I värsta fall måste samtliga noder itereras för att hitta en sökt nod. En tänkbar, men komplicerad, lösning är att spara

noderna i en hash-tabell med en hashfunktion som är beroende av nodens fysiska position.

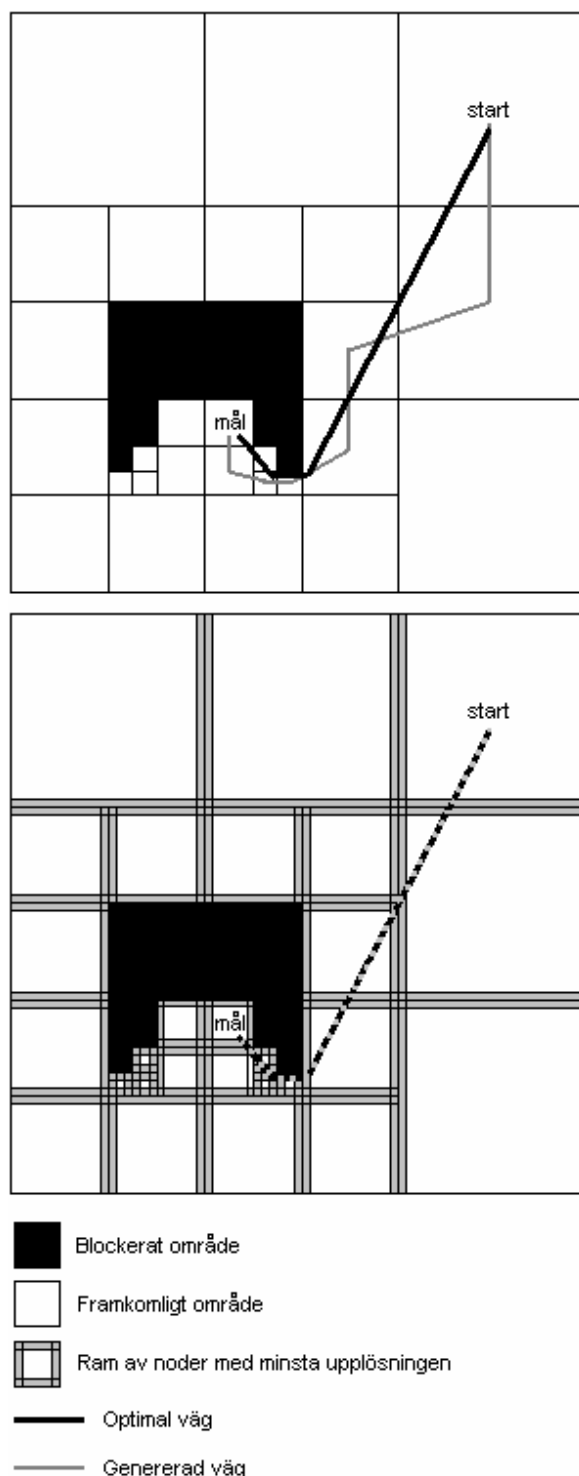


Figur 5 Ett område av polygoner (till vänster) och den motsvarande grafen (till höger).

Godtyckliga polygoner är effektiva på grund av sin förmåga att anpassa sig till världen men samtidigt kan det vara mycket beräkningskrävande att generera polygonerna så att de avbildar världen korrekt liksom att söka i den datastruktur där de lagras. Rutnätet å andra sidan var relativt okomplicerat att skapa och enkelt att hitta den ruta som motsvarar ett område i världen men inte alls lika flexibelt. Ett mellanting mellan rutnät och godtyckliga polygoner är quad-träd [16, 19]. Ett quad-träd är ett rutnät där varje fält kan delas upp i fyra nya fält ett godtyckligt antal gånger. Den tredimensionella motsvarigheten till ett quad-träd är ett oct-träd där varje volym kan delas upp i åtta nya volymer. Varje fält delas upp rekursivt till dess att den antingen är fri från hinder eller att den minsta fältstorleken nås. På så sätt kan även godtycklig precision uppnås i de fält där så önskas (se Figur 6). Quad-träd möjliggör en effektiv partitionering av världen eftersom ett enda fält kan användas för att koda ett stort område med lika parametrar. Quad-träd är egentligen oviktade sökträd med fyra barn per nod. Tidskomplexiteten för att hämta ett objekt ur ett sådant quad-träd är i värsta fall $O(n)$ men i medel $O(\log n)$. Vanligtvis begränsas upplösning i quad-träd vilket innebär att en övre gräns för antalet nivåer i sökträdet fås. Ett problem med quad-träd är att den genererade vägen inte alltid är optimal därför att den begränsas till att gå mellan mittpunkten i varje fält. Då vissa fält är större än den högsta precisionen kan vägen få onödiga svängar (se Figur 6) [16].

För att undvika detta kan datatypen modifieras något till ett inramat-quad-träd [16,19]. Ett inramat-quad-träd skiljer sig från vanliga quad-träd genom att varje fält har sin omkrets fylld med fält av den högsta precisionen. Egenskaperna för både quad-träd och inramade-quad-träd varierar i förhållande till en rutnät implementation. I [16] görs en jämförelse mellan sökning (med D*) i ett rutnät och ett inramat-quad-träd i sökrymder med varierande frekvens på blockerade områden. Resultaten visar att då världen är på förhand känd blir rutnätet betydligt effektivare vad gäller minnesåtgång för alla undersökta fall. Tidsåtgången blir mindre för inramade-quad-träd vid låg densitet av blockerade områden. Då densiteten ökar kräver rutnät implementationen mindre tid. I en okänd värld är däremot det inramade-quad-trädet bättre både med avseende på minnes- och tidsåtgång. Tyvärr är inte detaljerna, exempelvis hinderdensiteten, från jämförelsen redovisad varför det är svårt att dra några ytterligare slutsatser. En abstrahering snarlik inramade-quad-träd presenteras i [20]. Här införs portaler mellan de intilliggande fälten istället för en ram av små fält. En portal är en nod som kopplar samman två fält i världen.

Figur 6 Quad-träd högst upp, inramat quad-träd nederst. Med det inramade quad-trädet blir den genererade vägen även optimal (med avseende på upplösningen).



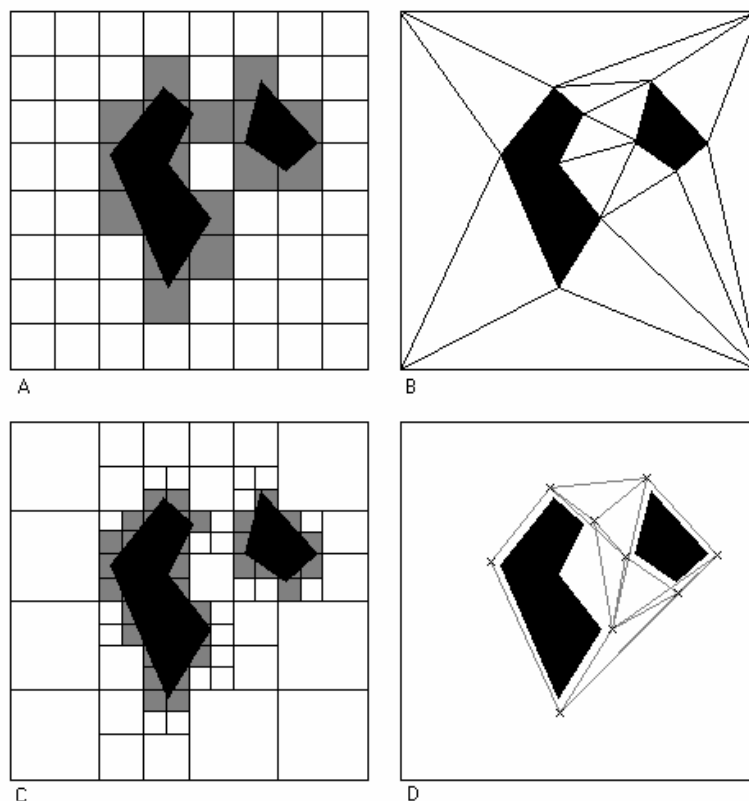
3.3.2 Skelettisering

Ett skelett är ett nätverk av vägar (eng *path lattice*). En annan vanlig benämning på skelettisering är *Points of Visibility*. Nätverket byggs upp utifrån en mängd punkter som är utplacerade i världen, manuellt eller automatiskt. Samtliga vägar mellan noderna testas och de vägar som inte är blockerade mellan ett nodpar bildar en båge mellan dessa. En sådan test tar alltid $O(n^2)$ tid där n är antalet noder. Om världen inte är dynamisk räcker det med att göra ett sådant test varje gång applikationen startas. Alternativt kan resultatet lagras på en fil för att sedan läsas in. Det nätverk som bildas används när en agent ska förflytta sig mellan två punkter. Vid förflyttning följer agenten helt enkelt den rätta linje

som bågen mellan ett nodpar bildar. För att finns en väg mellan noderna används någon sökalgoritm. [17]

Två allvarliga brister med denna metod är uppenbara: För det första måste noderna placeras ut på något sätt och för det andra representeras inte alla punkter i världen i nätverket [17]. Noderna kan placeras genom någon särskild algoritm avsedd för detta. Den enklaste är att statistiskt, med ett fast avstånd, placera noderna i världen, men då förvandlas skelettet till ett rutnät. Istället gäller det att finna noder som kan representera stora områden i världen vilket slutligen ger samma problem som i de mer avancerade cellrepresentationerna: Hur ska olika områden i världen generaliseras för att representera en nod i sökningen? Benämningen *Points of Visibility* kommer från idén att placera noder i alla konvexa hörn [38].

Problemet med att inte alla punkter finns representerade för sökningen är även det gemensamt med cellrepresentationen eftersom även cellernas upplösning vanligtvis har ett minimivärde. En avvägning mellan upplösning och beräkningstid är självklar men ofta är det irrelevant att diskutera extremt hög upplösning såvida systemet inte kräver mycket hög precision. [17]



Figur 7 Sammanfattning av fyra vanliga metoder för abstrahering. A celler, B polygoner, C quad-träd och D skelettisering.

3.3.3 Specialfall

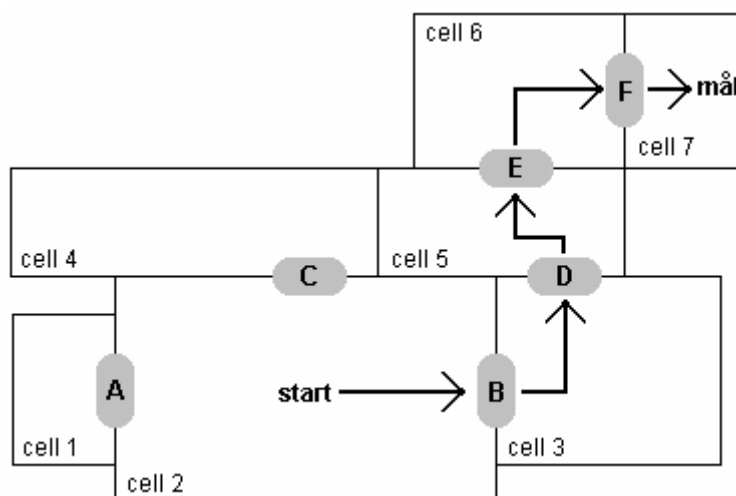
Ofta skiljer sig omgivningarna som agenten ska navigera genom avsevärt mellan olika områden. Ett sådant exempel är skillnaden på inom- och utomhusmiljö. Utomhusmiljöer karakteriseras ofta av stora ytor med samma egenskaper och spridda hinder. Inomhus är världen å andra sidan ofta uppbyggd av rum med trånga öppningar emellan. Det är stor skillnad på att navigera i en labyrint mot i öppen terräng. Då agenter ska ha förmågan att

röra sig både inomhus och utomhus kan problemet delas och lösas av två olika algoritmer, varje specialiserad på sitt område. Ett nytt problem blir att avgöra när agenten befinner sig i respektive miljö. Väljer man istället att använda samma algoritm i då agentens omgivning skiljer sig markant kan algoritmens effektivitet bli lidande.

3.3.4 Abstraktionslager

Sökrymden växer ofta mycket snabbt i förhållande till de parametrar som sökningen bygger på. Till exempel är antalet noder i sökningen kvadratisk mot storleken (antalet celler) på varje sida i ett rutnät. När världens storlek växer ökar alltså antalet noder som ska avsökas och vid någon storlek kommer sökningen kräva för mycket tid. Då kan man istället för att detaljplanera hela vägen göra en grovplanering [11]. När en människa ska förflytta sig långa sträckor tänker hon knappast på var varje fotsteg ska sättas utan snarare i form av delmål på vägen, tex först tunnelbanestationen, därefter parken och sedan mataffären. På så sätt får vi en ungefärlig bild av vår färdväg. Om en agent kan söka på samma sätt är det möjligt att söka långa vägar och samtidigt undvika fall där extremt många noder behöver genomsökas för att tillsist resultera i en misslyckad sökning [11]. Istället för att direkt söka i den finaste upplösningen av världen kan en sökning i en annan representation av världen göras; en sökningen i ett annat abstraktionslager. Någon speciell parameter eller egenskap i världen används för att skapa den nya representationen. Tar man exempelvis enbart hänsyn till avståndet mellan agent och mål går det enkelt att skapa ett abstrakt lager genom att längs linjen mellan agent och mål lägga in ett antal delmål [11]. Mellan varje delmål uppstår då ett mindre delproblem [10]. Om varje delproblem löses separat kan den genomsnittliga tidsåtgången bli mindre. Dessvärre finns ingen garanti för att agenten kan nå delmålen och lösningen är inte heller garanterat optimal [11].

Ett annat tillvägagångssätt är att förbehandla världen på något sätt och ur denna skapa upp lämpliga delmål eller regioner som används som noder i det abstrakta söklagret [11]. Detta tillvägagångssätt liknar metoden för att skapa en karta av godtyckliga polygoner. Skillnaden är egentligen bara att de olika lagren måste ha någon speciell egenskap, exempelvis lägre upplösning eller nya parametrar som påverkar sökningen. För att göra det lite enklare går det att använda rektanglar av en fix storlek som fält istället för godtyckliga polygoner [10]. Problemet blir hur man ska värdera varje fält, ofta är det svårt att uppskatta avståndet till målet. En lösning är något sorts medelvärde av alla faktorer som påverkar. Det kan dock kräva onödigt mycket beräkning.



Figur 8 Sökning i en värld med cell-portal-system.

Ett exempel på en abstrahering som bygger på delmål och regioner är det i kapitel 3.3 beskrivna cell-portal-systemet. Eftersom de olika cellerna i sig representerar skilda områden i världen kan dessa vara utmärkta att använda som noder i en graf där en sökning görs. Bågarna i grafen motsvarar då portalerna mellan cellerna. Representationen med celler och portaler är inte en fullständig beskrivning av världen utan endast en grov skiss. För att få en komplett beskrivning krävs mer detaljerade representationer av varje cell i sig. I Figur 8 visas hur en vägsökning i en värld med ett cell-portal-system kan gå till. Ur en graf med celler som noder och portaler som bågar fås en kedja av bågar (portaler) från start till målpunkten, i detta fall B, D, E, F. Därefter sker en sökning i den nuvarande cellen (cell 2) från startpunkten till portal B, sedan i cell 3 från portal B till D etc. Fördelar med detta tillvägagångssätt är att mindre delar av världen behandlas (cell 1 och 4 behöver inte abstraheras) och att sökvägen delas upp i mindre etapper som förhoppningsvis ger en lägre total söktid.

3.4 Sökning

Då agentens omgivning representeras av diskreta noder kan vägen från sökningens startnod S till målnoden G beskrivas av en graf. Genom att i denna graf söka den billigaste vägen från S till G fås en lösning till vägsökningsproblemet. Idag finns en stor mängd algoritmer för att hitta den billigaste vägen i en graf.

Några begrepp är gemensamma för flertalet sökmetoder. En första distinktion är huruvida lösningen ska vara optimal eller endast tillfredsställa en uppsättning villkor. Att finna en optimal lösning är ofta betydligt svårare. Ett alternativ är semioptimala lösningar. Semioptimala lösningar anses uppfylla då en funnen lösning ligger tillräckligt nära den optimala lösningen. I de flesta fall måste en avvägning göras mellan optimalitet och söktid varför semioptimala lösningar är vanliga då dessa underlättar avvägningen. Problemet med semioptimala lösningar är att garantera att dessa inte är *för* dåliga samtidigt som de inte får ta *för* mycket beräkningstid i anspråk. [1]

Ytterligare tre begrepp som är grundläggande för alla sökmetoder är kompletthet, effektivitet och uteslutning. En komplett sökmetod returnerar alltid en lösning om en sådan existerar. Med effektivitet menas i detta fall att varje nod endast undersöks en gång. Uteslutning (eng *pruning*) innebär att en mängd noder eller spår i lösningsgången inte behöver behandlas eftersom det på något sätt går att inse att dessa aldrig kommer att producera den sökta lösningen. [1]

Då en sökning pågår behandlas noder som representerar delar av lösningen. En nod skapas genom att data specifikt för den noden beräknas utifrån de förutsättningar som fanns i den föregående noden, föräldern. Den nya noden sägs då vara *genererad* och föräldern *utforskad*. Då en förälders samtliga efterföljande noder är genererade sägs föräldern vara *expanderad*. De genererade noderna får ofta en pekare tillbaka till sin förälder för att sökningens väg ska kunna återskapas då målnoden är funnen, så kallade bakåtppekare (eng *backpointer*). Under sökningen kan noderna delas in i fyra disjunkta kategorier:

- 1) Noder som har expanderats.
- 2) Noder som har utforskats men ännu inte expanderats.
- 3) Noder som har genererats men ännu inte utforskade.
- 4) Noder som inte är genererade.

Framförallt är distinktionen mellan grupp ett och tre viktig. I många algoritmer motsvaras grupp ett, expanderade noder, av en lista av noder som kallas stängda medan tillstånd hörande till grupp tre, utforskade noder, motsvaras av en lista med noder som kallas öppna. Detta brukar implementeras med två mängder med namnen CLOSED och OPEN. [1]

Sökalgoritmer kan delas in i två klasser beroende på vilka tidskrav de är anpassade för. Off-line algoritmer får en förfrågan och utför därefter en fullständig beräkning innan ett resultat returneras. En On-line algoritmer kan å andra sidan interagera med omvärlden under körning; ge delresultat vid godtyckliga tidpunkter och anpassa parametrar till ny indata. On-line algoritmer är oftast enkla eftersom de måste vara anpassade för att ge resultat i realtid. [35]

Rena realtidsalgoritmer är en underklass till on-line algoritmer som förutom möjligheten att dynamiskt ta hänsyn till omgivningen endast beräknar det nästkommande tillståndet i sökningen.

3.4.1 Definitioner för sökalgoritmerna

I de följande styckena kommer flera nya funktioner att presenteras. De är förklarade där de introduceras men samlas här för att kunna användas som referens.

- S = sökningens startnod.
- G = sökningens målnod.
- $g^*(X)$ = den lägsta kostnaden för alla vägar mellan startnoden S och någon nod X .
- $h^*(X)$ = den lägsta kostnaden för alla vägar mellan någon nod X och målnoden G .
- C^* = den lägsta kostnaden för alla vägar mellan S och G dvs $h^*(S)$.
- $g(X)$ = den hittills ackumulerade kostnaden till nod X där $g(S) = 0$.
- $h(X)$ = en uppskattning av $h^*(X)$ så att $h(G) = 0$.
- $f(X)$ = nodvalsfunktionen som är summan av $g(X)$ och $h(X)$.
- $k(G, X)$ = minimum av $h(X)$ vid dynamisk omgivning

3.4.2 Blinda sökmetoder

Blinda sökmetoder kräver ingen information om omgivningen. Inte ens målnodens placering spelar någon roll för ordningen som noderna expanderas i, förutom då termineringsvillkoret är uppfyllt och sökningen avbryts [1]. Dessa metoder är ofta ineffektiva och används främst som jämförelse mot andra mer komplicerade algoritmer. Det finns dock undantag då blinda sökmetoder faktiskt kan vara värda att ha i åtanke. Ett exempel är då antalet noder är litet, i det fallet är det möjligt att tid går förlorad om en mer komplicerad sökmetod används på grund av dess merkostnad.

Bredd-först-sökning (BFS) expanderar noder i förhållande till avståndet till startnoden. Först expanderas startnoden, därefter alla barn till startnoden, sedan alla barnbarn etc. Detta upprepas tills målnoden hittas bland någon av de expanderade nodernas barn [1, 2]. BFS använder en FIFO-policy (*first-in-first-out*) för att välja vilka noder från OPEN-listan som ska expanderas, alltså är en kö en lämplig datastruktur för implementering [1, 3]. BFS ger en optimal lösning, dvs den billigaste lösningen om någon lösning existerar. Priset för detta är hög minnes- och tidsåtgång då hela den avsökte grafen måste lagras och bearbetas.

Djup-först-sökning (DFS) väljer, till skillnad från BFS, att i första hand expandera den nod som ligger så långt som möjligt från startnoden. Först när en väg inte går att utforska längre går algoritmen tillbaka och väljer en nod närmare startnoden och fortsätter därifrån. En stor risk med DFS är att långa vägar som inte leder någonstans utforskas helt i onödan, därför brukar man ofta införa en gräns för hur djupt en väg får sökas innan en alternativ väg testas. DFS använder en LIFO-policy (*last-in-first-out*) för att välja vilken nod från OPEN-listan som ska expanderas. För detta ändamål är en stack en användbar datastruktur. [1]

En förbättring av ovan beskrivna blinda sökmetoder är Dijkstras algoritmen. I Dijkstras algoritmen väljs den nod ur OPEN-listan som har lägst kostnad, där kostnaden för varje nod definieras som totalkostnaden för vägen från startnoden till den aktuella noden. Kostnaden mellan två noder X och Y definieras av kostnadsfunktionen $c(X,Y)$. Kostnadsfunktionen kan till exempel återspegla avståndet mellan noderna i grafen eller hur mycket drivmedel som går åt för förflyttningen. Noderna som lagras i OPEN-listan sorteras sedan efter den totala kostnaden till noden. Totalkostnaden, $g(X)$, är summan av varje bågkostnad från startnoden till den aktuella noden. I det fall då samtliga bågkostnader är lika blir Dijkstra en BFS sökning men med varierande bågkostnader förbättras uteslutningen betydligt [1].

