Chapter 9
Representations for search-based methods

Dan Ashlock, Sebastian Risi, and Julian Togelius

Abstract One of the key considerations in search-based PCG is how to represent the game content. There are several important tradeoffs here, including those between locality and expressivity. This chapter presents several more new and in some respects more advanced representations. These representations include several representations for dungeon levels, compositional pattern-producing networks for flowers and weapons, and a way of representing level generators themselves.

9.1 No generation without representation

As discussed in Chapter 2, representation is one of the two main problems in search-based PCG, and one of the two concerns when developing a search-based solution to a content generation problem. In that chapter, we also discussed the tradeoff between direct and indirect representations (the former are simpler and usually result in higher locality, whereas the latter yield smaller search spaces) and presented a few examples of how different kinds of game content can be represented. Obviously, the discussion in Chapter 2 has only scratched the surface with regard to the rather complex question of representation. This chapter will dig deeper, partly relying on the substantial volume of research that has been done on the topic of representation in evolutionary computation [2].

In the first section of this chapter, we will return to the topic of dungeons, and show how the choice of representation substantially affects the appearance of the generated dungeon. The next section discusses the generation of maps for paper-and-pen role-playing games in particular. After that, we discuss a particular kind of representation that has seen some success recently, namely Compositional Pattern-Producing Networks, or CPPNs. As we will see, this representation can be used for both flowers and weapons, and many things in between. Finally, we will discuss how we can represent not only the game content but the content generator itself,

and search for good level generators in a search-based procedural procedural level generator generator.

9.2 Representing dungeons: A maze of choices

Dungeons or mazes (we mostly use the words interchangeably) are a topic that we have returned to several times during the book; the topic of most of Chapter 3 was dungeons, as well as the programming exercise in Chapter 2 and some of the examples in Chapter 8. The reasons for this are both the very widespread use of this type of content (including but certainly not limited to roguelike games) and the simplicity of mazes, allowing us to discuss and compare vastly different methods of generating mazes without getting lost in implementation details. It turns out that when searching for good mazes, the choice of representation matters in several different ways.

When the issue of representation arises, the goal is often enhanced performance. Enhanced performance could be improved search speed, creation of game features with desirable secondary properties that smooth ease of use, or simply fitting in with the existing computational infrastructure. In procedural content generation, there is another substantial impact of changing representation: appearance.

The pictures shown in Figure 9.1 are all level maps procedurally generated by similar evolutionary algorithms. Notice that they have very different appearances. The difference lies in the representation. All representations specify full and empty squares, but in different manners. The fitness function can be varied depending on the designer's goals and so is left deliberately vague.

Negative

The upper-left level map in Figure 9.1 starts with a matrix filled with ones. Individual loci in the gene specify where the upper left corner of a room goes and its length and height. The corridors are rooms with one dimension of length one. The red dots represent the position of a character's entrance and exit from the level. There is a potential problem with a random level having no connection from the entrance to the exit. If there is a large enough population and the representation length (number of rooms in each chromosome) is sufficient then the population contains many connected levels and selection can use these to optimise the level. This representation creates maps that look like mines.

Binary with content

The upper-right map in Figure 9.1 is created used a simple binary representation, but with *required content*. The large room with four pillars and the symmetric room with a closet opening north and south of it are the required content. They are specified

