Nota's bij slides – Robin Haveneers

Slide 1: Voorwoord
Het onderwerp dat ik graag aan jullie kom voorstellen vandaag heeft als titel "Spelinhoud generatie door middel van probabilistische programmeertalen. Een onderzoek naar de functionaliteit van ProbLog. "Spelinhoud generatie" klinkt in het Engels als "Game content generation" en toegegeven, in mijn ogen klinkt dat wel wat interessanter en indrukwekkender. De kans bestaat dat ik deze termen door elkaar gebruik maar ze betekenen dus hetzelfde. Nu we het eerste deel van de terminologie achter de rug hebben, zal ik even schetsen wat ik graag aan jullie kom vertellen.

Slide 2: Inhoudstafel
Beginnen doe ik met uit te leggen in welke domeinen of meer bepaald welke soorten domeinen gebruik maken van deze game content generation, en dan specifiek de "procedurale" vorm, die ik nog uitgebreid zal uitleggen. Ik tracht ook te schetsen waarom dit domein erg interessant is voor informatici, en welke kennis en die je momenteel al hebt al erg kan van pas komen, maar ook welke kennis je eventueel minder hebt die ook aan bod komt in dit onderwerp. Ik ga ook in het kort nog eens schetsen wat declaratief programmeren is aangezien dat erg relevant is voor dit onderwerp, maar ik vermoed dat de meesten – na de lessen declaratieve talen- hiermee wel vertrouwd zijn.

Verder ga ik toelichten wat procedural content generation, of PCG, juist is met bijhorend twee manieren om aan PCG te doen, toegelicht aan de hand van enkele voorbeelden. Verder zal ik verklaren hoe de onderzoeksgroep declaratieve talen en artificiële intellegentie hier aan het deperatement een 'variant' op ProLog heeft gemaakt die ProbLog heet, en wat hier zo anders en speciaal aan is. Afronden doe ik door te vermelden hoe ProbLog van toepassing zou kunnen zijn in het domein van PCG (wat dus mijn onderzoek is), en wat mijn visie tot nog toe hierover is.

Slide 3: Inhoudstafel met enkel PCG in kleur
Slide 4: Situering
Zonder meer, zal ik nu van start gaan met een korte schets te geven wat er in mijn papers precies werd behandeld. In mijn papers wordt vooral gepraat over het generen van simpele game content zoals doolhoven en kerkers. Hoewel dit in de papers maar eenvoudige voorbeelden zijn, is het ook voor de hand liggend dat dergelijke algoritmes en denkpatronen zeker ook gebruikt worden bij het creëren van grotere, complexere spellen. In mijn papers behandelt men vooral de aanpak waarbij gebruik gemaakt wordt van declaratieve talen, zoals AnsProLog en andere varianten of talen zoals ProLog. Hoewel het grafische aspect minder aan bod komt, krijg je toch een heel goed beeld van hoe je zelfs met pure ASCII-tekens een doolhof en of kerker kan construeren.

Slide 5: Relevantie
Voor het begrijpen van mijn papers was een basiskennis van Prolog vereist. Men behandelt op een gegeven moment ook een vorm van automaten in de tekst die gebruikt worden om bepaalde maps te genereren, en daarvoor kwam mijn kennis van automaten uit fundamenten van de informatica en automaten en berekenbaarheid erg van pas. Zoals nu wel zal opvallen, zijn mijn papers meer naar de theoretische en minder naar de praktische kant gericht,

desalniettemin heeft het mij echt wel inzicht gegeven hoe je zelf een kerker of doolhof zou generen moest je zelf een spel willen ontwikkelen. Ik heb 3 papers gelezen die meer handelden over het effectieve genereren van speelwerelden en 1 paper, die van de KU Leuven zelfs was, die in mijn ogen van iets hoger niveau was, behandelde het onderwerp over inferentie en 'leren' in probabilistische programmeertalen aan de hand van gewogen booleaanse formules. Deze laatste kwam wel erg van pas om ProbLog beter te begrijpen en in te zien "hoe" ProbLog deze kansen verwerkt onder de motorkap.