3.4.3 En heuristisk förbättring – A*

Dijkstras algoritmen som ger de lägsta kostnaderna till alla andra noder från startnoden känns intuitivt ineffektiv om det enda i lösningen som är intressant är kostnaden mellan två specifika noder. En första förbättring är att avbryta algoritmen så fort en billigasteväg hittats till målnoden G . Det besparar mycket arbete men är fortfarande inte tillfredsställande vid en jämförelse med en sökning i ett praktiskt exempel. Dijkstra blåser upp en bubbla runt startnoden med en radie av konstant kostnad. Den första effektiviseringen spräcker bubblan när den når målnoden men den tar ingen hänsyn till ”riktningen” mellan start och mål. [1]

A*-algoritmen inför denna riktning med hjälp av en heuristik och så gott som alla mer avancerade sökmetoder har A*-algoritmen som grund. En heuristik är en tumregel som avgör vilket av flera alternativ som är mest lovande för att uppnå ett bestämt mål. Normalt är en sådan regel en avvägning mellan enkelhet och en korrekt särskiljning i bra och dåliga alternativ. [35]

Den enda praktiska skillnaden mellan Dijkstra och A* är i vilken ordning noderna expanderas, dvs hur de är ordnade i OPEN-listan. Istället för att sortera på $g(X)$ introduceras en ny kostnadsfunktion $f(X) = g(X) + h(X)$ där $h(X)$ är en heuristikfunktion som utnyttjar den information som finns tillgänglig om världen. Om $h(X)$ är en optimistisk uppskattning av kostnaden mellan X och G , dvs $h(X)$ är mindre än eller lika med den faktiska kostnaden för att nå G , så kommer A* att producera en optimal lösning. Eftersom det bara är ordningen i OPEN-listan som ändras så är komplexiteten i värsta fallet densamma som för Dijkstras algoritmen men den ger en bättre medelprestation om det går att koppla en lämplig heuristikfunktion till problemet. [1]

En översikt av A*-algoritmen:

```

Generera startnoden och lägg till nodlistan

SÅ LÄNGE det finns fler noder på kön

    Hämta ut den billigaste noden på nodlistan

    OM noden är målet SÅ avbryt

    Generera nodens grannar och sortera in dem på nodlistan
    enligt  $f(X) = g(X) + h(X)$ 

    OM den sista noden var målnoden SÅ har algoritmen lyckats

```

Den bästa heuristiken är vanligen "avståndet från X till G" [35]. Ett krav som ofta ställs på heuristikfunktionen är att den ska underskatta avståndet till målet. En underskattande (eng *admissible*) heuristik garanterar fullständighet och optimalitet för A*. Vad som är ett lämpligt mått på avståndet beror på grafstrukturen och på avsökningsmetoden. Det euklidiska avståndet är lägre än antalet noder som måste besökas. Om ett rutnät används där diagonal förflyttning är otillåten det beräkningsmässigt fördelaktigt att använda ett viktat Manhattan-avstånd ($\Delta X + \Delta Y$) eftersom den beräkningstunga rotfunktionen som krävs för att beräkna euklidiska avstånd undviks. [1, 35]

Heuristikfunktionen $h(X)$ behöver inte beräknas förrän sökningen genererar nod X. Den ökade beräkningstiden för att beräkna heuristiken blir i praktiken liten eftersom bara ett fåtal noder vanligen måste besökas. [1]

Vid de flesta problem kan man använda olika heuristiker. Ett populärt exempel är 8-pusslet där åtta brickor ligger i en 3 x 3-matris och problemlösaren ska återskapa ett mönster på så få förflyttningar som möjligt. Ett alternativ till heuristikfunktion är att räkna antalet rutor som inte ligger på sin rätta plats och kalla resultatet h_1 , ett annat är det sammanlagda Manhattan-avståndet för alla brickorna, h_2 . En heuristik som ger ett högre resultat lägger h närmare h^* och ger en snabbare sökning. I exemplet med 8-pusslet ger h_2 aldrig lägre värden än h_1 och kommer därför att ge en effektivare sökning. [1]

Om terrängkostnaderna varierar över ett stort spektrum måste den lägsta kostnaden väljas för att få en undre uppskattning som uppfyller kravet på A*:s heuristikfunktion. Det betyder att A* bara kommer att ge en liten förbättring jämfört med Dijkstra. Genom att införa en konstant b vid kostnadsberäkningen, $f(X) = g(X) + b * h(X)$, går det att styra hur A* ska bete sig:

- $b = 0$ => Dijkstra
- $b = 1$ => Standard A*
- $b > 1$ => Viktad A*

Ett större b ger en lösning som går rakare mot målet men garantin för en optimal lösning förloras. [1, 35]

Garanterad terminering

A* terminerar alltid i en ändligt stor graf. Anledningen till detta är att varje sådan graf innehåller ett ändligt antal ickecykliska vägar och att vid varje nodexpansion läggs en ny länk till algoritmens sökträd. Varje länk representerar en ickecyklisk väg och till slut tar antalet möjliga vägar slut. Även noder som expanderas fler än en gång representerar nya vägar eftersom de måste vara strikt billigare för att läggas tillbaka på OPEN-listan. [1]

Fullständig

A* är också komplett på ändliga grafer. Algoritmen misslyckas endast när OPEN-listan blir tom och om en väg från S till G existerar så kan inte OPEN-listan bli tom innan denna har hittats. Det motsäger antagandet att S ligger på en lösningsväg eftersom varje nod som ligger på en lösningsväg också har en granne som ligger på samma väg. [1, 35]

Optimal lösning

En heuristisk funktion sägs vara underskattande om $h(X) \leq h^*(X)$ för alla X. En A*-algoritm som använder sig av en sådan funktion kan också visas ge optimala lösningar. Anta att en A*-sökning terminerar med en målnod för vilken $f(G) = g(G) > C^*$. A* undersöker termineringsvillkoret efter det att någon nod valts för expansion. Alltså uppfyllde g när den valdes för expansion $f(G) \leq f(X)$ för alla X i OPEN-listan. Det innebär att $f(X) \geq C^*$ gäller för alla noder i OPEN-listan. Det strider mot kravet att den heuristiska funktionen måste vara en optimistisk uppskattning av kostnaden till målnoden. Alltså måste $g(G) = C^*$ och A* returnerar en optimal väg. [1, 35]

Uteslutningsgrad

Kraften av den heuristiska uppskattningen $h(X)$ mäts över hur mycket uteslutning den orsakar och är beroende av tillförlitligheten hos den heuristiska uppskattningen. Om h är en exakt uppskattning av kostnaden till målet så kommer A* alltid att expandera optimala noder. Om istället h sätts till 0 blir bäst-först sökningen till en bredd-först sökning. De heuristiker som praktiskt används hamnar någonstans mittemellan dessa två. [1]

Komplexitet

För de flesta problem är antalet noder som behandlas av sökningen exponentiellt i förhållande till sökningens längd. Den exponentiella tillväxten uppstår såvida felet i den heuristiska uppskattningen inte växer fortare än logaritmen av det verkliga avståndet, dvs $|h(X) - h^*(X)| \leq O(\log^*(X))$. För de flesta praktiskt användbara heuristiker är felet åtminstone proportionellt mot det verkliga avståndet. [35]

3.4.4 IDA*

Det kan vara svårt att förutsäga tids- och minnesåtgången för en godtycklig A*-sökning. Om det är stora tidsskillnader mellan olika sökningar och det finns krav på att kunna styra beteendet kan *Iterative Deepening A** (IDA*) användas. Ett nytt brytvillkor läggs till den vanliga A*-algoritmen, ett maximalt sök djup som avbryter sökningen i förtid. Sök djupet kan styras av ett schemalägningsprogram. De sökningar som visar sig kräva ett stort sök djup kan ges lägre prioritet så de inte blockerar vägsökaren. [25, 35]

En intressant aspekt med IDA* är att man kan använda en djup-först-strategi vid nodexpansionen utan att riskera att fastna i oändliga cykler som vid vanlig DFS.

3.4.5 Sökning i dynamiska omgivningar

Ovan beskrivna heuristiska sökmetoder fungerar vanligtvis bra då agentens omgivning är statisk men när omgivningen förändras uppstår ett problem: Den beräknade vägen kan bli dyrare eller till och med ogiltig på grund av förändringen. Förändringar kan vara av två typer, antingen förändras målets position eller så förändras kostnaderna för att ta sig dit. Då målets position förflyttas är garanterat den beräknade vägen felaktig, eftersom en fullständig väg saknas. Däremot kan fortfarande den beräknade vägen vara den bästa när kostnader förändras i agentens omgivning. Kostnaderna kan förändras på olika sätt; på grund av rörliga spelare eller andra agenter, genom rörliga hinder, genom försvinnande eller nyskapade hinder och genom att agenten själv kan manipulera omgivningen [4]. Då är det ändå ofta nödvändigt att genomföra en ny sökning för att visa att den befintliga vägen fortfarande är bäst, om man vill garantera optimalitet.

Den enklaste metoden för att hantera dynamiska omgivningar är att helt enkelt planera en väg från start till mål baserat på den tillgängliga informationen och när en förändring detekteras beräknas en helt ny väg [5]. Andra metoder går ut på *explore-exploit*-strategier där agenten dels söker av omgivningen och dels använder nyupptäckt information för planering [5]. Då måste en avvägning göras för hur resurserna ska fördelas mellan att utforska och avsöka omgivningen. Eftersom en omplanering kan omfatta ett stort antal noder och kräver tid därefter blir det ohanterligt att använda denna metod i fall då omgivningen förändras oftare än tidsåtgången för en planering. För att det överhuvudtaget ska vara möjligt att göra omplanering måste följande kriterium vara uppfyllt:

$$tid_{plan} * antal_agenter + tid_{övrigt} < tid_{förändring}$$

tid_{plan} anger tidsåtgången för ett pass av planeringsalgoritmen.

$tid_{övrigt}$ anger tidsåtgången för alla beräkningar som inte ingår i planeringen.

$tid_{förändring}$ anger tiden mellan två förändringar.

För att minska tidsåtgången vid omplanering (tid_{plan}) kan olika optimeringstekniker användas. Exempelvis kan delar av den tidigare planerade vägen återanvändas. Sådana metoder medför ofta att den nya vägen inte kan garanteras vara optimal.

En annan metod för att hantera dynamiska omgivningar är prediktion i samband med sökningen. Då planeringen genomförs predikteras de förändringar som sker under agentens förflyttning mot målet. Sedan genomförs planeringen med hänsyn till dessa. Prediktion medför ingen garanti för att omplanering inte behövs men det kan minska frekvensen av onödiga sådana. Dessutom kräver prediktionen i sig processortid vilket medför att en avancerad sådan som kräver mycket processortid i realiteten bara slukar kraft som lika gärna kunde användas av sökningen. Prediktion förutsätter också att förändringarna i världen är deterministiska till viss del.

Fullständig omplanering är i allmänhet för kostnadskrävande och prediktion ger inga garantier för att omplanering inte krävs trots allt. Därför fordras ofta mer förfinade metoder avsedda just för dynamisk sökning. Två alternativ finns; en omplanerare som är mer effektiv än en fullständig omplanerare eller realtidsökning. Båda dessa metoder kan klassificeras som on-line algoritmer eftersom de dynamiskt ska kunna anpassa sin returdata under exekveringens gång. En effektivare omplanerare återanvänder information för att hitta den nya vägen snabbare. Realtidsalgoritmer planerar endast vilken som är nästa nod i sökningen, till skillnad från icke-realtidsalgoritmer (Dijkstra, A*

etc) som beräknar en fullständig lösning innan det första steget i planen exekveras. När en realtidsalgoritm valt en nästa nod exekverar agenten steget och sedan beräknas nästa steg osv.

*Dynamiska heuristiska metoder D**

En sökalgoritm specialiserad för omgivningar som är helt eller delvis okända för agenten är D* (*Dynamic A**) [5, 28, 40]. De metoder som beräknar en ny komplett väg varje gång omgivningen förändras är visserligen vanligtvis optimala i det avseende att de hittar den billigaste vägen men samtidigt mycket ineffektiva [5, 40]. D* är funktionellt likvärdig med dessa metoder men betydligt mer effektiv vid dynamiska omgivningar då omplanering krävs [5, 40]. D* är ingen realtidsalgoritm utan anpassar den beräknade vägen först då en diskrepans upptäcks i agentens färdväg. Däremot är det ibland möjligt att använda D* i applikationer som fungerar i realtid eftersom uppdateringar sker mycket snabbt [8].

D* är en vidareutveckling av A* där kostnaderna mellan noderna kan förändras under lösningsgången. Problemet definieras på samma sätt som för A*, med noder, bågar, start- och målnoder, kostnadsfunktioner etc. För varje nod X har D* en heuristisk uppskattning av kostnaden från X till G givet av funktionen $h(X)$. Uppskattningen ska vara så nära den verkliga kostnaden som möjligt. Utforskade men genererade noder finns på OPEN-listan som också används för att propagera information om förändringar i omgivningen mellan noder och beräkna vägstoknader. En parameter t i varje nod anger om det är NYTT (det har aldrig varit på OPEN-listan), ÖPPET (är just nu på OPEN-listan) eller STÄNGT (är inte längre på OPEN-listan). För varje nod på OPEN-listan finns en funktion $k(G, X)$ som definieras som minimum av $h(X)$ före kostnadsförändring och alla värden $h(X)$ antagit sedan X placerades på OPEN-listan. Med hjälp av $k(G, X)$ delas alla noder på OPEN-listan in i kategorierna HÖJ ($k(G, X) < h(X)$) eller SÄNK ($k(G, X) = h(X)$). Fallet $k(G, X) > h(X)$ blir aldrig aktuellt eftersom $k(G, X)$ är lika med det minsta $h(X)$ värdet hittills. Indelningen används för att sprida kostnadsförändringar i OPEN-listan på så sätt att HÖJ betyder att en kostnadsökning skett och SÄNK betyder kostnadsminskning. Vidarebefordringen av kostnadsförändringar sker när kostnaden för en nod som hämtas ur OPEN-listan och expanderas inte stämmer med agentens data. [5]

För att veta vilken nod som ska behandlas av algoritmen sorteras de i OPEN-listan efter en parameter k_{\min} . Denna definieras som $\min(k(x))$ för alla X som har $t = \text{ÖPPET}$, dvs kostnaden för den nod på OPEN-listan med minsta nuvarande eller tidigare beräknade kostnad. Vägar med en kostnad mindre än k_{\min} är optimala medan de med lika eller större kostnad inte behöver vara det. Förutom k_{\min} finns en parameter k_{old} som är lika med det värde k_{\min} hade innan den senaste noden plockades från OPEN-listan. Liksom i A* har noderna bakåtppekare för att den beräknade vägen ska vara tillgänglig för agenten. [5]

I huvudsak består D* av två delar: En del för att beräkna den optimala vägen till målet och en del för att sprida kostnadsförändringar och placera påverkade tillstånd på OPEN-listan. Från början är t satt till NYTT för alla noder och S placeras på OPEN-listan. Därefter beräknas den optimala vägen från start till mål som i A* men med tillägget att noderna i OPEN-listan är sorterade efter $k(G, X)$ och att ovan beskrivna parametrar uppdateras. Då en väg hittats, $t(X) = \text{STÄNGT}$ där X = agentens nod, följer agenten bakåtppekarna tills den antingen når målet eller upptäcker ett fel i $c()$, tex beroende på ett tidigare okänt hinder. Då ett hinder, eller en högre kostnad, upptäcks är de optimala vägar som innehåller bågen med den förändrade kostnaden felaktiga. Bågstoknaden uppdateras och den efterföljande noden placeras på OPEN-listan. Därefter expanderas noden på OPEN-listan för att sprida den nya kostnaden längs de felaktiga vägarna. De

noder som vidarebefordrar kostnadsökningen är HÖJ-noder medan de noder som kan sänka sin kostnad är SÄNK-noder som placeras på OPEN-listan. Genom upprepad expansion ökas HÖJ-nodernas kostnader. I motsats uppdateras kostnaden med ett lägre värde om en lättad upptäcks, tex i det fall som ett område såg ut att vara blockerat inte var det. Nu kan en ny optimal väg finnas. Den efterföljande noden markeras som SÄNK och läggs till OPEN-listan och genom upprepade expansioner av noderna på OPEN-listan beräknas nya optimala vägar från de berörda noderna. Därefter fortsätter agenten att följa den nodkedjan, som kan ha förändrats på grund av kostandsförändringen. [5, 28, 31]

D^* är optimal då $k_{\min} \geq h(X)$ [5]. D^* är fullständig om sökrymden innehåller ett ändligt antal noder [5]. Experiment som genomförts med en stokastisk sökrymd visar att D^* framförallt vinner tid mot fullständiga omplanerare då stora områden ska avsökas [5, 31]. Detta är naturligt då D^* inte tvingas till den omfattande uppgiften att beräkna en helt ny väg då ett hinder upptäcks utan bara modifierar den befintliga från hindret. D^* är också mest effektiv då förändringar upptäcks nära agentens position [5]. Detta beror på att förändringar nära agenten ger mindre förändringar i sökrymden. Komplexiteten för D^* är i bästa fall densamma som för A^* . Då fel i agentens information om omgivningen upptäcks krävs ytterligare beräkningar. Hur omfattande dessa beräkningar är beror på hur många fel som upptäcks. Alltså är tidskomplexiteten för D^* beroende av hur många uppdateringar som krävs.

Learning RealTime A^ (LRTA*)*

Grunden i LRTA* är en tabell med uppskattningar av avståndet mellan varje nod och målnoden. Avståndet för nod X som lagrats i tabellen fås med funktionen $a(X)$. Från början bygger uppskattningarna i tabellen på en heuristikfunktion, $h(X)$, som förutsetts ge en undre gräns för det verkliga avståndet. Genom att agenten upptäcker omgivningen uppdateras uppskattningarna i tabellen tills de konvergerar mot de exakta avstånden. LRTA* är en realtidsalgoritm.

Om X är det nuvarande tillståndet, $a(X)$ det uppskattade avståndet och $c(X, Y)$ är bågkostnaden mellan grannarna X och Y kan LRTA* sammanfattas i följande tre steg:

- 1) Beräkna $f(Y) = a(Y) + c(X, Y)$ för varje granne Y till X .
- 2) Gå till grannen med det lägsta $f(Y)$.
- 3) Uppdatera $a(X)$ till det lägsta $f(Y)$.

Då $a(X)$ uppdateras med $f(Y)$ fås en ny undre gräns för avståndet mellan X och målet, genom en av grannarna till X . Den kortaste vägen från X till målet måste vara åtminstone så stort som det minsta av dessa uppskattningar eftersom vägen till målet måste gå genom en granne. En viktig iakttagelse är att LRTA* endast utnyttjar kostnaden till grannar och en heuristikfunktion, alltså krävs endast information om grannarna och heuristiken. LRTA* är fullständig om det finns en väg från varje nod X till målet. LRTA* har ingen garanti för optimala lösningar, däremot kommer heuristikfunktionen konvergera mot de faktiska kostnaderna efter ett stort antal sökningar och varje sökning därefter kommer att ge en optimal lösning. Så fort en kostnad förändras är heuristiktabellen felaktig och för att få en optimal väg måste återigen ett stort antal sökningar göras för att uppdatera tabellen.

Moving-Target Search (MTS)

En speciell metod som bygger på LRTA* och heter *Moving-Target Search* har tagits fram för vägsökning till ett rörligt mål [7]. Metoden går i korthet ut på att agenten försöker skapa en tabell med exakta avstånd mellan alla noder, ungefär som i LRTA*. MTS garanterar att agenten når fram till målet om det rör sig långsammare än agenten, antingen genom en lägre hastighet eller för att målet ibland gör misstag och inte förflyttar sig bort från agenten. Minneskomplexiteten är i värsta fall $O(n^2)$, där n är antalet noder i sökrymden, då heuristikvärden för alla tillståndskombinationer behöver sparas. I normalfallet är den betydligt lägre och algoritmen kan dessutom förfinas så att bara förändringar av heuristiken behöver sparas, vilket minskar den normala minnesåtgången ytterligare. Tidskomplexiteten är $O(n^3)$, i värsta fall. [7]

I de praktiska testerna som utvärderat algoritmen rör sig agenten och målet i turordning. För att kompensera hastighetsskillnaden står den långsammare ibland över sin tur. En heuristikfunktion, som måste vara underskattande, ger ett uppskattat avstånd mellan två valfria tillstånd. Algoritmen bygger på en tabell eller matris med heuristikvärden som uppdateras efterhand. Från början fylls tabellen med värden från en heuristikfunktion. I stort finns två olika händelser i algoritmen; att agenten eller målet rör sig. Då agenten och målet har samma position avslutas algoritmen. När det är agentens tur att röra sig från tillståndet X beräknas $f(Y) = a(Y, G) + 1$ för varje granne Y till X där G är tillståndet målet befinner sig i. Istället för en kostnadsfunktion använder sig MTS av en fast kostnad med värdet 1. Av samma anledning som i LRTA* uppdateras $a(X, G)$ med $\min(f(Y) \forall Y)$ om $a(X, G) < f(Y)$. Då målet rör sig från G till G' observerar agenten förflyttningen och $a(X, G')$ beräknas som jämförs med $a(X, G)$ där X är agentens tillstånd. Om $a(X, G) < a(X, G') - 1$ sätts $a(X, G)$ till $a(X, G') - 1$. Denna uppdatering görs eftersom då avståndet mellan G och G' maximalt är ett så är avståndet till G åtminstone så stort som avståndet till G' minus ett. Slutligen uppdateras måltillståndet till G' . [7]

Kostnaden att förflytta sig mellan två noder är i grundbeskrivningen av MTS satt till ett. Det är möjligt att, som i LRTA*, använda en godtycklig kostnadsfunktion $c(X, Y)$ även för MTS.

