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 procedurale 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 op deze vlakken ProbLog 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 het ontwikkelen van eender welk soort spel en is ook al een gevestigd onderzoeksdomein. Het gebruik van Answer Set Programming (ASP) voor het genereren van spelinhoud is echter nog nieuw. 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, geef ik in deze introductie relevante achtergrondinformatie.

1.1 Generatie van spelinhoud

Het proceduraal 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 meer rekwisieten (wapens, kisten, muntjes ...), maar ook 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.

1.2 AnsProlog

Zoals vermeld is Answer Set Programming (ASP) reeds goed bestudeerd op het gebied van game content generation [Nelson and Smith, 2015], [Togelius *et al.*, 2011], [Smith *et al.*, 2011]. ASP is een vorm van declaratief programmeren gericht op moeilijke (NP-harde) zoekproblemen. Het is vooral bruikbaar in kennis-intensieve toepassingen, maar kan ook makkelijk gebruikt worden voor zijn generatieve mogelijkheden. Een ASP-programma is dus een programma dat kennis beschrijft, en niet de uitvoering van het programma. Er wordt beschreven *wat* moet berekend worden en niet *hoe*. Op het gebied van generatie van spelinhoud kan in zulke programma's dus beschreven worden aan welke voorwaarden de spelinhoud moet voldoen. In het programma wordt het ontwerp van de inhoud die dient te worden gegenereerd beschreven om nadien een bepaalde answer set, een mogelijke beschrijving van een wereld die voldoet aan het ontwerp, te berekenen.

De ASP-programma's beschouwd in deze paper, en waarop deze paper verder bouwt, zijn geschreven in AnsProlog: een syntax met regels vergelijkbaar aan Prolog, maar tegelijk ook verschillend van Prolog door onder meer de mogelijkheid om "keuze-regels" of "niet deterministische regels" te definiëren.Een voorbeed hiervan is `{s,t} :- p`, wat zo veel wil zeggen als: "als `p` in het model voorkomt, kies dan willekeurig welk van de atomen `s,t` (`s`, of `t`, of `s` en `t`, of geen van beide) worden toegevoegd". Op deze keuze-regels kunnen ook numerieke onder- en bovengrenzen worden opgelegd: zo zegt `1{s,t}1 :- p` dat er exact één zal worden toegevoegd. Een andere uitbreiding op de syntax ten opzichte van Prolog is de mogelijkheid om voorwaarden of constraints te bepalen die altijd waar of net nooit waar moeten zijn. 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 (de bete-

kenis hiervan is verder niet relevant). Elke oplossing, elke "answer set", voldoet aan deze regels.

1.3 ProbLog

ProbLog is een probabilistische uitbreiding op Prolog [De Raedt *et al.*, 2007]. Uniek aan ProbLog is dat voor elke regel kan gespecificeerd worden met welke kans deze regel optreedt, een voorbeeld hiervan is het volgende: `p::friend(A,B).`. Deze regel zegt dat `A` bevriend is met `B` met een kans `p`.

In een ProbLog-programma geef je ook bepaalde *queries*, 'vragen' op. Bekijk daarvoor het onderstaande voorbeeld .

```
0.5 :: color(green).
0.5 :: color(red).

query(color(C)).
```

ProbLog biedt de mogelijkheid een programma te sampelen. Bij het sampelen genereert ProbLog een mogelijke toekenning aan de opgegeven queries. Als het bovenstaande programma wordt gesampeld, zijn er drie mogelijke oplossingen:

- `color(red).`
- `color(green).`
- `color(green).` en `color(red).`
- lege oplossing

Het sampelen geeft dus één of meerdere mogelijkheden uit die subprogramma's (afhankelijk van hoeveel resultaten worden gesampelt).

Verder is het in een ProbLog-programma ook mogelijk om `evidence` te voorzien. Elke sample gegenereerd met het programma zal voldoen aan deze gegeven `evidence`. Zo kan het volgende schrijven geschreven worden: `evidence(friend(C,D)).` wat betekent dat persoon `C` altijd bevriend moet zijn met persoon `D`, in elke mogelijke oplossing van het programma.