Slide 6: Declaratief programmeren
Als afsluiter van deze inleiding wil ik nog snel even herhalen wat declaratief programmeren juist is. Op het eerste zicht is het misschien raar dat je een logische of dus declaratieve programmeertaal gaat gebruiken om spelinhoud te maken, maar het is dus wel degelijk mogelijk zoals je later zien. Zoals we allemaal weten is declaratief programmeren de tak van de informatica waarin je niet beschrijf *'hoe*' je programma moet uitgevoerd worden, maar dat je eerder beschrijft '*wat'* er aan de hand is en zo een declaratie neerschrijft van het programma, van daar de naam dus. Vaak berusten declaratieve talen dus op logica en tonen ze heel wat overeenkomsten met logische formules.

Slide 7: Inhoudstafel
Nu we de basis een beetje gezet hebben een klein overzicht hebben van wat mijn papers behandelen is het tijd om het leukere aspect te gaan bekijken. Ik zal beginnen met uit te leggen wat het reeds genoemde PCG is, of voluit 'procedural content generation'.

Slide 8: PCG Wat is het
PCG, of dus nogmaals, procedural content generation, wordt door Wikipedia omschreven als 'a method of creating data algorithmically rather than manually', of dus een manier om algoritmisch –beter dan manueel- data te creëren. Eerst en vooral wil ik jullie erop wijzen dat dit niet per sé hetzelfde is als 'dynamisch' content te genereren. Toegepast op het creëren van data voor spelletjes, betekent PCG dus niet per se dat levels, doolhoven en andere zaken bij runtime worden gegenereerd. Ik was in het begin een beetje verward door te denken dat PCG enkel werd gebruikt om levels en props te genereren afhankelijk van voorgaande keuzes. Die twee zijn dus niet noodzakelijk hetzelfde. PCG komt er eigenlijk op neer dat je verschillende soorten content kan genereren met eenzelfde algoritme. Hoe dit precies in zijn werk gaat, zal zeker en vast nog duidelijk worden. Maar om het dus samen te vatten is PCG dus een methode om bij de uitvoering van je spel een verschillend level te krijgen met hetzelfde algoritme en zelfs met (nagenoeg) dezelfde parameters. Ik kan me voorstellen dat het probabilistische deel hier al een iets duidelijker plekje krijgt, zo niet, wacht nog even en dan zal het wel zo zijn.

Slide 9: PCG met ASP
Een eerste paper in mijn selectie van papers handelde over PCG met ASP. ASP staat voor answer set programming, en betekent in mensentaal zo veel als het halen van mogelijke antwoorden uit een set van feiten. Ik zal dit proberen te illustreren aan de hand van een voorbeeld.

De gegeven set bevat vier lijnen. De eerste regel toont een verzameling van mogelijkheden aan: het kan zijn dat het geregend heeft, dat er een sproeier heeft aangestaan, beide tezamen

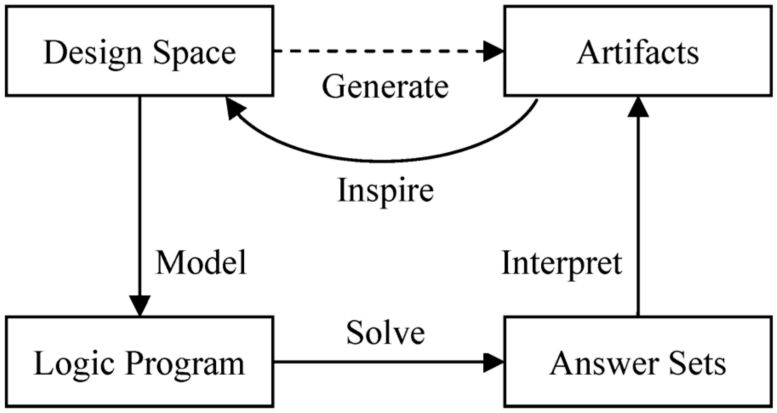
of geen van beide. Het zegt ook dat zowel regen als sproeier noodzakelijker wijs impliceren dat het nat is (herinner dat ‘: - ‘ in prolog stond voor de omgekeerde implicatiepijl ⇐). De laatste regel geeft ons de mogelijkheid, of beter de verplichting, om droogheid af te leiden als er geen reden is om natheid te veronderstellen.

```
{rain, sprinkler}.
wet :- rain
wet :- sprinkler.
dry :- not wet.
```

Nu zijn de mogelijke answer sets van dit stukje code de volgende:

```
dry.
wet, rain.
wet, sprinkler
wet, rain, sprinkler.
```

Slide 10: ASP Schema
Hoe wordt dit nu gebruikt bij PCG? Wel dat zal het volgende prentje hopelijk een beetje duidelijker maken.