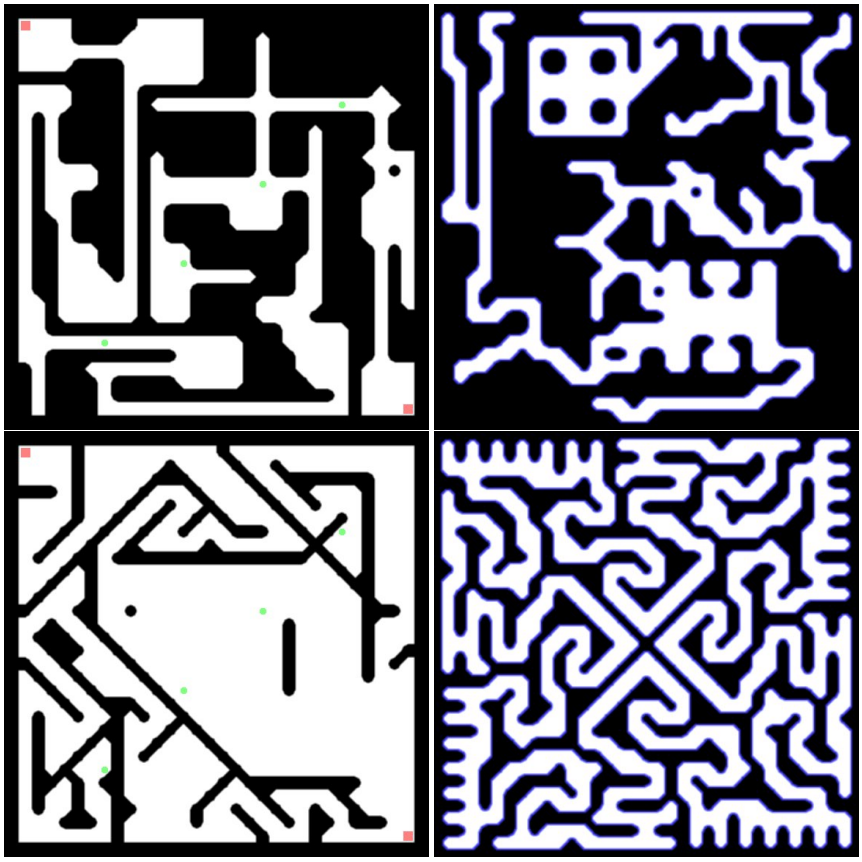


Fig. 9.1: Maps generated using four different representations. In reading order the representations are negative, binary with required content, positive, and binary with rotational symmetry. Adapted from [1]

in a configuration file. The first few loci of the chromosome specify the position of the required content elements. The remainder specify bits: 1=full 0=empty. The fitness function controls relative distances of the required content elements, and the entrances and exits. The required content represents elements the designer wants placed in an otherwise procedurally generated level. This representation generates maps that look like cave systems.

Positive

The lower-left map in Figure 9.1 uses a representation in which the loci specify walls. The starting position, direction, and maximum length of a wall are given as

well as a behavioural control. The behavioural control is 0 or 1. If it is zero it stops when it hits another wall, if it is one it grows through the other wall. This representation generates maps that look like floor plans of buildings. The example shown uses eight directions—eliminating the diagonal directions yields an even more building-like appearance.

Binary with symmetry

This representation specifies directly, as full and empty, the squares of one quarter of the level with a binary gene. Each bit specifies the full/empty status of four squares in rotationally symmetric positions. There are a large number of possible symmetries that could be imposed. The imposition of symmetry yields a very different appearance.

9.2.1 Notes on usage

An important additional factor is that we need to ensure levels are connected. In the plain binary representation, if the probability of filling a square is 0.5 then it is incredibly unlikely that there is any path between entrance and exit. Similarly, if the length of walls in the positive representation is close to the diameter of the level, connected levels are unlikely. In both cases a trick called *sparse initialisation* is used. Setting the probability of a filled square to 0.2 or the maximum length of a wall to 5 makes almost all random levels connected. They are also, on average, very highly connected and so not very good. This leaves the problem of locating good levels to whatever technique the search algorithm uses to improve levels. In the examples shown, the crossover and mutation operators of the evolutionary algorithm found this to be quite easy.

The representations shown to illustrate the impact of changing representation are relatively simple. Figure 9.2 shows a more complex version of the positive representation with three types of walls. If there are two types of players, one of which can move through water and the other of which can move through fire, this representation permits the simultaneous generation of two mazes, stone-fire and stone-water, that can be optimised for particular tactical properties. In this case the stone-water maze is easier to navigate than the stone-fire maze.

