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, een probabilistische declaratieve programmeertaal ontwikkeld aan de KU Leuven, op het gebied van procedurale creatie van spelinhoud. Het maakt de vergelijking met AnsProlog, een andere declaratieve programmeertaal, met als criteria: snelheid en variatie. Er wordt zo onderzocht waar ProbLog voordelen en nadelen heeft bij het proceduraal genereren van spelinhoud. De resultaten van dit onderzoek zijn bereikt aan de hand van de analyse van 3 soorten puzzels. Uit de resultaten van dit onderzoek kan worden afgeleid dat op gebied van snelheid ProbLog het niet haalt van AnsProlog. ProbLog bereikt echter wel een veel grotere variatie in gegeneerde oplossingen die voldoen aan het gevraagde model. We besluiten dat ProbLog in de praktijk mogelijkheden biedt als variatie de belangrijkste factor is en snelheid meer van ondergeschikt belang is.

1 Introductie

Generatie van spelinhoud 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 dit gegeven is echter nog nieuw. Vanaf ongeveer 2011 zijn onderzoekers begonnen met de toegevoegde waarde te onderzoeken van ASP op gebied van *game content generation*. Dit valt te lezen in onder andere [Yannakakis and Togelius, 2011] en [Smith and Bryson, 2015]. Alvorens dieper in te gaan op het onderzoek, wordt in deze introductie relevante achtergrondinformatie gegeven.

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 inhoud van spellen kan genereren.

Onder 'inhoud' verstaan we onder meer rekwisieten (wapens, kisten, muntjes ...) maar ook plattegronden en speelwerelden (kerker, platform, onder water ...).

Zaken zoals het spelmechanisme zelf en de AI achter niet-speelbare personages vallen niet onder deze definitie van spelinhoud.

1.2 AnsProlog

Zoals vermeld is ASP reeds goed bestudeerd op het gebied van game content generation. ASP, of "Answer Set Programming", is een vorm van declaratief programmeren gericht op moeilijke (NP-harde) zoekproblemen. Het is vooral bruikbaar in kennis-intensieve toepassingen.

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 ook verschillend van Prolog door onder meer de mogelijkheid om "keuze-regels" of "niet deterministische regels" te definiëren. Een voorbeeld 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` worden toegevoegd". Op deze kansregels kunnen ook numerieke grenzen 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 niet 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 betekenis hiervan is verder niet relevant). Elke oplossing, elke "answer set", voldoet aan deze regels.

In ASP worden zoekproblemen gereduceerd tot modellen waar nadien een solver wordt op losgelaten om zo tot een resultaat te komen. In tegenstelling tot gewone Prologprogramma's, eindigen deze solvers in principe altijd: het model is verwezenlijkbaar of het is niet verwezenlijkbaar [Lifschitz, 2008].

In mijn onderzoek heb ik voor het oplossen van de Answer Set Programs 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 een een solver, namelijk `clasp`, die answer sets berekent [Gebser *et al.*,].

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

1.3 ProbLog

In deze paper zal ik onderzoek doen naar ProbLog. ProbLog is een probabilistische uitbreiding op Prolog, ontwikkeld door de onderzoeksgroep DTAI[2] aan het Departement Computerwetenschappen van de KU Leuven [De Raedt *et al.*, 2007]. Uniek aan ProbLog is dat je voor elke regel kan specifiëren met welke kans deze regel optreedt. Dat is dan de kans dat die regel behoort tot een willekeurige gesampled programma. Een ProbLog programma definieert dus als het ware een kansverdeling over alle mogelijke niet-probabilistische subprogramma's.

Een ProbLog-programma bestaat dus uit regels geannoteerd met een kans, een voorbeeld hiervan is het volgende: `p::friend(A,B)`. Deze regel zegt dat `A` bevriend is met `B` met een kans `p`. Verder is het in een ProbLog-programma ook mogelijk om `evidence` te voorzien. Elke sample gegeneert met het programma zal voldoen aan deze gegeven `evidence`. Zo kan je bijvoorbeeld het volgende schrijven: `evidence(friend(C,D)` wat betekent dat persooon `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. De analyse gevoerd in deze paper onderzoekt dus de toepasbaarheid en de mogelijkheden van deze willekeurigheid van ProbLog op het gebied van game content generation.

2 Probleemstelling

Het laatste deel van de vorige paragraaf brengt ons naadloos bij de probleemstelling van dit onderzoek: de toepasbaarheid van ProbLog op gebied van game content generation. In dit onderzoek heb ik een analyse gedaan op gebied van snellheid, variatie en een aantal andere kleinere criteria. Hierbij heb ik AnsProlog/ASP vergeleken met ProbLog en ook ProbLog op zichzelf geëvalueerd.

2.1 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. Ik heb ook de vergelijking geaakt 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.