I [7], [9] och [10] ges en rad förslag på förbättringar av MTS. Exempelvis kan man tillåta längre intervall mellan uppdateringen av målets tillstånd, dvs ett mindre dynamiskt mål, eller blanda realtidssökning med viss del off-line-sökning. Med dessa tillägg undviks situationer där agenten långa perioder försöker hitta vägen ut ur lokala maxima i heuristikfunktionen. Ett problem med MTS är att ett tillstånd kan avsökas flera gånger av agenten.

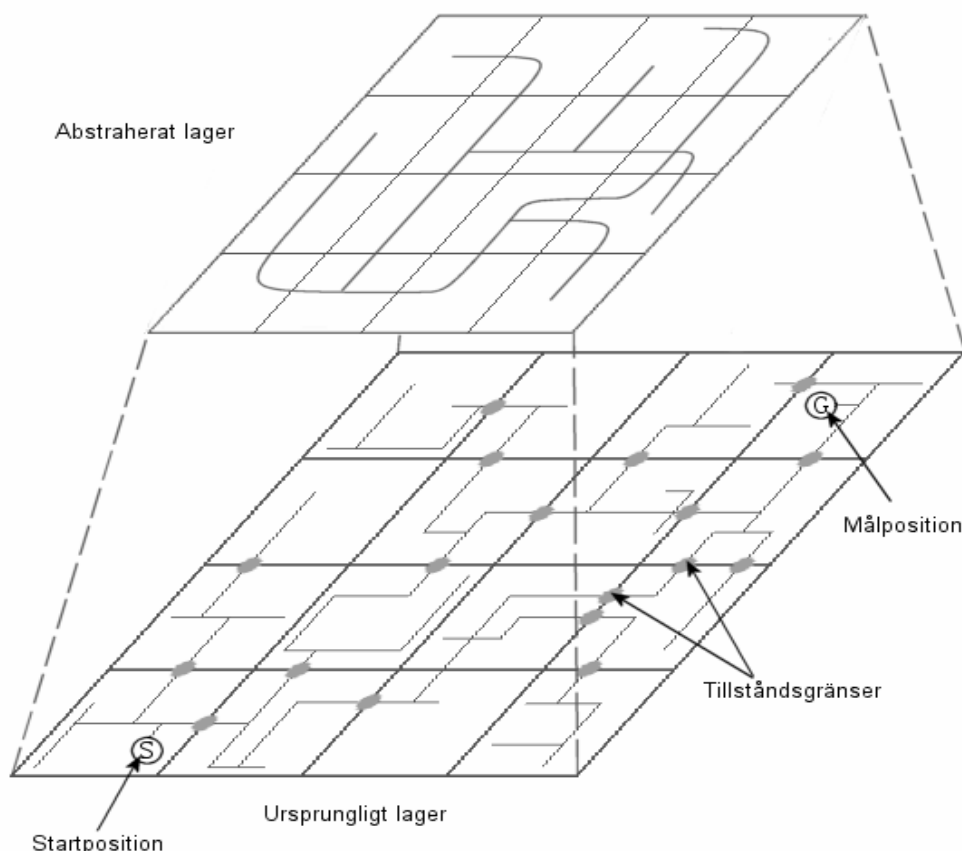
Märkning

En dynamisk metod för att dela in världen i områden kallas märkning (eng *marking*). Märkning innebär att tillstånd ges ett värde i som betyder att det är onödigt för agenten att förflytta sig till det tillståndet såvida inte även målet befinner sig i ett tillstånd med märkning i . Under sökningen förflyttar sig agenten endast till de tillstånd som är omärkta eller märkta med samma värde som målnoden. Det innebär att en rad tillstånd kan uteslutas ur sökprocessen. Experiment med märkning i kombination med realtidssökning (*Moving Target Search* och *Forgetfull Depth-First Search*) visar att söktiden kan kortas i vissa fall, men inte med någon signifikant skillnad. Däremot blir den genomsnittliga sträckan som agenten tillryggalägger mellan start och mål betydligt mindre jämfört med de andra algoritmerna. Anledningen till att sträckan kan variera är att dessa realtidialgoritmer inte garanterar en optimal väg. [9]

3.4.6 Sökning i flera lager

Under kapitlet Abstraktionslager beskrevs tidigare hur olika lager för sökning kunde skapas utifrån agentens kunskaper om världen. Hur dessa abstraktioner påverkar egenskaperna hos A* eller andra sökmetoder är en annan fråga. Med en abstraktion menas att ett söklager kompletteras med ett abstrakt lager som transformerar sökrymden. Resultaten från sökningen i det abstrakta lagret används sedan för att underlätta sökningen i det ursprungliga lagret [11]. Åtminstone två olika typer av abstraktioner är möjliga; den som går ut på att skapa flera mindre delproblem av det ursprungliga problemet och den som förbättrar heuristikfunktionen genom att använda resultaten från det abstrakta lagret som hela eller en del av heuristikfunktionen.

Ett exempel på hur delproblem skapas ges i [10] där en abstrakt karta partitioneras i ett antal delproblem. Varje delproblem har ett antal gränsnoder som bildar länkar med andra delproblem. Sökningen sker i två olika kartor; en lokal karta som innehåller information om motsvarande delproblem samt en abstrakt karta som visar hur de olika delproblemen länkas samman, dvs gränsnoderna. Den väg som hittas i den abstrakta kartan delas efter gränsnoderna upp i delproblem (se Figur 9). Sökalgoritmen som används i [10] är en variant av MTS och de resultat som artikeln ger går endast att knyta till denna algoritm. Praktiska försök visar att en abstrakt karta framförallt minskar tidsåtgången för underhåll av kartan [10].



Figur 9 Ett abstraktionslager genererat utifrån det ursprungliga lagret.

Då abstrakta lager används för att förbättra heuristikfunktionen är det grundläggande kravet att tiden man sparar i sökningen med en förbättrad heuristikfunktion inte får vara

större än tiden det tar att genom det abstrakta lagret beräkna den nya heuristikfunktionen [21]. Ofta är det enklast att analysera hur bra en abstraktion är genom att undersöka hur många tillstånd som expanderas totalt (i samtliga lager) jämfört med hur många tillstånd som expanderas då endast det ursprungliga lagret används. I A* måste ett heuristikvärde beräknas varje gång ett tillstånd ska läggas till i OPEN-listan. Det kan göras genom att söka i det abstrakta lagret med ett starttillstånd som motsvarar det nuvarande tillståndet och ett måltillstånd som motsvarar det ursprungliga måltillståndet. Då en väg hittas i det abstrakta lagret finns även det abstrakta avståndet mellan mål- och starttillstånden. Detta avstånd används tillsammans med andra eventuella faktorer för att beräkna det slutgiltiga heuristikvärdet för det nuvarande tillståndet. Det är uppenbart att varje beräkning av ett heuristikvärde implicit innebär att en sökning i det abstrakta lagret måste göras. Dessa värden bör lagras för att undvika dubbla beräkningar. Specifikt för vägsökning är det tänkbart att vikta samman resultatet från den abstrakta sökningen med det euklidiska avståndet från det aktuella tillståndet till måltillståndet för att få den slutgiltiga heuristikfunktionen. På detta sätt kan heuristikfunktionen styra agenten runt hinder, istället för att A* ska expandera tillstånden och upptäcka att de är blockerade.

För att skapa abstrakta tillstånd grupperas de ursprungliga tillstånden samman. En sådan abstraktion kallas *homomorfism* [21]. Om det ursprungliga lagret saknar heuristik och måste avsökas med en blind sökmetod är det möjligt att skapa ett abstrakt lager och utifrån detta beräkna en heuristikfunktion (med blind sökning) för det ursprungliga lagret. Därefter kan en väg sökas i det ursprungliga lagret med A*. Valtortas Barriär definieras som antalet tillstånd som expanderas då en sökrymd avsöks med en blind algoritm. Valtortas teorem innebär att det kommer att expanderas åtminstone lika många tillstånd under sökningen då det abstrakta lagret används som om endast det underliggande lagret avsöks med en blind algoritm. Det finns dock metoder att kringgå Valtortas Barriär, exempelvis genom att cache-lagra vissa värden och beräknade vägar samt att variera upplösningen i abstraktionen. I de fall när en heuristikfunktion finns för det abstrakta lagret är Valtortas teorem ej tillämpligt. [21]

3.4.7 Andra metoder

Utöver ovan nämnda sökmetoder finns några andra idéer för hur vägsökning kan göras. Dessa presenteras kortfattat nedan.

Logikbaserade

Tidigare beskrevs två olika tillvägagångssätt för planering, sök- respektive logikbaserad. Ett vägsökningsproblem kan översättas till en samling tillstånd och handlingar som skulle kunna behandlas som ett logiskt problem där ekvationer löses med hjälp av deduktion. Eftersom vägsökning är relativt statiskt, samma handlingar finns alltid och målet är alltid att hitta en optimal väg, så är logisk deduktion antagligen för flexibelt för att bli effektivt.

Kraftfält

Ett kraftfält är en matris där varje element motsvarar ett område i sökrymden. Elementen i matrisen tilldelas ett värde som ger en beskrivning av det motsvarande området i en viss tidpunkt. Målet med kraftfältet är att manipulera dessa värden så att en agent kan navigera genom världen med deras hjälp. Vanligtvis brukar agentens mål vara att förflytta sig i riktning mot målet till det område med lägst värde. [32]

Värdena i matrisen kan beräknas på olika sätt. Några exempel är:

- 1) Avståndet till målet.
- 2) Bäst-först-sökning, tilldela kostnadsvärden enligt sökningen.
- 3) Fyll passerbara element med ett ökande värde från målet och utåt.
- 4) Starta en fyllning från alla blockerade element och fyll med ett minskande värde.
- 5) Med hjälp av A^* .

Det står klart att kraftfältet endast är en representation av agentens omgivning och att de olika teknikerna för att beräkna matrisvärdena är själva sökningen. Punkt 3 och 4 är nya algoritmer för att hitta en väg och undvika hinder. När sökningen ska ske i realtid i stora sökrymder är de troligtvis för långsamma.

3.5 Speciella implementationstekniker

Nedan ges några optimeringstekniker för vägsökning. De flesta är avsedda att användas med A^* eftersom det är den dominerande sökalgoritmen för vägsökning.

3.5.1 Vägsökningshanterare

I en applikation med flera agenter som utnyttjar vägsökningen parallellt kan resurser ofta delas mellan de olika sökprocesserna och ibland behöver vissa processer prioriteras framför andra. För det ändamålet kan en vägsökningshanterare som fördelar resurser mellan de olika processerna användas. Vägsökningshanteraren kan både liknas vid en schemaläggare och en resursförmedlare. Schemaläggningen kan vara beroende av hur lång tid en sökning pågått, hur mycket agenten har kvar att förflyttas längs sin nuvarande väg etc. Resurser som fördelas kan vara kartor, misslyckade sökningar eller taktiska parametrar. [24]

3.5.2 Tidsfragmentering

Tidsåtgången för en sökning kan variera stort. En kort väg utan hinder kan genereras 100-tals gånger fortare än en komplex och lång färdväg. Detta orsakar stora problem för en förutsägbar schemaläggning. Genom att fragmentera beräkningen i mindre intervall binds inte processorn upp så länge att tidskritiska processer passerar sina tidsgränser.

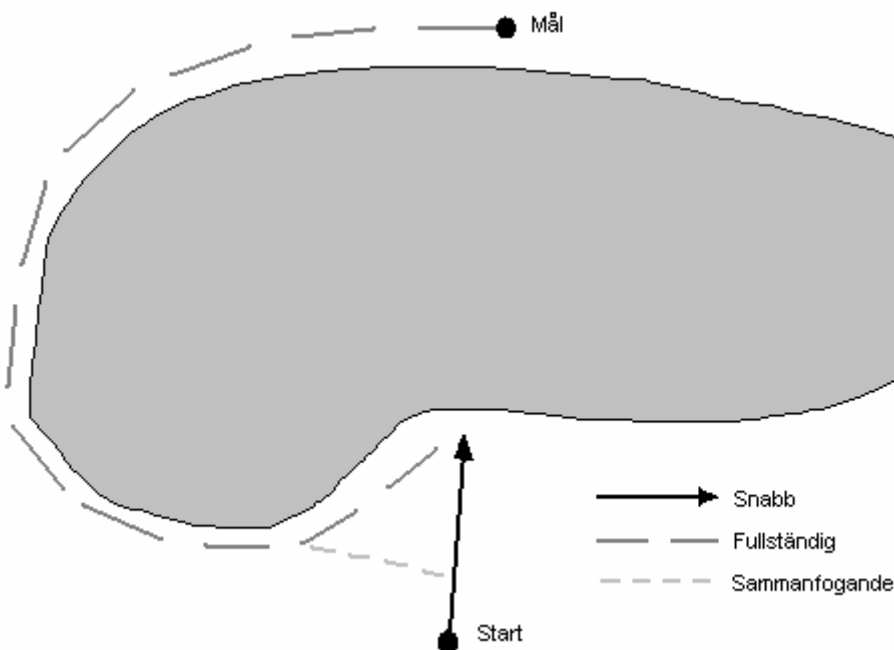
Funktionaliteten blir i en praktisk datorimplementation densamma som att låta varje beräkning köra på en separat processortråd men fördelen med en fragmentering är att vägsökaren själv kan få kontrollen över schemaläggningen och vilka sökningar som ska prioriteras.

3.5.3 En naturligare väg

Om det finns ett krav att agenten måste börja sin rörelse omedelbart efter aktivering kan man kombinera flera olika sorters beräkningar för att få ett naturligt beteende. En "Quick Path" använder en snabb vägsökningsalgoritm för en kort förflyttning i riktning mot målet [24, 37]. Syftet är att agenten ska påbörja sin förflyttning medan en optimal väg beräknas. I [24] föreslås en sträcka på 3-15 söknoder men siffran är relativ i förhållande till hastighet och terräng. I många fall räcker det att beräkna en Quick Path eftersom inga hinder finns på vägen, i dessa fall undviks mer krävande sökalgoritmer helt [38].

I samband med att agenten förflyttar sig längs en Quick Path beräknas den fullständiga vägen. Den fullständiga vägen går mellan Quick Path:ens sista nod och slutnoden och när den sökningen är redo kan agenten ha hunnit röra sig flera steg bort från startnoden. [24]

För att sammanfoga de två vägarna beräknas en väg från agentens nuvarande position till slutmålet. Med stor sannolikhet korsas den fullständigt avsökta vägen inom några noder och då kan sökningen terminera. Kombinationen av den snabbt beräknade vägen, den fullständiga och den sammanfogande bildar agentens slutliga färdväg, som i Figur 10. [24]



Figur 10 Tre olika vägar bildar den slutliga färdvägen.

3.5.4 Cheap List

Den datastruktur som håller tillstånden som används i sökningen påverkar i högsta grad algoritmens prestanda. I algoritmer som kräver en sorterad nodlista, normalt för OPEN-listan, är det vanligt att använda en prioritetskö baserad på heaps som både ger relativt snabb uthämtning och insättning av tillstånd ($O(\log n)$ för båda i värsta fall) [12, 27]. När tillstånd utforskas av A^* genereras antingen fyra eller åtta nya noder beroende på om fyra- eller åttakopplad sökning används. Det innebär att för varje uthämtning kan åtminstone fyra insättningar krävas, alltså är det rimligt att prioritera insättning framför uthämtning [27]. För konstant tidskomplexitet för insättning behövs någon form av osorterad lista men för en sådan är tiden uthämtning linjär vilket inte är acceptabelt för en OPEN-lista med tusentals poster. [27]

En metod som presenteras i [27] kallas *Cheap List* och är en kombination av en sorterad och en osorterad lista. Den sorterade listan är relativt kort och håller de billigaste tillstånden på OPEN-listan. För att hämta ett tillstånd returneras helt enkelt den första noden på den sorterade listan. Om den sorterade listan blir tom fylls den med de billigaste noderna från den osorterade listan, en operation som är linjär mot antalet noder på den osorterade listan. När ett tillstånd ska lagras kontrolleras först om det är billigt nog att läggas till den sorterade listan. I så fall görs en insättning som är linjär mot antalet noder på den sorterade listan. Eftersom antalet noder på den sorterade listan är litet går

insättningen snabbt trots den linjära tidskomplexiteten. Det innebär att den sorterade listan är av variabel storlek. I de fall då tillståndet inte platsar i den sorterade listan läggs det till den osorterade. [27]

3.5.5 Stäng blockerade tillstånd omedelbart

Istället för att lägga samtliga genererade tillstånd på OPEN-listan sparas endast de icke-blockerade tillstånden där. Blockerade tillstånd läggs direkt på CLOSED-listan. På så sätt undviks en OPEN-lista som blir onödigt lång på grund av att tillstånd som aldrig behöver expanderas lagras. I sökrymder med en stor del blockerade tillstånd blir en sådan effektivisering särskilt gynnsam. [25]

3.5.6 Cache för misslyckade sökningar

Ett scenario som en vägsökningsalgoritm till varje pris vill undvika är att söka en väg mellan två punkter där ingen giltig väg existerar. Genom att spara start-slutpunktspar där sökningen misslyckats tidigare kan upprepade anrop till vägsökningen för dessa par undvikas. Framförallt är det användbart att spara misslyckade sökningar om situationer där agenten fastnar kan uppstå. Om det finns mobila objekt i världen som kan blockera vägen för en agent är det inte lika effektivt att spara misslyckade sökningar. Det beror på att en väg som tidigare var blockerad kan vara fri efter det att ett blockerande objekt har förflyttats. [25]

3.5.7 Omplanering

En agent behöver ibland utföra flera upprepade vägsökningar, till exempel då den av någon anledning har fastnat eller för att kontrollera att den befintliga vägen fortfarande är korrekt. Det är onödigt att göra en ny vägplanering från början om en nästan korrekt lösning redan finns, något som exempelvis A^* kräver. Dynamiska algoritmer som D^* återanvänder inte heller gamla planer då nya genereras, däremot kan de ta hänsyn till förändringar som uppstår medan agenten förflyttar sig längs den planerade vägen. Många metoder för omplanering har utvecklats men få är anpassade till A^* . I [29] presenteras omplaneraren SHERPA som bygger på en A^* -algoritm (LPA^*), som både lagrar information om tidigare planer och om *hur* dessa konstruerades samt att kvaliteten på omplanerade planer är lika bra som om de skulle ha genererats helt från början. [29]

I värsta fall är omplanering lika krävande som att göra en fullständigt ny planering. Hur mycket omplanering förbättrar resultatet beror på vilka förändringar som skett i sökrymden. Experiment där prestandan (procentuell förbättring av nodexpansioner) hos SHERPA undersöktes visar på en klar förbättring då SHERPA-omplanering används. Förändringarna i sökrymden sägs vara omfattande men någon närmare förklaring ges ej. Sökrymden var också begränsad i storlek vilket innebär att effekterna av SHERPA-omplanering i större sökrymder är outforskad. Det är också viktigt att ha i åtanke den merkostnad som en mer avancerad omplaneringsalgoritm får jämfört med en enkel A^* , särskilt i de fall då omplaneringar är sällsynta. [29]

Realtidsalgoritmer planerar endast agentens nästkommande steg, till skillnad från de fullständiga lösningar som ges av icke-realtidsalgoritmer såsom A^* . Omplanering är därför inte intressant för realtidsalgoritmer. Däremot kan det finnas intresse av att spara och återanvända delar av den information som användes vid den föregående planeringen. Exempel på sådana realtidsalgoritmer är de som presenteras i [7, 10, 30].

3.5.8 Mellanmål

En A*-sökning som strävar efter en optimal lösning expanderar ett stort område runt startnoden. Genom att dela upp sökningen till att gå mellan punkter längs med en rät linje mellan start och mål erhålls flera små sökningar vars summa av nodexpansioner understiger en fullständig sökning. Nackdelen är att den optimala vägen kan gå en helt annan bana än längs den räta linjen. Om ett av mellanmålen råkar hamna i ett svåråtkomligt område eller på toppen av en kulle kommer den resulterande vägen bli oacceptabel. [13]

3.5.9 Tvåvägssökning

En liknande taktik som att använda mellanmål är att låta sökningen gå från start och mål samtidigt tills sökrymderna möts. De två bubblorna av expanderade noder som bildas blir summerade ändå mindre än en envägssökning. Ett problem som uppstår är hur algoritmen ska känna av att sökrymderna mötts. En implementation måste kunna hantera växlingen mellan söklägena och kostnaderna för de traverserade noderna måste sättas på ett sånt sätt att det enkelt går att förena de båda sökvägarna. [13, 35]

3.5.10 Riktad sökning

Genom att lägga en fast gräns på den maximala storleken hos A*-algoritmens OPEN-lista, eller godtycklig annan bäst-först-sökning lista på aktiverade noder, undersöks endast de noder som för tillfället tycks ligga i en intressant riktning. De engelska uttrycket *Beam Search* åsyftar den ficklampsliknande sökningen. Detta sparar minne men offerar garantin för en optimal lösning. Sökningen kan också fastna i ett lokalt optima om antalet noder på OPEN-listan inte räcker till för att hitta en väg runt hindret. [40]

3.5.11 Mjuka rörelsebanor

Ett problem med de banor som genereras av de flesta sökmetoder är att agenten tvingas till tvära svängar på stället då den ska byta riktning. Det beror på att banan är uppbyggd av en rad linjesegment som knyter samman punkterna (noderna eller delmålen) som sökningen resulterade i. En enkel lösning är att göra en spline-interpolation mellan punkterna men det får två följd problem; att ingen hänsyn tas till blockerade områden då interpolationen görs och att ingen hänsyn tas till agentens fysiska egenskaper såsom svängradie. [11, 33, 37]

Ett första steg är att göra interpolationer med hänsyn till agentens fysiska egenskaper. Det görs enklast genom att svänga maximalt tills agenten står rakt mot målet. Sedan kan även inbromsning läggas till för att minska svängradien och tillåta agenten att ta sig igenom områden där många tvära svängar krävs. Fortfarande återstår dock problemet med att den interpolerade banan kan skära områden som gör den icke-optimal eller t o m ogiltig. En lösning är att använda en riktningberoende sökning. Noderna utökas med en extra parameter, riktningen, och under sökningen kontrolleras inte enbart om en nod är blockerad utan även om svängen som agenten tvingas till för att ta sig till den nya noden är möjlig. Används en heuristisk sökning bör även heuristikfunktionen ändras så att noder med riktning mot målet premieras. En riktningberoende sökning ger r gånger fler tillstånd, där r är antalet riktningar som agenten kan röra sig i, och kräver dessutom omfattande beräkningar av rörelsebanan då sökningen är klar. [11]

Interpolationer är långsamma och går knappast att beräkna i realtid för alla vägar. En lösning kan vara att endast tillåta riktningssändringar i diskreta steg och ha färdigberäknade interpolationer för möjliga banor. [37]

3.5.12 Minneshantering

Under exekveringen av en A*-algoritm krävs en hel del minneshantering. Dels måste sökrymden hanteras på ett smidigt sätt och dels måste en mängd söknoder skapas. I en praktiskt implementation kan en teknik som kallas *memory pooling* användas för att låta flera simultana vägsökare dela på en resurs av förallokerade söknoder. Istället för att skapa ett nytt objekt och sedan ta bort det hämtas en begagnad söknod ut, används en kort stund och returneras sedan till den gemensamma strukturen. Om strukturen blir tom skapas nya noder när begäran om sådana kommer in. På så sätt stabiliseras antalet noder på en nivå som är det maximalt simultant använda..[24]

4 Implementation

I följande kapitel beskrivs den implementation av vägsökning som vi gjort. Motiv till de olika val som gjorts tas upp och alternativa lösningar diskuteras dessutom.

