Procedural Content Generation for Computer Games

Lecture 4 – Constraints in PCG

Vojtěch Černý
cerny@gamedev.cuni.cz

Constraints of PCG

- Generating some kinds of content is hard
  - notably, puzzles
  - but also strategy maps, dungeons, …
- There's a lot of rules you want satisfied
  - Solvability
  - Fairness of resources
  - All areas reachable
  - …

Dealing with Constraints

- Adjust your PCG
  - Be smart with your approach (e.g. generate puzzles from end to start)
  - Use a validation technique (e.g. solvers, AI players)
  - Validate constraints for building blocks
  - …
- **Or** you can tackle it directly as a constraint satisfaction problem (CSP)

Constraint satisfaction problems (CSP)

- The area is called Constraint satisfaction problems
  - Specify your domain
  - Specify constraints
  - Let a solver find solutions
- Thoroughly explained in Roman Barták's lecture - **NOPT042**

- The techniques are many and the theory is vast
- We'll focus on the common approaches for PCG

Agenda

- Answer set programming (ASP)
- Generating game rules
- Mixed initiative approaches
- Wave Function Collapse

The topics of this lecture are loosely centered around the theme of constraining PCG.

Answer set programming (ASP)

A little warning up-front: This may be the only theory-heavy part of this lecture. Be prepared.

Answer Set Programming (ASP)

- Form of Constraint Satisfaction Problems (CSP)
- Popular and powerful paradigm
  - Specifically useful if dealing with hard problems (NP and harder)
- Builds quite a lot on theoretical logic

- We're going to go through the *(basic)* theory in several slides
  - *Simplified and informal whenever possible, for the sake of clarity*
- Then we'll move on to practical ASP
  - *Which is sometimes simpler than the theory*

Please read the following slides slowly and carefully.

If you want something more thorough:
A relatively short paper trying to explain ASP and the theory behind it: http://ceur-ws.org/Vol-1145/tutorial1.pdf
A paper just describing ASP skipping the theory behind it:
https://www.cs.cmu.edu/~cmartens/asp-notes.pdf

Basic ASP

The *program* is a set of rules

- A *rule* is an implication:

$\mathbf{x_0 \leftarrow x_1 \wedge x_2 \ldots x_n}$

where $\mathbf{x_0 .. x_n}$ are literals (atom / ¬atom)

- Rule has *head* - $\mathbf{\{x_0\}}$ and *body* - $\mathbf{\{x_i \mid i = 1 \ldots n\}}$

Legend:
∧ = and. ∨ = or.
The rule notation may remind you of Prolog (the actual language we will use is called AnsProlog), but the program execution is completely different.
Notably, rules in ASP can be written in any order without some causing an infinite loop.

In full ASP, the head may be a disjunction of literals. Empty disjunction is defined as ⊥, so that empty head is naturally ⊥.

Prolog tries to prove the query you give him.
ASP tries to generate a model for your problem.

Special rules

A rule without body is a *fact*.

x_0.

A rule without a head is an *integrity constraint* (head is ⊥).

⊥ ← x_1 and x_2 … x_n.

⊥ = contradiction.

What is an answer set?

- Minimal (in inclusion) set of literals ***M***, that satisfy (model) the program

Example – "grass is wet" reasoning program:

sprinklers ∨ raining.

¬blue ← raining.

Possible models: **{sprinklers, raining, ¬blue}**, **{sprinklers, ¬blue}, {sprinklers}**, **{raining, ¬blue}**.

Answer sets: **{sprinklers}**, **{raining, ¬blue}** *(only)*

Motivation for the example (similar ones are used in Roman Barták's lectures) – the grass is wet. Reasoning about what happened.

sprinklers = sprinklers are on.
raining = is it raining
blue = sky is blue

Side note – in this example we used full ASP, with disjunction in head.

Default negation

- In logic, we have two negations:

Classical (**¬x**) ¬x ∈ ***M***

Default (**~x**) x ∉ ***M***

- Interpretation

¬x we know that x isn't true

~x we don't know that x is true

We never want **{x, ¬x}** ⊆ ***M*** *(contradictory model)*

For obvious reasons, we do not want contradictory models.

Reducts

Important when we want default negations in our programs.
In this slide negative = default negative. Positive is opposite of negative.

Reduct of program ***P*** by set of literals ***M*** ($\boldsymbol{P}|_{\boldsymbol{M}}$) is formed:

- Take only applicable rules (negative literals aren't in ***M***)
- Take only positive literals
  - (if head is negative, change it to ⊥)
- (The result is a program)

The result is a program without negations.
M is an answer set to ***P***, if it is an answer set to $\boldsymbol{P}|_{\boldsymbol{M}}$.

Reducts are kind of difficult to understand intuitively. The intuition is to reduce the program to only positive literals by knowledge in ***M***.

Note that the transformation of a rule to the reduct is independent on ***M***.
M only affects which rules will remain in the reduct.

Example of reduct

P = **{q ← ~p}**

$P/_{\{\}}$ = **{q}** the rule is applicable
$P/_{\{p\}}$ = **{}** the rule is not applicable
$P/_{\{q\}}$ = **{q}** the rule is applicable
$P/_{\{p, q\}}$ = **{}** the rule is not applicable