Deze **afbeelding** vertelt ons eigenlijk dat om artifacts, of props, onderdelen van een spel te genereren we eerst de design space modeleren in een logisch programma, zoals het geen hierboven. We draaien dan dit logisch programma door een ASP-solver en zo verkrijgen we answer sets die voldoen aan het model die we dan op hun beurt kunnen interpreteren als beschrijvingen van de gewenste ‘artifacts’.

Slide 11: Spelvoorbeeld
Ik kan me inbeelden dat dit nog niet volledig duidelijk is, dus laat ik nog een iets concreter voorbeeld aanhalen, specifiek voor games.

```
color(red;yellow;green;cyan;blue;magenta).
dim(1..6).

{cell(C,X,Y) : color(C)} 1 :- dim(X), dim(Y).
{start(X,Y):dim(X):dim(Y)} 1.
{finish(X,Y):dim(X):dim(Y)} 1.
```

De eerste twee regels zijn vanzelfsprekend. Color is een van de kleuren tussen haakjes en dim is een getal van 1 tot 6. De eerste regel van de onderste drie zegt in feite dat er 1 cel feit wordt geproduceerd voor elke toekenning van X en Y. Dat zijn dus 36 cellen. De laatste twee regels definiëren dat er 1 cel is die de startcel is en 1 cel dat de eindcel is. Breiden we zo een programma uit met meerdere regels en feiten dan zie je wel in dat je inderdaad op een nog relatief eenvoudige manier complexe puzzels kan maken.

Slide 12: Ansprolog kort

Wat ik nog niet vermeld was, was dat de bovengenoemde syntax AnsProlog heet en in feite een uitbreiding is op ProLog. Het meest functionele aan AnsProlog zal het volgende stukje code trachten te verklaren

```
x { p , q , r } y
```

Dit stukje code wil simpelweg zeggen: 'kies minstens x elementen p, q, r maar kies er niet meer dan y.

Wat ik ook nog even wil benadrukken is dat hoewel AnsProlog geen echte probabilistische programmeertaal is, er toch een notie van kansen aanwezig is: je weet namelijk niet hoe de uitvoering zal verlopen, aangezien je keuzes laat in je programma. Deze keuzes kunnen bij elke uitvoering verschillen. Uit onderzoek is gebleken dat deze aanpak, met ASP-solvers, erg in trek is aangezien PCG voornamelijk begaan is met twee problemen: de betrouwbare generatie van gewenste content uit een design space ontwerp en het feit dat dit snel en flexibel is en net dat is het geen wat erg naar voor komt bij het gebruik van dergelijke ASP-solvers.

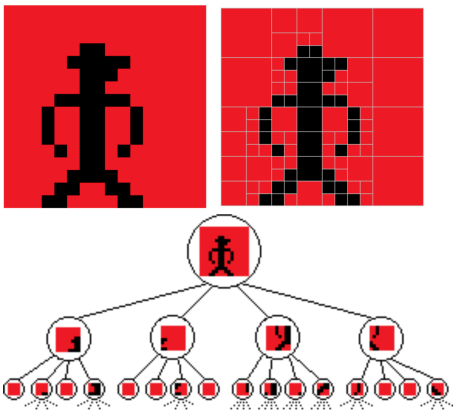
Slide 13: Kaarten met meer constraints

Als laatste notie wil ik erbij vertellen dat op het eerste zich zo'n levels, doolhoven, of kerkers maken eenvoudig lijkt in logisch programmeren. En dat is voor een stuk ook zo. Maar houdt er ook rekening mee dat als je een dergelijk algoritme schrijft dat het niet voor de hand liggend is dat elke uitvoer effectief een 'speelbaar' gebied is. Dit implementeren, dat elke map die er uit komt speelbaar is, is al minder vanzelfsprekend. Of wat als de speler een bepaalde volgorde moet volgen om het level te beëindigen, stel dat hij bijvoorbeeld een diamant nodig heeft om een bepaalde deur te openen, dan moet je er voor zorgen dat deze diamant bereikbaar is en toch niet te makkelijk te vinden is. Op de slides zie je enkele **afbeeldingen** van kaarten, telkens verbeterd om meerdere constraints te voldoen. Welke constraints en regels worden toegevoegd ga ik hier niet volledig uit te doeken doen, maar moest je geïnteresseerd zijn mag je dit uiteraard altijd vragen en het valt uiteraard ook te lezen in de papers.