Först beskrivs hur en abstrakt karta för sökningen genereras. Därefter behandlas hur en sökning genererar delmål ur abstraktionen. Slutligen undersöks den praktiska kopplingen mellan det överliggande skriptet och Pathfinder-objektet.

4.1 Världen i Project Entropia

Project Entropia är en tredimensionell värld där samtliga föremål representeras av objekt. Såväl rörliga, exempelvis NPC:er eller spelarkaraktärer, som fasta föremål beskrivs rumsligt utifrån listor med begränsningsvolym. Det finns även möjlighet att ge objekten en rad andra attribut och egenskaper, men dessa är inte intressanta för vägsökningen. Ett till formen komplicerat objekt kan bestå av en lång lista medan ett enklare endast har en begränsningsvolym som definierar dess form. Begränsningsvolymerna är rätblock som förutom storlek och position även kan ha en rotation runt någon punkt (pivot-punkt). Begränsningsvolymerna används framförallt till kollisionsdetektering då föremål i världen rör sig och även vid rendering. När en NPC rör sig testas kontinuerligt huruvida en kollision uppstår och om så är fallet hindras den fortsatta förflyttningen. En fundamental egenskap för en vägsökningsalgoritm är att undvika kollisioner varför begränsningsvolymerna i vårt fall utgör grunden för algoritmen.

Begränsningsvolymerna finns lagrade i systemet på två sätt. Dels genom varje objekt där man direkt har tillgång till listan med volymer och dels i ett oct-träd där alla volymer finns lagrade var för sig i förhållande till sin position i världen. Ett oct-träd är ett tredimensionellt quad-träd, dvs en volym som rekursivt kan delas upp i åtta mindre volymer om och om igen [23]. Oct-trädet som används är dessutom överlappande vilket innebär att angränsande noder delvis representerar samma område. Anledningen till överlappet är att objekt som ligger på en nodgräns inte ska flyttas högre upp i trädet, trots att de egentligen är för stora för den nod de tillhör [36].

Markytan i Project Entropia byggs upp av terrängrutor, likformade områden där terrängen har samma egenskaper. Varje ruta är åtta gånger åtta meter stor. Terrängrutorna beskriver lutningen för terrängen på en viss position och håller även information för rendering. Förutom terrängrutorna finns vegetationsrutor. Huvudsyftet med vegetationsrutorna är att fungera som bas för den vegetationsmotor som finns i Project Entropia. Vegetationsmotorn genererar träd, buskar, gräs och annan vegetation på en vegetationsruta efter givna parametrar. Genereringen är stokastisk för att undvika att samma vy upprepas för spelaren. Idag saknas begränsningsvolym för de objekt som genereras av vegetationsmotorn på grund av den extra belastningen på kollisionshanteringssystemet det skulle medföra. Därför finns det ingen möjlighet att ta med dessa då vägplaneringen genomförs, åtminstone inte i det lägsta lagret. På en viss höjd i världen ligger även en yta som simulerar vatten. Det innebär att samtliga vattentäckta områden ligger under den på förhand definierade höjden.

Under utveckling i Project Entropia är även ett system med portaler och celler. Huvudsyftet med detta är att förbättra och skapa nya förutsättningar för rendering men

det förenklar också uthämtningen av begränsningsvolymerna eftersom varje cell kommer ha en egen datastruktur med volymer.

4.2 Testning

Ett problem under utvecklingen av vägsökningssystemet var den begränsade möjligheten att testa lösningen. Miljön i Project Entropia är inte deterministisk varför det är omöjligt att genomföra upprepade testfall i det fullständiga systemet. Egentligen fanns tre olika möjligheter för att testa vägsökningen:

- 1) Testa enskilda delar var för sig och fristående.
- 2) Göra omfattande ändringar i källkoden för att möjliggöra upprepning av testfall av hela vägsökningen.
- 3) Testa vägsökningens funktionalitet i systemet men utan att kunna göra upprepade testfall.

Alternativ ett är lämpligt då prestanda och minnesåtgång ska undersökas för en väl avgränsad algoritm. Sådana tester gjordes exempelvis av A*-algoritmen och abstraheringen. De ändringar som krävs för tester enligt punkt två är mycket omfattande eftersom en del av agenternas beteende bygger på slumpfaktorer. Dessutom skulle olika hårdkodade testvärldar behövas. Framförallt har funktionalitetstester genomförts enligt punkt tre. Resultatet av sådana tester kan inte ses som statistiskt eller vetenskapligt säkra utan snarare som en indikation på att ett förväntat beteende eller en funktionalitet följer specifikationen.

4.3 Designbeslut för abstraheringen

Den första frågan att ta ställning till är hur omgivningen ska representeras för sökalgoritmen. I sin tur kan den delas in i två delfrågor: Vilka data från omgivningen är nödvändig för en fungerande algoritm och hur ska informationen lagras?

4.3.1 Analys av nödvändig data

Den mest grundläggande informationen för vägsökningsalgoritmen är vilka områden som är blockerade och vilka som är fria. Denna information finns lagrad i begränsningsvolymerna. Världen i Project Entropia är tredimensionell och NPC:er kan röra sig i sex frihetsgrader. Naturligt vore alltså att använda de begränsningsvolymerna som finns i systemet utan modifiering. Det finns dock några invändningar mot detta. För det första är NPC:erna ställda under tyngdlagen, dvs de kan inte flyga fritt i tre dimensioner utan endast röra sig kontrollerat på ett fast underlag. Detta innebär att en mycket stor del av den tredimensionella rymden kommer att vara oåtkomlig och därmed onödig att spara. För det andra ökar mängden data och därmed mängden noder som behöver avsökas proportionellt mot storleken på den tredje dimensionen om inte en avancerad struktur används för att lagra datat. En sådan avancerad datastruktur kräver dessutom att en speciell algoritm transformerar det tillgängliga datat till ett format som passar datastrukturen. För det tredje är ett system under utveckling som skulle kunna tillåta att nivåskillnader delas upp i enskilda celler. Då skulle exempelvis en trappa eller en stege vara en portal mellan två celler.

Fördelen med en tredimensionell representation är att den är mer generell vilket innebär att en rad specialfall inte behöver uppstå, exempelvis vid navigering under tak. Dessutom är chansen stor att NPC:erna rör sig naturligare då deras väg har genererats utifrån en tredimensionell representation istället för en tvådimensionell med olika tilläggsvillkor.

Tvådimensionella representationer är mindre minneskrävande och genererar färre tillstånd att söka i. Givetvis innebär en tvådimensionell representation att begränsningsvolymerna som finns i systemet måste representeras i två dimensioner istället för tre. Transformationen till den tvådimensionella representationen är en projektion på det plan som ska användas för sökningen, alltså ingen tidskrävande operation men den hade kunnat undvikas helt om en tredimensionell representation använts. Nackdelen med en tvådimensionell representation är att sökningen kräver en rad bivillkor för att simulera den tredje dimensionen. Exempel på sådana bivillkor är att objekt under en viss höjd, tex markytan eller NPC:ns fötter, kan ignoreras. På samma sätt kan objekt ovanför huvudet på NPC:n ignoreras. Dessvärre är inte dessa villkor allmängiltiga vilket visas senare då den faktiska implementationen beskrivs i detalj.

Slutsatsen är att en tvådimensionell representation bör användas, framför allt på grund av de komplicerade datastrukturer som krävs för att en tredimensionell värld skulle gå att hantera tillräckligt snabbt i sökalgoritmen. Komplicerade datastrukturer kräver även ofta en merkostnad eller större minne för vissa operationer, något som skulle försämra den slutgiltiga algoritmens prestanda. Utöver detta tar implementeringen av en avancerad struktur tid från andra, högre prioriterade, delar.

Med tanke på förutsättningarna i Project Entropia, huvudsakligen kravet att hundratals NPC:er ska röra sig samtidigt och behöva uppdatera sin väg ofta, drogs slutsatsen att en lösning som exekverar så snabbt som möjligt var den bästa. Dessutom var arbetets tid begränsad varför det var rimligt att välja en relativt enkel lösning för att något konkret resultat skulle kunna visas. I de fall som en NPC verkligen är beroende av att röra sig vertikalt finns två lösningar:

- 1) Små höjdskillnader mellan volymer och terrängen kan ignoreras eftersom en NPC kan röra sig upp för 0,5 meter höga trösklar utan problem.
- 2) Där höjdskillnaderna är stora bör världen delas in i celler med portaler som länkar mellan olika horisontella plan.

Förutom begränsningsvolymerna finns information om terräng, vegetation, vatten och andra agenter tillgänglig. I varje agent sätts en önskad hastighet, sedan kontrollerar fysiksystemet om denna är möjlig med tanke på agentens högsta tillåtna hastighet och övriga parametrar såsom gravitation och friktion. I uppförsbacke är det sällan så och agentens hastighet minskas, i utförsbacke är det troligt att hastigheten är tillåten eller t o m så hög att agenten tappar kontakten med marken. Det innebär att en agent skulle kunna tjäna på att gå runt en kulle eller välja en väg upp till en målpunkt med mindre genomsnittlig lutning. Det är dock svårt att säga vid vilka lutningar en viss väg är att föredra. I de fall då terrängens lutning är så brant att agenten omöjligt kan gå uppför den kan en genererad väg visa sig vara oframkomlig om inte hänsyn tas till lutningen. Ur terrängrutorna erhålls bla lutningen för ett område. Problemet är att beräkning av lutningen tar tid och att antalet tillstånd fyrdubblas. Antalet tillstånd ökar eftersom lutningen är beroende av från vilken riktning sökningen expanderar tillståndet. En lösning vore att införa ett abstrakt statiskt lager som håller information om lutningen. Slutsatsen är att förtjänsten som ett hänsynstagande av lutningen ger inte väger upp de extra beräkningar som krävs.

Vegetationen finns dels lagrad som manuellt utplacerade begränsningsvolymer och dels i vegetationsrutorna som automatiskt genererar vegetation. Den manuellt utplacerade vegetationen behandlas på samma sätt som övriga begränsningsvolymer. Eftersom den

automatgenererade vegetationen inte har några begränsningsvolymmer är det inte möjligt att behandla den på samma sätt som andra objekt. En möjlighet är att använda ett abstrakt lager av vegetationsrutor där vegetationsdensitet, och eventuellt typ, är avgörande för kostnaden för att förflytta agenten genom en ruta. Eftersom vegetationen endast är en visuell effekt och inte påverkar varken spelare eller agenter finns det ingen anledning att lägga ner processortid på att anpassa sökningen efter den. Först när spelare och / eller agenter berörs av den automatgenererade vegetationen är det intressant att anpassa vägsökningen efter den.

I verkligheten bör vatten vara ett hinder för vissa agenter, speciellt då det blir för djupt. Idag finns ingen sådan begränsning i den virtuella Project Entropia-världen. Både spelare och agenter har möjlighet att röra sig under vatten hur som helst. Eftersom vattenytan ligger på en konstant nivå i hela världen är det möjligt att helt enkelt göra samtliga områden under denna höjd opasserbara. Problemet är att det för varje område krävs en kontroll av terrängrutans höjd. I de fall då områdena i abstraktionen överlappar flera terrängrutor med olika höjd kan det också vara svårt att avgöra hur området ska behandlas. Ett annat alternativ är att skapa begränsningsvolymmer som täcker vattnet eller speciella områden som icke-simkunniga agenter är förbjudna att beträda. Då varken spelare eller agenter idag påverkas av vatten är det lämpligt att inte låta vägsökningen göra det heller, därför tas ingen hänsyn till vatten i den slutgiltiga algoritmen.

Till sist finns även information om andra agenter och spelare tillgänglig. Den skulle kunna användas för att skapa ett taktiskt agerande. Exempelvis kan områden i närheten av vänligt sinnade agenter ges en lägre kostnad. En annan möjlighet är att beräkna siktlinjer mellan agenter för att ge dessa förmågan att gömma sig för varandra. Det är kostsamt att hålla reda på alla icke-statiska objekt (spelare och agenter) och ännu mer krävande att för varje abstrahering beräkna förhållandena mellan dessa. Därför är det inte realistiskt att införa en taktisk påverkan av sökningen, åtminstone inte när hela kartan genereras dynamiskt.

4.3.2 Datastruktur för lagring

I huvudsak finns två olika metoder för att lagra de data som ger agenten en bild av omgivningen; som en cellrepresentation eller som ett skelett. En klar fördel med ett skelett, om det går att placera ut noderna på ett vettigt sätt, är att agentens väg är direkt genererade utifrån bågarna som kopplar samman noderna. Problemet att inte alla punkter i världen är representerade med ett skelett finns egentligen i en cellrepresentation också. Det är helt enkelt en fråga om upplösning. Tanken med ett skelett är att endast de punkter som är intressanta för agentens förflyttning ska representeras i abstraheringen medan en cellrepresentation i sitt enklaste utförande bara är en kopia av världen efter en given upplösning. Om fler och fler noder införs i skelettet (dvs upplösningen ökar) går skelettet mot att bli likvärdigt med en cellrepresentation. Valet föll alltså på att abstrahera världen med celler, på grund av nackdelarna med ett skelett.

Valet av datastruktur för lagringen av omgivningsdata i celler har flera alternativ. Den enklaste, och kanske mest intuitiva, lösningen är att använda en representation baserad på en tvådimensionell array. Mer avancerade strukturer, tex olikformade områden eller quad-träd, är mer effektiva men betydligt mer komplicerade att implementera.

Med de givna förutsättningarna, att minimera tidskomplexiteten på bekostnad av minnesåtgången, stod valet mellan att använda antingen en statiskt eller dynamiskt genererad karta. En statisk karta skapas helt och hållet då systemet startar och innefattar

hela världen medan en dynamisk karta genereras av agenten under exekvering och innehåller endast det område som är aktuellt för den kommande sökningen. Om en statisk karta implementeras som av en tvådimensionell array kommer den kräva en mycket stor minnestillgång på grund av världens storlek. Istället kan en mer avancerad struktur användas, tex ett quad-träd, för att både spara minne och ge en godkänd tidskomplexitet för uthämtning av data. En dynamisk karta är knappast värd att implementera med någon annan struktur än en matris eftersom den mindre storleken begränsar minnesåtgången betydligt.

Om en dynamisk karta implementerad med en array används kommer varje agent att kräva en minnesåtgång på $x_n * y_n$ bit, där x_n och y_n är storleken på den karta som krävs för den aktuella sökningen, om bara en bit lagras per cell (dvs blockerad eller icke-blockerad). Dessutom krävs en tillfällig karta med information om kostnader till avsökta tillstånd men eftersom denna är nödvändig såväl vid statiska som dynamiska kartor tas den inte med i det följande resonemanget. En statisk karta som lagras i en tvådimensionell array kräver alltid $x_m * y_m$ bit, där x_m och y_m är världens dimensioner. Det innebär att den totala minnesåtgången för samtliga agents abstraktion blir summan av $x_n * y_n$ för alla n då en dynamisk karta används och $x_m * y_m$ i fallet med en statisk karta. Alltså kan $(x_m * y_m) / (x_n * y_n)$ dynamiska abstraheringar hanteras parallellt innan en statisk blir mer minneseffektiv.

Exempel: Avvägning mellan en statisk och flera dynamiska kartor

En server ska idag kunna hantera ungefär 25 kvadratkilometer och 300 agenter. Alla agenter kan samtidigt ha en abstraktion av ungefär 290*290 meter innan en statisk karta blir mer minneseffektiv. Eftersom alla agenter inte vägsöker samtidigt och en karta av 290*290 meter är relativt stor är dynamiska kartor att föredra, sett ur ett minnesperspektiv, med dagens konfiguration. Däremot är dynamiska kartor mer processorkrävande under exekvering.

En mer avancerad datastruktur effektiviserar lagringen på bekostnad av insättning och uthämtning. Därför är det framförallt intressant att använda en sådan då abstraheringen är förberäknad i en statisk karta för att undvika merkostnaden av dynamiska insättningar. Tidsåtgången skiljer sig inte bara efter vilken datastruktur som används för att lagra abstraktionen utan även mellan en statisk och en dynamisk abstraktion, oavsett datastruktur. Om en dynamisk abstraktion används krävs abstrahering under sökningen. Då en statisk abstraktion används är den färdigberäknad när vägsökningen påbörjas. Det spar alltså värdefull tid under exekvering att ha en färdigberäknad karta. Den effektivaste lösningen om både minnes- och tidsåtgång tas i åtanke är förmodligen en statisk abstraktion som implementeras av ett quad-träd eller någon annan datastruktur som tillåter relativt snabb uthämtning av element av variabel storlek. En sådan lösning kräver att tillräckligt med minne finns för att hålla all data som en statisk abstraktion kräver, att uthämtning ur strukturen går tillräckligt snabbt och att den avsatta tiden för arbetet är tillräcklig för en avancerad implementation.

Exempel: Quad-träd

Statisk inläsning av hela världen till ett quad-träd kräver mycket tid då systemet startas. Insättning och, framförallt, uthämtning tar längre tid än från ett rutnät.

I datastrukturen för ett quad-träd behövs följande delar: 1 byte som anger storleken på området, 2 byte som anger position, 1 bit som anger om det är blockerat eller inte och nio pekare (till förälder, grannar och barn). Totalt 25 bit och 9 pekare, dvs 354 bit då en pekare är 32 bit. Troligen krävs mer utrymme för position och storlek i en verklig implementation.

En karta av storleken 13000 x 13000 rutor ger för ett rutnät med en bit per ruta totalt mindre än 22 mb minne. Ett quad-träd med samma minnesåtgång kan innehålla $22 \cdot 8 \cdot 10^6 / 354 \approx 500\,000$ stycken element. Dessa element motsvarar en kvadratisk värld med strax över 700 element per sida i genomsnitt. Det innebär att varje område i quad-trädet åtminstone måste omfatta 19 x 19 rutor från rutnätet i snitt.

Huruvida det är rimligt beror på densiteten och spridningen av de blockerade områdena.

Ett sista alternativ är att generera en statisk karta med färdigberäknade vägar mellan samtliga noder. En sådan implementation är realistisk om antalet noder är litet. När sökrymden innehåller tusentals noder blir minnesåtgången snabbt ohanterlig.

Vår slutsats blev att en statisk karta implementerad av en matris kräver för mycket minne. Det extra arbete som krävs för att implementera någon mer avancerad datastruktur, för att på så sätt minska minnesåtgången för en statisk karta, kunde inte motiveras tidsmässigt inom ramen för arbetet. Alltså föll valet på dynamiska abstraheringar, unika för varje agent. Slutligen kvarstod valet av vilken implementationsteknik som skulle användas för den dynamiska kartan. Frågan var om det fanns någon struktur som var bättre lämpad än arrayen för dynamiska kartor då högsta prioritet var snabb uthämtning och lagring av data samt en enkel implementation. I tabellen nedan finns en uppställning över tänkbara strukturer med de viktigaste för- och nackdelarna.

Implementationsteknik	Fördelar	Nackdelar
Rutnät	Snabb lagring och åtkomst av data. Mycket enkel och välbeprövad implementation. Enkel att abstrahera till.	Ineffektiv lagring vilket innebär minneskrävande och en stor sökrymd. Statisk storlek. Hackig väg.
Polygoner	Utnyttjar minnet effektivt. Lätt att använda om världen är representerad i polygoner.	Extra processorkraft krävs för att generera polygonerna. Underliggande datastruktur för lagring av polygonerna krävs.
Quad-träd	Utnyttjar minnet effektivt. Snabbt vid okända omgivningar.	Långsammare än matriser vid kända omgivningar. Relativt komplicerad implementation. Relativt långsam in- och uthämtning.
Inramat quad-träd	Som quad-träd med tillägget att vägen blir optimal i förhållande till den valda upplösningen.	Som quad-träd, dessutom blir implementationen något mer komplicerad.
Skelett	Den funna vägen är optimal. Liten sökrymd.	Svårt att placera ut noder automatiskt. Ojämn upplösning av världen.

