Genereren van spelinhoud met probabilistisch programmeren
Een onderzoek naar de toepasbaarheid van ProbLog

Robin Haveneers
Bachelor Informatica
KU Leuven, België
robin.haveneers@student.kuleuven.be

Abstract

Deze paper bespreekt de toepasbaarheid van ProbLog op het gebied van procedurele creatie van spelinhoud. In dit onderzoek wordt ProbLog vergeleken met ASP. Problog en ASP zijn beide declaratieve talen waarbij het grote verschil is dat ProbLog gebruik maakt van kansen. Het onderzoek wordt gevoerd op het gebied van snelheid, variatie en modellering en of ProbLog op deze vlakken voordelen biedt dankzij het gebruik van kansen. Uit de resultaten van dit onderzoek kan worden afgeleid dat op vlak van snelheid ProbLog het niet haalt van AnsProlog. ProbLog bereikt echter wel een veel grotere variatie in gegenereerde oplossingen die voldoen aan het gevraagde model. ProbLog biedt dankzij zijn probabilistisch aspect ook een aantal interessante modelleringsvoordelen. We besluiten dat ProbLog in de praktijk goede mogelijkheden biedt op vlak van game content generation indien variatie de belangrijkste factor is en snelheid meer van ondergeschikt belang is.

1 Introductie

Generatie van spelinhoud (Engels: "game content generation") is een belangrijk onderdeel bij de ontwikkeling van eender welk soort spel en is ook inmiddels een gevestigd onderzoeksdomein. Het procedureel genereren van spelinhoud is gedefinieerd als *"het genereren van spelinhoud aan de hand van een algoritme met gelimiteerde of indirecte invoer van de gebruiker"* [Togelius *et al.*, 2015]. Met andere woorden: het gaat over software die met één soort beschrijving verschillende sets van spelinhoud kan genereren. Onder 'inhoud' verstaan we onder andere rekwisieten (wapens, kisten, muntjes ...), evenals plattegronden en speelwerelden (kerker, platform, onder water ...). Het spelmechanisme ("game engine") zelf en de AI achter niet-speelbare personages vallen niet onder deze definitie van spelinhoud.

Het gebruik van Answer Set Programming (ASP) voor de generatie van spelinhoud is echter nog nieuw, maar wel reeds goed bestudeerd [Nelson and Smith, 2015], [Togelius *et al.*, 2011], [Smith *et al.*, 2011]. Vanaf 2011 zijn onderzoekers begonnen met de toegevoegde waarde te onderzoeken van ASP op gebied van *game content generation*, zie [Yannakakis and Togelius, 2011] en [Smith and Bryson, 2015]. Alvorens dieper in te gaan op het onderzoek, wordt relevante achtergrondinformatie gegeven.

2 Achtergrond

2.1 ASP en AnsProlog

ASP is een vorm van declaratief programmeren gericht op moeilijke (NP-harde) zoekproblemen. Het is voornamelijk bruikbaar in kennis-intensieve toepassingen, maar kan ook gebruikt worden voor zijn generatieve mogelijkheden. In een ASP-programma wordt beschreven *wat* moet berekend worden en niet *hoe*. Bij game content generation wordt in het programma het ontwerp van de spelinhoud beschreven. Aan de hand van dit programma wordt een answer set, een geldige beschrijving van een wereld die voldoet aan het ontwerp, berekend.

De ASP-programma's beschouwd in deze paper zijn geschreven in AnsProlog: een syntax vergelijkbaar aan Prolog. AnsProlog biedt de mogelijkheid om "keuzeregels" of "niet deterministische regels" met numerieke boven- en/of ondergrenzen te definiëren. Een voorbeeld hiervan is

```
1{s,t}1 :- p
```

dit wil zeggen: "als `p` in het model voorkomt, kies dan willekeurig exact één van de atomen `s,t` (dus `s`, of `t`) dat wordt toegevoegd". Een andere uitbreiding op de syntax is de mogelijkheid om voorwaarden of constraints te bepalen die altijd waar moeten of net nooit waar mogen zijn, dit zijn "integrity constraints" of "integriteitsvoorwaarden". Beschouw de volgende regels als voorbeelden:

`:- not victory` en `:- birds.`

Deze regels bepalen respectievelijk dat `victory` altijd moet optreden, en dat `birds` nooit mag optreden in de "answer set".

Het vinden van antwoorden, mogelijke modellen op deze ASP-programm's berust op het vertalen van het programma in een Booleaanse formule om er nadien een SAT-solver op toe te passen. In een volgende paragraaf wordt toegelicht dat de solver gebruikt in deze paper, `clingo`, dit patroon volgt en gebruik maakt van state-of-the-art algoritmes op vlak van Boolean constraint solving.

2.2 ProbLog

ProbLog is een probabilistische uitbreiding op Prolog [De Raedt *et al.*, 2007]. Een ProbLog-programma bestaat uit twee delen: feiten met een kans geannoteerd en een set van regels en niet-geannoteerde feiten. Een voorbeeld van een feit met een kans is het volgende: `p::friend(a,b).`. Deze regel zegt dat `a` bevriend is met `b` met een kans `p`.

Een ProbLog-programma beschrijft een kansverdeling over alle mogelijke werelden die gegenereerd kunnen worden met het programma. Elk feit van de vorm `p::f` geeft je de keuze, ofwel behoort `f` tot het model met kans p ofwel behoort het niet tot het model met een kans $1 - p$. Een totaalkeuze is nu een subset van alle mogelijke probabilistische feiten. Als er dus n feiten zijn met een keuze, zijn er 2^n totaalkeuzes. De kans dat een bepaalde wereld een geldig model is van het ProbLog-programma, is gelijk aan zijn totaalkeuze. De kans op een wereld dat geen geldig model is is 0.

ProbLog biedt nu de mogelijkheid om een programma te samplen. Dit samplen zal een bepaald aantal mogelijke modellen (afhankelijk van het gekozen aantal) terug geven. ProbLog doet aan rejection based sampling. Het genereert dus een set van werelden, haalt daar diegene uit die niet voldoen aan bepaalde voorwaarden uit het programma, haalt nadien diegene er uit die niet voldoen aan de query en geeft uit de overblijvende werelden één terug. Indien er `evidence`, de integriteitsvoorwaarden, wordt opgegeven met een heel kleine kans, is de kans dat het gegenereerde sample daaraan voldoet ook zeer klein, waardoor rejection based sampling lang kan duren.

