Node-Oriented Procedural Engine

version 1.0

by Filip Skwarski

"Everything small is just a small version of something big."

– Finn the Human, Adventure Time (S2E15)

How It Works

The goal of the project was to create a reusable method for procedurally generating tile maps of arbitrary size, complexity and purpose (dungeons, towns, continents, custom shapes…) by specifying a set of local rules (a generative grammar).

The rules generate a graph, which is then converted into a 2-dimensional map by recursively arranging and connecting its nodes from the bottom up until every required structure and road is generated. The end result then undergoes post-processing to make it look more detailed.

The 'windows_launcher' interface is a way to visualize the various results of specific grammars included in the maps folder. The actual maps (or rather the templates for generating an infinite number of maps from a specific set of rules) can be modified by editing the files in the 'maps' folder.

To create and test your own map, simply copy-paste one of the existing map folders under a different name and edit the included text files, or create a completely new folder with a .txt file containing different rules from scratch. The required map template format is explained in "Editing and Creating Map Templates".

1. Map template →	**2. Map graph →**	**3. Final map**
Grammar for generating map graphs, provided as a text file. Allows for multiple ways to construct the map graph from the top down. This is what .txt files in the 'maps' directory define.	One specific instance generated from the map template and a specific seed. Only one arrangement of areas and paths is selected. This is what the interface shows after loading a map, changing seed selects a different graph.	2-dimensional tile map generated from a specific map graph from the bottom up (with minimal randomness affected by the seed). This outcome is split into 2 major steps: an initial map, close to the graph, and a second step adding post-processing and graphics.

The basic stages and the order in which they happen are shown above. The conversion of the map graph (2) into the final map (3) is the most important step of map generation.

Interface (windows_launcher)

The launcher will likely only run on Windows and requires a mouse and keyboard to navigate – it's a Python 3 script which uses pygame for displaying UI and map graphics, compiled with auto-py-to-exe.

Running windows_launcher.exe (or main.py in the source files) launches a visualizer with the UI on the right, allowing for loading map rules, switching seed (version of the map generated), and generating/exporting results.

The UI includes the following settings:

- tile size – The zoom level of generated map, which can be freely changed at any point. Affects the exported image as well. Click left/right arrows to decrease of increase this value.
- seed – Levels with the same seed and map template will always end up looking the same. Seed can be freely changed before generating the map, but changing it with a map displayed will take more time because another map be generated. To prevent taking too much time, changing seed is disabled once a map is fully generated (but you can go back, change seed, and generate it again). Click left/right arrows to change the seed,

And the following actions:

- Load Map – Open one of the map templates included in the 'maps' folder. The graphs generated for that template will be displayed in the main window.
- Generate – Generate a preliminary map based on the currently open graph.
- Add Details – Apply postprocessing and tilemap to the generated map.
- Export Txt – Save an array of all tiles of the generated map in the main folder.
- Export Png – Save a full image of the generated map at current zoom level in the main folder.
- Back – Go back to the graph display (after generating a map).

Generation and postprocessing times depend on the size of the map – the more complex the graph, the longer the map will take to generate.

Besides generating a full map, it's possible to **click specific nodes of the graph to select them**. Choosing "Generate" with a node selected will only generate a part of the map corresponding with this node. To de-select a node, click the background outside of it.

If there are any bugs when loading the level, a specific warning message should appear towards the top of the screen. Usually, it means there is an error in the syntax of the map template (e.g. a node has no class defined, a [root] template is missing from the rules, etc.).

If a level successfully loads and it's possible to change graphs by changing the seed, BUT pressing "Generate" does nothing – it means something goes wrong when converting the graph into a tilemap. There are currently no interface alerts for those cases, but it's likely a result of an error in the map rules (e.g. a container appears as a final node, a road connects to a non-existing label, etc.)

Since it's an early version, it's also possible the script might crash during loading or generation. One scenario where this might occur is when the rules are self-referential and would generate infinitely long graphs, but there's also a chance of other unpredictable and exciting bugs.