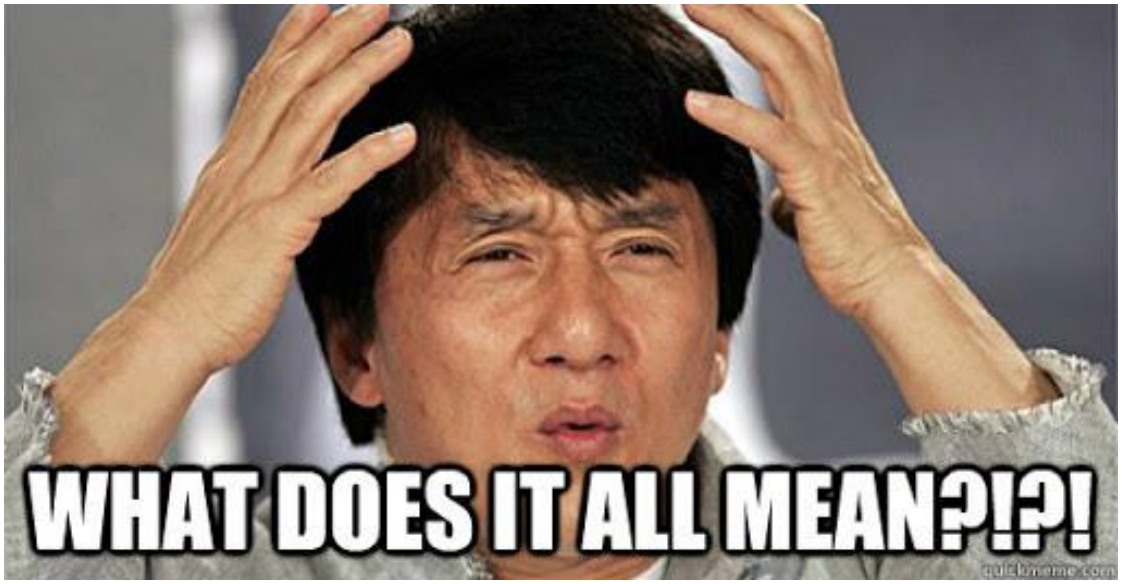
Only **{q}** is an answer set.
{p} isn't a minimal model for **{}**.
{} isn't a model for **{q}**.

Example of reduct 2

P = **{q ← ~p, p}**

$P/_{\{\}}$ = **{q, p}**
$P/_{\{p\}}$ = **{p}**
$P/_{\{q\}}$ = **{q, p}**
$P/_{\{p, q\}}$ = **{p}**

Only **{p}** is an answer set.



Maybe this was understandable or even easy. Maybe you feel like Jackie here. Either way it is time to move on, from theory to practice.

ASP in practice

Thankfully, it is easier to apply ASP in practice than to fully grasp the theory behind it. However, without knowing the theory, you may be (however rarely) stumped by some things happening in your program.

It's like knowing how to program in a high-level language without understanding the processes behind it (compilers, caching, complexity theory, etc.).

Legend

Meaning	Theory	ASP program
Rule	$\mathbf{x_0 \leftarrow x_1 \wedge x_2 \wedge x_3}$	x0 :- x1, x2, x3.
Classical negation	**¬x**	-x
Default negation	**~x**	not x

Practical programming in ASP

Thankfully, it's easier to use ASP in practice than to understand the theory. You're describing what you "want" in the solution.

Rule

```
y :- x.
```

- If there's x, there also must be y.

Fact

```
x.
```

- There must be x.

Constraint

```
:- x, y.
```

- There can't be both x and y.

Note: if you combine the rule, the fact and the constraint here, you get an unsatisfactory program.

Adding predicates

- In practical ASP, you also have predicates (it works in first order logic).

Example - graph coloring:

```
node(0). node(1). node(2).
edge(0, 1). edge(1, 2). edge(2, 0).
color(r). color(g). color(b).
```

They work identically to terms, but allow additional constructs.

ASP Shorthands

- Often used rules, which (drastically) simplify code

Ranges

`p(1..3).` shorthand for p(1). p(2). p(3).

`p(1;2;3).` shorthand for p(1). p(2). p(3).

ASP Shorthands 2

Choice rule:

`{x, y, z}.` "shorthand" for {x ∨ ¬x}. {y ∨ ¬y}. {z ∨ ¬z}.

Practically, this generates answer sets for all combinations of variables

Answer sets now are {}, {x}, {y}, {z}, {x,y}, {x,z}, {y,z}, {x,y,z}

Can be constrained by lower and upper bound:

`1 {x, y, z} 2.` (at least 1, at most 2)

{} and {x,y,z} are no longer answer sets

ASP Shorthands 3

- Choice rule with conditional literals

```
2 {q(X) : p(X)} 2.
```

Model has to have 2 q(X) from all possible p(X).

```
{q(X) : p(X)} = 1.
```

Alternative form of `1 {q(X) : p(X)} 1`

Adding predicates 2

Use variables.

```
connected(X, Y) :- edge(X,Y).
connected(X, Y) :- edge(Y,X).
```

As far as models are concerned, this means that for each edge(X,Y) in the model, also connected(X,Y) and connected(Y,X) will be added to that model.

Adding predicates 3

Choice as part of standard rules.

```
{colored(V, C) : color(C)} = 1 :- vertex(V).
```

For each vertex, there must be exactly 1 color assigned to that vertex.

Please, keep this rule in mind when you'll be starting with ASP (e.g. in labs). If you understand this one, everything else is simpler.

Example – graph coloring

```
vertex(0..2).
color(r;g;b).
edge(0, 1). edge(1, 2). edge(2, 0).
{colored(V, C) : color(C)} = 1 :- vertex(V).
:- edge(X, Y), colored(X, C), colored(Y, C).
```

The constraint specifies that for each edge, it cannot hold that both of its vertices have the same color.

Program structure

Programs are usually structured in 3 sections

- Generate
  - Describe how to generate a candidate solution
- Define (non-mandatory)
  - Additional “helper” rules to simplify program
- Test
  - Add constraints

Helper rules are sometimes even useful in the generate sections. As such, there also might be helper rules throughout the generate section of the program.

The previous program had just the Generate section (4 lines) and a Test section (1 line).

Important note – this is a user structure of the program, NOT how it is executed in a solver.