Aan de hand van inferentie kan ProbLog antwoorden vinden op de queries die worden meegegeven aan een ProbLog-programma. Deze inferentie gebeurt in twee stappen.

In de eerste stap wordt het programma samen met de evidence en de queries omgezet naar een gewogen Booleaanse formule om nadien op die formule inferentie uit te voeren. In deze stap wordt een variabel-vrij programma gezocht voor de opgegeven queries en evidence, vervolgens wordt dit programma omgezet in een logische formule en nadien herwerkt naar de overeenkomstige conjunctieve normaalvorm (CNF) [Mendelson, 1987]. Deze wordt tenslotte herwerkt naar zijn gewogen vorm: als het programma een regel `p::f` bevat, kennen we aan `f` de kans p toe en aan $\neg$`f` de kans $1 - p$.

In een tweede stap wordt inferentie uitgevoerd op de bekomen gewogen formule. Er zijn drie inferentie-taken die kunnen worden uitgevoerd, **MARG**, **EVID** en **MPE**.

1. Bij **MARG** is het de bedoeling voor een gegeven aantal queries de marginale kansverdeling van zo'n query, gegeven een bepaalde observatie te berekenen, of nog: $P(Q|E)$ met Q de kans en E de evidence.
2. **EVID** of 'de kans van evidence' is het berekenen met welke kans bepaalde evidence voorkomt in de modellen, of: $P(E)$.
3. **MPE** zoekt naar de meest waarschijnlijke combinatie van alle niet-evidence predicaten, gegeven de evidence.

ProbLog berust hoofdzakelijk op de **MARG** inferentie methode: het berekenen van de kans op slagen van een query. Zie ook [Fierens *et al.*, 2015].

3 Probleemstelling en hypothese

De probleemstelling van dit onderzoek baseert zich op de volgende vragen:

Q$_1$ Is het ProbLog-systeem geschikt om inhoud van spellen te genereren?

Q$_2$ Hoe scoort ProbLog ten opzichte van ASP/AnsProlog op vlak van snelheid en variatie?

Q$_3$ Wat zijn de modellerings voor- en nadelen bij het gebruik van ProbLog?

In deze paper wordt een vergelijking van ProbLog en ASP gemaakt, steunend op drie soorten puzzels die worden onderlegd aan een aantal criteria. De hypothese is dat de probabilistische kant van ProbLog een voordeel kan bieden bij het genereren van spelinhoud.

4 Aanpak

In dit onderzoek wordt voor de uitvoering van de ASP-programma's, geschreven in AnsProlog, gebruik gemaakt van `clingo` uit het Potassco-pakket[1]. `clingo` is de combinatie van een grounder, `gringo` die de programma's variabel vrij maakt en een solver, namelijk `clasp`, die answer sets berekent [Gebser *et al.*, 2010] aan de hand van een conflict-gedreven solver gebruikmakend van technieken uit het SAT-onderzoeksdomein. [Gebser *et al.*, 2012].

4.1 Puzzels

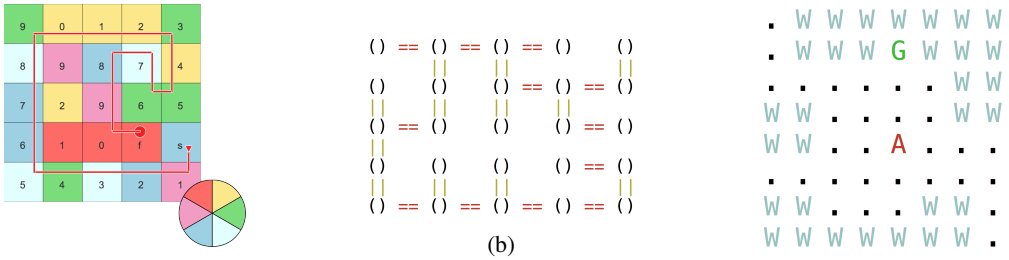
Voor dit onderzoek wordt beroep gedaan op drie soorten puzzels. De drie puzzels zijn de *chromatic maze*, *perfect maze* en de *dungeon*.

Chromatic Maze
De *chromatic maze* (cfr. [Smith and Mateas, 2011], figuur 1a) is een steeds vierkante puzzel en is een soort doolhof waarbij elk vakje een kleur uit een kleurenwiel heeft. Om de puzzel op te lossen moet een pad van start naar finish over de vakjes gevonden worden. Een overgang van het ene vakje naar een ander vakje is enkel toegelaten als de kleurenovergang die optreedt een overgang is in het gegeven kleurenwiel, met de klok mee of tegen de klok in.

Perfect Maze
De *perfect maze* (cfr. [Nelson and Smith, 2015], figuur 1b) is een steeds vierkante puzzel en is een perfect doolhof, dit wil zeggen dat vanuit elke positie een pad kan gemaakt worden naar het vakje linksboven `(1,1)`. Daardoor zal de puzzel steeds een pad bevatten van het vakje linksboven tot het vakje rechtsonder.

Dungeons
Een *dungeon* is een representatie van een soort kerker (cfr. [Nelson and Smith, 2015], figuur 1c). De *dungeon* is altijd perfect vierkant. In deze voorstelling van een kerker is er één altaar (`A`), één diamantje (`G`) en een variabel aantal muren (`W`). De rest van de vakjes hebben geen specifieke eigenschap. De gedachtegang van zo'n *dungeon* is om zich van de ingang naar de diamant te begeven, deze op te rapen, de diamant naar het altaar te brengen om zo de uitgang te openen en zich tenslotte naar die geopende uitgang te begeven. Tenzij anders vermeld, wordt voor al de experimenten een *dungeon* gebruikt waarbij er exact één diamant en één altaar is en waarbij voor de rest van de vakjes, willekeurig, ofwel een muur ofwel niks wordt gezet.

[1] Potsdam Answer Set Solving Collection, `http://potassco.sourceforge.net/`



(a) (b) (c)

Figuur 1: Voorbeeld van een geldige *chromatic maze* met oplossing (a) , en *perfect maze* (b) een *dungeon* (c)

4.2 Vertaling

Het eerste deel van dit onderzoek bestond er in de AnsProlog-programma's, die de eerdergenoemde puzzels beschreven, om te vormen tot ProbLog-programma's. Er moet rekening gehouden worden met de constraints die aanwezig zijn in de AnsProlog-programma's. Aangezien AnsProlog en ProbLog beiden varianten zijn op Prolog, vormde dit geen noemenswaardige problemen. Ze hebben beide functionaliteiten verschillend van Prolog, maar evenzo verschillend van elkaar. De vertalingen van de puzzels in ProbLog, met commentaar als leidraad, staan in de appendices. De vertalingen zijn gebaseerd op de puzzels gegeven in [Smith and Mateas, 2011] en [Nelson and Smith, 2015]. De vertaling gebeurde op hoogniveau waardoor enkele stukken zonder veel moeite konden worden overgenomen. De kern van de puzzels is niet veranderd ten opzichte van de ASP-tegenhanger: ze hebben dezelfde eigenschappen en hetzelfde 'doel'.