Tabell 1 Jämförelse av implementationstekniker.

Valet föll på att använda tvådimensionella arrayer som rutnät; framförallt på grund av snabb- och enkelheten men det motiveras ytterligare med att kartorna, och därmed arrayerna, inte blir så stora då agenten följer rörliga mål. Det beror på att eftersom agenterna vanligtvis kommer att uppdatera sitt mål med några sekunders intervall och då behöver inte det abstraherade området som lagras ha en större radie än den sträcka som agenten kan förflytta sig mellan två uppdateringar. Därför blir det inte tal om matriser med tusentals rader och kolumner. Då agenterna rör sig till fasta mål är kartstorleken direkt beroende av den parameter som avgör positionen för målet. Denna kan justeras för att passa vägsökningen.

Exempel: Kartstorlek

När en NPC rör sig efter ett rörligt mål fås en kartstorlek på ungefär 2000 kartnoder per sökning vilket motsvarar sidor på ungefär 45 x 45 noder. Resultatet är strakt beroende av de parametrar som styr NPC:n och spelarens rörelser. Vid fasta mål blir medelstorleken ca 9000 kartnoder, men det är direkt beroende av hur kartstorleken sätts vid fasta mål och hur långt bort målet kan ligga.

Kartor av dessa storlekar utgör inget problem för minnesåtgången då en kartnod endast innehåller en bit som anger om den är blockerad eller ej och en short int som anger kostnaden till noden.

Slutligen finns möjligheten att använda flera olika abstraheringar, i lager, för att på så sätt förbättra sökningen och eventuellt lagra informationen om världen på ett bättre sätt. Flera lager är inte en förutsättning för en lyckad abstraktion utan snarare en möjlighet att ta vara på speciella förutsättningar som ges av informationen abstraktionen bygger på. I

Project Entropia ser vi framförallt tre möjligheter till abstraktionslager, förutom det grundläggande som bygger på begränsningsvolymer. För det första finns vegetationsrutorna som vegetationsmotorn bygger på. Om agenterna ska navigera efter vegetation kan rutornas egenskaper ligga till grund för en grovsökning där områden med svårframkomlig terräng är kostsamma. Egenskaper som skulle kunna beaktas är exempelvis densiteten hos den utplacerade vegetationen, typ av vegetation eller markens egenskaper.

För det andra kan abstraheringen ta hänsyn till lutningen på underlaget där agenten rör sig. På så sätt bör en naturligare väg fås. Det finns två problem med ett sådant lager. Dels måste lutningen beräknas för varje område och dels blir antalet tillstånd betydligt fler på grund av att kostnaden varierar beroende på vilket tillstånd (område i världen) som NPC:n rör sig från när den går in i ett annat tillstånd

Det tredje alternativet är att använda det cell-portal-system som är under utveckling. Genom att skapa en graf där noderna är celler och bågarna portaler skulle det snabbt gå att hitta en väg på cellnivå. Den vägen kan sedan utnyttjas i den exakta vägsökningen. Ett sådant lager skulle dela upp problemet i flera mindre delproblem (att söka vägen i en cell mellan två portaler). Det innebär dels att endast data från den aktuella cellen behöver abstraheras och sparas, vilket förhoppningsvis skulle minska minnesåtgången, och dels att sökningen sker på kortare avstånd flera gånger, något som påminner om mellanmål.

Varken vegetationsmotorn eller cell-portal-systemet är fullt implementerade idag varför det är svårt att förbereda vägsökningen mot dessa system och omöjligt att testa prestandaförändringar och konsekvenser på funktionalitet. Höjdskillnader skulle medföra naturligare rörelser på bekostnad av processortid och minne. Dessutom skulle det vara lätt för en spelare att lura agenter som undvek branta höjdskillnader genom att själva röra sig upp och ner för dessa, något som skulle förstöra spelkvalitén. I ett första skede har vi valt att endast använda ett lager för sökningen men förbereda ett gränssnitt mot cellsystemet. Framförallt eftersom cell-portal-systemet är viktigt både för sökning i inomhusmiljö och mellan nivåskillnader.

4.3.3 Algoritmen för kartgenerering

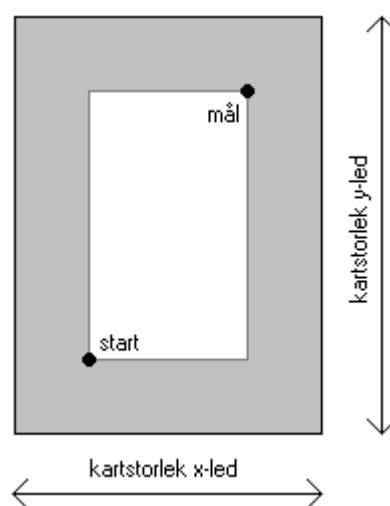
Syftet med kartgenereringen är att abstrahera den omgivning som NPC:n vill söka i till data som kan sparas i en matris. Ovan beskrevs både hur världen i Project Entropia är strukturerad och varför vi valt att lagra kartan i en tvådimensionell array som representerar ett rutnät. I korthet innehåller algoritmen följande steg:

- 1) Beräkna storlek och position för det område som ska abstraheras.
- 2) Hämta ut begränsningsvolymer som berör det aktuella området.
- 3) För varje volym som returneras från 2. Gör:
- 4) Roterar volymen kring z-axeln och beräkna de fyra hörnpunkterna. De fyra punkterna bildar nu en rektangel som är volymens skugga i xy-planet.
- 5) Klipp bort de delar av rektangeln som ligger utanför det område som ska abstraheras.
- 6) Avbryt för alla volymer som helt ligger under markytan eller ovanför NPC:ns huvudhöjd.
- 7) Markera de fält i rutnätet som motsvarar kanterna på de klippta rektanglarna.

Pseudokod för abstraheringen finns i Bilaga 2: Pseudokod på raderna 7 till 14.

Storlek och position

Områdets storlek beräknas utifrån agentens och målets positioner (start- respektive slutposition). En rektangel med diagonala hörnpunkter i startpunkten och slutpunkten utgör basen för området. Därefter läggs en buffert till omkring som kan varieras beroende på vilken typ av sökning kartan är avsedd för (se Figur 11). Området har också en maximal storlek som definieras av en konstant. Om det aktuella målet skulle ligga utanför en karta av maximal storlek beräknas ett temporärt mål som ligger på linjen mellan start- och slutpunkt, en konstant sträcka från startpunkten. I arrayen töms de element som motsvarar sökområdet. Tidsåtgången är beroende på hur stort området är som ska tömmas, $O(s_x s_y)$, där s_x och s_y är antalet element i x- respektive y-led i arrayen.



Figur 11 Kartstorleken bestäms utifrån start- och målpunkten. Det grå området är bufferten som läggs till för att ge sökningen spelrum.

Hämta begränsningsvolym

Begränsningsvolymerna hämtas som en lista direkt ur det oct-träd där de finns lagrade. Tidskomplexiteten för att hämta en volym från oct-trädet är $O(\log n)$ i medel och $O(n)$ i värsta fall. Oct-träd begränsas vanligtvis av ett maximalt djup (eller upplösning) [36]. Det innebär att komplexitetsberäkningarna inte riktigt stämmer då antalet volymer i löven blir stort och en lista i löven måste itereras för att hitta en viss volym. Ingen information om komplexiteten för att hämta ut alla volymer ur ett givet område har hittats. Egentligen är detta det intressanta för kartgenereringen då volymer från ett område som motsvarar kartan ska hämtas. Komplexiteten för en sådan operation är beroende av områdets storlek, trädets upplösning samt volymernas antal och position. I värsta fall kan tidskomplexiteten begränsas till $O(k \log k)$ där k är antal löv. Det uppkommer då ett område som motsvarar hela världen ska hämtas och varje löv håller minst en volym. För att hämta alla volymer måste algoritmen iterera ner till varje löv. En iteration tar $O(\log k)$ tid och det görs k gånger.

Ett intressantare fall är då en del av världen, med sidor av längden X och Y , hämtas. Ett område vars sidor förhåller sig som x/X respektive y/Y till storleken på hela världen kommer att ge färre antal iterationer ner till löven men längden av iterationerna kommer att vara lika långa, dvs $O(\log k)$. Om spridningen av objekt i oct-trädet är jämn kommer antalet iterationer minska proportionellt mot storleken på det område objekten hämtas från. Med beteckningarna ovan blir antalet iterationer $k^*(xy)/A$, där $A = XY$. Detta ger en uppskattad komplexitet på $O((k \log k)^*(xy)/A)$ för uthämtningen.

Antalet volymer som returneras från oct-trädet är i värsta fall alla n stycken som finns lagrade. Även denna siffra bör vara proportionell mot storleken på området i de fall då objekten är jämt fördelade i oct-trädet. Oct-trädet som håller begränsningsvolymer är ett så kallat överlappande-oct-träd. Det innebär att noderna i trädet överlappar varandra och får bli följden att volymer som ligger utanför det avsedda området returneras. Fördelen är att volymer som ligger på en nodgräns inte flyttas uppåt i trädet. Antalet returnerade volymer beror även på hur stort överlappet är.

Exempel: Returnerade volymer

Antalet volymer som returneras från oct-trädet är mycket stort. Under ett test som pågick i ungefär 10 minuter då en NPC förföljde en spelare i en normal värld returnerades totalt 75 000 begränsningsvolymer. Av dessa var endast 2 500 nödvändiga att rita in i kartan. En mycket stor del, ungefär 28 000, av volymerna låg utanför kartområdet. Att de returnerades beror på oct-trädets överlapp och upplösning. Resterande 44 500 volymer som sållades bort låg antingen under markytan eller över NPC:ns huvudhöjd.

Testet visar att en mycket stor mängd av de returnerade volymerna aldrig används i vägsökningen. Mängden kan minskas genom att överlappet i oct-trädet minskas. Ett minskat överlapp får dock andra oönskade effekter. Att ett så stort antal volymer sållas bort innebär också att ordningen som olika sållningstester görs i är viktig för kartgenereringens prestanda.

För att optimera kartgenereringen hämtas nya objekt från oct-trädet endast då start- eller målpunkt inte befinner sig inom det område som den föregående sökningen avsåg eller då kartbufferten ska förändras på grund av en misslyckad sökning. Endast i de fall då en nya lista med begränsningsvolymer hämtats behöver en ny karta beräknas, i annat fall kan den gamla återanvändas och algoritmen returnerar här.

Exempel: Kartåtervinning

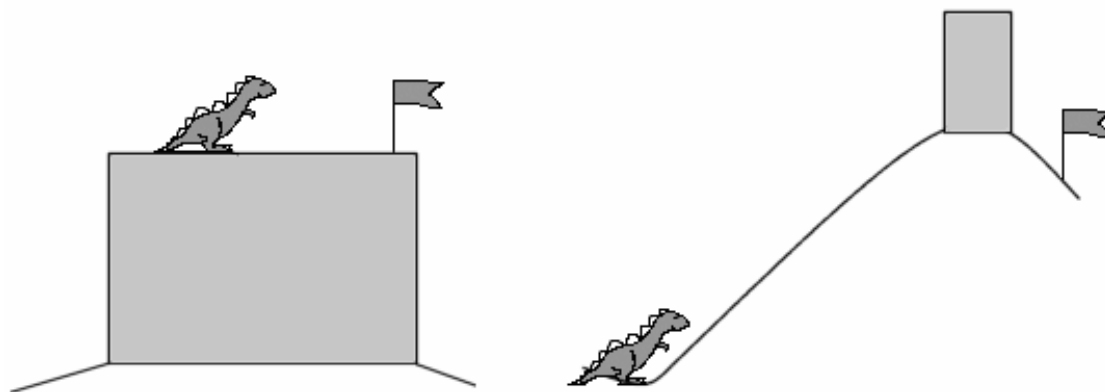
Hur ofta en karta kan återanvändas varierar kraftigt beroende på om målet är rörligt eller fast. Praktiska tester visar att då en agent förföljer ett rörligt mål återanvänds 40-60 % av alla kartor. Variationen beror på hur det rörliga målet betar sig och vilka parametrar som används vid kartgenereringen. Resultatet kan endast ses som en indikation på att återanvändningen fungerar tillfredsställande i en vald konfiguration. När agenter rör sig mot fasta mål återanvänds 20-40 % av de genererade kartorna. Detta är starkt kopplat till storleken på kartan och avståndet till det fasta målet.

Speciella avbrottsvillkor

Den aktuella volymen ignoreras om dess högsta punkt är mindre än 0,5 meter över markytan. Det beror på att agenter kan gå över hinder som är mindre än 0,5 meter höga. Volymer vars lägsta punkt ligger högre upp än höjden av markytan plus agentens höjd ignoreras eftersom det ska vara möjligt att gå under föremål, exempelvis i tunnlar och under trädskronor. I den lösning som valts kan problem uppstå i de fall då agenten har kommit upp ovanför markytan, exempelvis på grund av att den är tillräckligt nära målet för att använda FÖRFÖLJ-tillståndet (se avsnitt Skriptfunktionaliteten). Då kan volymer

ignoreras som är i samma horisontella plan som agenten vilket gör att vägsökningen större än den hjälper. NPC:n kan komma att fastna på plattor när han navigerar efter det plan som finns i terrängens höjdnivå. Problemet har förebyggts på skriptnivå genom att inte tillåta NPC:er att navigera nära bebyggelse som medför dessa risker, tex flervåningshus.

Med alternativet där volymer jämförs med agentens position i höjddled istället för markytan kan situationer uppstå där volymer ignoreras som hindrar vägen (se Figur 12).



Figur 12 Till vänster visas en situation där agenten inte når flaggan, eftersom volymer endast ignoreras då de ligger för högt i förhållande till markytan. I situationen till höger visas hur agenten skulle tro att vägen var fri till flaggan om jämförelsen skedde mot agentens position.

Rotation

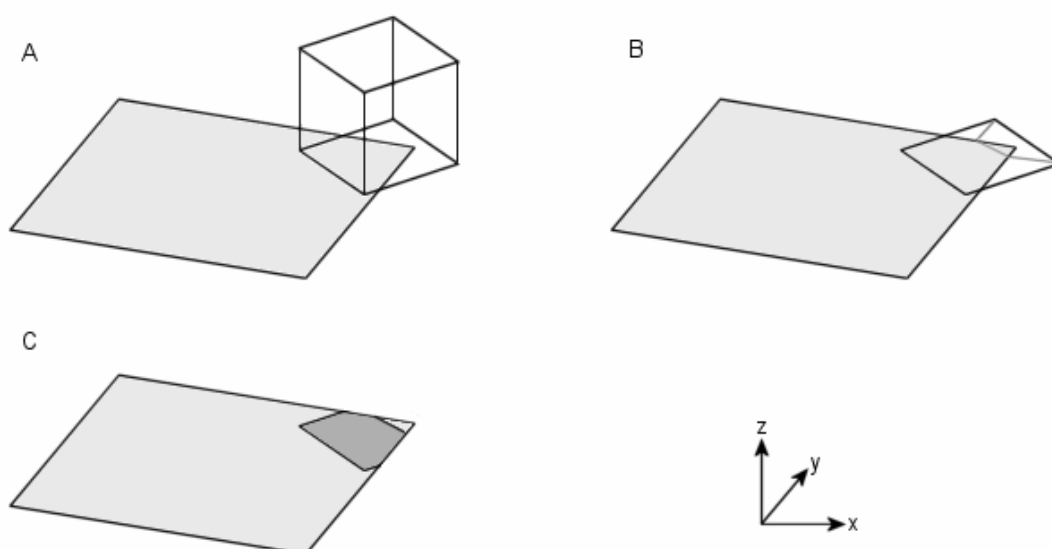
Begränsningsvolymer kan ha godtycklig rotation kring pivotpunkten. Rotationen i sig är inget problem att beräkna eftersom en komplett rotationsmatris för varje volym finns. Däremot krävs ett extra steg i algoritmen för att hantera volymer som är roterade runt x- eller y-axeln. Skillnaden mellan rotation kring z-axeln och x- eller y-axeln uppstår då volymens hörn inte ligger parallellt i xy-planet. Dels uppstår problem i klippningen av volymen och dels måste hörnen i volymen projiceras på xy-planet efter rotation och punkternas konvexa hölje beräknas för att volymens skugga ska bli korrekt. Det finns olika algoritmer för beräkning av konvexa höljen men ingen har en tidskomplexitet på mindre än $O(n \log n)$ där n är antalet punkter som höljet ska beräknas från. Förutom beräkningen av det konvexa höljet krävs en algoritm som klipper en volym mot en annan volym. En sådan är något mer avancerad än den enkla tvådimensionella klippningen.

Efter att ha undersökt förhållandena i Project Entropia framkom det att få volymer var roterade runt någon annan axel än z (vinkelrät mot markytan). Därför valde vi att endast implementera den enklare rotationen kring z-axeln. I en mer generell lösning bör rotation kring samtliga axlar beräknas och volymens faktiska projicering på xy-planet användas.

Klippning

Att klippa en figur innebär att skära bort delar som inte passar mot en annan figur. I vårt fall vill vi klippa bort de delar av, de på xy-planet projicerade, begränsningsvolymen som ligger utanför det aktuella sökområdet (se Figur 13). Klippning sker av varje linjesegment mellan två på varandra följande hörnpunkter. Varje punkt som ligger utanför det aktuella sökområdet flyttas in till dess kant (se Figur 13). Faktum är att en liten area ibland missas av klippningen, som i figurens övre högra hörn. Det beror på att den implementerade klippningsalgoritmen är förenklad med tanke på vägsökning. Att den lilla arean inte är markerad som blockerad gör nämligen ingenting eftersom den är helt blockerad av den inklippta arean och därför kan sökningen aldrig nå dit.

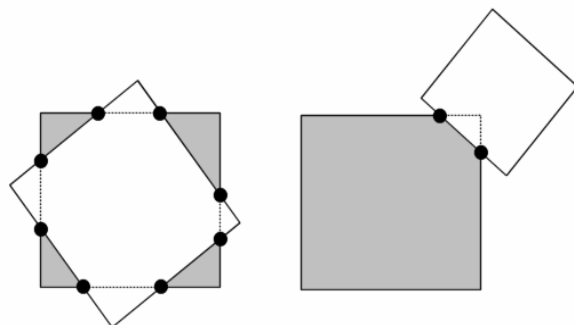
Med en godtycklig polygon som ska klippas har algoritmen den linjära tidskomplexiteten $O(n)$, där n är antalet hörnpunkter i polygonen. I vårt fall är n alltid fyra. Merparten av volymerna ligger antingen helt inom eller helt utanför det aktuella sökområdet vilket medför att relativt få volymer behöver klippas.



Figur 13 Klippning av en begränsningsvolym. I A visas volymen och kartan innan klippning. I B är volymen projicerad på kartan och hörnen utanför kartan ska flyttas in till kartkanten. I C är den projicerade arean klippt för att passa kartan (mörkt område).

Markera blockerade områden

Utifrån de hörnpunkter (mellan två och åtta stycken enligt Figur 14) som klippningen returnerar ska områden i matrisen markeras som blockerade. Det görs genom att linjer dras mellan hörnen så att polygonens kantlinjer blir ifyllda i sökrymden. För ändamålet används Bresenhams algoritm som är vanligt förekommande i tillämpningar inom datorgrafik då en linje mellan två punkter ska ritas i diskreta steg. Fördelen med Bresenhams algoritm är att inga dyra operationer, såsom multiplikation eller division, används och att alla beräkningar görs med heltal. Tidskomplexiteten för Bresenhams algoritm är konstant $O(\max(x,y))$ där x och y är avståndet i diskreta punkter mellan linjens start- och slutpunkt i x - respektive y -led.



Figur 14 De vita polygonerna klippas mot de grå. Till vänster fås en ny polygon med maximala åtta hörnpunkter. Till höger fås endast ett linjesegment, d v s minimala två hörnpunkter.

Eftersom agenten under förflyttning använder sig av sina känselspröt för att ta sig ur oförutsedda situationer är det inte bra om den går för nära hindrens begränsningsytor då den riskerar att fastna i ett kollisionsbeteende. Därför utökas alla begränsningsytor i kartgenereringen med en variabel storlek. Vid den första sökningen läggs en meter till på alla axlar som en navigeringsbuffert. Om detta innebär att start eller mål hamnar i en oåtkomlig punkt minskas den sedan ner stegvis för att i det sista försöket inte användas alls.

4.3.4 Sammanfattning av kartgenereringen

Den implementation av kartgenerering som vi valt är speciellt anpassad efter förutsättningarna i Project Entropia. Därför är den troligen inte lämplig för en generell vägsökning. Det finns ett antal parametrar som kan justeras i algoritmen; den maximala storleken på kartan, upplösningen, avbrottsvillkor och storleken på bufferten runt start- och slutpunkt. Alla dessa parametrar påverkar beräkningstiden för kartgenereringen och beteendet hos agenterna. En större karta gör att större sökningar kan hanteras, bättre upplösning innebär en bättre avbildning av världen, speciella avbrottsvillkor påverkar var NPC:n kan röra sig och buffertens storlek är kopplad till hur många sökningar som misslyckas. Kartgenereringen är alltså ett avvägande mellan tillgänglig processortid och i slutändan beteendet hos agenterna.

Komplexitet

Tidskomplexiteten för storleks och positionsberäkningen är beroende av tömningen av arrayen som i sin tur begränsas av den konstant som anger största möjliga arraystorlek. Uthämtningen av begränsningsvolymer från oct-trädet tar i värsta fall $O((k \log k) * (xy) / A)$ tid, där k är antalet löv i oct-trädet och $(xy) / A \leq 1$ och är ett storleksförhållande mellan det avsedda området och hela världen. Återstoden av algoritmen är direkt beroende av antalet begränsningsvolymer som returneras från oct-trädet. För varje uthämtad volym (totalt m stycken) har kontrollen av avbrottsvillkoren konstant komplexitet. Klippningen har $O(h)$ komplexitet, där h är antalet hörnpunkter i polygonen. För de projicerade begränsningsvolymererna är antalet hörnpunkter konstant fyra. Slutligen krävs $O(\max(x,y))$ tid för Bresenham's algoritmen. Den sammanlagda tidskomplexiteten blir $O(s_x s_y + n + m * (h + \max(x,y)))$.