Hamiltonian cycle (oriented graph)

- Graph instance described elsewhere by predicates edge/2, vertex/1
- Generate

```
{ham_edge(U, V)} :- edge(U, V).
```

- Define

```
reachable(0).
reachable(Y) :- reachable(X), ham_edge(X, Y).
```

- Test

```
:- 2 {ham_edge(X,Y) : edge(X,Y)}, vertex(X).
:- 2 {ham_edge(X,Y) : edge(X,Y)}, vertex(Y).
:- not reachable(X), vertex(X).
```

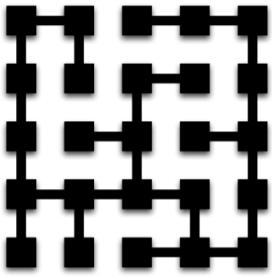
The constraints (Test section) mean (in order):
There cannot be two hamiltionian edges coming out of any vertex.
There cannot be two hamiltionian edges going into any vertex.
There cannot be an unreachable vertex.

Feel free to express these as non-negatives.

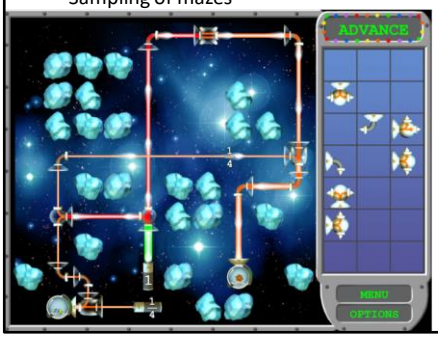
Final notes on ASP

- Powerful tool
  - Solves hard problems
  - Easy to write (with some experience)
- You can also look at it as “enhanced grammars”
  - With constraints
  - And more expressive language
- Resist any urges to write your own solver

Example usages of ASP



Sampling of mazes



Puzzle levels in Refraction

Base and oil placements in Warzone2100

AKA what is this good for.

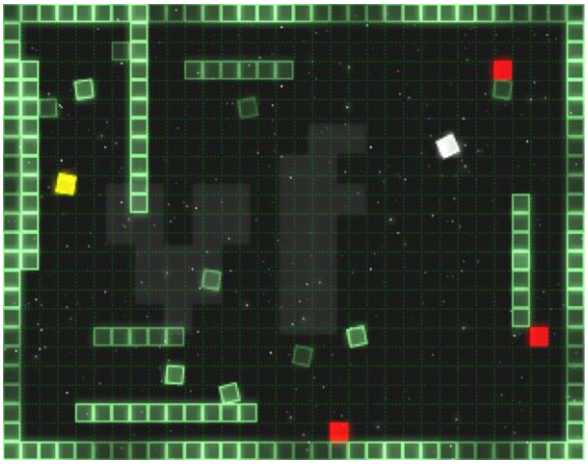
Warzone 2100 originally used evolutionary algorithms to place bases and oils, and then determined that ASP is more fit and reliable.
If you want to use PCG in a puzzle game, or maybe just create a solver as a validator for it, ASP may be a quick solution.

Examples usages of ASP

Variations Forever
Adam Smith and Michael Mateas

Generating simple mini-games
Including:
- topology
- character movements
- goal
- …

“Mixed-initiative” generation by adding constraints of what you don’t like



(Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games, Adam M. Smith et al, 2010)

The games consist more or less of “Pacman-likes” and “Asteroids-likes”.

Generating game rules

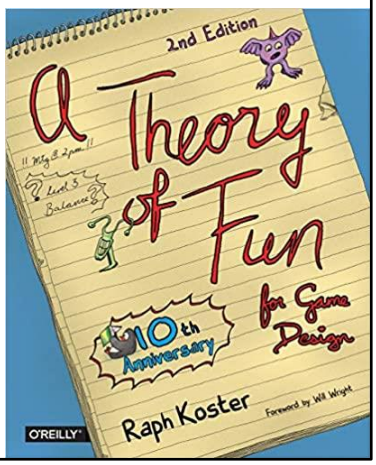
Okay, so thus far, this lecture was somewhat different and hardcore.
Now we're returning back to the "standard" form.

Generating game rules

- Constraints or goals are difficult to express

- We want games to be fun, but what is fun?
- How do you measure it?

- Three nice theories:
  - Csikszentmihalyi's theory of flow
  - Koster's theory of fun
  - Malone's three-component model
    - Challenge, fantasy, curiosity



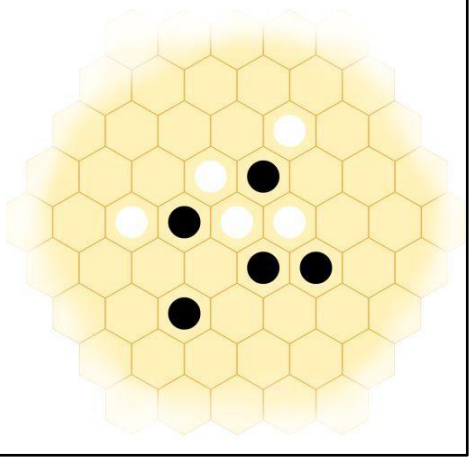
Flow != fun, but is quite related.
Using flow as an indicator of fun, we see that it is related to difficulty.
But difficulty is similarly hard to measure, it is also subjective, with people having different preferences, etc.

In the end, you want humans to be satisfied with your game. But in the process, you may benefit of some of the (mentioned or other) models.

(Flow: The Psychology of Optimal Experience, Mihaly Csikszentmihalyi, 1990)
(Theory of Fun for Game Design, Raph Koster, 2004)
(Toward a Theory of Intrinsically Motivating Instruction ,Thomas W. Malone, 1981)

Generating game rules

- We have already studied Yavalath
- Now we have seen an ASP approach
- Are there any PC games with successfully generated rules?
- Shamely, not quite. But let's look at interesting experiments.



Why are we looking at things that have failed, in the practical sense?