Het eerste aspect van de vertaling was de omzetting van de niet-deterministische regels uit AnsProlog naar ProbLog. Zo zegt de AnsProlog-regel hieronder dat voor elke tegel, elk paar van x en y coördinaten, exact één kleur uit alle mogelijke kleuren moet worden toegekend. Dit komt uit het voorbeeld van de *chromatic maze*.

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

Eén mogelijke manier om dit in ProbLog te vertalen is door te zeggen dat voor elke `tile(X,Y,C)` met `X` en `Y` respectievelijk het x- en y-coördinaat, één kleur `C` uniform moet worden geselecteerd uit de lijst van alle kleuren. De code die dit uitdrukt, staat hieronder.

```
tile(X,Y,C) :-
  dim(X),
  dim(Y),
  colors(Lc),
  select_uniform(id(X,Y),Lc, C, _).
```

Analoge voorbeelden waren er bij de *perfect maze* waarbij de verbonden tile uniform wordt gekozen uit de 4 mogelijkheden: verbonden met een vakje links, rechts, boven of onder.

Een ander interessant aspect kwam aan bod bij een alternatieve vorm van deze puzzel. Daarbij was het de bedoeling om een maximaal aan horizontale, en bijgevolg een minimaal aantal verticale, verbindingen te hebben. Het doolhof vertrekt zo van linksboven en maakt een soort slangbeweging tot rechtsonder.

In AnsProlog kan dit gemodelleerd worden door eerst te definiëren, op basis van een regel, wat een verticale verbinding is en daarna een zogenoemde 'minimize'-constraint op te leggen op deze verticale verbindingen. Een vergelijkbaar resultaat kon in ProbLog verkregen worden door, in plaats van de verbinding uniform te laten kiezen, de keuze niet-uniform te maken door gewichten toe te kennen aan de kansen met `select_weighted`. De horizontale verbindingen krijgen een kans van beide 45% en de verbinding boven en onder een kans van nog elk 5%. De gegenereerde puzzels zijn nagenoeg altijd puzzels met een maximaal aantal horizontale verbindingen.

Verder zijn er in AnsProlog ook de integriteitsvoorwaarden. Een voorbeeld hiervan is de voorwaarde die zegt dat er steeds `victory` moet zijn en dat deze moet bereikt worden in minder dan `max_solution` en meer dan `min_solution` stappen. Met `victory` bedoelen we het feit dat er in de *chromatic maze* een geldig kleurenpad bestaat van start naar finish.

```
:- not victory.
:- victory_at(T), T < min_solution.
:- victory_at(T), max_solution < T.
```

In ProbLog is het mogelijk een analoge constraint op te leggen door middel van `evidence`, waarmee wordt bedoeld dat elke toekenning moet voldoen aan `victory`.

```
victory :- victory_at(T), time(T).
victory_at(T) :-
  max_sol(Max),
  min_sol(Min),
  time(T),
  T > min_sol,
  T < max_sol,
  finish(X,Y),
  player_at(T, X, Y).
evidence(victory).
```

Een alternatieve aanpak voor deze `evidence` is om expliciet deze constraints in je programma neer te schrijven en er zo voor te zorgen dat geen evidence moet voorzien worden. Zo was er bij de *dungeon* de voorwaarde dat er maximaal één diamant en één altaar konden zijn. De alternatieve aanpak is door uit de lijst van vakjes eerst twee tiles te kiezen die respectievelijk de diamant en het altaar zijn en uit de overblijvende vakjes te selecteren welke vakjes muren worden en welke leeg blijven. Deze aanpak en de aanpak met `evidence` samen met hun voor- en nadelen, worden verder nog besproken.

4.3 Analyse

Het meest belangrijke deel van dit onderzoek is nagaan hoe het ProbLog-systeem presteert. Het grootste deel van de analyse was op vlak van variatie. De snelheid van de systemen wordt eveneens onderzocht en tenslotte komt ook de modelleringscomplexiteit aan bod.

Snelheid

Een eerste criterium dat is behandeld in dit onderzoek is de snelheid. Er wordt onderzocht hoe lang het duurt een aantal samples, oplossingen die voldoen aan het programma, te genereren.

Om de snelheid te meten wordt het ProbLog systeem tegenover het `clingo`-systeem geplaatst. Er worden met het `sample`-commando van ProbLog telkens 100 samples gegenereerd, dat is mogelijk met het argument `-N`. Voor elke puzzel zag het commando er als volgt uit: `problog sample -N 100 puzzel`. Een sample is, zoals eerder, een mogelijke 'wereld' die voldoet aan het programma.

Voor `clingo` zijn er eveneens modellen gegenereerd met als argumenten `--models=100` en `--rand-freq=1` welke respectievelijk er voor zorgen dat `clingo` 100 modellen genereert en 100 procent willekeurige beslissingen neemt, hoofdzakelijk bij het kiezen van variabelen. Het commando voor de AnsProlog puzzels zag er als volgt uit: `clingo --models=100 --rand-freq=1 puzzel`.

Voor beide is dan getimed hoe lang het duurde voor deze 100 puzzels te genereren. Dit timen is gebeurd op een Macbook Pro, 2.7 GHz Intel i5-processor, 8GB DDR3 RAM onder OS X 10.11.

Variatie

Het tweede criterium is de variatie tussen de gegenereerde puzzels. Dit is bij puzzels een zeer belangrijk criterium. Puzzels die gegenereerd worden en maar minimale verschillen vertonen, zijn doorgaans niet gewenst. Een vaak terugkerend criterium is 'afstand' tussen twee posities $P_1(X_1, Y_1)$ en $P_2(X_2, Y_2)$, meer bepaald de Manhattan-afstand, welke als volgt gedefinieerd is: $D(P_1, P_2) = |X_1 - X_2| + |Y_1 - Y_2|$. Dit is een goede afstandsmaat omdat in geen enkele puzzel diagonale bewegingen zijn toegelaten.