Nodes and Paths

Nodes

Every map is generated from a graph (more specifically, a hierarchical tree structure). Every node of that graph corresponds to a certain area of the final map (a node may represent a single tree, a forest containing that tree as well as other trees, or the entire map). Nodes can contain one or more child nodes. Every node also contains one parent node, unless it's the top node labelled 'root', which represents the map itself.

Nodes belong to one of two basic types based on whether or not they have child nodes:

1. **Leaf nodes** - nodes without children, which represent predefined structures of a certain size and material, e.g. trees, houses, rocks, decorations, fixed-size areas made of specific material.
2. **Containers** – arrangements of child nodes, e.g. forests containing trees, seas containing islands, towns containing buildings. Containers can contain any number of leaf nodes or other containers (or both). Containers have no fixed size – their size results from the area taken up by their child nodes and the container's empty space and margins.

So in the graph, leaf nodes without any children correspond to low-level predefined structures, their parent nodes represent arrangements of said structures, their grandparents represent arrangements of said arrangements, and so on, until all the nodes join at the 'root' node representing the entire map.

It's worth nothing that:

- The map template is defined **from the TOP DOWN**. Example: rules require that a map node must contain 3 towns and a mountain, a town must contain 2 districts, and a district must contain 5 buildings.
- The resulting map is generated **from the BOTTOM UP**. Example: generate buildings and mountain nodes first, arrange buildings into districts, arrange districts into towns, and finally arrange 3 towns and a mountain into a map.

Every node of the graph must have a label and a class. The label is arbitrary, but can be used to connect a specific node to another specific node with a path (see "Paths" below). The class, on the other hand, is required to determine the node's size and material in the final map.

Nodes may also have two special properties defined by the map – 'central', which makes them generate in the centre of their container, and 'overworld', which changes the way their contents are arranged. For more details, see "Defining Nodes" under "Template Class Definitions".

The graph visualization lists labels on every node (truncated when too long). For containers, it also lists classes. Central nodes are marked with a little black dot to the left of the label.

Paths

While parent-child relationships between nodes determine how areas include other areas, they say nothing about how areas of the same parent are connected – by default, rooms of a building will be completely separated by walls, and features of the world map will just litter the landscape randomly.

Paths between nodes are specified in the graph **separately**. They correspond to paths, corridors and doorways on the generated map. Paths are essential for giving the map a sense of progression and creating structured layouts.

In the map template, paths can be defined by the parent node for its child nodes. Two kinds of connections can be defined:

1. **Connection between two child nodes of the same parent** – in this case, the path will run from one structure to another inside the same container (linking path).
2. **Connection between a child node and the parent node itself** – in this case, the path will run from the structure to the edge of the container (outgoing path).

Nodes with different parents CAN'T be connected with a single path, but you CAN create longer paths by nesting them, e.g. connecting two different nodes with their respective parents and connecting the parents with each other to create one long path.

Every path must specify one of four possible directions – north (n), south (s), east (e) or west (w) – for both connected nodes. These directions determine which 'exit' of a node the path connects to when it's constructed, e.g. a path may run from the south end of a forest to the east end of a dungeon, or from the south end of a building to the northern edge of the area containing that building.

Paths can have different materials, by default 'road' or 'dirt' tiles are used (depending on container).

Paths have a lot of impact on how the structures are arranged. The generator tries to place structures in such a way that the paths between them are as short as possible, so a structure connecting to the southern edge of its container is more likely to end up in the south, and a structure whose eastern end connects to the western end of another is more likely to end up to the east of it. This may not always happen, however, especially when there are many structures connecting in many different directions.

While the generator takes some precautions against it, two paths connecting different pairs of structures **CAN cross one another** within the same container or even join to form a shared path. The likelihood of this happening is much lower if the directions let structures arrange into a shape which avoids it – e.g. if a room connects with 4 other similarly-shaped rooms in 4 different cardinal directions, it will likely end up in the centre and the resulting paths won't cross. So to prevent unwanted shortcuts in larger layouts like dungeons and the world map, it might help to define path rules in such a way that the obvious routes between connected rooms would form a 'grid' where all nodes and paths have similar sizes and distances. For more details, see "Defining Paths" under "Template Class Definitions".