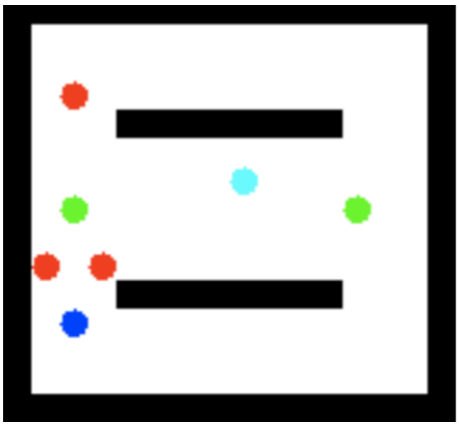
Well, even looking at Lecture 0, most of games that popularized some kinds of PCG weren't the first ones to feature it.
Even Rogue wasn't the first game with procedurally generated dungeons.

That is, in practice, usually some game invents a PCG feature and either flops or does mediocre, and then somebody figures out how to perfect this feature and fit it better in a really successful game.

And also, there's a real game - ROM CHECK FAIL, but the space of the mechanics in it are really small and don't produce much novelty.

An Experiment in Automatic Game Design

- J. Togelius and J. Schmidhuber
- PacMan-like games
  - Grid based
  - One player entity (cyan)
  - Only simple movements of other entities
- Rule space of several variables
- Fitness
  - Learnability
  - Tries to find not too easy, not too hard
- Results:
  - Simple mechanics, such as “chase blue”



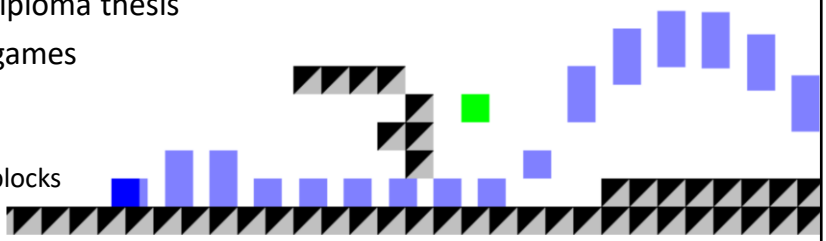
An Experiment in Automatic Game Design, Julian Togelius and Jurgen Schmidhuber.

Fitness – discard games that a random player plays “somewhat well”.
Then award games that could be learned quickly (by quite a simple evolutionary strategy, learning a simple neural network).

The rule space is quite simple, given by variables mapping:
- Number of items of each color (3 variables)
- Behaviour of item of color (5 possibilities, 3 variables)
- Collisions between items (3 possibilities, 4 x 4 variables)
- Score of collisions (3 possibilities, 4x4 variables)

Endless Runners

- Vojtěch Černý, diploma thesis
- Endless runner games
- Coevolution of:
  - AI controllers
  - Level building blocks
  - Game rules



- Some interesting results, research still ongoing
- Fitness by number of rules applied, actions used, etc.

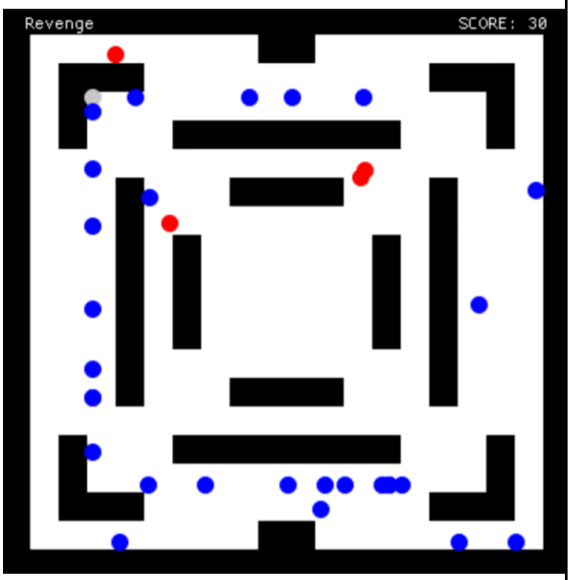
Yes, yes, shameless self-promotion. Move along.

The concept is that endless runner games are relatively narrow genre, and it might be simpler to generate
As such we decompose everything to building blocks (objects, actions, collisions, …) and let the generator pick what he wants

AI players were small evolved neural networks, layout formed of generated blocks, rules represented as a list with attributes and pointers to each other (similar to linear genetic programming)
Screenshot is from a game that was generated, including rules, actions, game speed, collisions, and layout blocks

ANGELINA$_1$

- Michael Cook
- Grid-based games
- Cooperative coevolution of:
  - Rules
  - Level design
  - Layout design
- Rules are formed by a grammar
- Players controlled by simple search
- Very few games even playable



Available from http://www.possibilityspace.org/publication.html

Actually, forming rules by a grammar follows quite natural creation. What happens in a collision can be combined with an if-else statement, multiple things can happen at the same time, etc.
Level design – bitmap of passable/impassable
Layout design – list of objects and some of their variables
Could probably be combined, it may be just difficult because of different representation (direct & indirect)
Note: In the screenshot, the map is statically designed by a human, yet it is the first one provided in the paper. Misleading!

ANGELINA$_2$

- Michael Cook
- Metroidvania games
- Similar to ANGELINA$_1$
- Instead of rules, evolves powerups
  - Several improvements to player
  - Often make a new map part accessible
- Much better playability
  - But that is expected given the base game is human designed

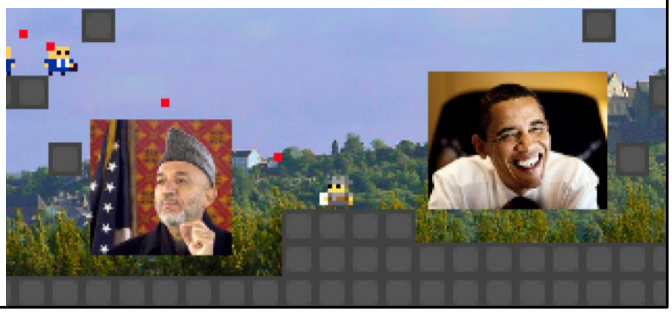
Available from http://www.possibilityspace.org/publication.html

ANGELINA$_3$

- Michael Cook
- builds on ANGELINA$_2$ (still Metroidvania's)
- analyses current news and online sources
- tries to add story and aesthetics to ANGELINA$_2$

- The games start to be weird (and thus, interesting)

- Example: Game based on war in Afghanistan



Available from http://www.possibilityspace.org/publication.html

The game is called HOT Nato, Angelina tries to finds titles that somewhat rhyme.