- **Chromatic Maze**
  Voor de *chromatic maze* worden er 4 criteria geëvalueerd. De totale score wordt dan gegeven door
  $$\frac{C_{start}}{N} + \frac{C_{finish}}{N} + \frac{C_{sf}}{2 \times N} + \frac{2 * C_{kleur}}{N^2 * ((2 * N) - 2)}$$
  met N de hoogte of breedte van het spelbord: deze zijn namelijk gelijk. Elke score wordt dus herschaald naar één en nadien gesommeerd.
  - *Afstand tussen start-vakjes.*
    Het eerste criterium is de Manhattan-afstand tussen de start-vakjes. Neem $start_1(X_1, Y_1)$ en $start_2(X_2, Y_2)$ als coördinaten voor het startvakje bij respectievelijk de eerste en tweede puzzel. De score wordt dan als volgt berekend: $C_{start} = D(start_1, start_2)$.
  - *Afstand tussen finish-vakjes.*
    De afstand tussen de finish-vakjes is op een analoge manier berekend, ditmaal met de coördinaten $finish_1(X_1, Y_1)$ en $finish_2(X_2, Y_2)$ voor het finishvakje, als volgt: $C_{finish} = D(finish_1, finish_2)$.
  - *Afstand tussen de kleuren.*
    Voor elk vakje in de eerste puzzel wordt gezocht naar het dichtstbijzijnde vakje met dezelfde kleur in de tweede puzzel. Als er zo een vakje bestaat, zal dat vakje nadien niet meer beschouwd worden. Elke vakje kan namelijk maar maximaal één tegenhanger hebben. De score (C_{kleur}) wordt gegeven door het algoritme A.1 gegeven in pseudocode in de appendix.
  - *Afstand tussen start en finish*
    Tenslotte wordt de afstand tussen het start- en finish-vakje van de eerste puzzel vergeleken met diezelfde afstand bij de tweede puzzel. Dit laatste criterium geeft een maat van complexiteit: als de afstand tussen het begin- en eindpunt gelijk is, zijn de puzzels erg analoog. De berekening gebeurd met $start_1(Sx_1, Sy_1)$, $start_2(Sx_2, Sx_2)$, $finish_1(Fx_1, Fy_1)$ en $finish_2(Fx_2, Fy_2)$, als volgt: $C_{sf} = |D(start_1, finish_1) - D(start_2, finish_2)|$.
- Perfect Maze
  De *perfect maze* is getest, steunend op twee criteria: het aantal overeenkomstige verbindingen en de lengte van het kortst mogelijke pad. Het aantal overeenkomstige verbindingen krijgt een groter gewicht dan het kortste pad, aangezien de kans dat het kortste pad gelijk is, relatief groot is. De totale score wordt gegeven door:
  $$\frac{3}{4} * \frac{C_{verb}}{2 * N + 3 * (N - 4)} + \frac{1}{4} * \frac{C_{pad}}{(2 * N) - 2}$$
  met N de hoogte of breedte van het spelbord.
  - *Aantal gelijkaardige verbindingen*
    In de *perfect maze* wordt een verbinding weergeven als volgt: `parent(3,2,-1,0)` (zie ook appendix B.2), dit wil zeggen dat het vakje (3,2) verbonden is met (2,2). Dat is analoog met: `parent(2,2,1,0)`. Op deze twee manieren wordt getest of een bepaald vakje in de ene puzzel verbonden is met een bepaald vakje in de andere puzzel. De score (C_{verb}) wordt berekend door algoritme A.2 gegeven in pseudocode in de appendix.
  - *Verschil in kortste pad*
    Voor elk van de twee puzzels wordt met bestaande verbindingen, gegeven door de `parent`-relaties, het kortste pad berekend, gebruikmakend van het algoritme van [Dijkstra, 1959]. Deze waarde wordt bijgehouden in C_{pad}.
- Dungeon
  De *dungeon* is onderworpen aan 6 criteria. De totale score wordt gegeven door:
  $$\frac{1}{6}\left(\frac{C_d}{N} + \frac{C_a}{N} + \frac{C_{da}}{N} + \frac{C_m}{N} + \frac{C_{am}}{N} + \frac{C_{al}}{N}\right)$$
  met N de hoogte of breedte van het spelbord.
  - *Afstand tussen diamanten*
    Neem $diamant_1(X_1, Y_1)$ en $diamant_2(X_2, Y_2)$ als coördinaten voor het diamant bij respectievelijk de eerste en tweede puzzel. De score wordt dan als volgt berekend: $C_d = D(diamant_1, diamant_2)$.

– *Afstand tussen altaren*
Neem $altaar_1(X_1, Y_1)$ en $altaar_2(X_2, Y_2)$ als coördinaten voor het altaar bij respectievelijk de eerste en tweede puzzel. De score wordt dan als volgt berekend: $C_a = D(altaar_1, altaar_2)$.
– *Afstand tussen diamant en altaar*
Voor twee puzzels wordt het verschil in afstand bepaald tussen de locatie van de diamant en de locatie van het altaar. Dit zijn het aantal stappen die de speler minimaal moet zetten om de uitgang te kunnen openen en geeft dus een maat voor complexiteit. Dit wordt berekend met diamanten $diamant_1(X_1, Y_1)$ en $diamant_2(X_2, Y_2)$ en altaren $altaar_1(X_1, Y_1)$ en $altaar_2(X_2, Y_2)$ en wordt gegeven door: $C_{da} = |D(diamant_1, altaar_1) - D(diamant_2, altaar_2)|$.
– *Verschil in aantal muren*
Als vierde criterium wordt het verschil in aantal muren berekend (C_m).
– *Afstand tot dichtstbijzijnde muur*
Als voorlaatste criterium wordt voor elke muur de afstand berekend tot de dichtstbijzijnde muur in de tweede puzzel. Eenmaal er geen muren meer zijn die kunnen dienen als overeenkomstige muur, wordt de afstand maximaal gezet. Dit wordt geëvalueerd aan de hand van algoritme A.3, gegeven in pseudocode in de appendix. Neem in dit algoritme t_1 = "muur".
– *Afstand tot dichtstbijzijnde lege vakje*
Met hetzelfde algoritme als hierboven wordt ook voor elk leeg vakje in de eerste puzzel een zo dicht mogelijk leeg vakje gezocht in de tweede puzzel. Neem voor deze berekening t_1 = "leeg" in het algoritme en vervang C_{am} door C_{al}.