Note that the graph visualization doesn't include paths (to prevent clutter), but you can generate maps of specific nodes to see what their child paths would look like without generating the whole map.

Editing and Creating Map Templates

The ‘maps’ folder contains directories corresponding to different map templates, which can be loaded via the interface. The name of the folders will correspond to the map names displayed on the list.

Every map folder must contain one or more text (.txt) files containing rules for generating maps in the correct format. The order and names of files are NOT important – as long as they are .txt files and their content is properly formatted, the interface collects info from every file in the folder to build its list of node classes and templates.

The grammar for generating maps must include two sections: a list of node classes and their properties, and a list of template classes and their structure. These sections can be either put in one .txt file or split across several files (their content will be collected and combined into one set of nodes classes and templates).

The files can also include empty lines or comment lines starting with #, which will be ignored. You can also add in-line comments using # (anything after # will be ignored).

The node class section must start with a header line labelled:

```
=== STRUCTURES ===
```

Everything underneath this line is considered part of node class definitions, until the end of file, or until another header, which starts template class definitions:

```
=== TREE ===
```

If NO header line appears before text in the file, its content is considered to contain template class definitions by default. Templates take more space, so for longer maps, you can split them across multiple files without any headers this way.

Here’s a short example of a map template defining both node classes and a ‘root’ template class:

```
=== STRUCTURES ===
tree            2/2/2, forest
forest          0/0/1, grass
cave            0/0/1, rock

=== TREE ===
[root]
cave_containing_forest (cave)
        small_forest (forest)
                tree1 (tree)
                tree2 (tree)
                tree3 (tree)
```

Node Class Definitions

Every node in the grammar will need to have a certain class assigned. These classes have to be defined in the node class definition section under the '=== STRUCTURES ===' header.

A node class definition needs to provide 5 values in a single line in the following format: the name of the class, and then (after a "break" made of one or more tabs/spaces), 3 more values for size and the name of material.

```
class_name          width/height/margin, material
```

What these values mean:

- class_name – An arbitrary unique name of the class which will be used to classify nodes in the graph. The name cannot include spaces, tabs or the # sign.
- width, height – Number of tiles filled by a structure with this class horizontally and vertically. When set to 0, the class is considered a container and will not have a fixed size, but its size will depend on the amount of space required to generate its child nodes.
- margin – Minimum amount of outer space between this and other nodes. For any neighbouring structures, the distance should be equal to the longest required margin of either. Applies to both structures and containers. The 'margin' space is always filled with the parent container's material.
- material – Name of the material which the structure's space will be filled with (see "Materials" for a closed list of all possible values).

Important: sizes and margins do NOT guarantee that the resulting structure will have the exact size in the final map, or even that it will appear at all. Generation involves a lot of post-processing and tweaking, and some results may be adjusted or overwritten in the second step. For example, "orphaned" tiles like 1-tile buildings or bits of grass among other terrain will be cleared up, the margin of water container will be expanded to account for shores, etc.

The **distinction between leaf nodes and containers** is automatically decided on whether or not a class has a size specified. If the class specifies width and height of 0, it's considered a container node and cannot occur unless it includes some child nodes down the line. Otherwise, it's considered a leaf node and cannot have child nodes.

Breaking this rule when defining rules (e.g. creating a graph where a leaf node has children, or where a container node appears without any children at the bottom of the structure) will result in errors, since every branch of a level graph must eventually end in leaf nodes, but it can contain an arbitrary number of nested containers along the way.