ProbLog verschilt van ASP doordat het niet-determinisme in ASP niet hetzelfde is als een kansverdeling, waarvan wel gebruik wordt gemaakt in ProbLog. ProbLog hanteert dus effectieve willekeurigheid, die in ASP niet van toepassing is. Deze paper onderzoekt dus de toepasbaarheid en de mogelijkheden, maar tegelijk ook de limieten en nadelen van de willekeurigheid van ProbLog op het gebied van game content generation.

2 Probleemstelling

De probleemstelling in dit onderzoek is als volgt geformuleerd: is ProbLog toepasbaar op het vlak van 'game content generation'? Is het systeeem daar voldoende geschikt voor? Op welke manieren kan de probabilistische kant van ProbLog een voordeel bieden bij het genereren van spelinhoud? Wat zijn de voor- en nadelen bij het gebruik van ProbLog?

In deze paper wordt een vergelijking van ProbLog en ASP gemaakt aan de hand van drie soorten puzzels die worden onderlegd aan een aantal criteria. De hypothese is dat de probabilistische kant van ProbLog zeker een voordeel kan bieden bij het genereren van spelinhoud maar dat het systeem, zoals het nu is, niet snel genoeg is om een groot praktisch voordeel te bieden.

3 Aanpak

In dit deel van het onderzoek zal ik toelichten hoe ik mijn experimenten heb opgezet en uitgevoerd. De resultaten van deze experimenten worden dan besproken in de volgende paragraaf.

In mijn onderzoek heb ik voor het oplossen 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].

3.1 Puzzels

Voor dit onderzoek heb ik gebruik gemaakt van drie puzzels. Sommige experimenten gebruiken echter een vereenvoudigde versie van de desbetreffende puzzel indien dit duidelijker was voor het experiment of wanneer het uitvoeren van programma's met de meer complexere voorwaarden te veel tijd in beslag zou nemen.

De drie puzzels waren de *chromatic maze*, *perfect maze* en de *dungeon*. Ik zal een korte beschrijving geven van deze puzzels.

Chromatic Maze

De *chromatic maze* (cfr. [Smith and Mateas, 2011], figuur 1a) is een altijd vierkante puzzel en is een soort doolhof waarbij elk vakje een bepaalde kleur uit een kleurenwiel heeft. Om de puzzel op te lossen moet een bepaald pad van start naar finish over de vakjes gevonden worden. Een overgang van het ene vakje naar een ander vakje is enkel toegelaten wanneer de kleurenovergang die optreedt zich in het gegeven kleurenwiel bevindt (bijvoorbeeld: vakje 1 is rood en vakje 2 is groen, dan moet om de overgang van 1 naar 2 geldig te maken rood naast groen in het kleurenwiel staan).

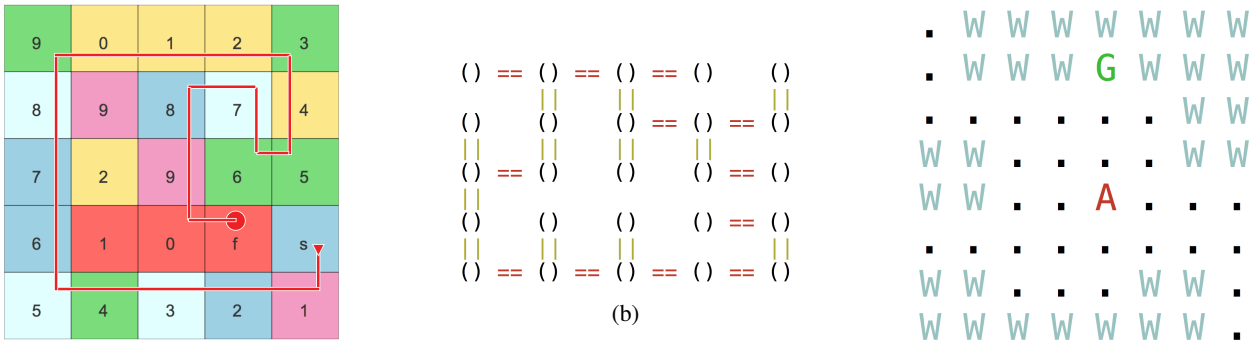
Perfect Maze

De *perfect maze* (cfr. [Nelson and Smith, 2015], figuur 1b) is een altijd vierkante puzzel en is simpelweg een perfect doolhof: vanuit elke positie kan een pad gemaakt worden naar het vakje linksboven `(1,1)`. Hierdoor zal de puzzel ook altijd een pad bevatten van het vakje linksboven tot het vakje rechtsonder.

