Progressive Deepening for Gametrees: An application for RoboRescue

By:
Aron Abbo aabbo@science.uva.nl
Stefan Peelen speelen@science.uva.nl

Supervisor:
Arnoud Visser

Special Thanks:
Maurits Fassaer
Stef Post

http://www.science.uva.nl/~arnoud/research/roboresc/

Abstract

In dit verslag wordt rapport gemaakt van ons onderzoek naar de applicatie van de zoektechniek Progressive Deepening op het domein van RoboRescue.
Er zal worden uitgelegd wat RoboRescue is en hoe er in de huidige implementatie gebruik wordt gemaakt van gametrees om tot oplossingen te komen. Verder zal worden verteld hoe wij hebben geprobeerd om het proces van het construeren van een zoekboom computationeel minder zwaar te maken door toepassing van een zoektechniek.
Er wordt ingegaan op hoe een gedeelte van een oude zoekboom opnieuw wordt gebruikt om als basis voor een volgende boom te dienen; er zal worden ingegaan op de implementatie van zo'n methode en de complicaties daarvan. Tenslotte zal worden uitgelegd dat het ons niet is gelukt binnen de tijd een werkende implementatie te realiseren, maar er wordt wel een conclusie getrokken aan de hand van de informatie die we wel ter beschikking hadden.

1. Inleiding

Roborescue is een onderdeel van de Robocup, een jaarlijks terugkerend evenement waarin wedstrijden worden gehouden voor autonome systemen.

In Roborescue wordt een stad nagebootst, waarin vervolgens een aardbeving wordt gesimuleerd. De simulatie is behoorlijk uitgebreid, en houdt rekening met dingen als geblokkeerde straten, brandende gebouwen en bedolven mensen.
Een team van autonome agents krijgt vervolgens de opdracht om zoveel mogelijk mensen te redden, en om te zorgen dat zoveel mogelijk brandende gebouwen geblust worden. Er is een bepaalde tijdslimiet gesteld, waarna wordt geteld hoeveel gebouwen en mensen de ramp hebben overleefd. Het team dat de beste score heeft behaald, wint de wedstrijd.

Een team bestaat uit drie typen agents: Brandweer, ambulance en politie. De brandweer is in staat om branden te blussen, de ambulances kunnen mensen redden uit ingestorte gebouwen, en de politie kan geblokkeerde wegen weer berijdbaar maken.

De simulatie verloopt in stappen, die ‘cycles’ worden genoemd. Iedere cycle mag een agent een actie ondernemen, een gelimiteerd aantal berichten verzenden en ontvangen, en iedere cycle zal de uitbreiding van brand worden gesimuleerd.

1.1 Gametrees in RoboRescue

Voor ons project was het de bedoeling om door te gaan op een agentenstructuur zoals die bedacht is door een tweetal andere studenten: Stef Post en Maurits Fassaer[1].
Zij hebben een achitectuur bedacht die gebaseerd is op gametrees, en zijn een team gaan programmeren. Ze zijn beiden een andere richting opgegaan met hun implementatie: Fassaer ging door met het oorspronkelijke plan van gametrees, terwijl Post momenteel een simpeler finite-state machine gebruikt om zijn agenten de gewenste richting in te sturen. Wij zijn echter gaan werken op de code van Maurits, die de eerdergenoemde gametrees gebruikt.

Een gametree is, zoals de naam al doet vermoeden, een boomstructuur die van de situatie een spel maakt. Hierbij is iedere agent een speler, evenals ‘de ramp’; ze mogen om de beurt een 'zet' doen. De gametree begint met een root node, die de huidige situatie voorstelt. Bij elke zet van een speler wordt nu een nieuwe laag aangemaakt, en splitst de boom in een aantal nieuwe takken. Elke nieuwe node bevat een representatie van de spelwereld, met daarin de veranderingen die de vorige zet teweeg heeft gebracht.
Eerst mogen alle agenten één voor één een zet doen, waarbij de boom steeds groter wordt. Vervolgens is de ramp aan 'zet'. Hierbij wordt een module gebruikt die probeert te voorspellen hoe de brand zich zal gaan verspreiden in de volgende beurt. De boom splitst zich hier dus niet, maar wordt wel een laag dieper.
Vervolgens zijn de agenten weer aan zet, en begint een nieuwe ronde in het spel.

Als een bepaalde stopconditie is bereikt, zal het algoritme stoppen met het aanmaken van nieuwe lagen. Deze stopconditie kan een bepaalde boomdiepte zijn, of een tijdslimiet waar het algoritme niet overheen mag.

Nu wordt van alle leaf nodes een score berekend, die aangeeft hoe waardevol deze situatie is. In de huidige implementatie wordt deze score berekend aan de hand van het aantal overlevende mensen, het aantal onbeschadigde gebouwen, en de schade die de agenten hebben opgelopen tijdens hun reddingsacties. Dit is zo gekozen omdat de score die de kwaliteit van het team bepaalt, ook gebaseerd is op deze factoren.
Met deze scores wordt vervolgens een beste leaf node bepaald, en het pad dat tot deze situatie heeft geleid wordt bekeken. Hierna zal de agent die de gametree heeft berekend zijn eigen aandeel van dit pad gaan doen.

Het idee is dat iedere agent op deze manier dezelfde gametree construeert, en dus ook op dezelfde conclusie uitkomt. En als dit zo is, en als de agenten dus ook erop kunnen vertrouwen dat de andere agents hetzelfde optimale pad hebben bepaald, zullen ze dus samenwerken om een optimale situatie te bereiken.

1.1.1 Efficiëntie in Gametrees

Er zijn een aantal zaken waarop getracht is om dit gametree-mechanisme sneller te maken.