Tidskomplexitet (värsta fall)

Storleks- och positionsberäkning	$O(s_x s_y)$
Uthämtning av begränsningsvolym	$O((k \log k)^*(xy)/A)$
Avbrottsvillkor	$O(1)$
Loop $\forall m$ Klippning	$O(h)$
Markering	$O(\max(x,y))$
	<hr/>
	$O(s_x s_y + (k \log k)^*(xy)/A + m*(h + \max(x,y)))$

Eftersom $s_x s_y$ begränsas av en konstant, $(xy)/A \leq 1$, h är konstant och $\max(x,y)$ också begränsas av arrayens storlek blir värsta-falls-komplexiteten $O(k \log k + n)$. Vilken av termerna som är dominerande beror på begränsningsvolymernas position och storlek, alltså hur de lagras i oct-trädet.

Tidsåtgången för algoritmen är inte detsamma som komplexiteten. Exempelvis kan de operationer som ingår i klippningen kräva betydligt mer tid än den dubbla for-loop som nollställer värdena i arrayen trots att komplexiteten för klippningen är konstant. Alltså kan det vara intressant att undersöka den reella tidsåtgången för att hitta delar som är lämpliga att optimera ytterligare. För detta ändamål finns speciell programvara och tester redovisas i kapitel 5.2.

Celler och portaler

I den slutliga versionen av vägsökningsalgoritmen är endast ett abstraktionslager fullt implementerat. Det system av celler och portaler som är under utveckling är ett naturligt lager att lägga ovanför sökningen i de genererade kartorna. Vid sökning i komplicerade tredimensionella strukturer, exempelvis inuti byggnader, är det mer eller mindre ett krav att använda celler och portaler för att en sökning i tvådimensionella kartor ska fungera. Däremot är det tveksamt hur mycket cell-portal-systemet förbättrar vägsökning i utomhusmiljö, eftersom en stor cell troligtvis kommer att representera hela denna vilket inte innebär någon betydande förändring mot dagens system.

Med ett fungerande cell-portal-system delas sökningen lämpligen i två lager. Det översta är en sökning mellan celler där de passerade portalerna sparas som delmål för den kommande finsökningen i det undre lagret. På så sätt minskas storleken på kartorna eftersom endast kartor för de passerade cellerna behöver genereras. Både själva finsökningen och kartgenereringen bör alltså ta mindre tid. Däremot är det tveksamt om den totala söktiden minskar eftersom sökningen i det abstrakta lagret kan neutralisera tidsvinsten i finsökningen.

4.4 Sökning genom abstraktionen

Vid valet av sökalgoritm var de två viktigaste aspekterna den statiska världen och kravet på ett naturligt beteende hos NPC:erna. Realtidsalgoritmer, som endast planerar och exekverar ett steg per iteration, avfärdades tidigt eftersom de hade varit svåra att anpassa utan en avancerad högnivåabstraktion av världen.

Omgivningen i Project Entropia är dynamisk med avseende på andra NPC:er och spelare. Terräng och begränsningsvolym är statiska och NPC:er har inte förmåga att manipulera sin omgivning. Den dynamik som uppstår på grund av andra NPC:er behandlas inte av vägsökningen utan i de fall då en kollision mellan två NPC:er uppstår faller beteendet tillbaka på stimuli-responsfunktionen i känslspröten. Det är möjligt att systemet i framtiden kommer att innehålla någon form av oförutsägbarhet och det var

därför lämpligt att antingen välja en dynamisk metod direkt eller en statisk som lätt kan skrivas om för att hantera dynamiken som uppstår.

När målet för en vägsökning är rörligt uppstår en form av dynamik som ger helt nya förutsättningar för sökningen. Sökalgoritmer anpassade för dynamiska omgivningar begränsas vanligtvis av att de inte kan hantera rörliga mål. Undantag finns, tex MTS, men dessa är alltför långsamma eller funktionellt begränsade för att kunna användas i praktiken. Då ingen sökalgoitm anpassad för dynamiska omgivningar hanterar rörliga mål tillfredsställande lades funktionalitet för prediktion av målpunkten till sökningen. Prediktionen förutser det rörliga målets position med avseende på avstånd, hastighet och riktning. Med ett visst intervall kontrolleras målets position och om det rört sig utanför en toleransradie från målpunkten görs en ny prediktion och därefter en sökning.

Det rutnät som levereras av kartgenereringen för sökning är så enkelt det kan bli med booleska variabler i en tvådimensionell array som markerar att ett område antingen är blockerat eller icke-blockerat. Möjligheten att exakt uppskatta avståndet mellan valfri nod till målnoden ger en utmärkt heuristisk funktion och gör en A*-algoritm i enklaste form lämplig för sökningen. Skillnaderna på A* och den dynamiska varianten D* är också så små att det skulle vara lätt att lägga till den extra funktionaliteten om behovet skulle uppstå. Eftersom A* är en off-line algoritm måste en hel sökning genomföras varje gång målet rört sig. Med hjälp av toleransradien begränsas antalet sökningar till rörliga mål eftersom små förflyttningar av målet inte utlöser en ny sökning.

4.4.1 Designbeslut för sökmetoden

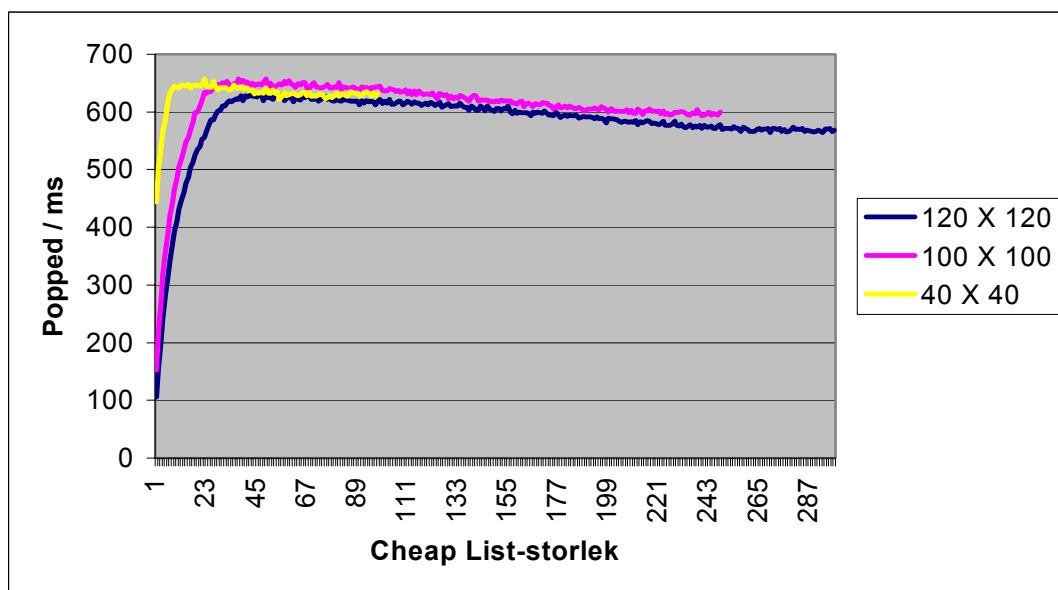
Vilken datastruktur som används för prioritetskölagring av avsökta noder är kritiskt för prestandan och med kravet att noderna antingen måste vara sorterade efter sin kostnad eller hämtas ut sorterat så finns det några alternativ. Dessa är tillsammans med deras värsta-falls-komplexiteter:

Datastruktur	O(insättning)	O(uthämtning)
Osorтерad lista	1	N
Sorterad länkad lista	N	1
Sorterad heap	log N	log N

Tabell 2 Tidskomplexitet för sorterad insättning eller uthämtning ur datastrukturer.

I nedanstående text är det viktigt att hålla isär **expanderade** noder från **genererade**. En nod som expanderas är den för tillfälligt billigaste noden i prioritetskön. Den hämtas ut och genererar sina grannar som stoppas in i kön. Därefter är den expanderade noden förbrukad. En observation av A*-algoritmen ger att det blir många fler genererade noder, dvs såna som måste sättas in i datastrukturen, än noder som blir expanderade, dvs såna som måste hämtas ut ur strukturen. De tester som utförts visar att en ren sorterad länkad lista och en heap-lösning har ungefär samma prestanda. I vårt fall ungefär 250 expanderade noder per millisekund. Antalet lagrade noder blir inte så stort att heap:en hinner löna sig eftersom den har en viss konstant merkostnad.

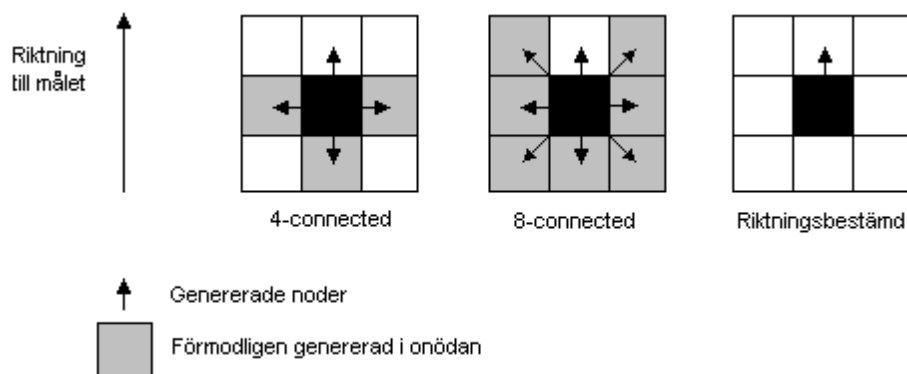
Nästan tre gånger så bra prestanda, 650 expanderade noder per millisekund, kunde nås genom en Cheap List implementation som beskrivits i kapitel 3.5.4. För att dimensionera storleken av Cheap List:en gjordes experiment med olika storlekar på sökområdet. Resultaten presenteras i Figur 15. För små kartstorlekar räcker det med en kortare lista och en för stor lista resulterar i sämre prestanda i samtliga fall. En avvägning gjorde att storleken fastställdes till 75 noder.



Figur 15 Prestandaberoende av Cheap List-storleken.

De 75 billigaste noderna hålls hela tiden på en kort sorterad lista och övriga läggs i en osorterad. Att hämta ut den billigaste noden får en tidskomplexitet på $O(1)$. En genererad nod som inte platsar bland de 75 billigaste placeras på den osorterade dyra med $O(1)$. Om den genererade noden ska in på den billiga listan går det åt $O(k)$ där k alltså som högst är 75. I praktiska försök visar det sig att den billiga listan sällan blir tom, dvs att man måste hämta ut 75 nya noder ur den dyra listan vilket är mycket dyrt.

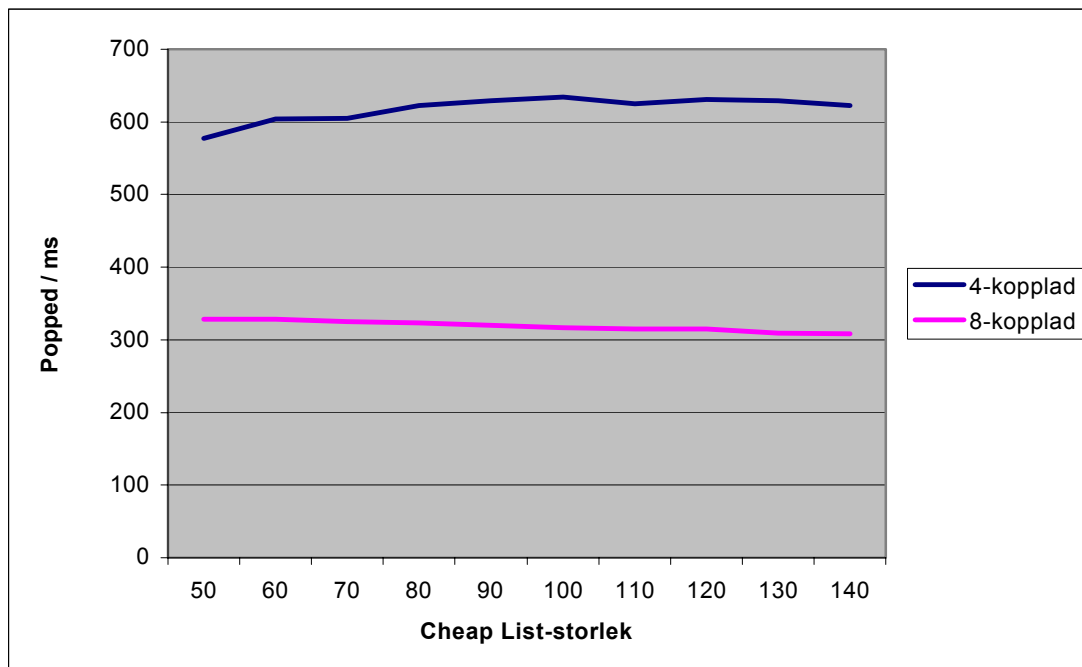
Vid expanderingen av en given nod kan man välja på olika strategier. De två standardalternativen är en 4-kopplad eller en 8-kopplad sökning (se Figur 16). En 4-kopplad genererar de fyra grannar som ligger på koordinataxlarna medan en 8-kopplad också tar med de diagonala grannarna. Vid en första anblick borde den 8-kopplade vara det snabbare alternativet eftersom antalet noder som måste expanderas blir färre om det är tillåtet att snedda. I praktiken händer det sällan att mer än en granne måste expanderas och därför blir det en stor merkostnad att generera de sju onödiga grannarna.



Figur 16 Översikt av expansionsstrategier.

Under praktiska försök fick vi nära en dubbling av prestandan med det 4-kopplade alternativet (se Figur 17). Försöken utfördes i en separat testapplikation. Ett tredje alternativ vid nodgenerering hade varit att bara generera noder som låg i riktning mot målet. Inga praktiska tester har utforskat den lösningen. Kostnaden för att beräkna

riktning och merkostnaden för att hantera hinder i de fall de uppstår är troligen överstigande det algoritmen skulle tjäna på färre insorteringar. I den undersökta litteraturen nämns alternativet på ett ställe som intressant att undersöka men liknande reservationer görs.



Figur 17 Jämförelse mellan genereringsstrategier

Sökningen kan terminera på fyra olika sätt:

- Sökningen når från start till mål.
- Sökningen når en kartkant innan målet uppnåtts.
- En tidtagare avbryter sökningen för att för lång tid har använts.
- Det finns inga fler noder att expandera.

Alternativ två och tre är egenvalda. Om sökningen når en kartkant avbryts sökningen och bufferten runt kartan dubblas. Detta inträffar relativt sällan. Med en större buffert från början skulle vi få färre sökningar utanför kartan men oftast en onödigt stor karta. Avvägningen som gjorts är en buffert på som till en början består av 20 noder extra i varje riktning. För varje misslyckad sökning dubblas denna buffert tills kartstorleken överstiger det maximala då hela sökningen anses misslyckad.

Ett alternativ är att inte bryta sökningen när en kartkant nås. Då skulle alla noder expanderas om inte målet hittas. Vid en första anblick kan det se konstigt ut att avbryta en sökning där det finns vägar kvar att undersöka men mycket ofta beror en sökning som går utanför kanten på att målet är blockerat i den aktuella kartan och en större krävs. Därför går onödigt tid till att tömma alla noder i OPEN-listan istället för att avbryta och börja med en större karta. Genom att modifiera vikten på heuristikfunktionen och buffertens storlek kan antalet sökningar som går utanför kartkanten kontrolleras.

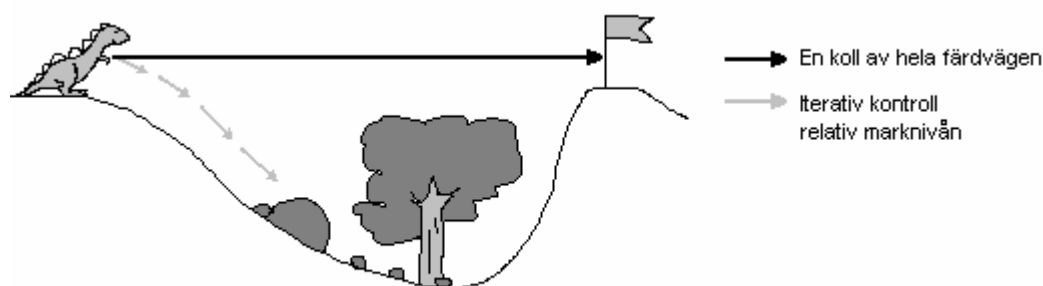
Tidtagaren bryter sökningen när mer än 200 millisekunder gått åt, men en normal sökning använder omkring en mikrosekund. Den är inkluderad för att hela världen inte ska hamna i ett låst läge pga något oförutsett programmeringsfel och har inte utlöst

under våra praktiska försök. Den första tanken med tidsbrytaren var att kunna använda tidsdelning (eng. *timeslicing*), att fördela sökningen över flera anrop. När algoritmen var tillräckligt färdig för att kunna testas visade det sig att själva sökningen i kartan tar en så liten del av beräkningstiden att det inte blev meningsfullt att dela upp denna.

Som tidigare beskrivits omges alla begränsningsytor av en navigeringsbuffert för att få NPC:erna att hålla sig på ett visst avstånd från hindren. Pga synkroniseringstid och en viss masströghet kan NPC:erna ändå råka halka in i områden som enligt navigeringskartan är oåtkomliga. När sökningen i kartan misslyckas kommer NPC:n att stanna så fort som möjligt och minska ner navigeringsbufferten. Den har ett antal chanser på sig att lyckas med sökningen, för varje gång minskas bufferten ner. I det sista försöket är navigeringsbufferten negativ och begränsningsytorna dras in en halv meter för att eliminera avrundningsproblem. Om fyra på varandra följande sökningar startar från samma absoluta koordinater så innebär det att sökalgoritmen har misslyckats. Skriptet tar emot en avbrytkod som retur från TickPathfinding-funktionen och väljer ett nytt mål.

Genom att använda en viktad A* som överskattar den heuristiska kostnaden ger sökningen en direktare väg och snabbare sökning på bekostnad av optimaliteten. Avvägningen som görs är minnesåtgång mot att få den optimala vägen utan någon putsning. Eftersom implementationen ändå gör om den beräknade färdvägen till ett fåtal delmål blir den praktiska skillnaden försumbar. I de praktiska försök som utförts har värden mellan 1.0, en ren A*-sökning som ger en optimal lösning, och 2.0 som är en stark viktning testats. För låg vikt gör att sökningen går bakåt från startpunkten så att den riskerar att hamna utanför den buffert som omger sökmatriken mellan start och mål. Hög vikt övervärderar kostnaden för att ta sig till målet vilket gör att hinder nära målet utforskas överdrivet grundligt innan en alternativ väg testas. I den valda implementationen fastställdes efter praktiska försök vikten till 1,6.

Ett sätt att slippa både sökningen och kartgenereringen är att testa för kollisioner längs med en vektor mellan start och mål i den tredimensionella världen. I en majoritet av fallen finns inga hinder och hela vägsökningen visar sig i efterhand onödig. Praktiskt skulle en sådan vektortest dock behöva göras iterativt i så små steg att det i sig skulle bli som en sökning i världen. Detta pga höjdskillnaderna i terrängen som gör att vektorer kan svepa under eller över objekt som under förflyttning visar sig hindra NPC:n.

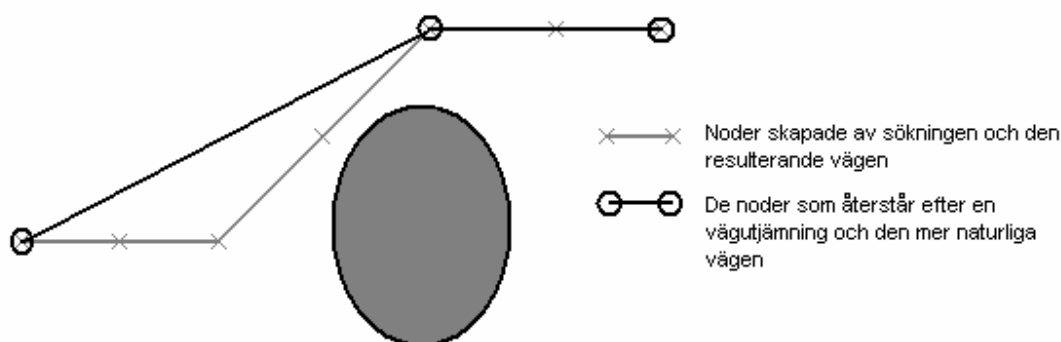


Figur 18 Problemet med en vektorkontroll av ohindrad väg.

4.4.2 Efterbehandling av sökningen

När nodkostnaderna från start till mål skapats stegar funktionen tillbaka från mål till start genom att alltid gå till den billigaste noden samt lagra denna i en array. Till slut har funktionen nått tillbaka till startnoden och har en lista med noder som ska passeras längs med vägen.

Det vore möjligt att returnera den här listan med alla koordinater som ska passeras direkt till skriptet men där vore den otymplig att navigera efter. NPC:n ska inte detaljstyras steg för steg utan algoritmen ska kunna leverera mer övergripande mål som måste passeras, tex runt vänstra hörnet på byggnaden, därefter till höger om brunnen. Genom att dra räta linjer mellan stegen parvis och eliminera dem som inte har något hinder mellan sig så får NPC:n istället de eftersträvade absolut nödvändiga koordinaterna som ska passeras. Detta moment kallas i litteraturen *Waypoint Smoothing*, delmålsutjämning. Förutom att skriptet får färre punkter att passera ger det också ett mer naturligt undvikande av hinder eftersom färdvägen rätas ut så den blir så rak som möjligt.



Figur 19 Delmålsutjämning.