Nu we het concept van ASP snappen, is er nog een andere manier om bepaalde aspecten te verwezenlijken op het gebied van PCG. Deze aanpak is minder specifiek en stelt eerder algoritmes voor om bepaalde voorwaarden van je spel te voldoen. Dit kan uiteraard gecombineerd worden met ASP om zo tot degelijke algoritmes te leiden. Een van zulke denkwijzes of algoritmes als je wil, is space partitioning en dit wordt in mijn papers vooral aangehaald om voornamelijk kerkers te genereren. Er zijn nog 3 andere algoritmes, maar ik zal voornamelijk dit algoritme uitleggen omdat dit het eenvoudigst te begrijpen is en in mijn ogen het 'tofste' is om te bekijken.

Slide 14: Space partitioning

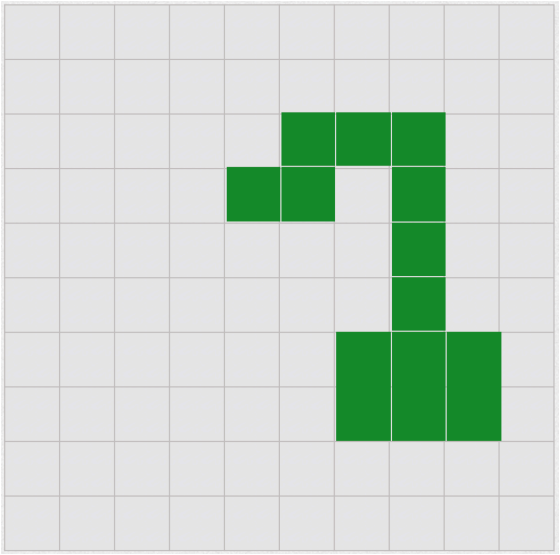
Dit gaat als volgt in zijn werk: neem een partitie van de ruimte (dus een subset van een 2D/3D space, of de gehele space) en verdeel deze in disjuncte sets zodat elk punt in de space in precies 1 van deze subsets ligt. De meest populaire methode hiervoor is binary space partitioning, en werkt –zoals je al kan horen aan de naam- door telkens de sets in twee te verdelen. Aangezien dit moeilijker is om in woorden uit te leggen, toon ik hierbij de volgende **afbeelding**.



De linkse afbeelding toont de originele afbeelding. Rechts zie je hoe de afbeelding opgesplitst is zoals weergeven in de boom onderaan. Je blijft dus splitsen tot er maar 1 kleur overblijft per subset. Je kan je nu wel inbeelden dat dit een recursief algoritme is. Specifiek bij het generen van kerkers lijdt dit tot de voorwaarde dat geen twee kamers van de kerkers overlappen. Pas je hier nu nog een algoritme op toe dat kamers verbindt met gangen, dan ben je in staat van speelbare en steeds correct dungeons te creëren.

Slide 15: Agent-based dungeon growing

Een ander algoritme of denkwijze is de agent-based dungeon growing. Deze is meer geneigd van chaotische, organische kerkers te generen in tegenstelling tot de georganiseerde kerkers van daarnet. Wederom zal ik dit uitleggen aan de hand van een **afbeelding**.



Je start van op een bepaalde positie en begint vanaf daar te graven naar een richting. Op een gegeven moment kan je bijvoorbeeld beslissen van hier een kamer neer te zetten en daarna verder te graven. Je kan je nu wel voorstellen dat dit kan leiden tot heel slechte en/of kleine

kerkers. Dit algoritme wordt in de tekst uitgebreid met look ahead checking en andere heuristieken. Dit zou ons opnieuw te ver leiden dit allemaal uit de doeken te doen.

Slide 16: Cellular automata
Verder heb je ook nog cellular automata die je kan gebruiken om kerkers te creëren. Dit is een heel interessant maar ook heel specifiek model dat erg onderzocht is geweest in computerwetenschappen. Zo'n cellular automata bestaat in feite uit een rooster van n dimensies een verzameling toestanden en een overgangsregel. Je geeft elke cel in je automaat een bepaalde toestand en deze toestand en zijn omgevingen bepaalt de toestand van een bepaalde cel in de toekomst. Deze cellular automata zijn heel veelzijdig en geven de mogelijkheid tot he creëren van oneindige kerkers bij runtime.

Slide 17: Grammar-based dungeon generation
Verder is er ook nog grammar-based dungeon generation. Dit wordt geïllustreerd op de volgende **afbeelding**. Tenslotte zijn er ook nog iets geavanceerdere platform generatie methodes zoals rythym-based methodes die –zoals het woord zegt- gebaseerd zijn op ritme.