Zo wordt er bijvoorbeeld als bewegingsmogelijkheden van de agenten niet uitgegaan van simpele acties, maar van behaviours. Het verschil daartussen is dat een actie maar één beurt kost, terwijl een behaviour een aantal acties is die achter elkaar worden uitgevoerd, om tot een bepaald doel te komen.
Hoewel een behaviour meerdere beurten kost, wordt echter alleen de eerste actie van de reeks gebruikt voor het voorspellen van een nieuwe situatie in een boom; de meeste gekozen behaviours worden dus niet afgemaakt.

Verder zijn er verschillende vormen van pruning toegepast om de grootte van de boom te beperken. Zo worden nutteloze behaviours eruit gefilterd, zoals het blussen van een huis dat niet in de brand staat. Een andere vorm van pruning die is toegepast is Alpha-Beta pruning.

1.1.2 Voordelen en tekortkomingen van Gametrees

Gametrees hebben een paar duidelijke voordelen en nadelen met betrekking tot ons probleem.
Zoals al is gezegd, is een groot voordeel dat er coöperatie ontstaat, zonder dat er strategieën hoeven worden te doorgecommuniceerd. Dit is niet alleen nuttig omdat er een communicatielimiet is, maar ook omdat onderlinge communicatie een vertraging met

zich meebrengt. De agents kunnen nu acties ondernemen, zonder daarbij van hun bronnen afhankelijk te zijn.
Verder is het te garanderen dat de agents een optimale strategie zullen vinden, als de boom volledig zou kunnen worden doorberekend tot aan de laatse cycle, de beschikbare informatie volledig zou zijn en de voorspelling van de branden perfect zou zijn.

Helaas is dit niet haalbaar. Er is een gelimiteerde hoeveelheid informatie voorhanden, en een zeer gelimiteerde rekentijd. Zelfs met de verschillende vormen van pruning kan de boom maar tot een zeer gelimiteerde diepte worden berekend, zelfs op een kleine kaart en met weinig agenten.

1.2 Doel van het project

En dit brengt ons tot het doel van onze opdracht. Het is onze taak om uit te zoeken tot hoever we deze gametree structuur kunnen versnellen, met behulp van een zoektechniek die Progressive Deepening heet. Verder zullen we kijken of deze versnelling significant is, en of het niet tot verslechtering leidt van de kwaliteit van de beslissingen van de agenten.
Volgens de opdracht bestaat Progressive Deepening uit twee delen: Time Control en hergebruik van oude bomen. In het volgende hoofdstuk zal een beschrijving worden gegeven van deze twee termen.

2. Methode

In de inleiding werd een beschrijving gegeven van gametrees, alsmede vermeld dat het onze opdracht was te onderzoeken in hoeverre de toepassing van Progressive Deepening zou leiden tot betere prestaties in het berekenen van een gametree voor een agent.
In de implementatie zoals wij die aangeleverd kregen wordt de gehele gametree verwijderd nadat deze is gebruikt om de beste move te genereren; een van de aspecten van progressive deepening is het hergebruiken van oude bomen om op die manier kostbare rekentijd te besparen.

2.1 Literatuurstudie

Om te beginnen zouden we voor onzelf duidelijk moeten maken wat Progressive Deepening precies is; en dit is precies het punt waar we tegen het eerste probleem van het project aanliepen: er was nauwlijks literatuur te vinden over deze specifieke zoektechniek. Het beste dat wij op internet konden vinden waren collegeslides en PDF artikelen waar het zeer kort genoemd wordt.
Het verassende aan deze artikelen was dat er vrijwel nooit gesproken werd over het hergebruiken van oude bomen, maar vrijwel alleen maar over het feit dat er een vorm van “time-control” geïmplementeerd kan worden om er zeker van te zijn dat een zoekproces tot een beslissing komt.
Omdat we zelf niet echt iets konden vinden hebben we hulp gevraagd aan de docent zoektechnieken van AI, Maarten van Someren. Hij vertelde dat hij eigenlijk ook niet precies wist wat het inhield, maar hij wist ons wel een boek te geven waarin het genoemd werd [3]. Dit zeer oude boek bleek echter ook van weinig hulp, omdat het geschreven was vanuit een voornamelijk psychologisch oogpunt, en alleen maar betrekking had op schaakposities en hoe schaakgrootmeesters tot beslissingen komen.

Uiteindelijk zijn we ervan uitgegaan dat er twee verschillende vormen van Progressive Deepening bestaan, zoals wordt beschreven in [2]. Bij de eerste vorm ligt de focus op het hergebruiken van oude bomen, en bij de tweede op het instellen van een tijdslimiet om zo gegarandeerd tot een antwoord te komen.

2.2 De Methoden

We hebben er uiteindelijk voor gekozen om het grootste deel van onze tijd te besteden aan het naar onze mening meest interessante aspect, namelijk het hergebruiken van oude zoekbomen om de rekentijd in volgende cycles te verkorten.
Toch zullen we in het volgende subhoofdstuk nog even aandacht schenken aan de notie van “time-control”.