4.5 Implementeringens gränssnitt

Den funktionalitet som utvecklats under examensarbetet består av en Pathfinder-klass i C++ som erbjuder BOLD ett fåtal metoder:

```
void    ActivatePathfinder(int p_iID)
void    DeletePathfinder(int p_iID)
void    SetPositionTarget(int p_iID, int p_iAX, int p_iAY)
void    SetMovingTarget(int p_iID, int p_iTargetID)
int     TickPathfinding(int p_iID)
int     GetNextWaypoint(int p_iID, int* p_pAX, int* p_pAY)
```

Dessutom implementerades avlusningsfunktioner för att rapportera händelser i BOLD ner till Pathfinder-objektet.

4.5.1 Praktiskt användande av vägsökaren

Den första inparametern, `p_iID`, i våra C++-metoder är ID-numret på ett MovableObject (MO). När ett MO vill förflytta sig sätter den sitt mål med endera ett fast koordinatmål med `SetPositionTarget` eller så slår den på målföljning av ett rörligt mål med `SetMovingTarget`. Båda funktionerna kontrollerar att det finns ett vägsökarobjekt redo, om så inte är fallet så skapas det ett och en pekare sätts från MO:et till vägsökaren.

När en NPC på detta sätt har aktiverats läggs den också till i skriptetekverarens schemaläggning. Tidsintervallet mellan dessa aktiveringar, sk ticks, är typiskt 300 ms. Vid en sådan tick uppdateras NPC:ns beteende. Normalt sätts endast rotation och hastighet till önskade värden men den kontrollerar också andra beteendestyrande faktorer som tex om den tagit så mycket skada att den ska fly och därmed sätta nya mål.

Under varje tick anropas vår metod `TickPathfinding`. Metoden uppdaterar vid behov NPC:ns sökmatrix och / eller sökning. Om returkoden signalerar till skriptet att endera har uppdaterats så hämtas en ny målpunkt ut med metoden `GetNextWaypoint`. Om inget delmål finns tillgängligt så kan det ha två orsaker. Antingen finns det en pågående sökning eller så har MO:et passerat en geografisk servergräns vilket innebär att vägsökarobjektet som anropas är nyskapat och inte har fått något mål satt. I båda fallen signaleras incidenten till skriptet som sätter om det gamla målet. För en mer detaljerad beskrivning av funktionerna se Bilaga 2: Pseudokod.

När ett delmål är uppnått begärs ett nytt delmål tills det att NPC:n är framme vid slutmålet. Pathfinder-objektet använder sig av olika returkoder för att ange statusen på det delmål som returneras:

- Ett delmål på vägen, ej det sista. Skriptet ska navigera mot den angivna koordinaten och när den passeras begära ett nytt delmål.
- Det sista delmålet. När det nås har skriptet uppnått sitt syfte och ska välja ett nytt mål eller ett nytt beteende.
- En ad-hoc-koordinat. Sökningen har misslyckats men vägsökaren försöker rädda situationen genom att expandera kartan eller dra in navigeringsbuffertar. Om NPC:n är på väg till en absolut koordinat returneras NPC:ns egen position. Det innebär att den kommer att stå stilla tills sökningen lyckas eller avbryter. Om NPC:n istället följer ett rörligt mål returneras målets sista absoluta koordinat. Det medför en risk att NPC:n springer in i hinder men det är att föredra framför att bli ett stillastående mål.
- En avbrytkod. Sökningen har misslyckats och det finns inget hopp om att den någonsin kommer att lyckas eftersom navigeringsbuffertar är indragna så långt det går och kartan är expanderad till maxstorlek. Detta inträffar endast när målet är oåtkomligt, tex bakom en mycket lång vägg eller inne i en byggnad utan åtkomliga öppningar. Skriptet måste välja ett nytt beteende.

När NPC:n inte längre har behov av vägsökarobjektet, antingen då den nått fram till ett slutgiltigt positionsmål, passerat en servergräns eller bragts om livet, körs `DeletePathfinder` för att ta bort de spår av sökningen som skriptet inte känner till.

5 Resultat

De mål som sattes upp vid arbetets början får anses uppfyllda i den mån som senare visade sig vara praktiskt möjlig. Algoritmen klarar både inom- och utomhusmiljöer utan någon särskiljning. Delmålslösningen visade sig vara lämplig för det gamla systemet precis som förstudien indikerade.

KAN-målen som inte har uppnåtts visade sig under arbetet olämpliga för körning i realtid. Hänsynstagande till höjdskillnader ger en dimension till i sökningen och en mångfaldigad tidsåtgång. Detsamma gäller för beräkning av andra rörliga agents påverkan i ett taktiskt lager. Inte förrän en självständig server är fullt implementerad kan fler lager bli aktuella.

5.1 Vägsökarens prestanda

I början av utvecklingen av vägsökaren lades mycket arbete ner på optimering av sökningen genom matrisen istället för genereringen av densamma. Som tidigare nämnts var det inte här energin var bäst spenderad. Det som är mest tidskrävande i implementationen är genereringen av kartan. Tidsåtgången i denna vid inläsningen av en ny karta kan delas upp på tre punkter:

- Noder som berörs av sökområdet återställs i den förallokerade matrisen.
- Statiska objekt hämtas ut ur kollisionshanteraren.
- Objekten roteras, klipps ner, filtreras och ritas in i matrisen.

Därefter tillkommer själva sökningen:

- A*-sökning från start till mål.
- Delmålsutjämning.

Den första optimeringen som litteraturen rekommenderar för vägsökning är att vägsöka så sällan som möjligt. När NPC:n jagar ett rörligt mål ställs det höga krav på ett naturligt följande av målet från spelarna. Det duger inte att uppdatera med några sekunders mellanrum eftersom spelaren då kan lura iväg NPC:n med snabba svängar. I implementeringen används två olika utlösare för att bedöma när färdvägen mot ett rörligt mål måste uppdateras.

- En tidtagare som jämförs mot en gräns som är linjärt beroende av avståndet till målet. I praktiken varierar den mellan .25 och 5 sekunder.
- En radie runt den senaste använda positionen hos målet som jämförs med den aktuella positionen.

Dessutom tillkommer en utlösare som är den enda som används om NPC:n navigerar mot ett fast mål. Efter 50 ticks, ca 20 sekunder, utan någon uppdatering av kartan tvingas en sådan fram. Anledningen till denna är att NPC:n kan ha hamnat i ett utrymme som kollisionsnavigeringen inte klarar av att hitta ut ur. Avvägningen är mellan hur ofta ett låst läge kan accepteras kontra hur mycket beräkningsarbete som ska sparas.

Om någon av ovanstående utlösare har aktiverats måste en ny färdväg beräknas. Dock är det inte säkert att en ny karta måste genereras. Punkt två och tre i den tidigare beskrivna

inläsningen av kartan kan undvikas om start och målpunkten ligger inom det gamla sökområdet. I så fall räcker det med att återställa sökvärdena för kartnoderna. Under de praktiska testerna har endast tider inom millisekundområdet kunnat mätas varför det är svårt att säga något konkret om tidsvinsten i en enskild sökning. När flera NPC:er navigerar ett svårt område med kartåtervinning jämfört med utan har vi märkt av en minskad processorbelastning på mellan 10 och 30 procent. Uppskattningen är grov eftersom det enda mätinstrument som då fanns att tillgå var det som är inbyggt i operativsystemet.

Den implementation som valts återanvänder inte någon del av den tidigare sökningen. Sökrymden är så liten att merkostnaden för att kontrollera om en ny sökning skär en gammal med stor sannolikhet överstiger den tidsvinst som kan erhållas.

5.2 Praktiska tester

De tester som redan behandlats under "Vägsökarens prestanda" har rört den krävda processorkapaciteten och hur det går att optimera själva sökningen. Det praktiskt intressanta, givet att sökningen håller sig inom tillgängliga resurser, är NPC:ernas beteende som det uppfattas av spelarna.

5.2.1 Empiriska jämförelser

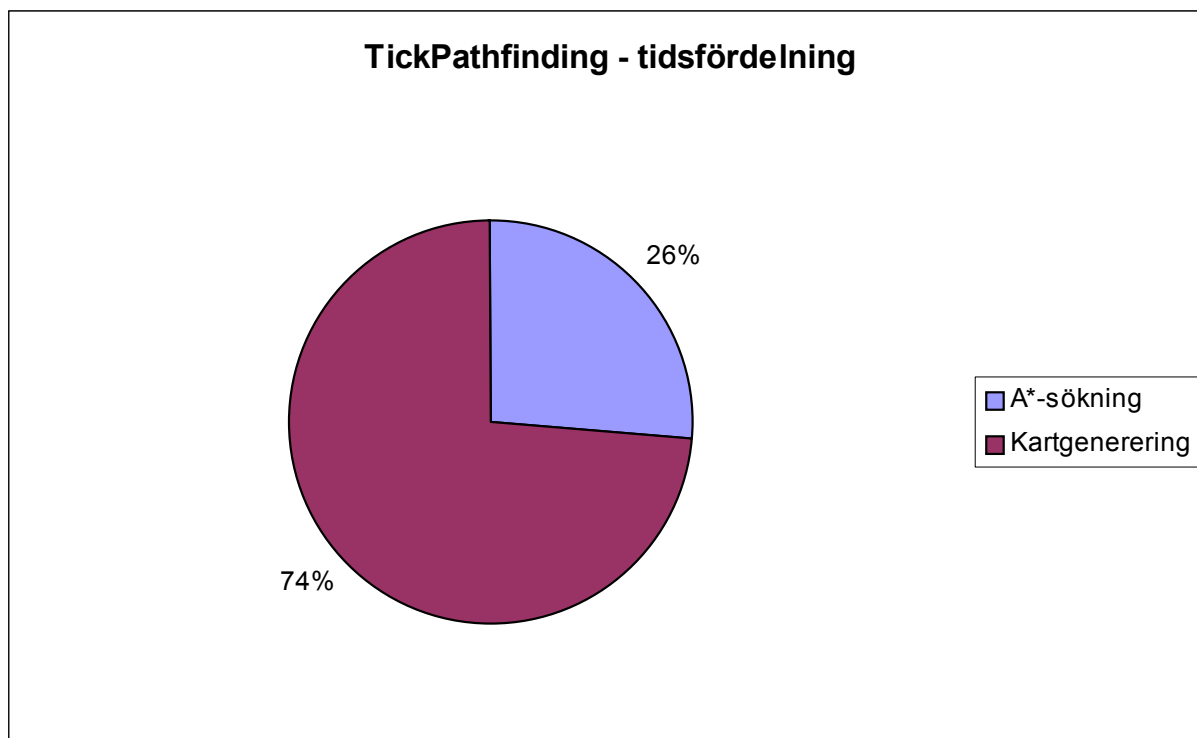
Det som finns att jämföra med är det rena kollisionsundvikande beteende som vägsökaren utnyttjar sig av för navigering mellan delmål och som var det enda som fanns i bruk innan examensarbetet påbörjades. NPC:erna styrde rakt mot sitt mål tills något av känselspröten signalerade för en kollision. Agenten styrde då åt motsatt håll. Spelarna kunde lätt lura in NPC:n i terräng som var svår att navigera igenom vilket resulterade i att de blev stillasittande mål.

Testfallen vi har valt är dels en öppen terräng med träd, kallad Utomhusmiljön, och en Inomhusmiljö med en labyrint av containrar. Inomhusmiljön är inte navigerbar utan vägsökning eftersom NPC:n snart fastnar i en icke-konvex sammansättning av begränsningsvolymmer. I Utomhusmiljön klarar det gamla systemet sig hyggligt. NPC:n springer gärna in i träd men den väjer runt dem och lyckas följa ett rörligt mål.

Med vägsökningen påslagen klarar NPC:n av vad spelarna kan kräva. Den rundar hinder och följer den kortaste vägen till målet. Pga de tidigare nämnda orsakerna, synkronisering mellan klient och server och yttre påverkan, kan NPC:n ändå råka kollidera med hinder. Det kollisionsundvikande systemet klarar i en majoritet av fallen dessa och efter upprepade misslyckanden beräknas en helt ny färdväg runt hindret.

Den brist som kan irritera spelarna är NPC:ns totala världsbild. Målets position är alltid känd. Det går inte att lura av sig NPC:n inuti en byggnad tex eftersom den vet allt om omgivningen. En funktion är förberedd för att låta bli att uppdatera ett rörligt måls position om siktlinjen är bruten av en begränsningsyta. Den lösningen är inte optimal eftersom de träd som genereras av vegetationsmotorn skapas på klientsidan och inte har några begränsningsvolymmer på servern. Detta medför att NPC:n fortfarande kan tyckas vara medveten om spelaren för tidigt.

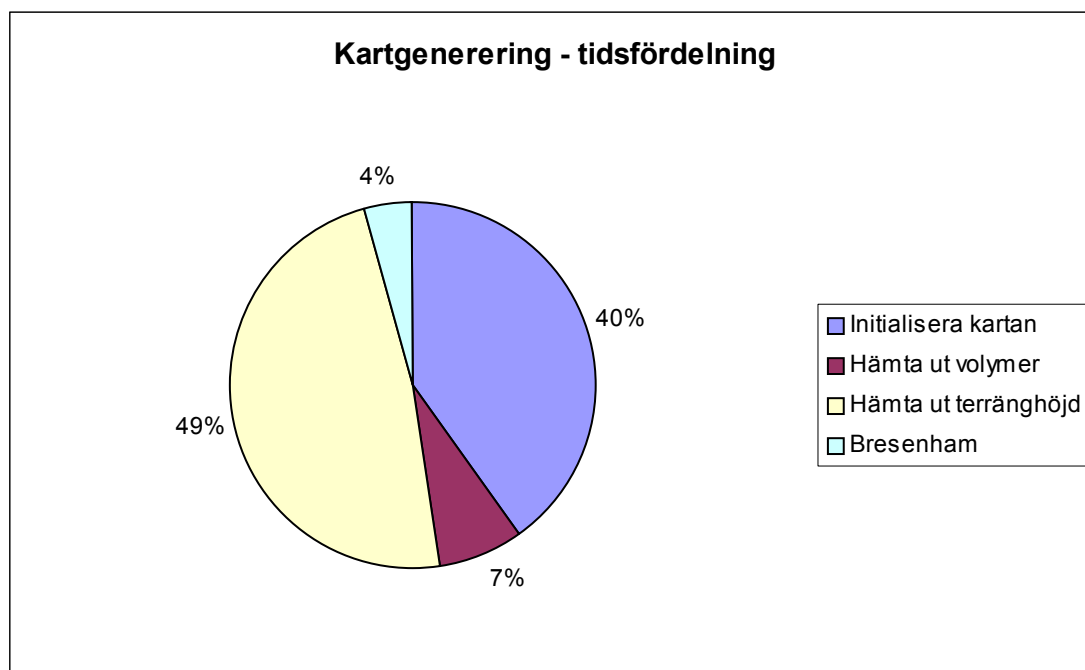
5.2.2 Statistisk analys



Figur 20 Tidsfördelning mellan abstraheringen och sökningen.

Diagrammet ovan är skapat ur data genererat av programmet Intel® VTune™. Det som framstår tydligt är att, med våra valda parametrar för kartåtervinning, så är det kartgenereringen som tar nära $\frac{3}{4}$ av tiden vid en exekvering. Exemplet ovan är en blandad miljö med NPC:er som jagar dels positionsmål och dels rörliga mål. Om uteslutande rörliga mål följs kommer andelen tid som förbrukas av sökningen att stiga då kartan ofta kan återanvändas och tvärtom om fasta mål följs eftersom det då oftast räcker med en kartgenerering och en sökning.

Tidsåtgången för sökningen är jämnt fördelad mellan beräkningen av heuristikfunktionen och hanteringen av söknoderna. Som heuristikfunktion används det euklidiska avståndet vilket innebär flera multiplikationer och en rotfunktionsberäkning; funktioner som är allmänt kända som särskilt beräkningsmässigt dyra. Att byta till ett Manhattanavstånd ger dock inte tillräckligt stora vinster för att kunna försvara de egenheter som uppstår i NPC:ernas undvikande av föremål. Eftersom noder genereras i en bestämd ordning, Nord-Syd-Öst-Väst, så undviker NPC:er tex oftast föremål på en bestämd sida och de kan få för sig att springa långa omvägar eftersom de inte förstår tidsvinsten med att springa diagonalt.



Figur 21 Medeltidsåtgång vid en kartgenerering.

Kartgenereringen står alltså för den största delen av exekveringstiden. De operationer som intuitivt upplevdes som mest tidskrävande, uthämtningen av volymer ur oct-trädet och inritningen av dessa i kartan med Bresenham's algoritmen, överröstas av de små operationer som utförs många gånger.

Det som är dyrt i kartinitialiseringen är den dubbla for-loop som itererar igenom och återställer sökrummet. Det är inte troligt att en omskrivning till en sk memset-operation skulle ge någon tidsvinst eftersom en sådan kompilerad till maskinkod ger samma funktionalitet och instruktioner.

Överraskande är att tiden för att läsa av terränghöjden under en begränsningsvolym står för en tredjedel av den totala körtiden. Det har visat sig att det är en oväntat avancerad operation som löser planets ekvation för den terrängyta som den undersökta punkten ligger på. Förmodligen skulle en enklare operation, tex ett medelvärde av höjderna hos terrängytans hörn, ge ett praktiskt acceptabelt resultat till en mycket lägre kostnad.

5.3 Slutsatser

De mål som ställdes upp efter förstudierna har uppnåtts. Vissa aspekter av implementationen är värda att framhålla som särskilt lyckade medan andra har blivit kompromisser mellan vad som önskades och vad som var praktiskt möjligt.

Systemet fungerar i sitt nuvarande skick. Det har integrerats med MindArk AB:s tidigare lösning och är fullständigt bakåtkompatibelt. Vägsökningen kan slås av och på under körning friktionsfritt eftersom navigeringen faller tillbaka på det ursprungliga systemet automatiskt om det inte finns några färdiga beräknade delmål.

Ett annat problemområde har varit passerande av servergränser som ligger mellan de delar av världen som respektive server har ansvar för. Det hanteras också utan problem med återsättning av mål om servern upptäcker att en NPC som navigerar med en vägsökare hamnat under dennes ansvar.

Med hjälp av analysverktyg har algoritmens exekveringstid kunnat reduceras ner till mindre än en fjärdedel av den ursprungliga implementationen. Det som återstår är den tid som går åt för att hantera data och de matematiska operationer som krävs. Om någon ytterligare optimering skulle bli aktuell måste denna troligen ske på maskinskodsnivå. Det nya systemet har visat sig kunna hantera nästan lika många NPC:er som det gamla. Det får anses som mycket bra med tanke på den betydligt mer avancerade funktionaliteten som lagts till.

De delar som är mindre tilltalande är bl a att serverns skriptinterpretator anropar vägsökaren synkront. Det innebär att andra, eventuellt tidskritiska, händelser får stå tillbaka tills vägsökarens beräkning är färdig. En uppdatering av vägsökaren tar i medel mindre än en mikrosekund och den maximala körtiden är grovt uppskattad en knapp millisekund. Hur stora praktiska problem detta innebär är inte klarlagt.

Målsättningen från början var att kunna inkludera flera taktiska lager som beräknades i realtid. Det visade sig senare inte vara praktiskt möjligt pga de krav det skulle ställa på servern. Det finns idag inte heller något stöd för dylika söklager även om det vore ett mindre arbete om det skulle bli aktuellt. Ett taktiskt lager skulle förmodligen bidra till en bättre upplevelse för spelarna då NPC:erna rör sig mer naturligt.

Eftersom MindArk AB har valt att inkludera vägsökarfunktionaliteten i den skarpa versionen av spelet antas att de ändå blivit nöjda med resultatet av arbetet.

6 Utvecklingsmöjligheter

Under utvecklingen av implementationen har vi valt bort ett flertal möjliga alternativ på lösningar och andra har dykt upp för sent för att hinna inkluderas. De är ordnade efter effekt per mängd arbete med den mest lovande först.

6.1.1 Separat server

Något som nämnts från början av arbetet men som ändå valdes bort på grund av en för snäv tidsram och examensarbetets avgränsningar var att lägga ut vägsökningen på en separat server. Den valda implementationen är ändå framtagen med detta i åtanke och det praktiska arbetet för att lägga till en RMI-liknande struktur bör vara begränsat. Det större problemet blir istället att sätta sig in i hur kartstrukturen är lagrad på servern idag och att ta fram ett protokoll för kommunikationen.

Genom att lägga ut vägsökningen separat begränsar man funktionaliteten eftersom det inte blir möjligt att komma åt dynamiska objekt från servern utan en omfattande kommunikation.

Det finns stora fördelar med att kunna använda en förgenererad terräng. Flera av de senare behandlade punkterna om Taktiska Lager och Vatten kan förberäknas och behöver då inte kosta någon tid under körning.

6.1.2 Celler och Portaler

Ett sätt att minska ner antalet objekt som måste hanteras i kollisionshantering och grafikrendering är att dela upp världen i celler som är separerade från varandra med portaler. Spelaren kan oftast inte se ut ur den cell han befinner sig i och därmed behöver servern endast hantera objekt som ligger i samma cell.

För vägsökaren finns också fördelar eftersom portalerna blir en lämplig lösning på problemet med höjdskillnader och broar. I ett flervåningshus kan en portal förbinda olika våningsplan med en trappa. En bro kan ha en portal i varje ände och underlätta för navigeringen hos NPC:erna. Det extra abstraktionslagret kan också effektivisera sökningen eftersom summan av flera delsökningar kan understiga en fullständig sökning.

En svårighet som uppstår är valet av heuristik. Det finns inget bra sätt att definiera avståndet mellan en godtycklig cell och målcellen eftersom portalerna inte pekar i någon bestämd riktning. Istället är det lämpligt att använda en Dijkstra eller en ren BFS för att söka sig genom portalnätet.

Under utvecklingen av den implementation som presenteras i examensarbetet har ett stöd för portalsystemet testats men eftersom det inte är färdigutvecklat måste gränssnittet skrivas om när det ska inkluderas i den skarpa versionen.