2.2 Variatie

Het tweede criterium waarop dit onderzoek zich baseert is de variatie tussen de gegenereerde puzzels. 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 zo veel mogelijk variatie te verkrijgen zodat de speler die het spel speelt niet het geovel krijgt een bepaald palttegrond of lay-out al eens tegen gekomen te zijn.

2.3 Andere

Verder zijn er ook andere, kleinere criteria onderzocht. Ik heb onder meer gekeken naar de effectieve syntactische omzetting zoals de lengte van de verkregen ProbLog-code. Daarnaast heb ik in ProbLog ook gekeken naar het samplen met en zonder `evidence` en of dit een invloed heeft gehad op de snelheid.

3 Aanpak

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

3.1 Puzzels

Voor dit onderzoek heb ik gebruik gemaakt van drie puzzels. Voor sommige experimenten heb ik gebruik gemaakt van een vereenvoudigde versie van de puzzel als dit duidelijker was voor het experiment of als 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* 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 over de vakjes gevonden worden. Een overgang van het ene vakje naar een ander vakje enkel is toegelaten als de kleurenovergang die dan optreedt zich in het gegeven kleurenwiel bevindt.

Perfect Maze

De *perfect maze* 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 rechstonder.

Dungeons

Een *dungeon* is zoals de naam zegt een soort kerker. 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 geopend uitgang te begeven. Aan dit soort werelden kan een heel aantal contstraints worden toegekend: de verhouding 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

[2]Declaratieve Talen en Artificiële Intelligentie, `https://dtai.cs.kuleuven.be/problog`

textitdungeon 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 om te vormen tot ProbLog-programma's waarbij rekening moest worden gehouden 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. Enkele voorbeelden van hoe bepaalde zaken in AnsProlog werden voorgesteld met de equivalente ProbLog code vind je hieronder.

Zoals eerder vermeld, beschik je in AnsProlog over de mogelijkheid om niet deterministische regels toe te voegen. Zo zegt de regel hieronder dat voor elke tegel (elk paar van x en y coördinaten) exact één kleur uit alle mogelijke kleuren moet worden toegekend.

```
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. Dat zie je hieronder.

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

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 zijn 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`, waarme 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).
```

Dit waren de twee zaken die het meeste aandacht vereistten bij het vertalen van AnsProlog naar ProbLog.

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 gebied van snelheid en variatie tussen de puzzels.

Snelheid

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 gegeneerd, 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 je hebt opgegeven.

Voor `clingo` heb ik telkens 100 modellen laten genereren 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. Het commando voor de AnsProlog puzzels zag er dan als volgt uit: `clingo --models=100 --rand-freq=1 puzzel`.

Deze heb ik dan beide afzonderlijk getimed met het `time`-commando ingebouw in alle Unix-systemen[3]. Deze puzzels zijn gedraaid op een Macbook Pro, 2.7 GHz Intel i5-processor, 8GB DDR3 RAM onder OS X 10.11.

Variatie

Een tweede belangrijk criterium was de variatie in de gegenereerde/gesampelde puzzels. Dit vereistte voor elke puzzel een aparte aanpak om de variatie te kunnen bepalen.

Na de criteria opgesteld te hebben, die hieronder worden besproken, heb ik opnieuw 100 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). Hoe hoger de score, hoe meer variatie de puzzels vertonen.

- **Chromatic Maze**
  Voor de *chromatic maze* heb ik 3 soorten criteria geëvalueerd. Elk van deze heeft een gewicht van $\frac{1}{3}$ en de resulterende score wordt herschaald naar 100.
  - *Afstand tussen start-vakjes.*
    Ten eerste heb ik gekeken wat de Hamming-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 kom je relatieve waarden uit die je kan vergelijken voor elke soort puzzel.
  - *Afstand tussen finish-vakjes.*
    Op dezelde wijze als bij de start-vakjes heb ik de afstand berekend tussen de finish-vakjes.
  - *Afstand tussen de kleuren.*
    Tenslotte 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

[3] http://man7.org/linux/man-pages/man1/time.1.html

de twee puzzels verschillend zijn), zet ik de waarde op de maximale afstand dat 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.