2.2.1 Time control

Een probleem met het maken van een zoekboom is dat het een computationeel vrij zwaar proces kan zijn. Zeker als de diepte en breedte van de boom relatief groot gaan worden, is het lang niet altijd zeker dat de hele boom binnen een redelijke tijd uitgerekend kan worden.
In het geval van RoboRescue is dit een zeer zwaar wegend nadeel van de gametree methode, omdat er geredeneerd moet worden in een niet al te lang tijdsbestek. Als een agent er te lang over doet om de boom aan te maken, is de informatie waarmee hij aan het redeneren is al verouderd, of nog erger, hij is nog steeds aan het nadenken terwijl er eigenlijk gehandeld moet worden.
In zo'n geval zou een vorm van "time-control" gewenst zijn; door de agent te dwingen na een bepaalde tijd tot een conclusie te komen, wordt in ieder geval gegarandeerd dat het op dat moment best beschikbare behaviour uitgevoerd gaat worden. Het kan gebeuren dat dit niet het optimale is, omdat de boom nog niet volledig doorgerekend is, maar het alternatief is dat de agent helemaal geen actie onderneemt.
Een nadeel inherent aan deze methode is dat niet zeker is dat een agent zijn optimale behaviour uitvoert, maar slechts hetgene dat op dat moment het beste lijkt.
Een nadeel van een andere categorie is het feit dat het nu lastiger is geworden voor de agents om samen te werken. Als van iedere agent met zekerheid kan worden gezegd dat deze zijn de boom doorrekent tot een bepaalde diepte, kunnen alle agents ervan uitgaan dat elke andere agent het behaviour dat hij daarvoor voorspeld heeft uit zal voeren; dit omdat elke agent uitgaat van hetzelfde wereldmodel. Dit leidt tot cooperatie tussen de agents onderling.
Deze cooperatie zal verminderen met een tijdslimiet, omdat agents er dan niet meer vanuit kunnen gaan dat andere agents even "diep" zijn gekomen met het berekenen van hun boom. Dit heeft tot gevolg dat de ene agent een andere optimale behaviour uit gaat voeren dan datgene dat de andere agents voor hem voorspeld hadden, en hierdoor vermindert de kans dat samenwerking succesvol wordt.

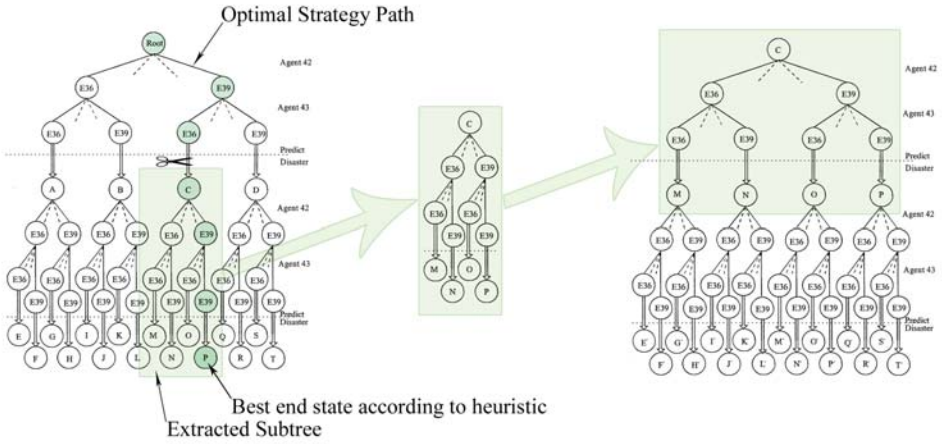
2.2.2 Hergebruik van oude bomen

Dit is de methode waar wij ons meer in verdiept hebben, en die wij tot uitvoering wilden brengen in de gametree implementatie van RoboRescue agents.
In de huidige implementatie wordt elke boom die door een agent berekend wordt aan het begin van de volgende cycle weer weggooid om vervolgens weer een nieuwe boom te berekenen.

Het doel achter ons onderzoek was om te kijken of er geen manier is waarop de bomen die nu nog worden weggooid behouden kunnen worden, om als basis te dienen voor een volgende cycle.
Om te beginnen zijn we in de structuur van de code gaan kijken, en we kwamen erachter dat het vinden van een optimale strategie als volgt gebeurt:

- Een agent genereert zijn volledige boom
- Een heuristiek kijkt welke node de optimale eindsituatie bevat
- Er wordt teruggezocht vanaf die eindsituatie tot aan de root
- Het pad dat zo gevonden wordt is de optimale strategie
- De boom wordt verwijderd

Het is nu dus zaak om de eerste vier stappen nog steeds uit te laten voeren, maar als volgt een verandering aan te brengen in de vijfde:
Nadat de optimale strategie bekend is geworden, moet nogmaals worden teruggezocht in de zoekboom, maar niet helemaal tot de root. In plaats hiervan wordt teruggezocht totdat de eerste disaster stap na de root van bovenaf wordt gevonden.
De node na de disaster stap moet nu tot nieuwe root worden gemaakt, en de subtree die onder deze nieuwe root hangt moet als basis dienen voor een volgende te genereren zoekboom. Op deze manier zal de agent het eerste gedeelte van zijn optimale behaviour niet meer opnieuw uit hoeven te rekenen, wat scheelt in rekentijd. Zie ook de figuur hieronder:



figuur: Subtree wordt als basis voor nieuwe boom gebruikt. Figuur afkomstig van onze poster.

3. Uitwerking

Nu bekend is hoe gametrees werken in een RoboRescue omgeving en hoe wij willen proberen deze methode efficienter te maken, is het tijd te kijken hoe dit op implementatie niveau in zijn werk zou moeten gaan.
We beginnen met te zeggen dat we er niet aan toegekomen zijn om een poging te wagen het 'time control' gedeelte te implementeren; we moesten vanwege de korte tijd die nog beschikbaar was een keuze maken, en het omzetten van zoekbomen leek ons interessanter dan een het instellen van een tijdslimiet.