ANGELINA$_4$

- still Michael Cook
- expands on ANGELINA$_3$'s thematization
  - uses a total of 15 services for textures with tags, music with moods, …
- utilizes ANGELINA$_1$-like mechanics
- creates 3D maze-like games
- outputs game description
- entered a GameJam

Available from http://www.possibilityspace.org/publication.html

This time, it is inputted a short phrase – in GameJam, the theme.
Didn't do too bad, the game is at least playable and interestingly weird ☺.
http://ludumdare.com/compo/ludum-dare-28/comment-page-1/?action=preview&uid=29184

Oh, and if you wandered what ANGELINA stands for – A Novel Game Evolving Labrat I've Named Angelina.

Mixed-initiative approaches

Mixed-initiative approaches

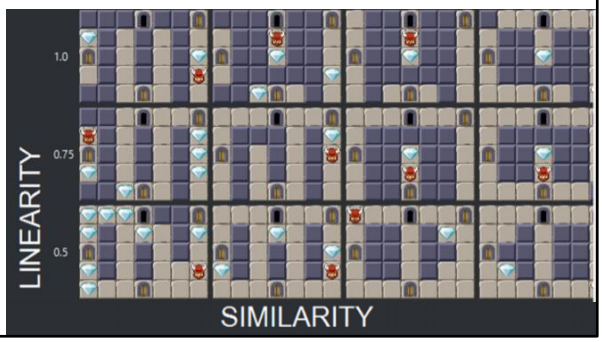
- PCG approaches requiring substantial input from a human at runtime, where the initiative is passed between human and computer
  - We do not count Spelunky to be mixed-initiative, since the human input is already part of the generator
  - However, interactive fitness evaluation in search-based PCG is mixed-initiative
- A dialogue between human and computer
- Human and computer may not have equal say
  - Computer-main methods
  - Human-main methods

The human input is the constraint here. It may be difficult whether it is a hard constraint (necessity) or a soft one (preference).
In mixed-initiative PCG what the human wants is usually a hard constraint.

All approaches require some input from humans, at least pressing the “Generate” button. By “substantial”, we mean something more than that.

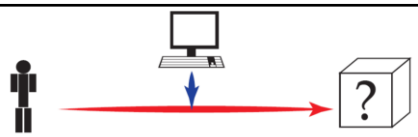
Computer-main

- Interactive evaluation
  - Can be taken a bit further by letting the human adjust evolution parameters
    - mutation rate, etc.



For illustrations on procedural terrain, you can check out videos at https://www.world-machine.com/

Human-main



- Computers just fill in the details
  - Procedural texturing
  - Procedural vegetation (SpeedTree) on a map
  - Diamond-square algorithm
  - Add small-scale noise to handcrafted terrain

Sui Generis had a pretty nice computer-aided terrain tool – check video at https://www.kickstarter.com/projects/1473965863/sui-generis (at 4:10)
Shame the game development died.

Quite cool automatic 3D tiling:
http://oskarstalberg.com/game/planet/planet.html

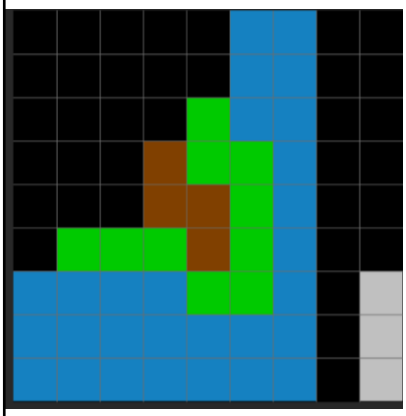
The three initiatives of dialogue

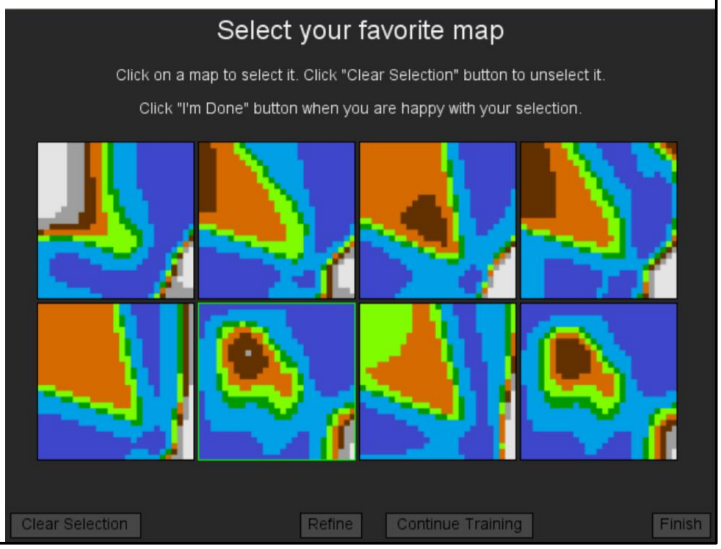
- Task initiative
  - Who decides what the dialogue will be about?
- Speaker initiative
  - When will who speak?
- Outcome initiative
  - Who decides the consensus / end of dialogue?

