GA-ASP Dungeons

Using Genetic Algorithms and Answer Set Programming
to create roguelike Dungeon levels

by

Anders Hartzen

andershh@itu.dk

and

Tróndur Justinussen

tjus@itu.dk

for

Procedural Content Generation

E-2011

Julian Togelius

Contents

Introduction

This paper is written as part of a project for the course ”Procedural Content Generation” with Julian Togelius at the IT University of Copenhagen.

For the course we had to either develop a new PCG algorithm, or apply an existing algorithm or combination of algorithms to some game domain in a novel way. Ideally, the results would be publishable.

Problem statement

The subject we chose and are examining in this paper is the possibilities of combining the declarative nature of Answer Set Programming with the adaptability of Evolutionary Programming in order to evolve answer sets that solve whatever problem posed with a mixed-initiative algorithm.

Our case study was to create code that can produce random level generators for roguelike games. In essence we are working to create a Procedural Procedural Level Generator Generator for those kinds of games.

Desired result

The desired end result is an algorithm that takes a few randomly generated or manually seeded ASP instruction sets, evolves them a number of generations, evaluates each set by running agents through the levels represented, and outputs a number of answer set that have been deemed the most ”interesting” by our own metrics.

The resulting AnsProlog code would in theory be able to procedurally create a number of interesting levels quickly enough that it could be used as a random level generator in roguelike games.

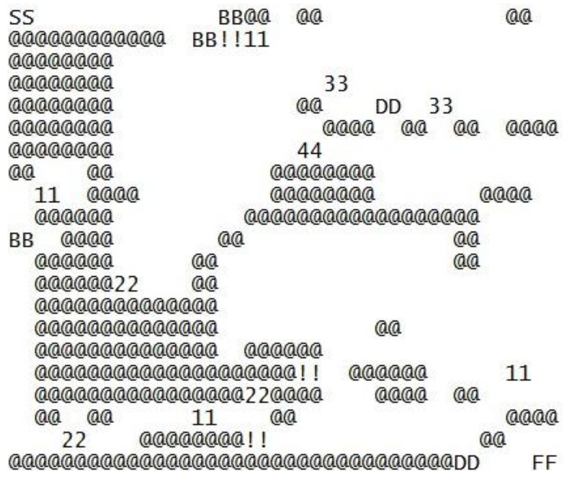
Background

The game Rogue was created in 1980 and had a novel random level generation feature that since has become the stereotypical use of procedurally generated content in games. Rogue was very successful and spawned a host of successors, including such famous titles as Nethack and the Diablo series. The more successful of the older roguelike games, including Rogue itself, are still supported by an active fanbase that regularly updates, refines and makes new editions of these great games.

Since the release of Rogue quite a few level generating algorithms have been created. Examples include *BSP Dungeon generation* (2011) and *Cellular automata for real-time generation of infinite cave levels* (2010).

Our approach to this problem is to evolve AnsProlog declarative code with simulation based fitness functions in order to produce ASP code that generates interesting levels. Using ASP to generate dungeon levels has already been proposed in *Answer Set Programming for Procedural Content Generation: A Design Space Approach* (2011), but combining it with evolutionary programming is a new way of taking advantage of ASP.

Dungeons based on this method will probably not look like the corridor-and-room solutions produced by the more standard roguelike level generators, but should still be able to create interesting dungeon levels.

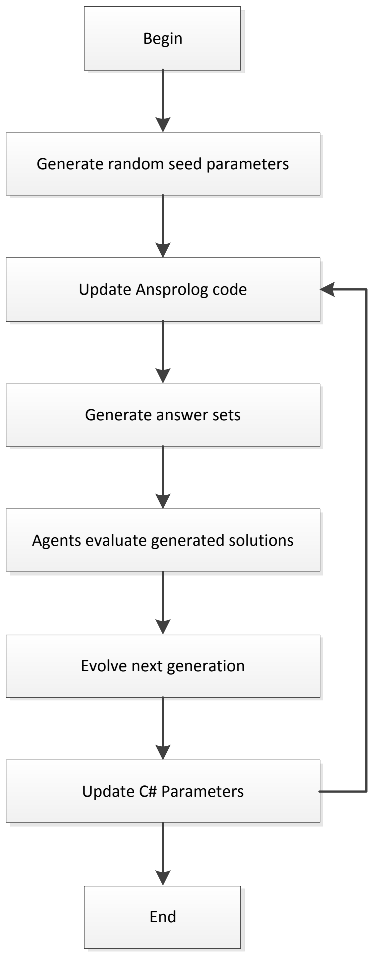


```
SS              BB@@  @@              @@
@@@@@@@@@@@@  BB!!11
@@@@@@@@
@@@@@@@@                33
@@@@@@@@              @@    DD  33
@@@@@@@@                @@@@  @@  @@  @@@@
@@@@@@@@              44
@@    @@            @@@@@@@@
  11  @@@@          @@@@@@@@        @@@@
  @@@@@@          @@@@@@@@@@@@@@@@@@
BB  @@@@        @@                @@
  @@@@@@      @@                  @@
  @@@@@@22    @@
  @@@@@@@@@@@@@@
  @@@@@@@@@@@@@@            @@
  @@@@@@@@@@@@@@  @@@@@@
  @@@@@@@@@@@@@@@@@@@@!!  @@@@@@      11
  @@@@@@@@@@@@@@@@22@@@@    @@@@  @@
  @@  @@      11    @@                @@@@
    22    @@@@@@@@!!                @@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@DD    FF
```