3.1 Implementatie

Het was voor ons lastig om in een korte tijd een toch redelijk ingewikkelde procedure te implementeren. Hiervoor zijn een aantal redenen te noemen, waarvan de voornaamste is dat wij allebei niet erg bekend waren met C++, de taal waarin het programma is geschreven. Een ander probleem was dat de code waarmee wij moesten werken niet bijzonder goed gedocumenteerd was, het heeft ons veel tijd gekost om uit te vinden wat precies waar staat in de code, en welke functies wat doen.
Hierbij zijn we af en toe geholpen door Maurits Fassaer, die de code heeft geschreven; helaas had hij niet veel tijd voor ons, en dus hebben we vaak in het duister zitten tasten.

3.1.1 Het initiële idee

Het was duidelijk wat er moest gebeuren, onze code zou vanaf de best gevonden eindnode terug door de boom moeten gaan om bij de eerste disasterstap uit te komen, de node na deze stap zou als nieuwe rootnode moeten gaan fungeren.

In eerste instantie hebben we een functie geschreven die vanaf de beste eindnode terug door de boom liep, deze functie keek bij elke node waar hij langskwam of deze van het type actor( ) (een agent) was. Als dit niet zo is, is het een disaster, en wisten we dat de volgende node de nieuwe rootnode moest worden.
Als de goede node gevonden was moest deze als het ware worden 'losgeknipt' van de rest van de boom, waarna de rest van de boom weggegooid zou kunnen worden.
Dit 'losknippen' hebben wij in eerste instantie als volgt aangepakt: Elke node heeft een 'strategy'; dit is de behaviour waarmee er van de vorige node naar de huidige wordt gegaan (verwarrend hier was dat in de literatuur 'strategy' iets anders was dan in de implementatie, waardoor wij een tijdje in de verkeerde hoek hebben gezocht voor deze procedure). Deze strategy is te zien als een tak van de zoekboom die twee knopen met elkaar vebindt.
De strategy's staan in verbinding met de nodes doormiddel van een verwijsstructuur; elke strategy heeft een prev( ) en een next( ) die verwijzen naar de nodes waartussen ze verbonden zitten. Verder heeft iedere node ook nog verwijzingen terug naar de strategies.

Één verwijzing naar de strategy boven de node, en ook nog een lijst van verwijzingen naar de strategies.

Wat ons in eerste instantie was verteld was dat het enige dat een rootnode onderscheidt van een gewone node het feit is dat een rootnode geen ‘strategy’ heeft. Uitgaande van deze eigenschap hebben wij besloten dat nadat de nieuwe rootnode was gevonden, zowel de verwijzing van de nieuwe root naar zijn strategy als de next( ) van deze strategy op NULL gezet moesten worden, waardoor deze in feite geen strategy meer boven zich heeft.
Nu de gewenste subboom los stond van de rest van de boom (de strategy die de subboom met de rest van de boom verbond is ‘doorgeknipt’), konden we veilig de oude boom verwijderen en met de subboom die overgebleven is verder redeneren.

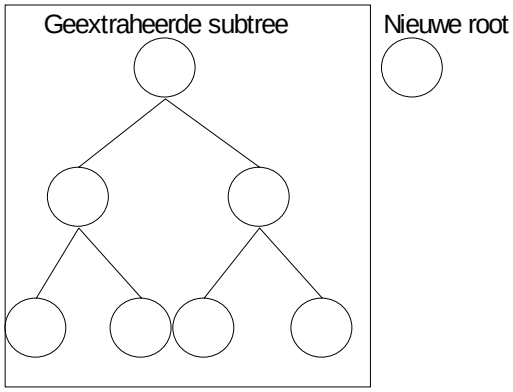
3.1.2 Complicaties en oplossingen

Zodra we alles hadden geïmplementeerd bleken er fouten op te treden bij het draaien van het programma. Na raadpleging bij de schrijver ervan bleek dat een rootnode er fundamenteel anders uitzag dan dat wij hadden verondersteld.
Het verschil tussen een rootnode en een gewone node is niet alleen het wel of niet aanwezig zijn van een strategy, er is ook een verschil in de inhoud van de node. Een node is in deze implementatie een ‘situatie model’, een beschrijving van de omgeving. Een strategy is nu het behaviour dat een verandering in een situatie model veroorzaakt, wat op zich weer resulteert in een nieuw situatiemodel.
Het eigenlijke verschil tussen een rootnode en een gewone node is nu dat in de rootnode alle objecten in het situatiemodel opgeslagen zijn, dus alle gebouwen, wegen, actoren etc., terwijl in niet–rootnodes alleen de *veranderingen* in deze objecten ten opzichte van de vorige node worden gerepresenteerd.
Dit verklaart de problemen in onze initïele implementatie. We hadden daar gewoonweg de gevonden node tot rootnode gemaakt, maar door bovengenoemde eigenschappen hield dit in dat de nieuwe rootnode niet uit alle objecten bestond, maar alleen maar uit veranderingen daarin. De rest van de nodes hadden nu dus allemaal verwijzingen naar objecten die niet meer bestonden omdat de oude rootnode tegelijk met de oude boom weg was gegooid.

Om om dit probleem heen te werken hebben we aantal verschillende oplossingen bedacht, waarvan we er maar van een hebben geprobeerd om dit te implementeren.
Het was duidelijk wat er moest gebeuren; de nieuwe rootnode moest worden aangepast zodat deze alle relevante objecten bevat, om dit doel te bereiken dachten wij eraan om:

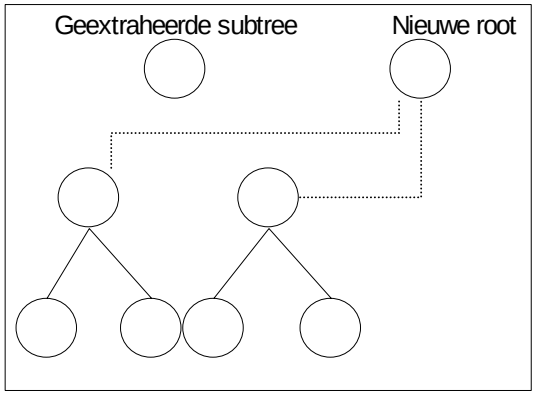
- De oude root over de nieuwe heen te kopieëren. Deze bevat nu wel de objecten, maar die zijn niet aangepast door de veranderingen die in de loop van de boom door de behaviours van de agents zijn toegebracht. Dus nadat de node gekopieërd is, moet worden teruggezocht in de boom en moet de nieuwe root worden geupdate met alle veranderingen die in de voorgaande nodes hebben plaatsgevonden

- De nieuwe rootnode te voorzien van de benodigde objecten doormiddel van een functie die ‘ExtractCurrentSituation’ heet. Hiermee wordt het situatiemodel (node) waarop je het toepast ‘gevuld’ met alle objecten die van belang zijn. De node wordt in feite een rootnode, premisse is wel dat dit alleen op een lege node kan gebeuren. Wat dus moet gebeuren is het volgende: Er moet een lege node worden aangemaakt, waar vervolgens ‘ExtractCurrentSituation’ op wordt toegepast. Om van deze node de nieuwe root te maken, zal hij met de rest van de boom moeten worden verbonden, terwijl de eerder gevonden node zal moeten worden losgemaakt. Dit wordt verduidelijkt in de volgende figuren:



figuur: Er is een subtree gevonden die de basis is voor de nieuwe boom

→



figuur: Deze subtree krijgt een rootnode die dmv ExtractCurrentSituation voorzien is van objecten

We hebben geprobeerd de laatst genoemde manier toe te passen, maar na het implementeren hiervan bleek dat deze oplossing op zichzelf ook weer problemen met zich meebracht, waarvoor we nieuwe oplossingen moesten verzinnen. We hebben het op dit punt opgegeven omdat we simpelweg geen tijd meer hadden om dit nieuwe probleem goed te analyseren en oplossingen hiervoor te verzinnen; en omdat we het idee hadden dat als we dit probleem op zouden lossen, we simpelweg tegen het volgende aan zouden lopen.

4. Resultaten

Om te bepalen wat het nut van het behouden van subtrees is, hebben we een aantal metingen gedaan om te zien wat nu eigenlijk de winst is bij de toepassing van Progressive Deepening.
Zoals in hoofdstuk 3 is verteld hebben we het Progressive deepening algoritme niet werkend gekregen: Maar het is moglijk om alsnog te berekenen wat de winst zou zijn. Progressive Deepening kan bij een boom van drie lagen diep, bijvoorbeeld, een boom extraheren van twee lagen diep. Door de simulatie te draaien met een maximale boomdiepte van twee, is de tijd te bepalen die daarvoor nodig is. Door deze twee waarden van elkaar af te halen, hebben we een schatting kunnen maken van de hoeveelheid tijd die een algoritme nodig heeft dat wel gebruik maakt van Progressive Deepening.

Zie bijlage 1 voor de resultaten van deze tests.

We hebben de code een aantal keer laten draaien waarbij de gametree op verschillende dieptes werd berekend. Iedere node werd gepruned tot de drie beste nodes. De tests zijn allemaal gedaan op een kleine oefenmap, met 2 tot 4 brandweeragenten en 2 tot 4 initiële brandhaarden.

4.1 Opbrengst van Progressive Deepening

Zoals is in de tests te zien is het tijdsverschil zeer minimaal. Het verschil wordt zelfs nog kleiner naarmate de boomdiepte groter wordt, iets waar niet direct een oorzaak voor te vinden is. Immers, het gedeelte van de boom dat iedere cycle 'gered' kan worden, hoort relatief gezien juist groter te worden. Met een branching factor *b* en diepte *d* zouden normaal gesproken b^{d+1}-*b* nodes berekend moeten worden. Met Progressive Deepening kunnen telkens b^{d}-*b* nodes opnieuw worden gebruikt.

Uit onderstaande tabel met een branching factor van 3 blijkt dat het nut van Progressive deepening dus juist zou moeten toenemen met de diepte. De getallen geven aan hoeveel nodes er berekend zouden moeten worden, of zouden worden bespaard.

	Zonder PD	**PD winst**	**In %**
2	24	6	25,0
3	78	24	30,7
4	240	78	32,5

Figuur: resultaten van progressive deepening ten opzichte van normale implementatie

Het is dus zo dat onze testresultaten wijzen op een steeds minder wordende prestatie van progressive deepening naarmate de boom groter wordt, terwijl de theorie zegt dat het omgekeerde waar zou moeten zijn. Het is ons niet geheel duidelijk aan welke kant de fout zit.

Een andere, belangrijkere trend is te zien tussen de twee grafieken met twee agenten. Het geval met meer branden lijkt het slechter te doen: Dit is te wijten aan het feit dat er meer verschillende behaviours zijn die bekeken moeten worden. Hierdoor wordt de boom breder, waardoor de subtree die opnieuw gebruikt kan worden relatief kleiner wordt.
Een andere factor die de boom breder maakt is de toevoeging van een aantal teamgenoten. Bij 4 brandweer agenten is de winst al gelijk zeer klein, terwijl dit niet zo'n zeldzaam geval zou moeten zijn. Deze factor is veel sterker dan het aantal branden, aangezien deze verbreding niet zo makkelijk is te prunen.