Dungeons

Een *dungeon* is zoals de naam zegt 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, één diamantje en een bepaald aantal muren. De rest van de vakjes hebben geen bepaalde eigenschap. De gedachtengang 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. Aan dit soort werelden kan een heel aantal contstraints worden toegekend: de verhouding

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



(a) (b) (c)

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

muren ten opzichte van het aantal lege vakjes, de afstand tussen de diamant en het altaar ... Tenzij anders vermeld, gebruik ik voor al mijn experimenten een *dungeon* 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.

3.2 Vertaling

Het eerste deel van mijn onderzoek bestond er in de AnsProlog-programma's die de eerdergenoemde puzzels beschreven, om te vormen tot ProbLog-programma's. Uiteraard moest er rekening moest gehouden worden met de constraints die ook aanwezig waren in de AnsProlog-programma's. Aangezien AnsProlog en ProbLog beiden varianten zijn op Prolog, was dit over het algemeen niet moeilijk. Ze hebben beide functionaliteiten verschillend van Prolog, maar ook van elkaar. Zo biedt AnsProlog de mogelijkheid tot het invoeren van niet-determinisitische regels alsook het neerschrijven van bepaalde voorwaarden die altijd waar of net nooit waar mogen zijn. Je vindt de vertalingen van de puzzels in ProbLog, met commentaar als leidraad, in de appenidces.

Het eerste aspect van de vertaling was de omzetting van de niet-deterministische kansen 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 `t`, één kleur 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 zo veel mogelijk horizontale, en dus zo wenig mogelijk verticale, verbindingen te hebben. Het doolhof vertrekt dan van linksboven en maakt een soort slangbeweging tot rechtsonder.

In AnsProlog kan dit gemodelleerd worden door eerst te definiëren, aan de hand van een regel, wat een verticale verbinding is en nadien 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. Zo is er in ProbLog `select_weighted` in de plaats van `select_uniform`. De horizontale verbindingen krijgen een kans van beide 45% en de verbinding boven en onder een kans van nog elk 2.5%. De gegenereerde puzzels zijn nagenoeg altijd puzzels met een maximaal aantal horizontale verbindingen.

Verder was er in AnsProlog ook de mogelijkheid om bepaalde voorwaarden op te leggen die nooit of net altijd moesten voorkomen. Een voorbeeld hiervan is de voorwaarde die zegt dat er altijd `victory` moet zijn en dat deze moet bereikt worden in minder dan `max_solution` en meer dan `min_solution` stappen.

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

In ProbLog kunnen we een analoge constraint opleggen aan de hand van `evidence`, waarmee we willen zeggen dat elke toekenning moet voldoen aan `victory`.

```
victory :- victory_at(T), time(T).

victory_at(T) :-
  max_sol(Max),
  min_sol(Min),
  time(T),
  finish(X,Y),
  player_at(T, X, Y).

evidence(victory).
```

Een alternatieve aanpak voor deze evidence is om er voor te zorgen dat werelden met deze bepaalde constraints gewoon nooit kunnen voorkomen. Zo was er bij de *dungeon* de voorwaarde dat er maximaal één diamant en één altaar konden zijn. Dit kan in ProbLog voorgesteld worden aan de hand van `evidence`, maar een andere 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 twee alternatieve aanpakken, en welke nu een voordeel biedt op vlak van modellering worden verder nog besproken.

3.3 Analyse

Het meest belangrijke deel van mijn onderzoek was uiteraard de analyse: effectief gaan bestuderen hoe het ProbLog-systeem presteert. Het grootste deel van de analyse was op variatie. De snelheid van de systemen wordt ook onderzocht en tenslotte komt ook de modelleringscomplexiteit aan bod, waar ook wordt gesproken over de effectieve voordelen die ProbLog biedt (hoofdzakelijk waar de probabiliteit van ProbLog een positief effect kan hebben).

Snelheid

Een eerste criterium dat ik heb behandeld in mijn onderzoek is de snelheid. Ik heb aan de hand van verschillende puzzels gekeken hoe lang de geschreven programma's met bijhorende software (ProbLog en `clingo`) er over deden om een aantal samples, oplossingen die voldoen aan het programma, te genereren. Ook heb ik de vergelijking gemaakt tussen de tijd nodig bij het berekenen van alle mogelijke kansen bij ProbLog en de tijd nodig om alle mogelijke samples op te lijsten met `clingo`. Het is uiteraard in een toepassing wenselijk, zelfs noodzakelijk, om een zo snel mogelijk systeem te hebben dat puzzels in real-time kan genereren. Spelletjes waarbij de speler lang moet wachten alvorens het spel kan beginnen, zijn gedoemd om te falen.