4.4 Modellering

Verder wordt de effectieve modellering onderzocht. Er wordt een analyse gemaakt naar waar de probabiliteit van ProbLog aan bod komt en waar die eventueel extra mogelijkheden biedt. Er wordt nagegaan hoe goed de `evidence` van ProbLog presteert en of er eventueel eenvoudigere alternatieven voor deze `evidence` bestaan bij de puzzels.

Verder is het in ProbLog mogelijk om de integriteitsvoorwaarden voor te stellen aan de hand van `evidence`. Het is eveneens duidelijk geworden dat dit op een alternatieve manier mogelijk is, zie sectie 4.2. Een voorbeeld hier van staat in appendix B.3, bij het model van de *dungeon* met en zonder `evidence`.

5 Resultaten

In deze paragraaf worden de resultaten besproken die bekomen zijn met de experimenten uit de vorige paragraaf.

5.1 Snelheid

De analyse op gebied van snelheid is zoals verwacht en niet bijzonder: het ProbLog-systeem is opmerkelijk trager dan `clingo`. ProbLog maakt gebruik van *rejection based sampling*, terwijl AnsProlog gebruik maakt van complexere methodes waaronder *no-good learning* gecombineerd met *backtracking*. `clingo` leert uit gemaakte 'fouten': wordt er een variabele toegekend die tot een mislukking leidt, zal deze variabele nooit nog worden toegekend in de volgende samples. We kunnen concluderen dat het `clingo`-systeem veel sneller werkt: voor bepaalde puzzels was ProbLog zelfs niet in staat een sample te genereren.

Hetzelfde geldt voor inferentie in ProbLog, het berekenen van de kansen van alle queries in ProbLog. Dit wordt bereikt met het commando `problog`, zonder extra opties. Bij puzzels met hogere dimensies, ingewikkeldere constraints of `evidence` met kleine kansen, sloeg ProbLog er nooit in om het model te evalueren. Voor puzzels met een kleine 100 000 mogelijkheden was `clingo` in staat om tot 30 miljoen samples weer te geven alvorens vast te lopen, ProbLog vond geen sample. `clingo` genereert ook consequent samples (elke X seconden), terwijl dit bij ProbLog zeer variabel was: dit viel op bij alle soorten puzzels. Waarom `clingo` consequenter puzzels kon genereren heeft te maken met de variatie: opeenvolgende gesamplede puzzles tonen bijna geen variatie, zoals besproken in volgende paragraaf.

5.2 Variatie

Het experiment om de variatie te bepalen verliep als volgt: na de criteria, die besproken zijn in paragraaf 4.3, per puzzel opgesteld te hebben, zijn er 20 puzzels gegenereerd. Er wordt vervolgens een score op 100 berekend van elke mogelijke combinatie van puzzels (puzzel 1 vs. puzzel 2, puzzel 1 vs. puzzel 3 ... puzzel 2 vs. puzzel 3 ... puzzel $(n-1)$ vs. puzzel n). Dit leidt tot $\frac{20!}{2!18!} = 190$ vergelijkingen. Hoe hoger de score, hoe sterker de variatie.

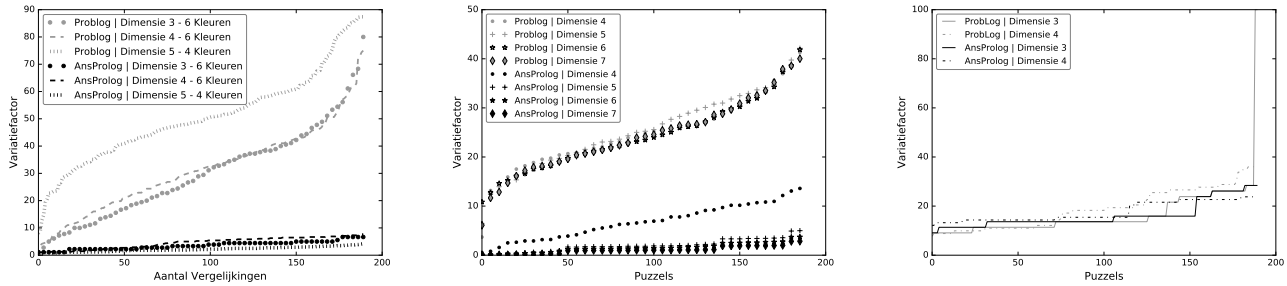
In figuur 2a wordt de variatie tussen de verschillende *chromatic mazes* van verschillende dimensies afgebeeld. De bekomen resultaten zijn gesorteerd van laag naar hoog om een beter overzicht te verkrijgen. Het is duidelijk dat ProbLog meer variërende puzzels genereert. Zelfs als de `--rand-freq`, de willekeurigheid waarmee keuzes van variabelen worden gemaakt in AnsProlog, op 100% stond en manueel de seed van de random generator werd aangepast, bezorgde dit nagenoeg identieke resultaten.

Een analoog resultaat kan worden afgeleid voor de *dungeons*, te zien in figuur 2b. Dit zijn samples van *dungeons* met verschillende dimensies, waarbij er één diamantje en één altaar is en waarbij de rest van de vakjes (willekeurig) ofwel een muur ofwel niets zijn.

In figuur 2c valt op dat zowel ProbLog als `clingo` een relatief lage variatie bereiken. Dit is te wijten aan het feit dat er een klein aantal mogelijkheden is voor de *perfect maze*. Er valt eenvoudig theoretisch te berekenen dat er bij dimensie drie 192 mogelijkheden zijn en bij dimensie vier zijn dat er iets meer dan 100 000. In vergelijking: bij de *chromatic maze* zijn er vanaf dimensie drie meer dan 50 miljoen mogelijkheden. Het is vrij evident dat de variatie lager ligt: er zijn veel minder puzzels om uit te kiezen.

Bij de eerste twee puzzels is het duidelijk dat de variatie bij ProbLog veel hoger ligt vergeleken met variatie bekomen aan de hand `clingo`. Dit is dankzij het feit dat bij elke sample opnieuw ProbLog effectief willekeurige keuzes maakt, terwijl `clingo` nooit echte 'randomness' bereikt.