4.2 Kwaliteit van Redeneren

Een ander punt dat zeer doorslaggevend is voor nut van Progressive Deepening is de mate waarin het de kwaliteit van het beslissingsalgoritme wordt aangetast door deze manier van optimalisatie. Helaas hebben wij dit niet echt kunnen testen, maar wij kunnen wel een voorspelling doen aan de hand van onze kennis van het systeem.
Zoals gezegd gaat Progressive Deepening door met een wereld zoals die voorspeld is in de vorige cycle. Deze wereld is in een zekere mate verouderd en incorrect, dat komt door de volgende zaken:

- De kennis over de wereld is incorrect, waardoor het moeilijk is om voorspellingen te doen over hoe de wereld erut ziet op de volgende cycle.
- De voorspellingsimulatie is opzettelijk zeer grof gehouden, zodat deze snel kan rekenen. Zo gaat de brandsimulator uit van een worst-case scenario: hij gaat er van uit dat iedere beurt elk gebouw in de brand vliegt dat een brandende buur heeft.
- Nieuwe observaties en berichten worden niet meegenomen, en het model is dus verouderd.
- Het is niet zeker dat de andere agenten wel hetgene gaan doen dat is voorspeld: Zij redeneren op een andere set van informatie, en kunen dus op andere conclusies zijn gekomen.

Al met al zit er dus nogal wat “ruis” in de aannames van de agent met betrekking tot de staat van de wereld. Dit is een nadeel voor onze methode omdat het voor kan komen dat een agent tot een bepaalde strategie is gekomen op basis van foute assumpties. Wat er dan gebeurd als de oude boom wordt hergebruikt, is dat de agent door blijft redeneren op basis van een wereldbeeld dat in eerste instantie al fout was; dit zou niet gebeuren als de boom wel na de eerste cycle was weggooid.

5. Conclusie en Discussie

5.1 time control

Wij hebben deze zoekmethode niet geïmplementeerd, maar hebben wel gekeken naar de implementeerbaarheid en waarschijnlijke complicaties van dit algoritme.
De implementatie lijkt op het eerste gezicht niet erg ingewikkeld: Het is vrij goed mogelijk om te zorgen dat het gametree algoritme stopt met het genereren van nieuwe nodes na een bepaalde tijdseenheid. Maar er zullen nog wel een aantal problemen de kop op steken:
Zo zal er alleen gekeken moeten worden naar lagen in de boom die zijn afgemaakt. Half complete lagen hebben nog niet gegarandeerd de optimale oplossing bekeken, en als een andere agent eerst andere nodes heeft bekeken, zal hij tot een andere conclusie komen. Verder is het ook niet te garanderen dat iedere agent evenveel nodes kan berekenen: En dus zal niet iedere agent tot een zelfde diepte komen. En dit zorgt er ook weer voor dat verschillende agents andere conclusies zullen trekken.
Het zou mogelijk kunnen zijn om zekerheid te verkrijgen door berichten rond te sturen, maar dit zorgt voor vertraging: en samenwerken zonder communicatie is nu juist het sterke punt van Game Trees.

5.2 Hergebruik van bomen

Uit hoofdstuk 5 kwamen de volgende punten naar voren:

- De relatieve tijdswinst is niet bijster groot. In de (zeer beperkte) testwereld was de winst maximaal zo'n 25 procent, maar in een 'echte' situate zal deze eerder rond de 5 procent liggen.
- De relatieve tijdswinst wordt kleiner bij bredere en diepere bomen. En dit zijn nu juist de momenten waar snelheid extra nodig zijn. Vooral de teamgrootte is van invloed op de winst.
- Het wereldbeeld is matig, en wordt snel slechter naarmate er langer wordt doorgegaan met deze foute informatie. Hierdoor zal de kwaliteit van de beslissingen snel achteruit gaan. Als de ene agent iets anders voorspelt als een ander, en ze gaan allebei op eigen houtje een actie ondernemen, zullen de wereldbeelden daarna zo veel van elkaar verschillen dat coöperatie daarna alleen maar moeilijker wordt.

De vraag is nu hoe nuttig Progressive Deepening nu eigenlijk is: is de magere snelheidswinst verslechtering van de behaviour waard?
Het is in ieder geval duidelijk dat in dit geval Progressive Deepening niet kan werken zonder dat om de zoveel tijd een nieuwe root node aan te maken met de nieuwste informatie die is verkregen met observatie en communicatie. Het gelimiteerde

voorspellend vermogen, de onvolledige informatie en de grote hoeveelheid spelers zorgen er voor dat Progressive Deepening alleen niet betrouwbaar genoeg is.

Een idee is dus om Progressive Deepening dus alleen om de cycle te gebruiken, en om daarna weer een op de normale manier een boom te maken. Er komt niet zoveel informatie binnen door observaties, omdat de agent niet ineens naar een volledig onbekende plek kan rijden in één beurt. En de informatie over de wereld die de agent krijgt van de andere agents, komt maar eens in de zoveel tijd.

Het huidige communicatie model is namelijk zo geïmplementeerd, dat alle informatie eerst zogenoemde centers lopen, aangezien agenten van verschillende types niet met elkaar mogen praten [4]. Informatie gaat van agent naar center, dan van center naar center, en dan weer terug naar de agents. Hierdoor duurt het even voordat er weer nieuwe informatie binnenkomt over de wereld, met grote hoeveelheden tegelijk. Deze vertraging is momenteel zo'n 4 cycles.

Het zou dus ook een optie kunnen zijn om alleen een compleet nieuwe gametree te creeëren na een communicatieronde, om daarna zo'n 3 cycles met Progressive Deepening te werken. Maar deze tijd is lang genoeg om behoorlijk van de echte wereld vervreemd te raken, dus het zou raadzaam zijn om hier een oog op te houden.
Zo zou er een module kunnen worden gemaakt die iedere cycle controleert tot hoever de voorspelde wereld overeenkomt met de huidige. Als dit verschil te groot is kan worden besloten om een volledig nieuwe boom te construeren; of er zou een andere voorspelde node kunnen worden gekozen, die beter lijkt op de huidige wereld. Dan kan deze node met de bijbehorende subtree als basis kunnen dienen voor de voorspellingen van de volgende cycle.