Methods

In our method, the ASP instruction set functions as a genotype, describing the constraints for the population of levels. The levels generated out of the genotype represent all possible phenotypes that are allowed by it. A number of random phenotypes are chosen as representatives of their genotype to be evaluated by our agents, whose combined work becomes the fitness function of our genetic algorithm. The most successful genotypes are then evolved into a new generation and the process is repeated.

The workflow is as follows:



We generate random seed parameters for 30 different instruction sets in our C# code.

These parameters fed into the ASP instructions file and executing one at a time, generating the first set of solutions.

Each of these is evaluated by our agents and rated on how ‘interesting’ they are.

Favouring the parameter set that generated the more interesting solutions, we then evolve the parameters for the next generation of ASP instruction sets.

The process then runs in a loop until a pre-determined point has been reached or we decide to stop it.

From this entire workflow we will end up with all the different ASP instruction sets that have been generated through the generations as well as their evaluation. Ideally, the latest generation will be the one closest to the requirements we evolved towards.

Our approach is divided into 3 major areas. APS instructions, agent based fitness evaluation and the genetic algorithm.

ASP instructions

The AnsProlog code is divided into 2 main sections: Parameters and rules.

The parameters define how big the level should be, positions of exits, the general openness of the level, the minimum distance should be travelled from entrance to exit and how many of the different features should be created. These are defined in simple '{parameter} = #' instructions that are easy to update from the C# code.

Rules define how the parameters, generating the attributes and features of the levels based on these numbers. Also, the rules define the soundness of the level by making sure that the features are reachable, not on top of each other etc.

Genetic algorithm

Genetic algorithms are part of the field of evolutionary computation, based on the scientific works of British scientist Charles Darwin and especially his monumental work *The Origin of Species*. Genetic algorithms take the basics from Darwin's theory of evolution and apply them to find optimal solutions to a given problem. The only requirement being that you can encode a solution to your problem as a chromosome, and that a solution can be evaluated by a fitness function. These are the basics Genetic Algorithms (GA), which we have tried to apply to our problem of generating a good level generator.

Chromosome (solution representation)

Height	Width	Length	Start X	Start Y	Exit X	Exit Y	Traps	Openness	M1	M2	M3	M4	B_TS	B_TD	B_E

In our case a solution is an instantiation of the different parameters we have defined in the ASP instruction set, parameters such as height, width, number of traps, number of monsters of each type, etc. There are 16 different parameters (or Genes in the GA terminology) that define our input into the ASP instruction set:

• Height	Height of the map
• Width	Width of the map
• Length	Minimum number of steps from start to finish
• Start X	X coordinate of the entrance
• Start Y	Y coordinate of the entrance
• Exit X	X coordinate of the exit
• Exit Y	Y coordinate of the exit
• Traps	Number of traps
• Openness	Number of open tiles
• M1	Monster of type 1;
• M2	Monster of type 2;
• M3	Monster of type 3;
• M4	Monster of type 4;
• B_TS	True sight buff that shows all traps in the map;
• B_TD	Buff for disarming next trap;
• B_E	Buff player for next monster encounter.

Genotype

The genotype for our representation is an ASP file using the Gene's values. This ASP is generated by writing the values to an .lp file containing them together with our already define answer set rules. This ASP file can then be fed into the ASP solver, which then can generate the Phenotype.