Slide 18: Spelletjes voorbeelden
Enkele voorbeelden van spelletjes die gebruik maken van deze methodes van platform generatie zijn Spelunky, en Infinite Mario Bros. (**afbeeldingen**)

Slide 19: Inhoudstafel
Nu we deze concepten onder de loep hebben genomen, is het tijd om kennis te maken met ProbLog.

Slide 20: ProbLog – Wat is het
ProbLog is een taal ontwikkeld hier aan het departement computerwetenschappen en geeft je alle functionaliteiten van ProLog waarbij je ook nog eens onzekerheden aan de hand van kansen kan toevoegen. De kracht achter ProbLog schuilt in het leerzaam aspect. ProbLog kan marginale kansen leren gegeven bepaalde bewijzen en feiten en uit (partiele) interpretaties. Het neemt je programma met zijn queries en bewijs en converteert dit naar een gewogen booleaanse formule. Dit kan opgelost worden met verscheidene algoritmes. De hele theorie achter ProbLog en zijn inferentie taken zou ons te ver leiden en kan makkelijk nagelezen worden op de site van ProbLog of in de paper die ik hierover heb gelezen. Het leek me interessanter om een verschil te geven tussen de eerste en de tweede, recente versie van ProbLog en een vergelijking te maken met ander programmeertalen.

Slide 21: ProbLog – Vergelijking
Vergeleken met de meeste andere probabilistische programmeertalen is ProbLog expresiever met respect tot regels die niet toegestaan zijn in een programma. Bijvoorbeeld PRISM en ICL stellen de voorwaarde dat regels in een programma niet cyclisch mogen zijn. In ProbLog daarentegen kan dit wel. Een voorbeeld van een cyclische regel is de volgende `smokes(X) :- smokes(Y), influences(Y,X).` PRISM stelt ook de voorwaarde dat regels met unificerende heads 'mutual exclusive' bodies hebben. Dit wil dus zeggen dat ten hoogste een van de bodies waar kan zijn, maar geen twee tegelijk. Voorbeeld hiervan is de volgende:

`alarm :- burglary, alarm :- earthquake.` Hierin wordt gesteld dat een alarm kan afgaan door een inbraak of een aardbeving of beide. Dit zou niet kunnen in PRISM. ProbLog2 is nog eens uitzonderlijk daar het met meerder queries tegelijk kan rekening houden. Zo is ProbLog2 eigenlijk een mooie hybride van vele andere probalistische programmeertalen.

Slide 22: Voorbeeld

Zie het voorbeeld hieronder, hierin zie je eenvoudig hoe ProbLog 2 afleidingen maakt.

```
1  %%% Probabilistic facts:
2  0.5::heads1.
3  0.6::heads2.
4
5  %%% Rules:
6  twoHeads :- heads1, heads2.
7
8  %%% Queries:
9  query(heads1).
10 query(heads2).
11 query(twoHeads).
12
```

Evaluate

Query ▼	Location	Probability
heads1	9:7	0.5
heads2	10:7	0.6
twoHeads	11:7	0.3

Slide 23: Inhoudstafel

Slide 24: ProbLog en PCG, mijn visie

Tenslotte rest er mij nog te vertellen hoe ik denk dat ProbLog toepasbaar zou zijn op het gebied van PCG. Dit leek me op het eerste zicht nogal evident. Je maakt bepaalde ProbLog regels die een kans geven aan bijvoorbeeld de verhouding muren en wandelpad, of monsters, of hoeveel centjes er te rapen vallen. Dit is echter volgens mij slechts een tipje van de sluier en ik vermoed, wel ik ben vrij zeker dat er veel meer kan verwezenlijkt worden met ProbLog. Ik heb bijvoorbeeld vrij laat ontdekt hoe ProbLog leert en hoe goed het daar in is. Je kan op deze manier misschien werelden en levels creëren afhankelijk van de speelstijl van de speler. Duurt het de speler soms te lang om bijvoorbeeld om een bepaald voorwerp te vinden, zorg ervoor dat in het volgende level dit voorwerp een pak eenvoudiger te vinden is of net niet. Neemt de speler bijvoorbeeld op een T-splitsing het vaakst de afslag naar links, zorg er dan voor dat deze afslag nergens naar lijdt, of net wel, afhankelijk van hoe moeilijk je het de speler wil maken.