- Perfect Maze
  De *perfect maze* is getest aan de hand van drie criteria, namelijk het aantal dezelfde verbindingen en de lengte van het kortste pad dat mogelijk is met die verbindingen. Het aantal dezelfde verbindingen krijgt een gewicht van 75% en de afstand van het kortste pad een gewicht van 25% aangezien dat de kans dat het korste pad hetzelfde is, relatief groot is.
  - *Aantal dezelfde verbindingen*
    In de *perfect maze* wordt een verbinding weergeven als volgt: `parent(3,2,-1,0)`, dit wil zeggen dat het vakje (3,2) verbonden is met (2,2). Uiteraard is dat hetzelfde als schrijven `parent(2,2,1,0)`. Er wordt dus getest of een bepaald vakje in de ene puzzel verbonden is met een bepaald vakje in de andere puzzel op deze twee manieren. Voor elke overeenkomstige verbinding wordt bij de score één punt 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 bereken gebruikmakend van het algoritme van Dijkstra. Het verschil in deze twee paden wordt beschouwd ten opzichten van het kortste pad mogelijk ($2 * n + 3 * (n - 4)$, 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 mee en levert, na herschaling, opnieuw een punt op 100.
  - *Afstand tussen diamanten*
    Als eerste wordt de Hammingafstand 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 die van het altaar, om zo te zien hoeveel stappen de speler als het ware 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 min2 (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 dichtsbijzijnde muur*
    Als voorlaatste criterium wordt voor elke muur de afstand berekent tot de dichtsbijzijnde muur in de tweede puzzel. Als er geen muur meer is die kan dienen als overeenkomstige muur, wordt de afstand maximaal gezet, 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.

4 Resultaten

4.1 Snelheid

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 gegeneerd voor dimensie drie en vier met zes kleuren en ook voor dimensie vijf voor vier kleuren (waardoor het aantal mogelijkheden al lager komt te liggen). De bekomen resultaten heb ik gesorteerd van laag naar hoog om een beter overzicht te verkrijgen. Je ziet hier 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 zie je voor de *dungeons* in figuur 2b. Dit zijn samples van *dungeons* met verschillende dimensies, waarbij er 1 diamantje en 1 altaar is en waarbij de rest van de vakjes (willekeurig) ofwel een muur ofwel niets zijn. Opnieuw ligt de variatie met ProbLog veel hoger.

4.3 Andere

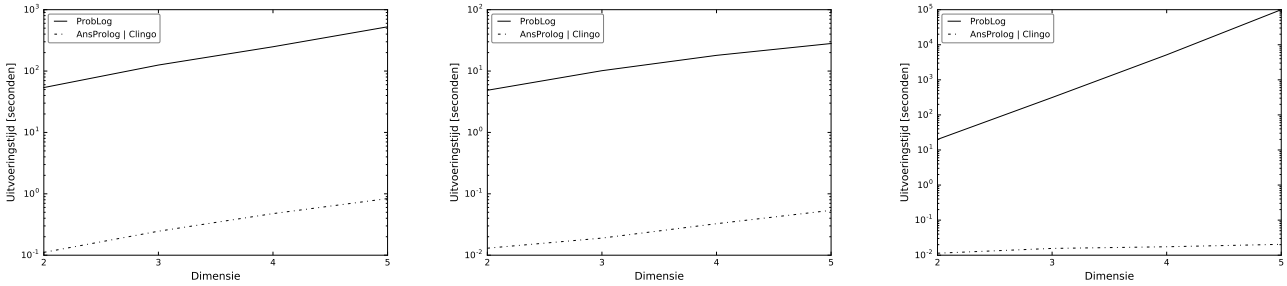
5 Toekomstig werk

Dankwoord

6 Referenties

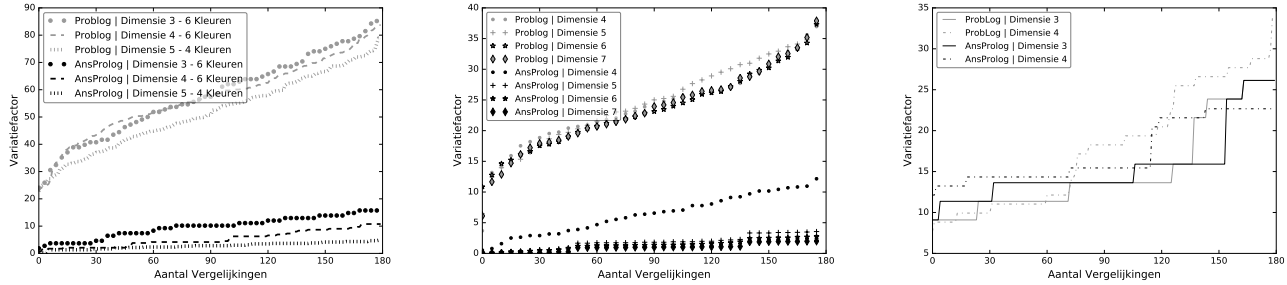
Referenties

[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.



(a) (b) (c)

Figuur 1: Uitvoeringstijd bij 100 gesampelde puzzels van verschillende dimensies met logaritsmisch geschaalde y-as van (a) *chromatic maze*, (b) *dungeons* en (c) *perfect maze*.



(a) (b) (c)

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

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

[Lifschitz, 2008] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597, 2008.

[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.

[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.