- Thus far, PCG mixed-initiative is typically only considering the speaker initiative
  - The others are 100% decided by humans

(What is Mixed-Initiative Interaction?, Norvick and Sutton, 1997)

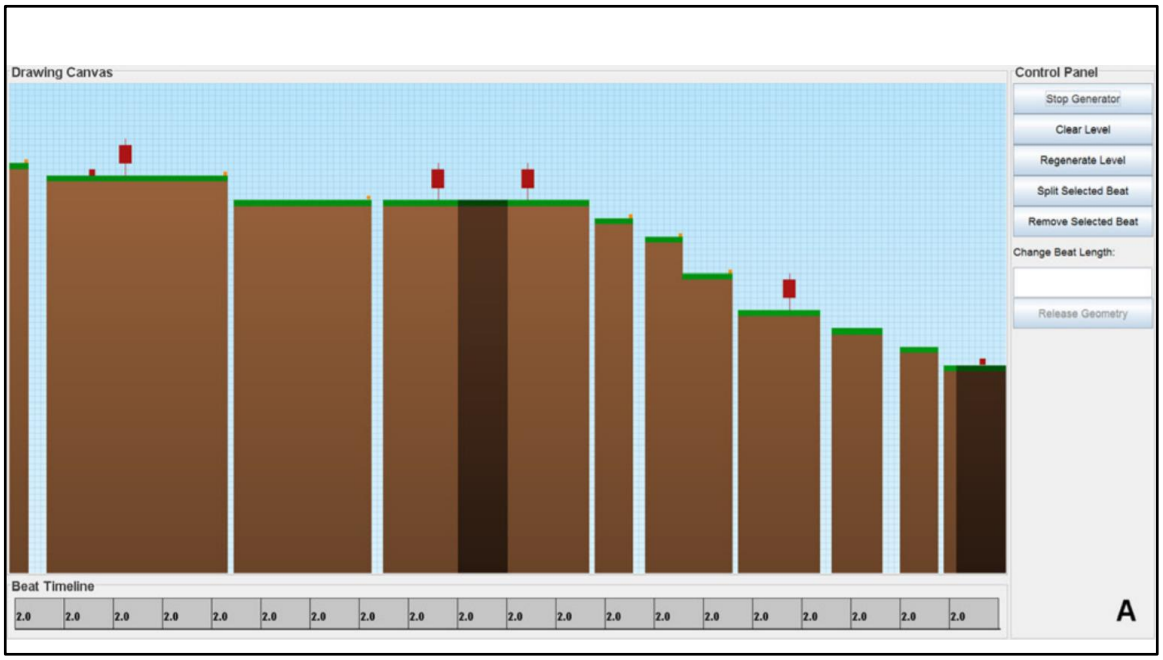
Examples of more even mixed-initative





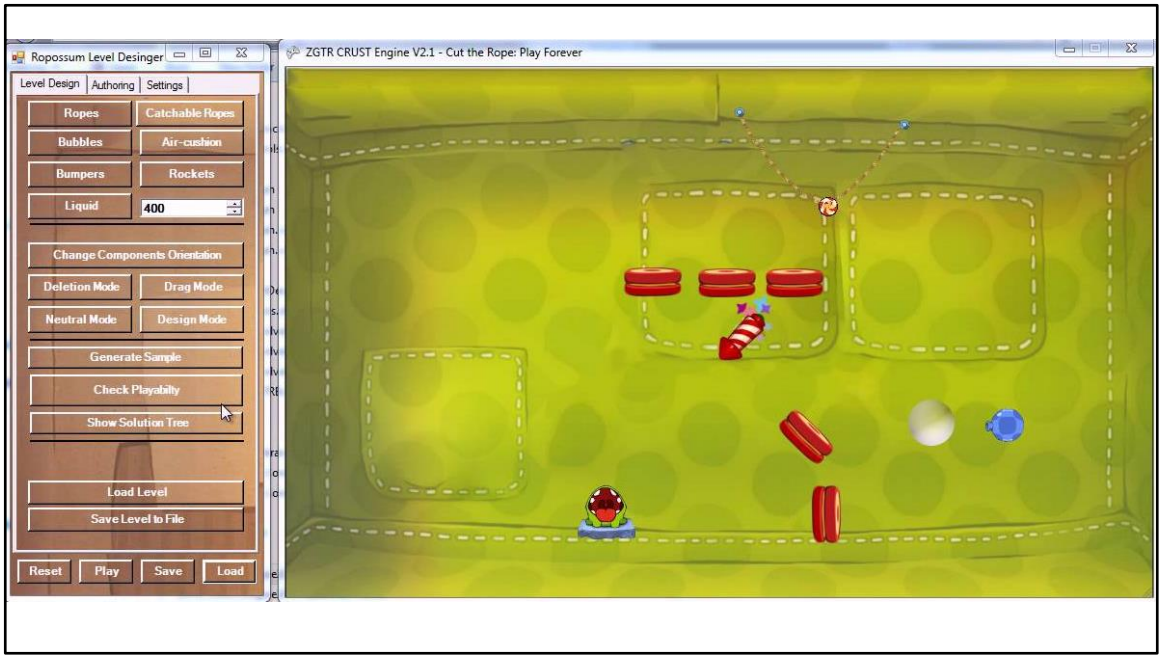
(Sentient World: Human-Based Procedural Cartography, Liapis et al., 2013)

A neural network tries to learn your patterns and then suggests higher resolution of what you're drawing.



(Tanagra: A Mixed Initiative Level Design Tool, Smith et al, 2010)

Uses beats as underlying structure – tries to control the pacing of the level.
The human can manipulate the beat-timeline, regenerate the level, and also modify the level directly.