Om de snelheid te meten heb ik het ProbLog systeem tegenover het `clingo`-systeem geplaatst. Ik heb met het `sample`-commando van ProbLog telkens 100 samples gegenereerd, dat is mogelijk met het argument `-N`. Voor elke puzzel zag het commando er dus als volgt uit: `problog sample -N 100 puzzel`. Eén sample is een mogelijke 'wereld' die voldoet aan het programma, en dus de constraints die werden opgegeven.

Voor `clingo` heb ik eveneens modellen laten genereren met als argumenten `--models=100` en `--rand-freq=1` welke respectievelijk er voor zorgen dat ook `clingo` 100 modellen genereert en 100 procent willekeurige beslissingen neemt. Het commando voor de AnsProlog puzzels zag er dan als volgt uit: `clingo --models=100 --rand-freq=1 puzzel`.

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

Variatie

Het tweede criterium waarop dit onderzoek zich baseert is de variatie tussen de gegenereerde puzzels. Dit is bij puzzels uiteraard een zeer belangrijk criterium. Puzzels die gegenereerd worden en maar minimale verschillen vertonen, zijn uiteraard niet wensbaar in een systeem dat levels van spellen moet construeren. Op het vlak van game content generation is het beter naar zo veel mogelijk variatie toe te streven. Zo krijgt de speler die het spel speelt niet het gevoel een bepaalde palttegrond of lay-out al eens tegen gekomen te zijn.

De variatie kwantitatief voorstellen vereistte voor elke puzzel een aparte aanpak om de variatie te kunnen bepalen, aangezien de constraints per puzzel verschillend zijn.

Na de criteria, die hieronder worden besproken, opgesteld te hebben, heb ik 20 puzzels gegenereerd om vervolgens de score te berekenen 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 meer variatie de puzzels vertonen.

- **Chromatic Maze**
  Voor de *chromatic maze* heb ik 4 criteria geëvalueerd. De eerste drie samen tellen voor 50% mee en het laatste criterium telt ook nog eens voor 50% mee.
  - *Afstand tussen start-vakjes.*
    Ten eerste heb ik gekeken wat de Manhattan-afstand was tussen de vakjes die dienden als start bij beide puzzels. Dit heb ik dan herschaald naar een getal tussen 0 en 1 door te delen door de breedte van de puzzel. Zo worden relatieve waarden bekomen die vergelijkbaar zijn voor elke soort puzzel.
  - *Afstand tussen finish-vakjes.*
    De afstand tussen de finish-vakjes is op een analoge manier, met Manhattan-afstand, berekend als de start-vakjes.
  - *Afstand tussen de kleuren.*
    Verder heb ik voor elk vakje in de eerste puzzel gezocht naar het dichtsbijzijnde vakje met dezelfde kleur in de tweede puzzel. Als er geen vakje meer is dat dezelfde kleur heeft (of stel bijvoorbeeld in het slechtste geval dat alle kleuren tussen de twee puzzels verschillend zijn), zet ik de waarde op de maximale afstand die in die puzzel mogelijk is $((2 * n) - 2$, met n de breedte). Dit wordt eveneens herschaald naar een getal tussen 0 en 1 door te delen door (aantal vakjes $*$ maximale afstand mogelijk) $= n^2 * ((2 * n) - 2)$, met n de breedte.
  - *Afstand tussen start en finish*
    Tenslotte heb ik 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 aangezien de puzzels die worden gegenereerd altijd oplosbaar zijn: als de afstand tussen het begin- en eindpunt dan hetzelfde is, zijn de puzzels erg analoog. Indien de variatie wordt berekend met enkel de eerste drie criteria kan deze al relatief hoog liggen. Dit laatste criterium is echter doorslaggevend als maat voor complexiteit. Dit ene criterium krijgt 50% van het totale gewicht.
- Perfect Maze
  De *perfect maze* is getest aan de hand van twee criteria,

namelijk het aantal overeenkomstige verbindingen en de lengte van het kortste pad dat mogelijk is met die verbindingen. Het aantal overeenkomstige verbindingen krijgt een gewicht van 75% en de afstand van het kortste pad een gewicht van 25%, aangezien de kans dat het korste pad hetzelfde is, relatief groot is.