Phenotype
The phenotype of our problem representation is the set of maps generated by the ASP-solver when using the genotype. However since the solution space for an ASP can be very large, and thus takes long for the ASP-solver to generate, we saw it necessary to only select a few maps.

Selection and replacement Strategy
We decided to use truncation selection for selecting the genes from which to create the next generation of genes. The fittest 50% of gene sets from each generation move on to the next generation. The bottom 50% is replaced by the children of the top 50%.

Children are created by first cloning the top 50%, and then mutating the clones before inserting them into the population.

Mutation Strategy
Our mutation revolves around changing the values of the individual parts of our chromosomes. When a chromosome mutates we cycle through each part of its genes, applying a probability check of 30% (i.e. each gene has a 30% of being changed). The amount changed is based on the average value for that property in the whole population. For instance if the height gene is selected, the average height is computed and applied to the gene. Whether the gene will go up or down is randomly selected.

Fitness evaluation
The fitness evaluation is one of the cornerstones of the GA algorithm; it is with the fitness function that we determine which solutions are better than others. Our goal is to find level generators that make interesting levels – so that is what our fitness function should measure – in other words the "interestingness" of the levels generated by the level generator.

Interestingness
The basic premise on which we base our metric, is that an interesting level is one where a skilled player does better than a less skilled one. The higher the difference there is between the two, the better the level.

Using agents to measure interestingness
To measure this difference we have created two agents, with different skill levels. By measuring the difference in total damage our two agents took from finishing the level, we can simulate how much player agency affects that player's performance.

A* Agent
The A* agent is the one considered the less skilled player. It employs the A* algorithm with a straight line distance heuristic attached, to compute its route from start to finish.

Smart Agent
The smart agent is considered the smart player. It also uses the A* algorithm with the straight line distance heuristic, but also with a heuristic which values map nodes based on their type, making so that the agent will attract towards obtaining buffs, and avoiding traps and monsters.

Computing the fitness value

Selecting representative maps
For each ASP instruction set we used the first 20 maps to evaluate on.

Calculating interestingness

Our final evaluation function was as follows:

$$Fitness = \left(1 - \frac{straightLineDist}{A\ Star\ Steps}\right) + (1 - \frac{Smart\ Agent\ Damage}{A\ Star\ Damage})$$

Where *straightLineDist* is the straight line distance from start to finish, *AStarSteps* is the number of steps taken from start to finish by A* Agent, *AStarDamage* is the damage suffered by A* Agent and *SmartAgentDamage* is damage suffered by Smart Agent

Results

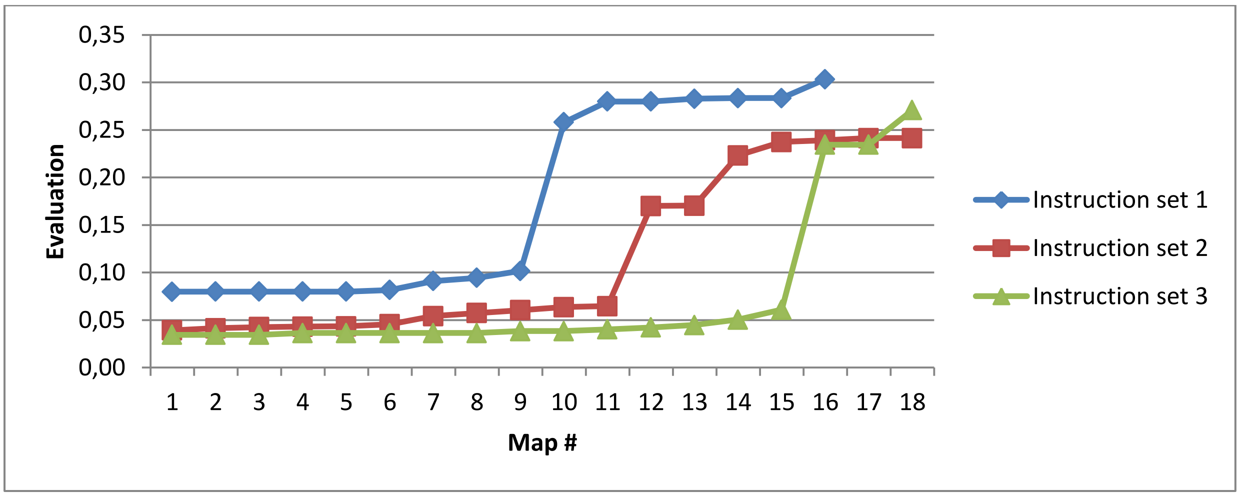
ASP instruction sets

The AnsProlog code worked well with the genetic algorithms, but there were some issues that had to be addressed in the implementation.