9.3 Generating levels for a fantasy role-playing game

An under-explored application of procedural content generation is the automatic creation of pen-and-paper (i.e. played without a computer) fantasy role-playing (FRP) modules. Popular examples of fantasy role-playing games include *Dungeons*

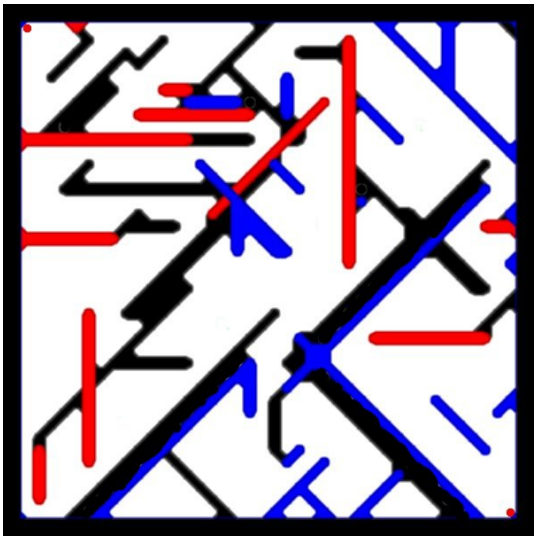


Fig. 9.2: An example of a maze, using a positive representation, with three sorts of walls: stone, fire, and water

and Dragons and the associated open gaming licence D20 systems which are used for heroic fantasy settings, *Paranoia* set in a dystopian, Orwellian future, *Champions* which is used for comic-book-style environments, and *Deadlands*, set in a haunted version of the old west. These are typically pencil-and-paper games in which players run characters and a *referee* (also known as a *game master*) interprets their actions with the help of dice, though some of these games have also been adapted into computer role-playing games. The system described here is intended to generate small adventure modules for a heroic fantasy setting.

There are a number of ways to structure generation of this type of content. The one presented here starts with required-content generation of a level. This means that the designer specifies blocks of the map, such as groups of rooms, that are forced into the level. The rest of the level is generated by filling in the area to match the relative distance between objects specified by the designer. This technique permits us to used search-based content generation to create many different levels all of which have basic properties specified by the designer. An example of a level generated in this fashion appears in Figure 9.3. Room 14 is an example of required content as is the block represented by rooms 7, 8, 11, and 12. These four rooms are a single required-content object.

Once the level map has been generated, the ACG system then automatically identifies room-sized open spaces on the map—this includes the rooms in the required content but also other spaces generated by the search algorithm optimising the level. The rooms are numbered and a combinatorial graph is abstracted from the map with rooms as vertices. The adjacency relation on the rooms is the existence of a path be-

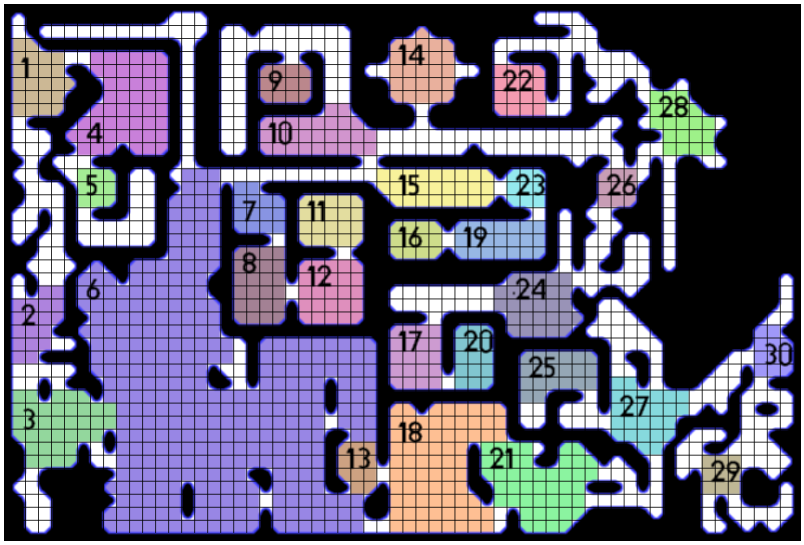


Fig. 9.3: Example of a level with automatically detected and numbered rooms

tween the rooms that does not contain a square in any other room. The graph for the map in Figure 9.3 is shown in Figure 9.4. The rooms are coloured to show which grid cells belong to them. The map with the numbered rooms, probably *sans* colours, is saved for use by the referee. The graph is handed off to the room-populating engine.

We now look at the details of the level generator. Each of these modules is an exemple and can be swapped for alternative methods with other capabilities.