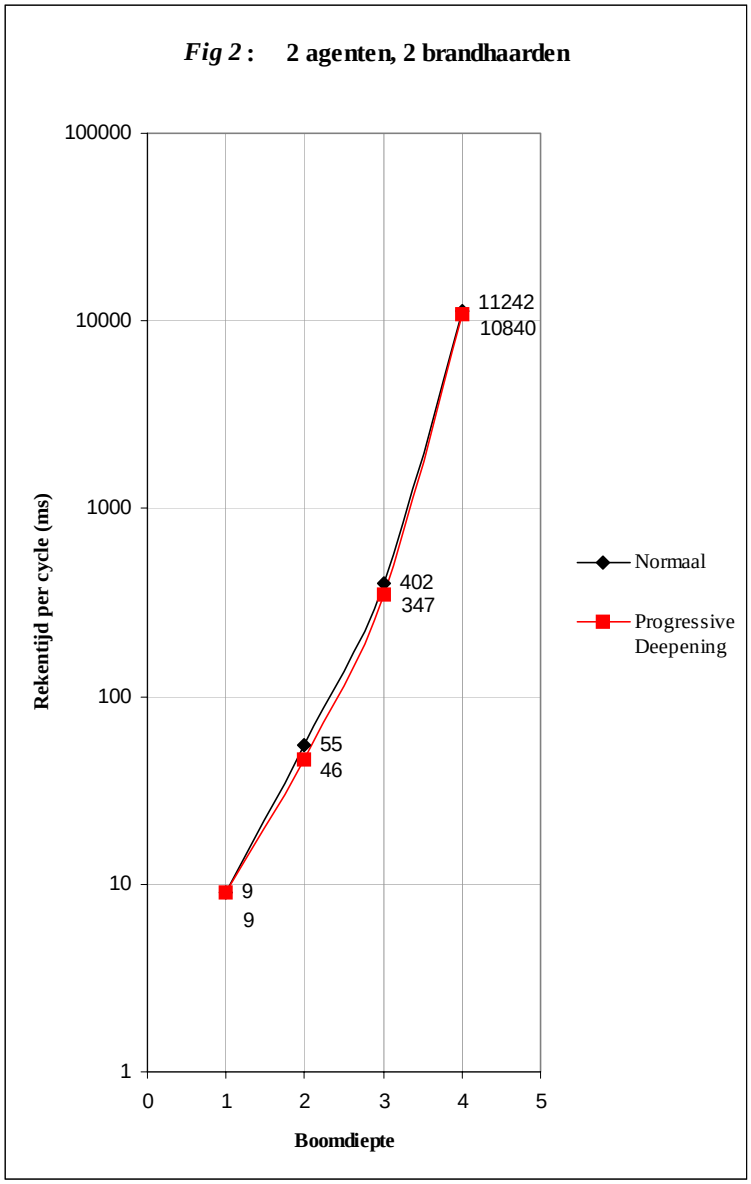
5.3 Taakverdeling

We hebben dit project voornamelijk als team gedaan, en de resultaten die we hebben geboekt zijn dus niet representatief voor één van de leden van ons tweetal. Wel is er tijdens het project af en toe wat werk verdeeld: Zo heeft Stefan wat meer tijd geïnvesteerd in het vinden van achtergrondinformatie, terwijl Aron iets meer heeft nagedacht over de implementatie, omdat hij iets meer ervaring had met C++. Ook bij het schrijven van het verslag hebben we deze verdeling min of meer aangehouden. Wel hebben we constant elkaar op de hoogte gehouden van vondsten en ideeën, en alles is in overleg tot stand gekomen.

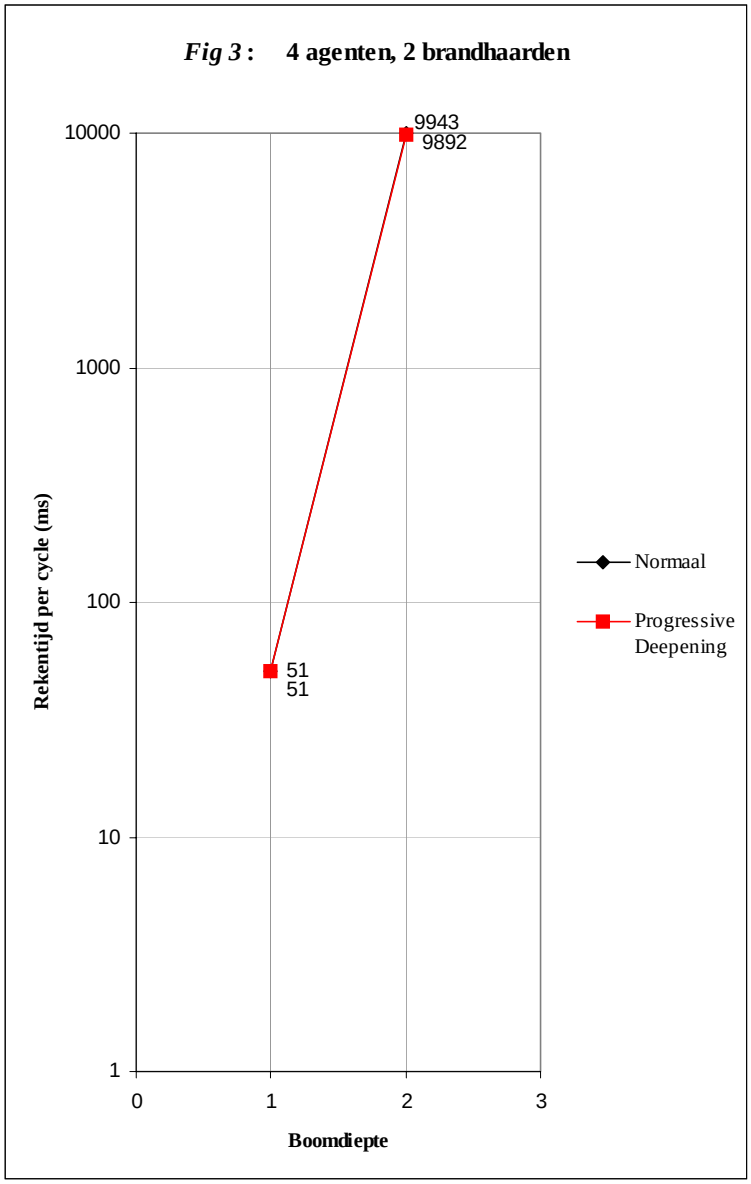
Bijlage 1: Resultaten van progressive deepening