Examples of Node Classes

Here is an example where 5 different node classes are defined for the map:

```
=== STRUCTURES ===
tree            2/2/2, tree
temple          7/5/3, building
pebble          1/1/0, rock
forest          0/0/1, grass
lake            0/0/0, water
```

Assigning the classes defined above to specific nodes would result in the nodes acting as, respectively:

- a 2x2 structure made of 'tree' tiles, generating at least 2 tiles away from other structures
- a 7x5 structure made of 'building' tiles, generating at least 3 tiles away from other structures
- a 1-tile structure made of a 'rock' tile, which can directly touch other structures
- a container with a surface made of 'grass' material, which must include other structures and will generate at least 1 tile away from other structures
- a container with a surface of 'water' material, which can freely touch other structures

An example including template definitions as well (a 'root' template) – cave, forest & trees:

```
=== STRUCTURES ===
tree            2/2/2, forest
forest          0/0/1, grass
cave            0/0/1, rock

=== TREE ===
[root]
cave_containing_forest (cave)
        small_forest (forest)
                tree1 (tree)
                tree2 (tree)
                tree3 (tree)
```

In the example above, 'tree' is a leaf structure (a 2x2 tree), while 'forest' and 'cave' are containers made of grass and rock materials. The map starts from a node with the label 'cave_containing_forest' of class 'cave', which contains another node with label 'small_forest' of class 'forest', which in turn contains 3 nodes labelled 'tree1', 'tree2' and 'tree3', all of which are assigned the class 'tree', making them 2x2 trees which require 2 tiles of space away from one another.

Note that the forest is made of 'grass' rather than 'forest' in order to create empty space between trees. Using 'forest' as material would create solid space filled entirely with trees.

List of Materials

Because the final step of generating a map uses tile graphics loaded from an image (tileset.png), the range of available materials is limited. The full list of working materials which can be used for node classes (and as road materials) is as follows (basic materials on the left, objects on the right):

- grass
- forest
- water
- building
- wall
- road
- plaza
- rock
- mountain
- destructible
- dirt

- chest
- pc
- npc
- npc2
- npc3
- enemy
- boss

Many of the materials have special properties when it comes to the final outcome of map generation.

- ‘grass’ will fill unused empty space away from its child nodes and roads with trees.
- ‘water’ will generate shore tiles between itself and other materials.
- ‘building’ structures will contain a 1-tile façade at the bottom in addition to the roof and generate doors where paths enter them from the south. 2x2 buildings become pillars, 1-tile-wide buildings become fences.
- ‘wall’ structures will adjust shape depending on the containers they’re inside. This is somewhat unreliable, but aims to make the result look more like a stone wall built around a city in a semi-logical way rather than a giant cube of concrete dumped from a helicopter to contain a city’s zombie outbreak. 1-tile-wide walls become fences.
- ‘plaza’ and ‘road’ materials are both cobblestone and can be used somewhat interchangeably.
- ‘rock’ and ‘mountain’ are both rock materials, the difference is in their postprocessing in the end result when surrounded by other materials (mountains will be smoothened to blend with surrounding terrain, rocks will always fill all of the required space).
- ‘dirt’ is specifically a material for interiors of rock structures (caves or dungeons) and doesn’t currently transition nicely into other materials, like grass.
- ‘chest’ and all the materials listed after it (chest, player character, non-player characters…) are special materials which generate objects and are handled separately from the tilemap. In practice, they’re just images drawn over the containers they’re in, rather than filling space with different material. The size of chests is best defined as 2/2 and the size of ‘character’ objects is best defined as 1/2, larger space will just put the images in the upper right corner and not change anything else.

Empty unaccessible space will be “blended” between materials, e.g. distant space away from solid nodes may end up filled with water, rock or trees depending on whether or not it’s the closest to water, rock/mountain, or grass/forest structures. If the level is big enough, distant space or space between non-neighbouring grass-surface structures will also end up filled with rocks to create mountains further behind the trees.

Template Class Definitions

A template represents a possible part of the graph. When a template class is defined, the structure following it can be reused and inserted at any other point of the graph if a node label is assigned a class with the same name – either inside the root node or inside other templates.

Templates are the main mechanism for generating and randomizing the map. Referencing templates inside other templates allows for generating large parts of the map without repeating content, and the more templates a template class allows, the more variety the resulting maps will have.