One of the main problems was how to select maps that properly represent the solution space of the procedurally generated ASP code. Generating all the answers is something that can take quite a while, time our GA would have to wait, before running agents through all of them. This coupled with GAs notorious running time quickly proved impractical for us. At first we tried to modify the ASP-rules to jump ahead in its solution space and return evenly distributed samples of the solution space. Sadly this didn’t pan out for us, and in the end we were forced to just look at the first generated maps in order to keep time usage to a minimum. This turned out to be acceptable as our tests showed that the solutions were varied enough in all but the most simple instruction sets.

Other problems included process would producing very big maps that used up so much memory that the process crashed, and the high likelihood of unsolvable or very restrictive constraints that resulted in time wasted on useless instruction sets. This was all minimized by creating limits on the values of each map parameter.

The evaluation of the ASP Instruction sets turned out to be highly dependent on the maps randomly chosen to represent them. The graph below shows how much the evaluation can vary for 18 maps from 3 sample ASP files. In order to remedy this increased the amount of sample maps from 5 to 20. We expected this to be enough to remedy the worst effects of the random map selection.



Evaluation functions

The biggest challenge of this project turned out to be defining the agents and evaluate the maps. The first agents had random elements and we found out that their fitness evaluations had a very high variance. We tried to minimize this by the use of weights for certain inputs, but it wasn’t enough. As we already had variance from the randomly selected levels, we instead decided to make all the agents deterministic, to keep the randomness to a minimum.

The table below shows the variations encountered in our first 3 evaluation functions done 8 times on the same level map.

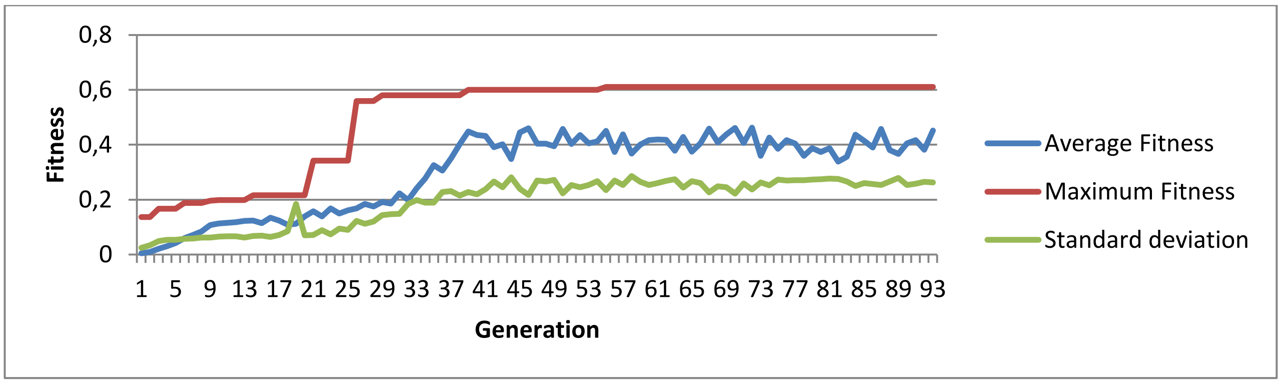
	Stochastic	Stochastic (80%-20%)	Deterministic
	6917,00	1648,00	1,146
	8910,00	2799,00	1,146
	14241,00	2806,00	1,146
	15464,00	3063,00	1,146
	15575,00	3191,00	1,146
	16457,00	4632,00	1,146
	24265,00	4716,00	1,146
	25117,00	4870,00	1,146
Mean	15868,25	3465,63	1,15
Standard deviation	6412,39	1153,68	0,00

Running the algorithm

We made several attempts to generate interesting maps, each time trying different evaluations methods. This was mostly because the algorithms generated results of a kind we didn't expect. For example our first runs only evaluated only on damage difference between the two agents. Of course without selection pressure for walls, the resulting maps had no walls, but were filled with monsters instead.
Later we tried different ways of combining damage and movement distance evaluations, each time generating unexpected but very logical outcomes.

Each run would take about 3 minutes per generation. In our trial runs it would take around 20-50 generations to reach a local maximum.

Below is an example of the evaluation numbers for a run of 92 generations. In this example, the there is a clear progression for the first 40 generations or so, before reaching a local maximum.