Een kleine opmerking hierbij is, hoewel de variatie van ProbLog hoger ligt, de gegenereerde *chromatic mazes* bij-



(a) (b) (c)

Figuur 2: Variatie bij 20 gesamplede puzzels van verschillende dimensies van (a) *chromatic maze*, (b) *dungeons* en (c) *perfect maze*.

voorbeeld niet altijd ingewikkeld zijn, op vlak van oplosbaarheid. Ongeveer de helft van van de *chromatic maze* hebben een oplossing die meer dan vijf stappen vereiste om van start naar finish te geraken. De rest van de puzzels mag dan wel een hoge variatie hebben op vlak van welke vakjes welke kleuren hebben en waar de start en finish liggen, de afstand tussen start en finish kan heel vaak gelijk zijn bij lage dimensies.

5.3 Modellering

Op vlak van modellering werd al besproken dat de vertaling tussen AnsProlog en ProbLog relatief eenvoudig gaat. Duidelijk moeilijker was het opstellen van de integriteitsvoorwaarden, daar zijn twee methodes voor: ofwel `evidence` voorzien, ofwel het programma herschrijven zodat bepaalde constraints, niet kunnen voorkomen. Deze laatste aanpak was nodig bij bijvoorbeeld de *dungeon*. Als `evidence` wordt opgelegd van bijvoorbeeld maximaal één diamant en één altaar te hebben, werd de uitvoeringstijd onnoemelijk lang. Constraints als deze zijn eenvoudig om te vormen naar niet-evidence. Zonder `evidence` kan ProbLog tot dimensie vijf binnen een aanzienlijke tijd samples genereren, terwijl met `evidence` het systeem na vele (> 8) uren uiteindelijk vastloopt en geen uitkomst genereert.

Er waren echter bij de *dungeon* ook veel complexere, specifiekere voorwaarden: zoals de afstand van het minimale pad dat de speler moet afleggen. Uit alle geldige werelden hebben werelden met zo'n voorwaarden een enorm kleine kans op voorkomen. Werelden genereren met deze constraints voorgesteld als `evidence` is niet, of toch zeer moeilijk, haalbaar. Pogen deze `evidence` te herschrijven naar code die er voor zorgt dat deze werelden gewoonweg niet kunnen voorkomen, vereist meer werk dan de `evidence`-aanpak en is ook vrij specifiek per puzzel. ProbLog vertoont duidelijk een beperking op gebied van game content generation: moeilijke constraints met een kleine kans vereisen een specifieke (en soms complexe) aanpak. Het gebruik van `evidence` in zo'n gevallen is niet voordelig.

Daarnaast is het duidelijk, zoals in het voorbeeld van de *perfect maze* met een gemaximaliseerd aantal horizontale verbindingen, dat de randomness van ProbLog een voordeel kan bieden. Het biedt de mogelijkheid een bepaalde voorwaarde op een alternatieve manier te formuleren, die niet mogelijk is in het ASP model.

6 Conclusie en toekomstig werk

Uit de voorgaande analyse kunnen we een antwoord geven op onze vragen uit sectie 3.

- $\mathbf{A}_1$ Het ProbLog-systeem is geschikt om inhoud van spellen te genereren, als snelheid van ondergeschikt belang is.
- $\mathbf{A}_2$ Op vlak van snelheid komt ProbLog niet in de buurt van ASP, maar op gebied van variatie biedt ProbLog een enorm voordeel. De puzzels gegenereerd met ProbLog tonen veel meer variatie.
- $\mathbf{A}_3$ ProbLog biedt interessante voordelen dankzij zijn randomness in het definiëren van integriteitsvoorwaarden.

Naar de toekomst toe is het interessant om te onderzoeken hoe puzzels presteren als we het gebruik van `evidence` zoveel mogelijk vermijden. Dit vereist wel per constraint een erg gedetailleerde aanpak. Een andere mogelijkheid zou zijn om de sampler te verbeteren.

Dankwoord

Graag wil ik dr. Kimmig bedanken voor haar nuttige feedback en bijdrage bij het hele proces van deze bachelorproef.

Referenties

[De Raedt *et al.*, 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI*, pages 2462–2467, 2007.

[Dijkstra, 1959] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[Fierens *et al.*, 2015] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15:358–401, 5 2015.

[Gebser *et al.*, 2010] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A users guide to gringo, clasp, clingo, and iclingo, 2010.

[Gebser *et al.*, 2012] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice, 2012.

[Mendelson, 1987] Elliott Mendelson. *Introduction to Mathematical Logic; (4th Ed.)*. Wadsworth and Brooks/Cole Advanced Books & Software, Monterey, CA, USA, 1987.

[Nelson and Smith, 2015] Mark J. Nelson and Adam Smith. Chapter 8: Asp with applications to mazes and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[Smith and Bryson, 2015] Anthony J. Smith and Joanna J. Bryson. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*. 2015.

[Smith and Mateas, 2011] Adam M. Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):187–200, 2011.

[Smith *et al.*, 2011] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):201–215, 2011.

[Togelius *et al.*, 2011] Julian Togelius, Jim Whithead, and Rafael Bidarra. Guest editorial: Procedural content generation in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:169 – 171, 2011.

[Togelius *et al.*, 2015] Julian Togelius, Noor Shaker, and Mark J. Nelson. Introduction. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[Yannakakis and Togelius, 2011] Georgios N. Yannakakis and Julian Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2:147 – 161, 2011.

A Algoritmes

A.1 Kleurafstand

1: $\mathbf{bord_1}$: de vakjes van het eerste bord
2: $\mathbf{bord_2}$: de vakjes van het tweede bord
3: $n \leftarrow$ dimensie van $bord_1$ (of $bord_2$)
4: **for** elk vakje $v_1(x_1, y_1)$ in bord$_1$ **do**
5: $k_1 \leftarrow$ kleur van v_1
6: **for** elk vakje $v_2(x_2, y_2)$ in bord$_2$ **do**
7: $k_2 \leftarrow$ kleur van v_1
8: **if** $k_1 = k_2$ **then**
9: $C_{kleur} \leftarrow C_{kleur} + D(v_1, v_2)$
10: verwijder v_2 uit $bord_2$
11: **else**
12: $C_{kleur} \leftarrow C_{kleur} + ((2 * N) - 2)$
13: **end if**
14: **end for**
15: **end for**