A template definition must start with a [template_class] header in square brackets, followed by a representation of the graph structure.

```
[template_class]
top_node_label (top_node_class)
        …
```

Example of a template defining the structure and contents of a room:

```
[dungeon_room]
dungeon_room (dirt_background)
        treasure_chest (chest)
        rock1 (small_rock)
        rock2 (small_rock)
        rock3 (small_rock)
        rock4 (large_rock)
```

Root

Every map template must contain at least one template class called [root], which corresponds to the top node of the map.

It's possible to include ONLY the root template and simply define the entire map graph inside it. In this case, the maps for different seeds will have very little variety – the map below will always arrange two grassy 'forest' areas, one of which contains 2 trees and the other contains 3 trees:

```
[root]
area (grass_background)
        forest1 (grass_background)
                tree1 (tree)
                tree2 (tree)
        forest2 (grass_background)
                tree1 (tree)
                tree2 (tree)
                tree3 (tree)
```

Nesting Templates

It's also possible to split the map into separate templates representing specific areas and make the root reference them by including "leaf" nodes with the class corresponding to [template_class]. For example, the following example is identical with the previous one, but uses 3 separate templates:

```
[forest_with_2_trees]
forest_with_2_trees (grass_background)
        tree1 (tree)
        tree2 (tree)

[forest_with_3_trees]
forest_with_3_trees (grass_background)
        tree1 (tree)
        tree2 (tree)
        tree3 (tree)

[root]
area (grass_background)
        forest1 (forest_with_2_trees)
        forest2 (forest_with_3_trees)
```

The classes 'grass_background' and 'tree' have to be defined in the class definition section, but the classes 'forest_with_2_trees' and 'forest_with_3_trees', which appear as leaf nodes under parent 'area' in [root], are references to templates.

When a full graph for the map template above is compiled, any structure with class 'forest_with_2_ trees' will be **substituted for the entire structure** of [forest_with_2_trees]. If that structure also contained nodes whose classes are templates, those nodes would also be substituted, recursively.

Multiple Templates per Template Class

When multiple templates with the same [template_class] are included, one of the templates will be selected at random whenever a map graph is compiled, depending on the seed.

```
[forest]
forest (grass_background)
        tree1 (tree)
        tree2 (tree)

[forest]
forest (grass_background)
        tree1 (tree)
        tree2 (tree)
        tree3 (tree)

[root]
area (grass_background)
        forest1 (forest)
        forest2 (forest)
```

The graph in the example above uses the same 'forest' class for both of the root's child nodes ('forest1' and 'forest2'), but [forest] defines TWO possible templates (either a container with 2 trees, or with 3).

Maps generated based on this template will randomly substitute every 'forest' for one of these versions, so there will be either 2 or 3 trees selected for every 'forest' area, and the final map will have anywhere between 4 and 6 trees, as opposed to a total of 5 as in the previous examples.

In a complete map graph generated from a map template (as visualized by the interface), nodes which correspond with the top node of a template will have the class [template_class_number], whose properties are the same as the template's top node class, and where the 'number' references which of the candidate templates was chosen (e.g. for the example above, any child of root may have a class 'forest_1' or 'forest_2' depending on which variant was randomly selected).

Template Pooling

There is a also an optional **template pool** mechanism ensuring that various templates do not repeat within a single map. In the visual interface's implementation of the algorithm, this is disabled by default, but you can mark specific nodes to use the pool mechanism by including an asterisk following the template class name (e.g. (class*) instead of (class)). The opposite designation is (class%), which means a pool is never used for that node, even if the code is changed to use it by default.

The template pool starts as a list of all possible templates for a given template class. Every time a node with a certain template class occurs, a random template is selected for it, but that template is also removed from the pool. When another node has the same template class, the template selected for it will be one of the templates remaining in the pool (and it will also get removed afterwards). When the pool is empty, it restarts to the original value.

Using a pool means that the templates will **not repeat** unless the number of templates to select exceeds the number of templates in the pool. In the map template below, 'forest1' will either have 2 trees or 3 trees, but because the selected version is then removed from the pool, 'forest2' will always take the other template, and [root] will always have one forest with 2 trees and one with 3 trees, even though their order will be randomised.

```
[forest]
forest (grass_background)
        tree1 (tree)
        tree2 (tree)

[forest]
forest (grass_background)
        tree1 (tree)
        tree2 (tree)
        tree3 (tree)

[root]
area (grass_background)
        forest1 (forest*)
        forest2 (forest*)
```

However, if there were 3 possible ‘forest’ templates, ‘forest1’ and ‘forest2’ would only use up 2 randomly selected templates out of 3, and one would randomly remain unused.