Taxonomy

If one were to organize the GA-ASP method into the taxonomy proposed in *Search-based Procedural Content Generation: a Taxonomy and Survey* (2011) the result would be as follows:

• Online/Offline	Running our method takes too long to be online but the resulting ASP can be used online if the constraints aren't too restrictive.
• Necessary/Optional	Level generators in general tend to be necessary.
• Random/Parameter	We used random seeded within certain min/max levels.
• Stochastic/Deterministic	ASP solutions are very stochastic in nature
• Constructive/Generate	We generate and test during the evaluation of levels

Discussion

On a design level, AnsProlog has a few problems. The main one being the lack of overview of the solution space without generating all the solutions for an instruction set. The solution space can be very large or very small depending on constraints, and it is difficult to know how many solutions one will get. Additionally, very restrictive constraints will result in long creation time, while lax ones will create less controlled solutions that might not be usable.

So if this method is to be used for creating level generators, success will hinge on coding evaluation functions that can handle such a free exploration of the solution space.

There is probably a need for more research on the area of declarative code in procedural content generation, since it would be usable for any problem statement that can be coded into an instruction set. In our experience, the problem is just to frame the question carefully, so that you get an answer you can use.

Further development

If the scope of this assignment had been bigger, we would have worked on creating better and more varied evaluation functions, added more parameters like distances between objects on the levels, added feature sets like keys and locks, and streamlined the entire process into a neat package.

Finally it would have been interesting to export the maps to an actual roguelike game, so one could explore the created maps first-hand.

References

Adam M. Smith (2011). *A Map Generation Speedrun with Answer Set Programming.* Retrieved December 14, 2011 from http://eis-blog.ucsc.edu/2011/10/map-generation-speedrun

Adam M. Smith and Michael Mateas(2011): *Answer Set Programming for Procedural Content Generation: A Design Space Approach.* In Simon M. Lucas' (Ed.) *Computational Intelligence and AI in Games, IEEE Transactions on. Issue 3, Sept. 2011.* Retrieved dec. 17, 2011 from https://blog.itu.dk/MPGG-E2011/files/2011/09/tciaig-asp4pcg.pdf

Anderson, M. (2011). *Basic BSP Dungeon generation.* Retrieved dec. 17, 2011 from http://roguebasin.roguelikedevelopment.org/index.php/Dungeon-Building_Algorithm

Doull, A. (2007-2008). *Death of the Level Designer.* In *Ascii Dreams.* Retrieved dec. 17, 2011 from https://blog.itu.dk/MPGG-E2010/files/2010/09/the-death-of-the-level-designer.pdf

Johnson, L., Yannakakis, Y. N., & Togelius, J. (2010) *Cellular automata for real-time generation of infinite cave levels.* In *Proceeding PCGames 10, Proceedings of the 2010 Workshop on Procedural Content Generation in Games.* Retrieved dec. 17, 2011 from https://blog.itu.dk/MPGG-E2011/files/2011/09/johnson2010cellular.pdf

RogueBasin (2011) *Basic BSP Dungeon generation.* Retrieved dec. 17, 2011 from http://roguebasin.roguelikedevelopment.org/index.php/Talk:Basic_BSP_Dungeon_generation

Ross, P., & Hallam, J. (1999). *Neural networks: an introduction*, AI Applications Institute, University of Edinburgh. Retrieved dec. 18, 2011 from http://www.itu.dk/courses/MVAI/E2008/notes.pdf.

Russell, S., & Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*, 2nd Edition, International Edition. Upper Saddle River, New Jersey, USA: Prentice Hall

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne C. (2011). *Search-based Procedural Content Generation: a Taxonomy and Survey.* In Simon M. Lucas' (Ed.) *Computational Intelligence and AI in Games, IEEE Transactions on.* Retrieved dec. 17, 2011 from https://blog.itu.dk/MPGG-E2011/files/2011/08/togelius2011searchbased.pdf

Yannakakis, G. N. (2011). *Genetic Algorithms.* Retrieved dec. 18, 2011 from https://blog.itu.dk/MAIG-E2011/files/2011/09/lecture4_ga.pdf

Blizzard North (2007). *Diablo* [Video Game]. Irvine, California, USA: Blizzard Entertainment.

Toy, M., & Wichman, G. (1980). *Rogue* [Video Game].

Stephenson, M., et al. (1987) *Nethack* [Video Game].