9.3.1 Required content

The underlying representation for creating the levels is a simple binary one in which 1=full and 0=empty. It is modified with specifications of *required content*. An entry in the required-content configuration file looks like this:

```
12 12
111111111111
100100000001
100000000001
100111011001
100100001001
100100001001
100130001001
100111111001
100100000002
100100000002
100100000002
111111111111
```

The object specified is a 12×12 area. The representation specifies the position in the level of the upper-left-hand corner of the room, which is part of the optimisation performed by the search algorithm. The values 0,1 are mapped directly into the level, forcing values. The value 2 means that those squares are specified by the binary gene used to evolve the level. This means that some of the squares in the required content are seconded to the search algorithm. The 3 is the same as a zero—empty space—but it marks the checkpoint in the required-content object from which distances are measured. Distances are computed by dynamic programming and the fitness function uses distances between checkpoints as part of the information needed to compute fitness.

9.3.2 Map generation

The map is generated by an evolutionary algorithm. The chromosome has $2N$ integer loci for N required-content objects that are reduced modulo side length to find potentially valid places to put required-content objects. If required-content objects overlap, the chromosome is awarded a maximally bad fitness. The remainder of the position in the map, including 2's in required content, are specified by a binary gene. This gene is initialised to 20% ones, 80% zeros to make the probability the map is connected high. This is *sparse initialisation*, described earlier. The fraction of ones in population members is increased during evolution by the algorithm's crossover and mutation operators.

9.3.3 Room identification

The room identification algorithm contains an implicit definition of what a room is. The rooms appearing in the required content must satisfy this implicit definition—if

not they will not be identified as rooms. For that reason a relatively simple algorithm is used to identify rooms.

Room identification algorithm

```
N=0
Scan the room in reading order
      If a 3x3 block is empty
        mark the block as in room N
        iteratively add to the room all squares with three neighbours
          already in the room
        N=N+1
      End If
End Scan
```

Once a square is marked as being part of a room, it is not longer empty, forcing rooms to have disjoint sets of squares as members. The implementation reports the squares that are members of each room and the number of squares in each room.

9.3.4 Graph generation

The rooms form the vertices of the graph of the dungeon. Earlier, a painting algorithm was used to partition space. The adjacency of rooms is computed in a very similar fashion. For each room, a painting algorithm is used to extend the room into all adjacent empty spaces until no such spaces are left. The rooms that the painting algorithm reaches are those adjacent to the room that was its focus. Each room is extended individually by painting and the paint added is erased before treating the next room. While the painting could be done simultaneously for all rooms, this might cause problems in empty spaces adjacent to more than two rooms.

The adjacency relationship has the form of a list of neighbours for each room but can be reformatted in any convenient fashion. The graph in Figure 9.4 was generated with the *GraphVis* package from an edge list—a list of all adjacent pairs of rooms.

9.3.5 Room population

The adjacency graph for the rooms is the simplest object to pass to a room population engine. The designer knows which room(s) and entrances and typically supplies this information to the population engine. The engine then does a breadth-first traversal of the graph placing lesser challenges, such as traps and smaller monsters, in the first layer, tougher monsters in the next layer, and treasure (other than that