In the opposite direction, if there were 3 ‘forest’ template classes but only 2 ‘forest’ templates, one template would be used twice, and one template would be used only once.

Template pooling can be used for handling some useful scenarios:

- It allows for randomizing placement of predefined elements without any additional instructions – if you specify 3 ‘dungeon’ templates, and insert 3 ‘dungeon*’ nodes somewhere in the map template, the map will always include 3 separate dungeons without repeating them. The same mechanism can be used to randomize item placement across the map, randomize order and connections between rooms in a dungeon, etc.
- It makes the result more diverse and coherent by maximizing the use of included possibilities, instead of omitting some entirely and only using one of the versions for every case.
- If can be scaled down if necessary by simply including the same identical templates multiple times (e.g. include every template twice), therefore increasing their presence in the pool and allowing them to repeat at the expense of omitting other templates.

Map Seed

The generation of a graph based on a map template and a seed is deterministic – the same seed with the exact same set of rules will always result in the same map.

This means that if you encounter any bugs, sharing the files and the seed used should be enough to reproduce them.

On the technical side, the ‘seed’ for the generation is not the same as the program’s random seed, and is simply used to reset that seed to specific values for specific cases – this is also why the generator’s seed must be an integer.

Randomness and Recursion

The way template substitution rules work has some interesting properties and quirks:

- It’s possible to have templates which are nothing but a single leaf node (this way, one of possible leaf nodes can be selected, e.g. a building with different sizes available).
- It’s possible to have templates which are nothing but references to other templates (this way, one of possible intepretations of a template class would be a ‘redirect’ to a random template of another class), though this mechanism only works up to a point to prevent infinite loops.
- It’s possible to have multiple templates for [root], much like for any other template class.
- It’s possible to affect the probability that a specific template will be chosen over another by including the same structure multiple times in a row as a template with the same template class (this will also work if template pooling is disabled in code).
- As in HTML, it’s sometimes possible to achieve intended results, like adding more padding to areas, by stacking containers with the same class or template class inside other another.
- Self-referential recursion (e.g. a rule calling itself, or a rule calling another which calls back the first rule) is possible in theory, but will be stopped at a certain graph depth to prevent crashing!

Graph Syntax

The syntax for representing graphs is the biggest part of writing rules for generating maps. The syntax is shared between map templates and graphs for a particular map – a map graph is generated by simply combining the content of templates into a single complete graph.

Defining Nodes

As mentioned before, every node which is required to appear must have a label and a class. The class can be one of the classes defined in the node class section, or correspond with a particular template header.

```
node_label (node_class)
```

Labels do not have to be unique (you can have the same label for multiple nodes), but labels for child nodes of the same part must be unique if you want to connect them with paths, which is often the case.

The parent-child relationship (which corresponds with areas being composed of other areas) is represented through **indentation using tabs**. A single tab means a single level of nesting, e.g. a node with 3 levels of indentation will be a child node of the last node with 2 levels of indentation. Below, 'child_node1_label' and 'child_node2_label' are both children of 'node_label'.

```
node_label (node_class)
	child_node1_label (child_node_class)
		grandchild_node_label1 (grandchild_node_class)
		grandchild_node_label2 (grandchild_node_class)
		…
	child_node2_label (child_node2_class)
		grandchild_node_label1 (grandchild_node_class)
		grandchild_node_label2 (grandchild_node_class)
		…
	…
```

Nodes can have two special features which define their relationship with other nodes:

- **Central** (represented with '!') – The node will always generate in the centre of the container. Other child nodes will generate around it. Only one central node can be defined, defining multiple will lead to just one being selected anyway. This is useful for important structures you want to put in the middle of an area.
- **Overworld** (represented with '%') – The node's child nodes and paths will be arranged and post-processed slightly differently when the map is generated, trying to reduce and even out distances and leaving more space around roads. This feature is intended for landmass-level areas to make them more accessible.