6.1.3 Effektivare datastruktur för abstraktionen

Eftersom terrängen utgörs av sammanhängande och förhållandevis stora blockerade eller icke-blockerade områden är minnesåtgången onödigt stor då en tvådimensionell array med fast storlek per element används. Istället kan en datastruktur som effektivt lagrar dessa områden som ett element användas. Ett exempel på en sådan datastruktur är quad-

träd. Om upplösningen i quad-trädet begränsas fås också en rimlig gräns på tidsåtgången för att hämta begränsningsvolymerna inom ett givet område.

6.1.4 Vatten

Ett flertal av NPC:erna ska vara hindrade av vatten. Idag finns inget stöd för detta eftersom det inte finns något enkelt sätt att markera ut vattnet i sökmatriken. Som tidigare nämnts ligger vattennivån på en viss global höjd i terrängen. För att kunna markera in detta till sökningen måste vi i varje söknod hämta ut ett höjdvärde ur terrängen. Det är inte praktiskt möjligt i det system som används idag pga den beräkningsbelastning det skulle medföra. Även om genereringen tar ut ett höjdvärde på en större yta än söknoderna så blir det en avvägning mellan NPC:ernas beteende nära vatten och beräkningskraven.

Ett enkelt alternativ är att lägga ut begränsningsytor över alla vattentäckta områden som endast kan passeras av vissa NPC:er och spelare. Det skulle göra kartgenereringens jobb enkelt eftersom ytorna skulle ritas in precis som andra statiska hinder men det skulle betyda ett stort merarbete för personalen som designar världen. Servern skulle dessutom behöva lägga in en kontroll vid varje kollisionsjämförelse för att se om det är två objekt som kan kollidera. Den extra belastningen är inte acceptabel.

Med vägsökningen på en separat server kan allt vatten ritas in på förhand utan särskilda begränsningsvolym genom att läsa av höjdkartan i varje punkt.

6.1.5 Upplösning

I den lösning som implementerats används en statisk upplösning i sökmatriken. Det finns stora vinster med att göra den relativ till NPC:ns storlek eftersom den nu är anpassad efter de högsta kraven på upplösning. NPC:erna varierar från hamsterstorlek till större dinosaurier. De största NPC:erna är inte intresserade av att kunna navigera med samma noggrannhet som de minsta. Både minnesåtgång och beräkningstid skulle minska proportionellt mot storleken på noderna i sökmatriken. I litteraturen föreslås en minsta nodbredd motsvarande halva storleken hos NPC:n för att alla passager ska kunna navigeras.

6.1.6 Trådbarhet

Den lösning vi implementerat använder sig endast av synkrona anrop. Det kan leda till problem eftersom interpretatorn körs på en enda processortråd. Av funktionerna som vägsökningen använder sig är det endast TickPathfinding som kan behöva trådas. De övriga gör endast kortare operationer som att sätta eller hämta ut variabler och där skulle det ställa till med mer problem än nytta att skicka ut anropen på en separat tråd. Utan att ha undersökt det närmare är det troligt att en trådad lösning skulle klara sig utan semaforer för vägsökarobjektet. Mellan ett TickPathfinding-anrop och ett därpå följande GetNextWaypoint går det minst ett Tick, dvs åtminstone 300 millisekunder. Det är högst otroligt att TickPathfindingen inte hinner schemaläggas och exekveras under den tiden och även om så sker så finns de gamla delmålen lagrade ända tills metoden returnerar.

6.1.7 Siktlinjer

NPC:erna har en komplett bild av sin omvärld. Det är acceptabelt att de känner till alla statiska objekt men de bör inte kunna känna av hur en spelare rör sig inne i ett hus eller när den är skyddad av terrängen. Ett lätt sätt att införa detta är att vid en uppdatering av målets position kontrollera vektorer från NPC:n till ett antal punkter på målets kropp. Om alla eller en stor andel av dessa är skyddade av fasta objekt låter NPC:n bli att

uppdatera positionen och jagar mot den senast kända. Det skulle innebära att spelare kan utnyttja höjdskillnader och byggnader på ett intressantare sätt för att undkomma eller lura in datorstyrda agenter i taktiskt fördelaktiga situationer.

Ett intressantare beteende kan dock slå fel om spelarna kommer på ett sätt att systematiskt utnyttja det till sin fördel. Alla dylika ändringar måste testas ingående innan de släpps ut i den skarpa världen.

6.1.8 Taktiska lager

I den A*-lösning som beskrivits används endast boolska-variabler för att beräkna kostnaden för att traversera en nod. I intervallet $[0, \infty]$ finns ett spelrum för att lägga till andra kostnader som representerar hur lämplig noden är för NPC:n. De uppenbara lagren är tex kostnader kopplade till höjdskillnader och varierande kostnad för olika terräng men det blir också möjligt att ta hänsyn till spelare som utgör hot och att lägga till tex flockbeteenden. De statiska lagren för terrängkostnader kan förberäknas och lagras, antingen tillsammans med de statiska objekten eller separat. Taktiska aspekter måste dock beräknas i realtid och servern där vägsökningen körs måste också vara medveten om rörliga objekt. Behovet av mer beräkningstid talar för en separat server men om för många rörliga objekt måste synkroniseras mellan serverna väger det upp fördelarna.

6.1.9 Dynamisk sökning

En svaghet med den valda lösningen är att den kräver en statisk omgivning. Två NPC:er som håller på att kollidera är omedvetna om varandra tills deras kollisionsundvikande känselspröt reagerar. Dörrar som öppnas och stängs uppdateras inte i kartan förrän en ny lista av statiska objekt hämtas ut och blir då låsta i det läget tills en ny uppdatering sker. En variant på A* som nämnts flera gånger tidigare är D* som klarar av att hantera oförutsedda hinder utan att göra en total omplanering. Viss ny funktionalitet måste tas fram i gränssnittet mellan skript och C++-kod, som rapporteringen av diskrepanser mellan sökmatrisen och den verkliga världen, men det är ett litet problem om det skulle bli ett krav att kunna hantera dynamiska situationer.

6.1.10 Axeljustering av sökmatrisen

Mycket onödig data genereras i områden som med stor sannolikhet inte kommer att beröras av sökningen. Om sökmatrisen inte skulle använda sig av koordinater på världsaxlarna utan istället vara vriden mot målet, som det grå området i Figur 4, så kunde kartbredden minskas ner avsevärt samtidigt som bufferten runt kartan kunde användas på ett intelligentare sätt. Den största nackdelen är beräkningarna som krävs för att översätta koordinater från världskoordinater till sökmatrisrelativa. Än så länge är algoritmen mer begränsad i beräkningstid än i minnesåtgång men om kravet på minnesförbrukning någon gång skärps kan stora vinster göras här.

7 Ordlista

A*	A*	Bäst-förstsökning med heuristik.
Bakåtppekare	Backpointer	Pekare till den föregående noden i en sökväg. Används för att återskapa kedjan av noder mellan start och mål.
D*	Dynamic A*	En A* sökning som kan anpassa lösningen efter förändringar i omgivningen.
Delmål	Waypoint	Ett mål på vägen som agenten ska passera för att följa vägen som sökalgoritmen givit.
Dominans	Dominance	En algoritm A1 dominerar en annan algoritm A2 om alla noder som expanderas av A1 också expanderas av A2. Strikt dominans är när dominansen är enkelriktad.
Fullständig	Complete	En algoritm är fullständig om den avbryter med en lösning när en sådan existerar.
IDA*	Iterative Deepening A*	Sökning som iterativt utökar antalet noder som sökningen får omfatta från startnoden räknat.
Informerad	Informed	En heuristik, h_1 , är mer informerad än en annan, h_2 , om $h_1(X) > h_2(X)$ för alla icke-mål-noder X.
LRTA*	Learning RealTime A*	Algoritm för realtidssökning som anpassar sig till omgivningen.
Mellanmål	Intermediate goals	Mål som sätts upp längs med vägen mot målet om slutmålet ligger på för stort avstånd för att kunna hanteras av vägsökningen
MMORPG	Massively Multiplayer Online RolePlaying Game	Kategori av datorspel där spelarna antar roller som personer i en virtuell värld och agerar med och mot varandra och andra datorstyrda agenter.
MTS	Moving Target Search	Algoritm för realtidssökning som hanterar rörliga mål.
NPC	Non-Player Character	Benämning på datorstyrd agent i Project Entropia.
Off-line sökning	Off-line search	Innebär att en fullständig sökning utförs utan interaktion med omvärlden, i motsats till on-line sökning.
Omplanering	Replanning	Att skapa en ny plan efter att en tidigare visat sig oanvändbar.
On-line sökning	On-line search	En on-line algoritm kan under exekveringen ta hänsyn till nya parametrar från omvärlden och därför påverka resultatet dynamiskt.

Optimalitet	Optimality	En algoritm är optimal jämfört med en grupp algoritmer om den dominerar alla andra algoritmer i gruppen.
Realtidssökning	Realtime search	Algoritmen beräknar endast vad nästa steg i planen ska vara.
Riktad sökning	Beam search	Ett maximalt antal aktiva söknoder riktar sökningen direktare mot målet.
Råkraft	Brute-force	Att använda beräkningskraft istället för intelligentare metoder för att lösa ett problem.
Söndra-och-härska	Divide-and-conquer	Problemlösning genom att dela upp det ursprungliga problemet i flera mindre och lösa dessa var för sig.
Skelettisering	Skeletonization	Metod för att dela upp sökrymden i ett nät av noder med fria vägar emellan.
Tvåvägssökning	Bidirectional search	Sökningen startar från både start och mål nod samtidigt. När lösningarna möts sammanfogas de två delarna.
Underskattande	Admissible	Underskattande. En algoritm är underskattande om den garanterat ger en optimal lösning när en sådan finns.
Uteslutning	Pruning	Ett mått på hur många noder som kan uteslutas under sökningen, utan att expandera dessa.

8 Källförteckning

- [1] Pearl, Judea. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley, 1984. ISBN 0-201-05594-5.
- [2] German, Oleg V. och Ofitserov, Dimitri V. *Problem Solving: Methods, Programming and Future Concepts*. Elsevier Science B.V., 1995. ISBN 0-444-82226-7.
- [3] Aho, Alfred, Hopcroft, John, Ullman, Jeffery. *Data structures and algorithms*. Addison-Wesley, 1983. ISBN 0-201-00023-7.
- [4] Reese, Björn och Stout, Bryan. "Finding a Pathfinder", I Proceedings of the AAAI 99 Spring Symposium on Artificial Intelligence and Computer Games. Sida 69-72.
- [5] Stentz, Anthony. "Optimal and Efficient Path Planning for Partially-Known Environments", i Proceedings IEEE International Conference on Robotics and Automation. 1994.
- [6] Jönsson, Markus. *An optimal pathfinder in real-world digital terrain maps*. Examensarbete på Institutionen för Numerisk Analys och Datalogi, KTH. 1997.
- [7] Ishida, Toru och Korf, Richard. "Moving-Target Search: A Real-Time Search for Changing Goals", IEEE Transactions on pattern analysis and machine intelligence. Vol 17, no 6, sida 609-619, 1995.
- [8] Stentz, Anthony. "Map-Based Strategies for Robot Navigation in Unknown Environments", i Proceedings AAAI 96 Spring Symposium on Planning with Incomplete Information for Robot Problems.
- [9] Melax, Stan. "New Approaches To Moving Target Search", AAAI Fall Symposium 1993 on Games: Planning and Learning.
- [10] Sasaki, Takahiro, Chimura, Fumihiko och Tokoro, Mario. "The Trailblazer Search with a Hierarchical Abstract Map", i Proceedings of 14-th International Joint Conference on Artificial Intelligence (IJCAI-95). Sida 259-265.
- [11] Pinter, Marko. "Toward more realistic pathfinding", Gamasutra. http://www.gamasutra.com/features/200103014/pinter_01.htm, acc 020618.
- [12] Stout, Bryan. "Smart Moves: Intelligent Pathfinding", Gamasutra. <http://www.gamasutra.com/features/19970801/pathfinding.htm>, acc 020620.
- [13] Yang, Qiang. *Intelligent Planning: a decomposition and abstraction based approach*. Springer-Verlag, 1997. ISBN 3-540-61901-1.
- [14] Migdalas, Athanasios och Göthe Lundgren, Maude. *Kombinatorisk Optimering: problem och algoritmer*. Linköpings universitet, Matematiska institutionen, 1996.
- [15] Bäckström, Christer. "Five Years of Tractable Planning", New Directions in AI Planning. IOS Press, 1996. ISBN: 90 5199 237 8.
- [16] Yahja, Alex, Stentz, Anthony, Singh, Sanjiv och Brumitt, Barry. "Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments", i Proceedings, IEEE Conference on Robotics and Automation. 1998.
- [17] Tozour, Paul. "Building a Near-Optimal Navigation Mesh", AI Game Programming Wisdom. Charles River Media, 2002. ISBN 1-58450-077-8.

- [18] Pottinger, Dave C. "Terrain Analysis in Realtime Strategy", i Proceedings Game Developers Conference 2000.
- [19] Szczerba, Robert J och Chen, Danny Z. "Using Framed-Subspaces to Solve the 2-D and 3-D Weighted Region Problem", Computer Science and Engineering Technical Report 96-18. University of Notre Dame, Department of Computer Science and Engineering. 1996.
- [20] Smith, Patrick. "GDC 2002: Polygon Soup for the Programmer's Soul: 3D Pathfinding", Gamasutra.
http://www.gamasutra.com/features/20020405/smith_pfv.htm, acc 020821.
- [21] Holte, R.C. Perez, M.B. Zimmer, R.M. och MacDonald, A.J. "Hierarchical A*: Searching Abstraction Hierarchies Efficiently", i AAAI/IAAI. Vol 1, sida 530-535, 1996.
- [22] Forbus, Kenneth D. Mahoney, James V. Dill, Kevin. "How qualitative spatial reasoning can improve strategy game AIs", i IEEE Intelligent Systems. Vol 17, no 4, sida 25-30.
- [23] Brigger, Patrick. *Shape oriented region representation*.
http://ltswww.epfl.ch/pub_files/brigger/thesis_html/node21.html acc 021002.
- [24] Higgins, Dan. "Pathfinding Design Architecture", AI Game Programming Wisdom. Charles River Media, 2002. ISBN 1-58450-077-8.
- [25] Cain, Timothy. "Practical Optimizations for A* Path Generation", AI Game Programming Wisdom. Charles River Media, 2002. ISBN 1-58450-077-8.
- [26] Higgins, Dan. "How to Achieve Lightning-Fast A*", AI Game Programming Wisdom. Charles River Media, 2002. ISBN 1-58450-077-8.
- [27] Weiss, Mark Allen. *Data Structures and Algorithm Analysis in C++*. The Benjamin/Cummings Publishing Company, 1994. ISBN 0-8053-5443-3.
- [28] Stentz, Anthony, "The Focussed D* Algorithm for Real-Time Replanning", i Proceedings of the International Joint Conference on Artificial Intelligence. 1995.
- [29] Koenig, Sven, Furcy David och Bauer, Colin. "Heuristic Search-Based Replanning", i Proceedings of the Sixth International Conference on Artificial Intelligence Planning & Scheduling (AIPS). Sida 310-317, 2002.
- [30] Edelkamp, Stefan. "Updating Shortest Paths", 13th European Conference on Artificial Intelligence. John Wiley & Sons, 1998.
- [31] Stentz, Anthony och Hebert, Martial. "A Complete Navigation System for Goal Acquisition in Unknown Environments", Autonomous Robots. Vol. 2, no 2, 1995.
- [32] Baert, Stefan. "Motion Planning Using Potential Fields", Gamedev.net.
<http://www.gamedev.net/reference/articles/article1125.asp> acc 021024.
- [33] Pinter, Marco. "Realistic Turning between Waypoints", AI Game Programming Wisdom. Charles River Media, 2002. ISBN 1-58450-077-8.
- [34] Möller, Tomas och Haines, Eric. *Real-Time Rendering*. A.K. Peters Ltd, 1999. ISBN 1-56881-101-2
- [35] Russell, Stuart och Norvig, Peter. *Artificial Intelligence: a modern approach*. Prentice Hall, 1995. ISBN 0-13-103805-2.
- [36] Ulrich, Thatcher. "Loose Octrees", Game Programming Gems. Charles River Media, 2000. ISBN 1-58450-049-2.

- [37]** Rabin, Steve. "*A* Aesthetic Optimizations*", Game Programming Gems. Charles River Media, 2000. ISBN 1-58450-049-2.
- [38]** Rabin, Steve. "*A* Speed Optimizations*", Game Programming Gems. Charles River Media, 2000. ISBN 1-58450-049-2.
- [39]** Snook, Greg. "*Simplified 3D Movement and Pathfinding Using Navigation Meshes*", Game Programming Gems. Charles River Media, 2000. ISBN 1-58450-049-2.
- [40]** Patel, Amit. "*Amits notes about pathfinding*", Amits Game programming information.
<http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#S10> acc 021105.

Bilaga 1: Arbetsgång

W. 225 Planering Startup	Systemet, kopplingen skript vs c- kod.	Design mål problem	Teori källor metoder	Semester
W. 230 Semester	Forts teori Återkoppling design	->	Praktiska försök med algoritmer utanför systemet	Abstraherings- försök. Koppling till algoritmer
W. 235 ->	Teori + design, återkoppling.	Utvärdering av implementationer. Val av lösning.	Optimering och verifiering av vald lösning.	-> Buffer
W. 240 Rapport	->	->	Avrundning. Förberedelse av presentationen.	Buffer

Tabell 3 Arbetschema v 1.0 vecka 1

Som kuriositeter presenteras här det första utkastet till en tidsfördelning för examensarbetet. Det var väntat att det skulle behöva revideras men en återblick nu när arbetet är inne i slutfasen visar att det inte var mycket som gick enligt planen. Den andliga meningen i schemat, att mycket av det teoretiska arbetet skulle läggas i början istället för att hastas ihop de sista veckorna, har ändå kunnat följas.

Kopplingen mot det befintliga systemet framstod som enkel att implementera under designen men tiden som krävts för att sätta sig in i skriptspråket och de problem som uppstått där har varit dominerande. En stor del av hanteringen av NPC:ernas beteende har behövt skrivas om för att passa ihop med de problem som kan uppstå under en vägsökning.

En annan del som verkade minimal inledningsvis var hanteringen av begränsningsvolymerna när de väl hämtats ut ur servern. Flera gånger har den behövt skrivas om när nya specialfall har dykt upp, ofta kopplade till problematiken med transformationen från tre till två dimensioner.

Bortsett från det nyss nämnda undantaget har implementeringen i C++ gått oväntat friktionsfritt. Så länge funktionaliteten har funnits isolerad till vägsökaren har det varit lätt att överblicka. Svårigheterna har istället uppstått i kontaktpunkterna med den virtuella världen.

Bilaga 2: Pseudokod

- 1 **Metod TickPathfinding** (returnerar en returkod)
- 2 OM Vägsökning är globalt avslagen SÅ returnera LYCKAD_SÖKNING
- 3 OM målets position har förändrats ELLER agenten är för nära en kartkant SÅ avbryt en eventuell tidigare sökning och initialisera sökdata
- 4 OM vägsökarens status är LYCKAD_SÖKNING ELLER MISSLYCKAD_SÖKNING OCH sökdata finns sen tidigare SÅ returnera INGEN_NY_VÄG
- 5 OM det inte finns tidigare sökdata SÅ
- 6 OM målet är för långt bort SÅ välj ett mål på maximalt avstånd i rätt riktning som inte ligger i en begränsningsvolym
- 7 Sätt kartstorlek beroende på agentens och målets position
- 8 OM agenten sökt från samma koordinater mer än tre gånger ELLER kartstorleken överstiger det godkända SÅ returnera MISSLYCKAD_KARTGENERERING
- 9 OM tidigare kartdata kan återanvändas SÅ rensa endast sökdata ANNARS rensa både sök- och kartdata
- 10 OM tidigare kartdata inte kunde återanvändas SÅ läs in begränsningsvolym. För varje volym:
- 11 Lägg till navigeringsbuffert
- 12 Roter runt z-axeln
- 13 Klipp mot kartgränserna
- 14 OM volymen hamnar inom kartan OCH är på en intressant höjd SÅ plotta in den i kartan
- 15 SÅ LÄNGE tidsgränsen inte överskrids OCH det återstår noder att undersöka:
- 16 Hämta ut den mest lovande noden
- 17 OM noden är målet SÅ generera delmål och returnera LYCKAD_SÖKNING
- 18 OM noden ligger på en kartkant SÅ returnera MISSLYCKAD_SÖKNING
- 19 Generera nodens fyra grannar och lägg in på nodlistan
- 20 OM tidsgränsen överskreds SÅ returnera SLUT_PÅ_TID ANNARS returnera MISSLYCKAD_SÖKNING

- 21 **Metod GetNextWaypoint** (returnerar absoluta kartkoordinater och en målstatus)
- 22 OM Vägsökning är globalt avslagen ELLER inget delmål är tillgängligt SÅ returnera målets position tillsammans med MISSLYCKAT_MÅL
- 23 OM det finns mer än ett delmål kvar SÅ returnera nästa mål tillsammans med FLER_MÅL ANNARS returnera sista målet tillsammans med SISTA_MÅL

Returkodernas betydelse

LYCKAD_SÖKNING = en väg till målet har hittats, delmål finns att hämta GetNextWayPoint()

MISSLYCKAD_SÖKNING = sökningen lyckades ej

INGEN_NY_VÄG = en tidigare beräknad väg är ska användas, delmål finns

MISSLYCKAD_KARTGENERERING = sökningen har misslyckats tidigare, agenten är inlåst eller så är målet oåtkomligt

SLUT_PÅ_TID = tidsgränsen överskreds

FLER_MÅL = fler delmål finns att hämta efter detta

SISTA_MÅL = det returnerade målet var det sista