A.2 Gelijkaardige verbindingen

1: $\mathbf{bord_1}$: de parent-relaties van het eerste bord
2: $\mathbf{bord_2}$: de parent-relaties van het tweede bord
3: $C_{verb} \leftarrow 0$
4: **for** elk parent $p_1(x_1, y_1, x_{v1}, y_{v1})$ in bord$_1$ **do**
5: **for** elk vakje $p_2(x_2, y_2, x_{v2}, y_{v2})$ in bord$_2$ **do**
6: **if** [($x_2 = x_1$ en $y_2 = y_1$ en $x_{v2} = x_{v1}$ en $y_{v2} = y_{v1}$)
7: ***of*** ($x_2 = x_1 + x_{v1}$ en $y_2 = y_1 + y_{v1}$ en $x_{v2} = -x_{v1}$ en $y_{v2} = -y_{v1}$)] **then**
8: $C_{verb} \leftarrow C_{verb} + 1$
9: **end if**
10: **end for**
11: **end for**

A.3 Afstand tot dichtsbijzijnde vakje

1: $\mathbf{bord_1}$: de vakjes van het eerste bord
2: $\mathbf{bord_2}$: de vakjes van het tweede bord
3: $C_{am} \leftarrow 0$
4: **for** elk vakje $v_1(x_1, y_1, t_1)$ in bord$_1$ **do**
5: $afstand \leftarrow (2 * N) - 2$
6: $teVerwijderen \leftarrow \varnothing$
7: **if** $t_1 = t_2$ **then**
8: **for** elk vakje $v_2(x_2, y_2, t_2)$in bord$_2$ **do**
9: **if** $D(v_1, v_2) < afstand$ **then**
10: $afstand \leftarrow D(v_1, v_2)$
11: $teVerwijderen = v_2$
12: **end if**
13: **end for**
14: $C_{am} \leftarrow afstand$
15: **end if**
16: **if** $teVerwijeren \neq \varnothing$ **then**
17: verwijder $teVerwijderen$ uit $bord_2$
18: **end if**
19: **end for**

B ProbLog

B.1 Chromatic Maze

```
% Genereer een lijst van getallen
list_of_integers(L,U,R) :-
 findall(M, between(L,U,M),R).
```

```
% Vorm alle mogelijke paren met
% een lijst van getallen
pairs(P) :-
 dim_list(L1),
 dim_list(L2),
 findall((A,B),
( member(A, L1),member(B, L2)),P).

% Definieer de bordgrootte
% en maximale tijd
size(5).
t_max(35)

% Definieer de dimensies
dim(D) :-
  size(S),
  between(1,S,D).

% Genereer een lijst van dimensies
% Gebruikt bij 'pairs'
dim_list(L) :-
  findall(N, dim(N), L).

% Kies een start en finish-vakje
% uniform uit de lijst met
% alle mogellijke vakjes
start(X,Y) :-
 start_and_finish((X,Y),(_,_)).
finish(X,Y) :-
 start_and_finish((_,_),(X,Y)).

start_and_finish((A,B),(C,D)) :-
  pairs(P),
  select_uniform(1,P,(A,B),R),
  select_uniform(2,R,(C,D),_).

time(T) :- t_max(M), between(0,M,T).

% Definieer wat aanliggende
% vakjes zijn
adjacent(X,Y,Nx,Y) :-
  dim(X),
  dim(Y),
  Nx is X+1.
adjacent(X,Y,Nx,Y) :-
  dim(X),
  dim(Y),
  Nx is X-1.
adjacent(X,Y,X,Ny) :-
  dim(X),
  dim(Y),
  Ny is Y+1.
adjacent(X,Y,X,Ny) :-
  dim(X),
  dim(Y),
  Ny is Y-1.

% Lijst van kleuren
colors([red, yellow, green,
cyan, blue, magenta]).

color(C) :-
  colors(L),
  member(C,L).
```

```
% Mogelijke kleurenovergangen
% in het kleurenwiel
next(red,yellow).
next(yellow,green).
next(green,cyan).
next(cyan,blue).
next(blue,magenta).
next(magenta,red).

% Toegelaten overgangen van vakjes
ok(C,C) :- color(C).
ok(C1,C2) :- next(C1,C2).
ok(C1,C2) :- next(C2,C1).

% Definieer welke mogelijke
% overgangen er zijn
passable(SX, SY, X, Y) :-
    adjacent(SX,SY,X,Y),
    tile(SX, SY, C1),
    tile(X, Y, C2),
    ok(C1,C2).

% Definieer wat een 'tile' is
tile(X,Y,C) :-
  dim(X),
  dim(Y),
  colors(Lc),
  select_uniform(id(X,Y),Lc, C, _).

% Definieer de positie van de speler
% op elk mogelijk ogenblik.
% Een speler mag niet blijven staan,
% en kan enkel op een vakje terechtkomen
% als de overgang geldig is.
player_at(0,X,Y) :- start(X,Y).
player_at(T, X, Y) :-
    time(T),
    T1 is T-1,
    player_at(T1, SX, SY),
    passable(SX, SY, X, Y),
    list_of_integers(0, T1, R),
    players_at(R, X, Y).

players_at([],_,_).
players_at([H|T], X, Y) :-
    \+ player_at(H, X, Y),
    players_at(T,X,Y).

% Definieer wanneer victory optreedt.
victory :- victory_at(T), time(T).

victory_at(T) :-
  time(T),
  finish(X,Y),
  player_at(T, X, Y).

% Predicaten nodig voor de visualisatie
tile_grid(S,S) :- size(S).
tile_char(X,Y,R) :-
  player_at(T,X,Y),
  T > 0,
  R is (T mod 10),
  not start(X,Y),
  not finish(X,Y).
```

```
tile_char(X, Y, s) :- start(X,Y).
tile_char(X, Y, f) :- finish(X,Y).
tile_color(X,Y,C) :- tile(X,Y,C).

% De zaken die worden opgevraagd
query(tile_grid(S,S)).
query(tile_char(X,Y,T))
 :- dim(X), dim(Y).
query(tile_color(X,Y,C)).

% De evidence: victory
% moet plaatsgevonden hebben
evidence(victory).
```