Features are defined by providing their symbol ('!' or '%') following a comma after the label and class:

```
node_label (node_class), property
```

Example – 'house' will always be generated in the center, surrounded by trees:

```
continent (grass_background), %
        forest (grass_background)
                house (building), !
                tree1 (tree)
                tree2 (tree)
                tree3 (tree)
```

Defining Paths

Paths are defined by opening the line (after any required indentation) with the @ symbol. Any line that begins with an @ is going to be considered a definition of a path.

Paths which **link between two nodes** use the following format, specifying two node labels and directions separated with an ASCII arrow (->). Each node label must be followed by a full stop and a direction character (s/n/e/w). The direction of the two nodes don't matter, a path running in the opposite direction will look the same.

```
@node1_label.direction -> node2_label.direction
```

'direction' can take only one of four values: 'n', 's', 'e', 'w'. If other directions or no direction are used, the generation will either fail or a default 's' direction will be used instead.

Direction represents the side of the node that the paths starts or ends at it. For example, the following path will run from the south end of node1 towards the west end of node2 – so it will actually run towards the EAST:

```
@node1_label.s -> node2_label.w
```

Paths which are outgoing, i.e. **link to the edge of their parent**, use a format where the child node is specified on the left side, and only a direction is specified on the right. In this case, the left side of the path must always be a node and the right side must always be a direction (the order can't be flipped).

```
@node1_label.direction -> direction
```

In the following example, the path will run from the south of end of node1 towards the western end of its parent/container – in this case, it will actually run towards the WEST:

```
@node1_label.s -> w
```

Outgoing paths have an important restriction – they will ONLY be generated in one of two cases:

1. When the parent node has another path attaching to it from that direction – e.g. if a building connects to the east side of town, but no path connects to the town from the east side, the path towards the edge will not be generated to prevent it from leading into empty/solid space. This mechanism works for entire sequences of paths.
2. When the path's material is specified – in that case, an outgoing path will always generate.

So if there is an area containing a structure which should have a path leading to it, but the area itself may be connected to from different directions (which depend on other templates), you can define 4 connections between the structure and all 4 edge directions, and then depending on which direction the area connects to other areas, the most appropriate path will be generated to connect that entry point with the structure.

Paths MUST be defined at the same level of indentation as the nodes they connect, and they can only connect nodes of the same parent. Paths are technically children of the parent node, like its child nodes.

There is no limit to how many paths a node can have. It is very useful to connect the same node with paths to multiple other nodes, and/or one of the container's edges. It's also possible to connect nodes with multiple edges to affect their placement on the map.

```
node (node_class)
        child1 (child_node_class)
                grandchild1 (grandchild_node_class)
                grandchild2 (grandchild_node_class)
                @grandchild1.e -> grandchild2.w
                @grandchild1.s -> s
        child2 (child_node_class)
        @child1.s -> child2.s
```

Path Materials

By default, paths use 'road' material in almost all cases, except when their container's material is 'rock' or 'dirt', in which case the path's default material will also be 'dirt'.

Unlike nodes, whose materials are determined by the class, paths can have materials directly specified by including it after their definition in brackets, like so:

```
@node1_label.n -> node2_label.s (grass)
@node1_label.s -> s (dirt)
```

If such cases, the path will be made of whatever material is selected.

It he path's material is the same as the container's material, the path will be invisible, but will STILL affect placement – so connecting the north side of a structure in a 'dirt' room to the northern edge with a 'dirt' path will cause the structure to be generated in the north, which is useful for arranging maps.

Unlike nodes, whose exact order during generation is irrelevant, paths are generated in order. This means that if their materials differ, the material of more recently generated paths will cross and override material of previously generated paths.

Path width cannot be changed and is 1 tile by default. It's also worth nothing that you can also 'force' paths running through containers by creating a dummy 1x1 node with the same material as paths, marking it as a central node (!) and connecting it to two opposing edges.