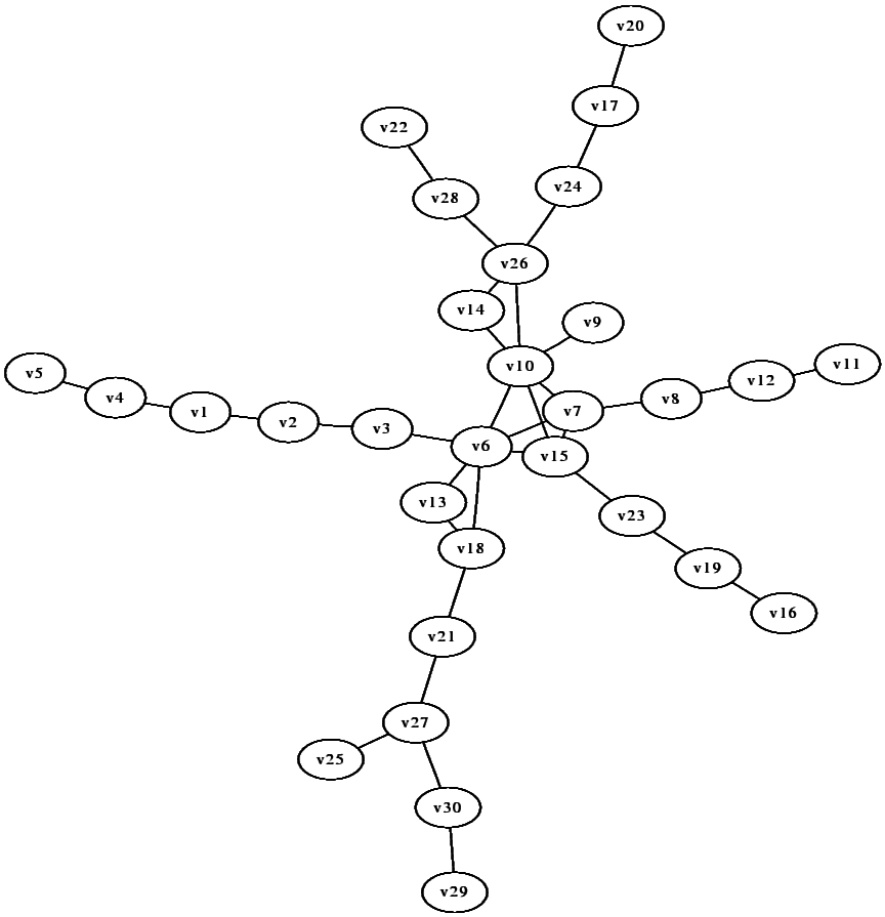


Fig. 9.4: The room adjacency graph abstraction for the level shown in Figure 9.3. Vertex v**n** represents room **n**

carried by monsters) in the next layer. Exits to the next level are typically in the last layer.

The required content is tagged if there is a special population engine connected with it. An evil temple, a crypt, a dragon's den or other boss can be placed with required content to make sure they always appear in the automatically generated level. Correct design of the fitness function ensures that the encounters appear in an acceptable sequence, even in a branching level, and so enable replayability.

The population engine needs a database of classified opponents, traps, and treasures scaled by difficulty. It can select randomly or in a fashion constrained by "mood" variables. A dungeon level in a volcano, for example, might be long on fire elementals and salamanders and short on wraiths, vampires, and other flammable

undead. A crypt, on the other hand, would be long on ghouls or skeletons and short on officious tax collectors. The creation of the encounter database, especially a careful typing system to permit enforcement of mood and style, is a critical portion of the level creation. The database needs substantially more encounters not associated with required content than it will use in a particular instance of the output of the level generator.

9.3.6 Final remarks

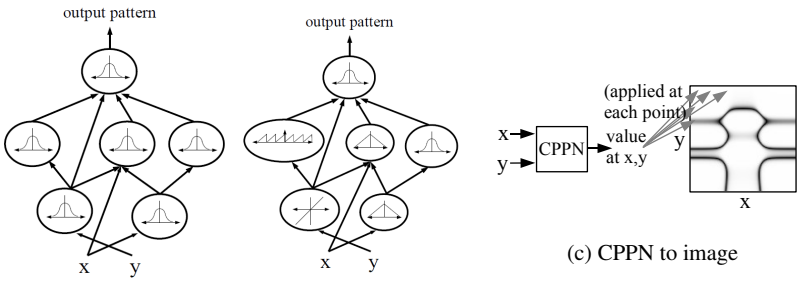
The FRP level generator described here is an outline. Many details can only be filled in when it is united with a particular rules system. The level generator has the potential to create multiple versions of a level and so make it more nearly replayable even when one or more of the players in a group has encountered the dungeon before. While fully automatic, the system leaves substantial scope for the designer in creating the required content and populating the encounter database.

9.4 Generating game content with compositional pattern-producing networks

In Chapter 5 we saw how grammars such as L-systems can create natural-looking plants, and learned that they are well suited to reproducing self-similar structures. In this chapter we will look at a different representation that also allows the creation of lifelike patterns, called *compositional pattern-producing networks* (CPPNs) [10]. Instead of formal grammars, CPPNs are based on artificial neural networks. In this section, we will first take a look at the standard CPPN model and then see how that representation can be successfully adapted to produce content as diverse as weapons in the game *Galactic Arms Race* [4] and flowers in the social videogame *Petalz* [7]. In *Petalz*, a special CPPN encoding enables the player to breed an unlimited number of natural-looking flowers that are symmetric, contain repeating patterns, and have distinct petals.