B.2 Perfect Maze

Deze versie is geoptimaliseerd in de zin dat de hoekjes apart worden behandeld: deze hebben maar twee mogelijke verbindingen.

```
% Definieer de bordtgrootte
width(3).

dim(D) :- width(W), between(1,W,D).

% Paarfunctie voorr
% een unieke identifier
identifier(X,Y,I) :-
 I is ((X+Y+X+Y+1)/2) + Y.

% De lijst van mogelijke verbindingen
list_of_directions([(0,-1),(1,0),
( -1,0),(0,1)]).

% Parent kiezen voor
% vakjes op een rand
edge_parent((X,Y),Xp,Yp,L) :-
  identifier(X,Y,I),
  list_of_directions(List),
  select_weighted(I,L,List,(Xp,Yp),_).

% Rechtsonder
parent(X,Y,Xp,Yp) :-
  width(Max),
  X == Max,
  Y == Max,
  identifier(X,Y,I),
  select_uniform(I,[(-1,0),(0,-1)],
(  Xp,Yp),_).

% Rechtsboven
parent(X,Y,Xp,Yp) :-
  width(Max),
  X == Max,
  Y == 1,
  identifier(X,Y,I),
  select_uniform(I,[(-1,0),(0,1)],
(  Xp,Yp),_).

% Linksonder
parent(X,Y,Xp,Yp) :-
  width(Max),
  X==1,
  Y==Max,
  identifier(X,Y,I),
  select_uniform(I, [(1,0),(0,-1)],
(  Xp,Yp),_).

% Alle 'gewone' vakjes:
% die niet op een rand liggen
% of geen hoekje zijn
parent(X,Y,Xp,Yp) :-
  width(Max),
  X \== 1, Y \== 1,
  X \== Max, Y \== Max,
  identifier(X,Y,I),
  list_of_directions(List),
  select_uniform(I,List,(Xp,Yp),_).

% Linker rand,
% behalve hoekje
parent(X,Y,Xp,Yp) :-
  width(Max),
  Y \== 1,
  Y \== Max,
  X == 1,
  edge_parent((X,Y),Xp,Yp,
  [1/3,1/3,0,1/3]).

% Rechter rand,
% behalve hoekjes
parent(X,Y,Xp,Yp) :-
  width(Max),
  Y \== Max,
  Y \== 1,
  X == Max,
  edge_parent((X,Y),Xp,Yp,
  [1/3,0,1/3,1/3]).

% Bovenste rand,
% behalve hoekjes
parent(X,Y,Xp,Yp) :-
  width(Max),
  X \== 1,
  X \== Max,
  Y == 1,
  edge_parent((X,Y),Xp,Yp,
  [0,1/3,1/3,1/3]).

% Onderste rand,
% behalve hoekjes
parent(X,Y,Xp,Yp) :-
  width(Max),
  X \== 1,
  X \== Max,
  Y == Max,
  edge_parent((X,Y),Xp,Yp,
  [1/3,1/3,1/3,0]).

% Definieer wanneer vakjes
% verbonden zijn
linked(1,1).
linked(X,Y) :-
 parent(X,Y,DX,DY),
 dim(X),
 dim(Y),
 T is X+DX, S is Y+DY,
 linked(T,S).
```

```
% De opgevraagde zaken
query(parent(X,Y,Xp,Yp)) :-
 dim(X), dim(Y),(X,Y) \== (1,1).

% De evidence: alle vakjes
% moeten verbonden zijn met (1,1)
evidence(linked(X,Y)) :-
 dim(X),dim(Y),(X,Y) \== (1,1).
```

B.3 Dungeon

Dit is een simpele versie van de *dungeon*: de enige constraints zijn dat er maximaal één diamant en één altaar is. De rest van de vakjes zijn een muur of niets

Met evidence

```
% Definieer de breedte,
% en mogelijke dimensies
width(3).
dim(D) :-
 width(W), between(1,W,D).

% Definieer paren van
% co\"ordinaten
pairs(P) :-
 dim_list(L1),
 dim_list(L2),
 findall((A,B),
( member(A, L1),member(B, L2)),P).

% Lijst van dimensies
dim_list(L) :-
 findall(N, dim(N), L).

% Mogelijke sprites
sprites([gem,altar,wall, none]).

% Unieke identifier
identifier(X,Y,I) :-
 I is (X+Y+X+Y+1)/2 + Y.

% Definieer de tile
tile((X,Y)) :- dim(X), dim(Y).

% Bepaal waar start en
% finish liggen
start((1,1)).
finish((W,W)) :- width(W).

% Haal voor elke tile
% een sprite op
tiles([],[]).
tiles([(X,Y)|T], [F|P]) :-
  sprite((X,Y),F),
  tiles(T, P).

% Definieer de sprites
sprite((X,Y),(X,Y,R)) :-
sprites(List),
 identifier(X,Y,I),
 select_uniform(I, List, R,_).

% Definieer dat er slechts
% \'e\'en gem en \'e\'en
% altaar mag zijn
only_one :-
 findall((X,Y),
( tile((X,Y)),sprite((X,Y),
( X,Y,gem))),
 R),
 length(R,1),
 findall((Z,Q),
( tile((Z,Q)),
 sprite((Z,Q),
( Z,Q,altar))),
 T),
 length(T,1).

% Genereer alle tiles
query(tiles(P,R)) :- pairs(P).
% Evidence
evidence(only_one).
```

Zonder evidence

```
width(10).
minwall(1).
dim(D) :- width(W), between(1,W,D).

pairs(P) :-
 dim_list(L1),
 dim_list(L2),
 findall((A,B),
( member(A, L1),
 member(B, L2)),
 P).

dim_list(L) :-
  findall(N, dim(N), L).

regular_sprites([none,wall]).
special_sprites([gem,altar]).

identifier(X,Y,I) :-
 I is (X+Y+X+Y+1)/2 + Y.

tile((X,Y)) :- dim(X), dim(Y).

start((1,1)).
finish((W,W)) :- width(W).

% Haal eerst de speciale tiles op
% (gem en altaar), gebruik
% dan de overige tiles
% om de de muren te plaatsen
tiles(H, (A, B), (C,D), Rest) :-
  get_special_gems(H, R, (A,B), (C, D)),
  get_regular_gems(R,Rest).

get_special_gems(H,R, (A, B), (C,D)) :-
  select_uniform(1, H,(A,B), R1),
  select_uniform(1, R1, (C, D),R),
  not((A,B)==(C, D)).

get_regular_gems([],[]).
get_regular_gems([(X,Y)|T], [F|Tail]) :-
  sprite((X,Y),F),
  get_regular_gems(T,Tail).
```

```
sprite((X,Y),(X,Y,R)) :-
 regular_sprites(List),
 identifier(X,Y,I),
 select_uniform(I, List, R,_).

get_solution(Gem, Altar, Rest) :-
  pairs(P),
  tiles(P, Altar, Gem, Rest).

gem(G) :-
  get_solution(G,_,_).
altar(A) :-
  get_solution(_,A,_).
rest(R) :-
  get_solution(_,_,D),
  member(R,D).

query(gem(G)).
query(altar(A)).
query(rest(R)).
```