(Ropossum: An Authoring Tool for Designing Optimizing and Solving Cut the Rope Levels, Shaker et al., 2013)

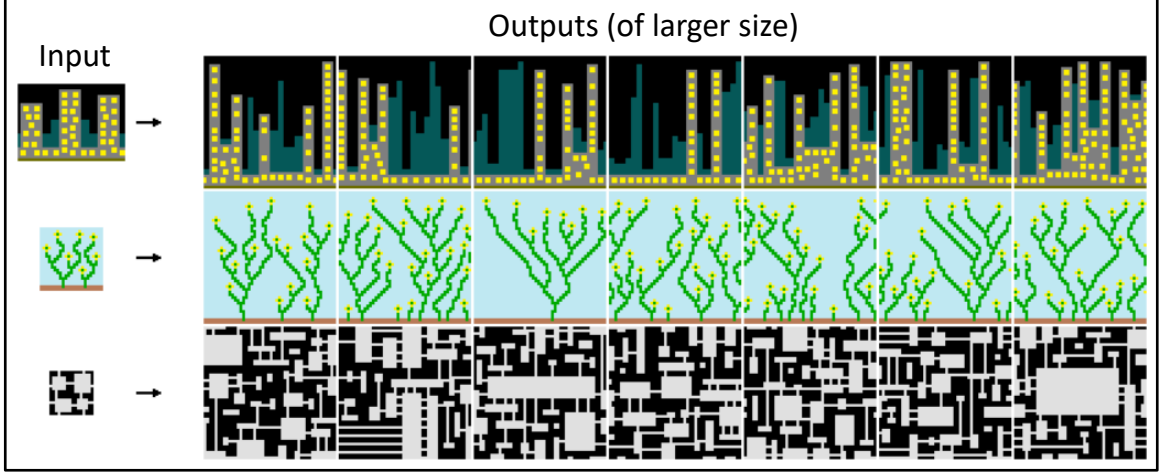
Features an AI solver to validate your design (by searching scripted player actions). Parts of the level may be regenerated.
By itself generates just simple levels, but can provide variants of human designed works.

Mixed-initiative approaches are still a largely unexplored territory.

Really, the more interesting approaches are few and far between.

Wave Function Collapse

Wave Function Collapse (WFC)



https://github.com/mxgmn/WaveFunctionCollapse (started in 2016, check the repo for gifs and links ☺)
It is an algorithm that can take a really small input, and turn it into output.
It seems quite a magical at first glance, but the inner workings are actually pretty simple :).

Wave Function Collapse (WFC)

- An algorithm for creating outputs that are locally similar to inputs
- Local similiarity:
  - 1. Each N*N pattern in output exists in the input
  - 2. (weak) probability of meeting an N*N pattern in output should be close to density of the pattern in input
- Inspiration in quantum mechanics
  - Wave collapse of superposition into a single state by observation
- Also a specific usage of CSP

Uses path-consistency.

WFC pseudocode

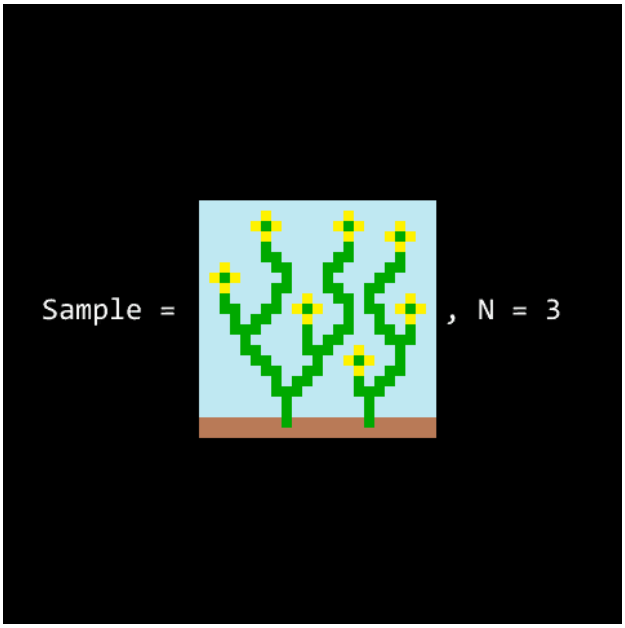
Algorithm 3: Wave function collapse (Overlap)

Data: bitmap B, tile size N, output size W, H
patterns ← *all* $N \times N$ *patterns of B, with weights = #occurences in input*
constraints ← *which patterns neighbors each other in B*
output ← array (W × H) *of zeros*
possible ← array (W × H) *of lists, each containing* patterns
while not *output filled* **do**
  pick uncollapsed $x, y \in W \times H$ with minimal entropy of patterns
  output$[x, y]$ ← *random (by weights) pattern from* possible$[x, y]$
  update possible *by* constraints
end

Yes, the updating possibilities by constraints is still quite fuzzy here.
There are in fact several options:
- you start by checking only the surrounding of already placed cell, removing possibilities there
- you can (and usually should) also propagate the changes in possibilities
  - that is, whenever you remove a possibility, check all surrounding cells if you should remove a possibility there

This is thoroughly explained in Roman Barták's lecture (look for Node Consistency, Arc Consistency and Path Consistency)
The take away is, it is a difficult tradeoff, more propagating constraints = less searching, but also more work on propagating constraints.

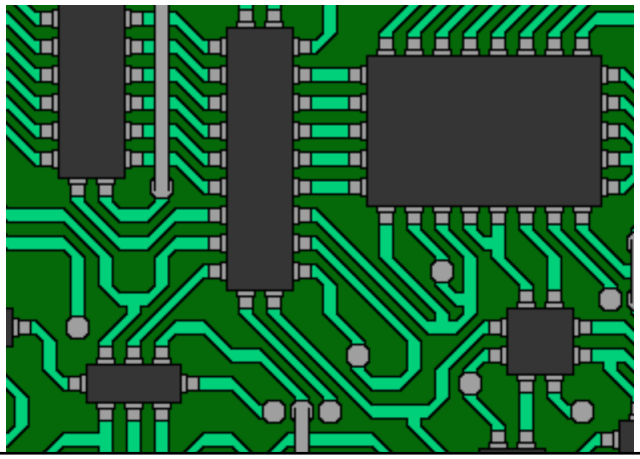
Visualising GIF



WFC, tile version

Input

Example output



WFC, tile version

- Works similarly to “overlap” version
- Instead of learning weights and placement constraints from bitmap, they are specified directly as input
- Essentially a method of tiling with Wang tiles

https://en.wikipedia.org/wiki/Wang_tile

WFC pseudocode - tiles

Algorithm 4: Wave function collapse (Tiles)