  - *Aantal gelijkaardige verbindingen*
    In de *perfect maze* wordt een verbinding weergeven als volgt: `parent(3,2,-1,0)` (zie ook appendix A.2), dit wil zeggen dat het vakje (3,2) verbonden is met (2,2). Uiteraard is dat hetzelfde als schrijven `parent(2,2,1,0)`. Op deze twee manieren wordt dus getest of een bepaald vakje in de ene puzzel verbonden is met een bepaald vakje in de andere puzzel. Voor elke overeenkomstige verbinding wordt bij de score één opgeteld. Dit wordt nadien ook herschaald naar 75% van de totale score.
  - *Verschil in kortste pad*
    Voor elk van de twee puzzels wordt aan de hand van de bestaande verbindingen het korste pad berekend, gebruikmakend van het algoritme van Dijkstra. Het verschil in deze twee paden wordt beschouwd ten opzichten van het kortst mogelijke pad ($2 * n + 3 * (n - 4)$, met n de breedte) en nadien herschaald naar 25% van de totale score.

- Dungeon
  De *dungeon* is getest aan de hand van 7 criteria. Elk van deze criteria telt even zwaar door en levert, na herschaling, opnieuw een punt op 100.

  - *Afstand tussen diamanten*
    Als eerste wordt de Manhattanafstand berekend tussen de diamanten van beide puzzels. Dit wordt dan herschaald naar een score tussen 0 en 1 door te delen door de maximaal mogelijke afstand.
  - *Afstand tussen altaars*
    Het tweede criterium is de afstand tussen de altaars en is volledig analoog aan de diamanten hierboven.
  - *Afstand tussen diamant en altaar*
    Het derde criterium kan beschouwd worden als een maat voor de moeilijkheid van de puzzel. Per puzzel wordt de afstand berekend tussen de locatie van de diamant en de locatie van het altaar. Er wordt zo berekend hoeveel stappen de speler minimaal moet zetten om de uitgang te kunnen openen. Het verschil in deze afstanden tussen beide puzzels wordt opnieuw herschaald door te delen door de maximaal mogelijke afstand.
  - *Verschil in aantal muren*
    Als vierde criterium wordt voor beide puzzels het aantal muren berekend en daarvan het verschil genomen. Dit wordt herschaald door te delen door het aantal vakjes min 2 (voor de diamant en het altaar).
  - *Verschil in lege vakjes*
    Analoog aan hierboven wordt als vijfde criterium het verschil in aantal vakjes berekend die geen muur, diamant of altaar zijn.
  - *Afstand tot dichtstbijzijnde muur*
    Als voorlaatste criterium wordt voor elke muur de afstand berekend tot de dichtstbijzijnde muur in de tweede puzzel. Als er geen muur meer is die kan dienen als overeenkomstige muur, wordt de afstand maximaal gezet. Dit gebeurt op dezelfde manier als bij de *chromatic maze* hierboven.
  - *Afstand tot dichtsbijzijnde lege vakje*
    Analoog aan hierboven wordt ook voor elk leeg vakje in de eerste puzzel een zo dicht mogelijk leeg vakje gezocht in de tweede puzzel.

3.4 Modellering

Verder heb ik ook andere, kleinere criteria in verband met de effectieve modellering onderzocht. Ik heb onder meer een analyse gemaakt naar waar nu effectief de probabiliteit van ProbLog aan bod komt en waar die eventueel extra mogelijkheden biedt. Eveneens heb ik onderzocht hoe goed de `evidence` van ProbLog presteert op gebied van game content generation en of er eventueel eenvoudigere alternatieven voor deze evidence bestaan bij de puzzels.