Boom Diepte	2 agenten, 2 brandhaarden			4 agenten, 2 brandhaarden			2 agenten, 4 brandhaarden		
	Wel PD	Geen PD	Verschil in %	Wel PD	Geen PD	Verschil in %	Wel PD	Geen PD	Verschil in %
1	9	9	0,0	51	51	0,0	20	20	0,0
2	55	46	16,4	9943	9892	0,5	83	63	24,1
3	402	347	13,7				512	429	16,2
4	11242	10840	3,6				12134	11622	4,2

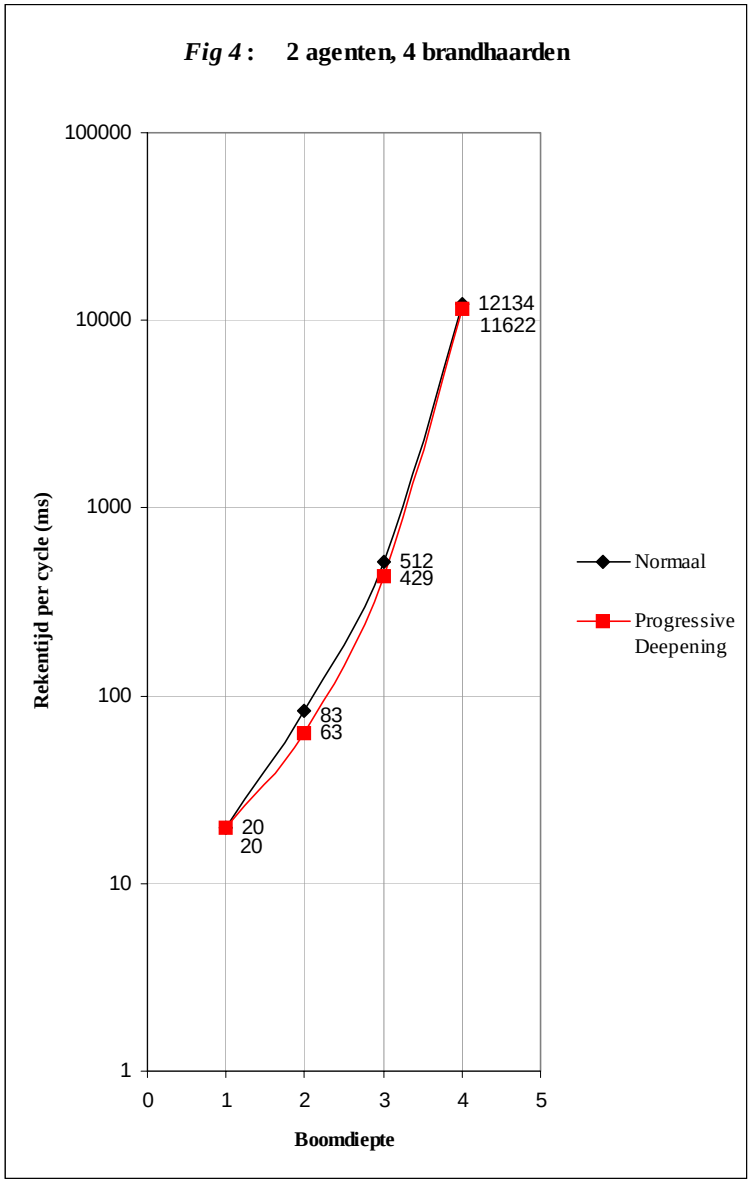
Figuur 1: tabel van de resultaten in verschillende situaties



Bijlage 1: resultaten van progressive deepening



Bijlage 1: resultaten van progressive deepening



Bijlage 2: Literatuurlijst

[1] S. Post en M. Fassaer; the design and implementation of software agents for the ‘robocuprescue simulator system’

[2] I. Bratko; Prolog, programming for Artificial Intelligence

[3] Adriaan D. de Groot; Thought and Choice in Chess

[4] RoboCupRescue Technical Committee; How to develop a RoboCupRescue Agent

[5] Verschillende websites, waaronder:

- http://www.compapp.dcu.ie/~tonyv/iterative.html
- http://www.engr.uvic.ca/~aschoorl/ceng420/notes/Game_Extras.pdf (p.89)
- http://cindy.cis.nctu.edu.tw/AI/ai1/ai-6.htm
- http://www.insight.demon.co.uk/Computer_chess/introduction/introduction.htm
- http://www.cs.miami.edu/~geoff/Courses/CSC545-S04/Content/GameSearch.shtml

Op deze sites wordt Progressive deepening steeds heel kort genoemd, maar uitgebreidere konden wij niet vinden.