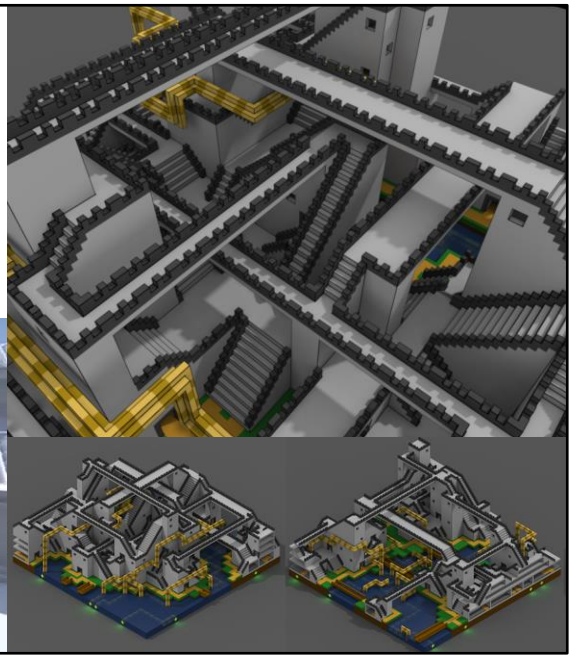
Data: tiles $T_1..T_n$, weights $W_1..W_N$, tile placement constraints C, output size W, H
output ← array (W × H), empty
possible ← array (W × H), tiles *in each cell*
while not *output filled* **do**
  pick uncollapsed $x, y \in W \times H$ with minimal entropy of possible tiles
  output$[x, y]$ ← *random (by weights) tile from* possible$[x, y]$
  update possible *by* constraints
end

- *Note that picking minimal entropy reduces to picking cell with minimal number of possible options when $W_1..W_n = 1$*

Also, instead of picking minimum entropy (which is actually quite human-like way of working) and randomizing by weights, you can experiment with custom algorithms for picking.
You risk higher possibility of running into a dead end (more on this later), but may be able to tweak the algorithm into e.g. preferring accessible regions.

Wave Function Collapse

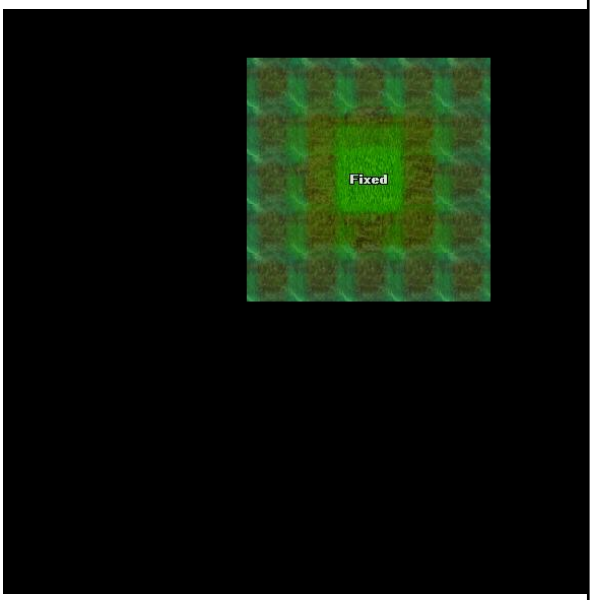
- The algorithm can also work in 3D



youtube.com/watch?v=bHr-A2r9N9s

Wave Function Collapse

- The algorithm can also work mixed-initiatively



Animated GIF, make sure to switch to presentation mode.

http://oskarstalberg.com/game/wave/wave.html a different, interactive demo running in browser

WFC Final notes

It is possible that the generation will fail due to “running into a dead end”

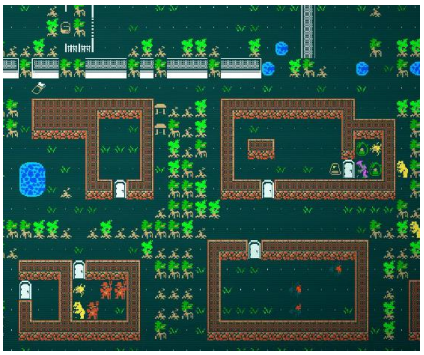
- There aren’t any possibilities for some tile due to constraints
- In practice, with good tiles / bitmaps, this is rare
- If it happens, you can:
  - restart
  - backtrack (and forbid possibilities as you go)

Restarting might be a better if dead-ends are rare (and they usually are).

WFC Examples

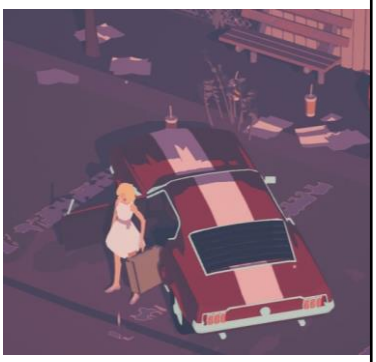
Bad North

3D WFC for islands



Caves of Qud

WFC for maps



Dead Static Drive

WFC for large scale tiling

https://www.badnorth.com/
https://store.steampowered.com/app/333640/Caves_of_Qud/

Bad North uses the tiling version of Wave Function Collapse.
Caves of Qud (considering the map as a bitmap) uses the overlap version (with additional passes to e.g. ensure connectivity)
Dead Static drive uses the tiling version for quite large tiles, and then modifies details of these tiles to avoid visible repetition.

Final Notes

Final points

- ASP is an efficient way to generate hard things
  - Even though it may be somewhat daunting at first sight
- Generating game rules is difficult
  - But can be done – take Yavalath
- Mixed-initiative approaches are usually about letting human affect computer processes
  - Or vice versa
- Wave Function Collapse is a cool and powerful algorithm that can procedurally generate tilemaps and bitmaps with little effort

Q & A

cerny@gamedev.cuni.cz
discord.gg/Zts98PGw6z

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Material has been produced within and supported by the project
„Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“
kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.