Het voorstellen van de niet-uniforme keuzeregels uit AnsProlog, is mogelijk aan de hand van het `select_uniform` en het `select_weighted` predicaat. Dit laatste biedt een grotere variatie aan mogelijkheden aan aangezien het mogelijk is mogelijk is bepaalde elementen of aspecten aan je spelinhoud met een grotere kans te laten voorkomen. Dit is onder meer te zien bij de puzzel van de perfect maze (appendix A.2). Deze puzzel is herschreven zodat overbodige assignments, die toch niet kunnen voorkomen, niet gegeneerd worden. Zo zie je dat voor tegels die op de rand liggen slechts wordt gekozen uit 3 van de 4 mogelijke verbindingen. Dit gebeurt door met `select_weighted` de mogelijke verbindingen een kans van $\frac{1}{3}$ te geven de verbinding die niet mogelijk is een kans van 0 te geven. Dit zou ook gemodelleerd kunnen worden door uniform te selecteren uit een lijst waarin enkel de drie mogelijkheden voorkomen, maar de aanpak met `select_weighted` is iets eenvoudiger.

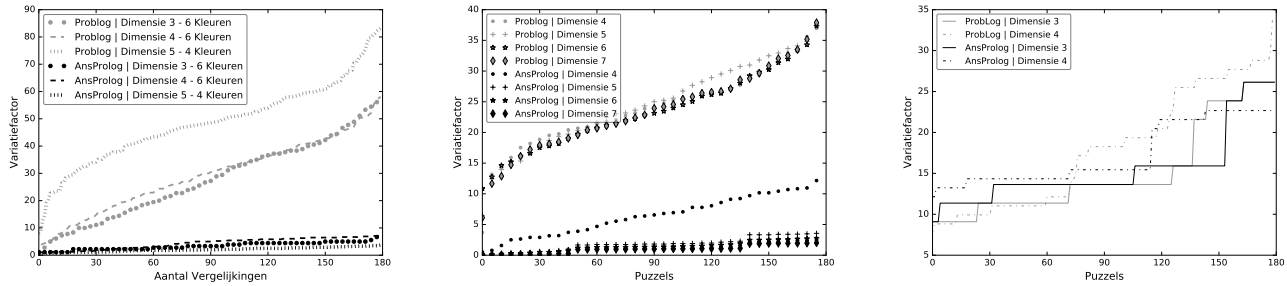
Verder is het in ProbLog dus ook mogelijk om de integriteitsvoorwaarden (de voorwaarden die altijd waar of net nooit waar mogen zijn) voor te stellen aan de hand van evidence. Het is eveneens duidelijk geworden dat dit ook op een alternatieve manier gaat, door je programm zo te schrijven dat modellen die niet voldoen aan je constraints gewoon niet te laten voorkomen. Een voorbeeld hier van vind je in appendix A.3, bij het model van de dungeon met en zonder evidence.

4 Resultaten

In deze paragraaf analyseer ik de resultaten bekomen uit de experimenten besproken in de vorige paragraaf.

4.1 Snelheid

De analyse op gebied van snelheid is zoals verwacht: 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*. ProbLog genereert dus als het ware een sample, controleert daarna of



(a) (b) (c)

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

deze voldoet aan de opgegeven voorwaarden, en start volledig opnieuw als dit niet zo is. Als er *evidence* wordt opgegeven met een heel kleine kans, is de kans dat het gegenereerde sample daaraan voldoet uiteraard ook zeer klein.

`clingo` daarentegen leert uit gemaakte 'fouten': wordt er ergens een variabele toegekend die tot een mislukking leidt, dan zal deze variabele nooit meer worden toegekend in de volgende samples. We kunnen dus concluderen dat het `clingo`-systeem veel sneller werkt: voor bepaalde puzzels was ProbLog zelfs niet in staat een sample te genereren terwijl `clingo` dit nog wel kon in een acceptabele tijd.

Hetzelfde geldt voor inferentie in ProbLog, het berekenen van de kansen van alle queries in ProbLog (dat gebeurt 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. Zelfs voor puzzels met een kleine 100 000 mogelijkheden, kwam het niet tot een resultaat. `clingo` daarentegen was in staat om tot 30 miljoen samples weer te geven alvorens vast te lopen.

4.2 Variatie

Voor de variatie verliep het experiment als volgt: voor elk van de puzzels zijn telkens 20 samples gegenereerd voor verschillende dimensies. Nadien heb ik deze puzzels allemaal (elke combinatie) met elkaar vergeleken aan de hand van de criteria besproken in paragraaf 3.

In figuur 2a zien we de variatie tussen de verschillende *chromatic mazes* van verschillende dimensies. Ik heb samples gegenereerd voor dimensie drie en vier met zes kleuren en ook voor dimensie vijf met vier kleuren (waardoor het aantal mogelijkheden al daalt). De bekomen resultaten heb ik 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 worden gemaakt in AnsProlog, op 100% stond en ik manueel de seed van de random generator aanpastte, bezorgde dit nagenoeg dezelfde 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 maar net 100 000. In vergelijking: bij de chromatic maze zijn er vanaf dimensie drie al meer dan 50 miljoen mogelijkheden. Het is dus vrij evident dat de variatie lager ligt: er zijn veel minder puzzels om uit te kiezen.

Bij de eerste twee puzzels is het wel heel duidelijk dat de variatie bij ProbLog veel hoger ligt dan bij het gebruik van `clingo`. Dit is dankzij het feit dat bij elke sample opnieuw ProbLog effectief willekeurige keuzes maakt, terwijl `clingo` nooit echte 'randomness' bereikt.

Een kleine bemerking hierbij is, hoewel de variatie van ProbLog hoger ligt, de gegenereerde *chromatic mazes* bijvoorbeeld niet altijd ingewikkeld zijn. Ongeveer de helft van van de *chromatic maze* hebben een oplossing die meer dan vijf stappen vereistte.

4.3 Modellering

Op vlak van modellering heb ik reeds besproken dat de vertaling tussen AnsProlog en ProbLog over het algemeen relatief eenvoudig gaat. Iets moeilijker was het opstellen van constraints die altijd of net nooit moesten waar zijn, daar zijn twee methodes voor: ofwel simpelweg `evidence` voorzien, ofwel het programma herschrijven zodat bepaalde constraints, die normaal worden voorzien aan de hand van evidence, simpelweg niet kunnen voorkomen. Deze laatste aanpak was nodig bij bijvoorbeeld de *dungeon*. Als de `evidence` wordt opgelegd (met dus het echte `evidence`-sleutelwoord) 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.

Er waren echter bij de dungeon ook veel complexere voorwaarden. Eén voorbeeld hiervan is er voor zorgen dat er altijd een pad bestaat tussen de diamant en het altaar, en dat het altaar nooit omsloten is door muren en dat het gem net wel altijd door minimaal 3 muren omsloten is ... Uit alle mogelijke werelden hebben werelden met zo'n voorwaarden een enorm kleine kans op voorkomen. Deze constraints voorstel-

len als `evidence` is niet, of toch zeer moeilijk, haalbaar. De uitvoeringstijd gaat hierdoor enorm de hoogte in. Zonder evidence kan ProbLog nog voor vele dimensies binnen een aanzienlijke tijd samples genereren, terwijl zonder evidence het systeem na vele uren uiteindelijk vastloopt en geen uitkomst genereert. Proberen deze evidence te herschrijven naar code die er voor zorgt dat deze werelden gewoonweg niet kunnen voorkomen, is erg complex. ProbLog vertoont dus duidelijk een limitatie op gebied van game content generation: moeilijke constraints met een kleine kans vereisen een specifieke aanpak en zijn moeilijk te implementeren.

Daarnaast is het wel 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. Het kan een beetje beschouwd worden als 'vals spelen', maar leidt wel tot hetzelfde resultaat. Dit is dan weer een aspect waar ProbLog een voordeel biedt op het vlak van game content generation.

5 Conclusie en toekomstig werk

Uit de voorgaande analyse kunnen we besluiten dan ProbLog mogelijkheden biedt op het gebied van game content generation. Indien variatie een belangrijke factor is wint het ProbLog systeem duidelijk van het `clingo`-systeem. Op vlak van snelheid daarentege,, maakt ProbLog geen schijn van kans: de sampler is te naïef en te traag om snel oplossingen van puzzels te kunnen geven.

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 mogelijheid zou zijn om de sampler te verbeteren.

De code om de puzzels zelf te draaien alsook de code die werd gebruikt om de variatie en snelheid te berekenen, is te vinden op `https://github.com/RobinHaveneers/bachelorpaper-probabilistic-programming`.

Dankwoord

Graag wil ik dr. Angelika 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.

[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.

[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 ProbLog

A.1 Chromatic Maze

```
%% Hulpfuncties

% 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).
```

A.2 Perfect Maze

Deze versie is geoptimaliseerd in de zin dat de hoekjes apart worden behandeld: deze hebben namelijk minder 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).
```

A.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)).
```