9.4.1 Compositional pattern-producing networks (CPPNs)

Because CPPNs are a type of artificial neural network (ANN), we will first introduce ANNs and then show that can modify them to produce a variety of different content. ANNs are computational models inspired by real brains that are capable of solving a variety of different tasks, from handwriting recognition and computer vision to robot control problems. ANNs are also applied to controlling NPCs in games and can even serve as PCG evaluation functions. For example, neural-network-based



(a) Neural network (b) CPPN (c) CPPN to image

Fig. 9.5: While traditional ANNs typically only have Gaussian or sigmoid activation functions (a), CPPNs can use a variety of different function, including sigmoids, Gaussians, and sines (b). The CPPN example in this figure takes two arguments x and y as input, which can be interpreted as coordinates in two-dimensional space. Applying the CPPN to all the coordinates and drawing them with an ink intensity determined by its output results in a two-dimensional image (c). Adapted from [4]

controllers can be trained to drive like human players in a car-racing game to rate the quality of a procedurally generated track [12].

An ANN (Figure 9.5a) is an interconnected group of nodes (also called neurons) that can compute values based on external signals (e.g. infrared sensors of a robot) by feeding information through the network from its input to its output neurons. Neurons that are neither input nor output neurons are also called hidden neurons. Each neuron i has an activation level y_i that is calculated based on all its incoming signals x_j scaled by connection weights w_{ij} between them:

$$y_i = \sigma\left(\sum_j^N w_{ij}x_j\right), \tag{9.1}$$

where σ is called the activation function and determines the response profile of the neuron. In traditional ANNs the activation function is often the sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-kx}}, \tag{9.2}$$

where the constant k determines the slope of the sigmoid function. The behaviour of an ANN is mainly determined by its architecture (i.e. which neurons are connected to which other neurons) and the strengths of the connection weights between the neurons.

While ANNs are usually used for control or classification problems, they can also be adapted to produce content for games. CPPNs are a variation of ANN that function similarly but can have a different set of activation functions [10]. Later we will see how special kinds of CPPNs can produce flowers in the *Petalz* videogame and weapons in GAR. While CPPNs are similar to ANNs, they have a different

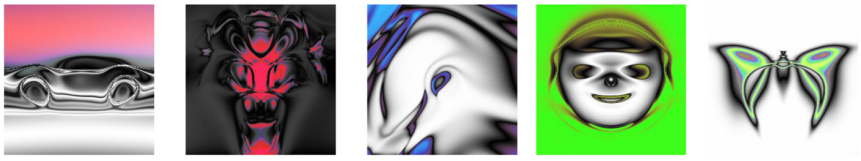


Fig. 9.6: Examples of collaboratively evolved images on Picbreeder. Adapted from [9]

terminology because CPPNs are mainly used as pattern generators instead of as controllers. Let us now take a deeper look at the differences in implementation and applications between CPPNs and ANNs.

Instead of only sigmoid or Gaussian activation functions, which we can also find in ANNs (Figure 9.5a), CPPNs can include a variety of different functions (Figure 9.5b). The types of functions that we include has a strong influence on the types of patterns and regularities that the CPPN produces. Typically the set of CPPN activation functions includes a periodic function such as sine that produces segmented patterns with repetition. Another important activation function is the Gaussian, which produces symmetric patterns. Both repeating and symmetric patterns are common in nature and including them in the set of activation functions allows CPPNs to produce similar patterns artificially. Finally, linear functions can also be added to produce patterns with straight lines. The activation of a CPPN follows the ANN activation we saw in Equation 9.1, except that we now have a variety of different activation functions.

Additionally, instead of applying a CPPN to a particular input only (e.g. the position of an enemy) as is typical for ANNs, CPPNs are usually applied across a broader range of possible inputs, such as the coordinates of a two-dimensional space (Figure 9.5c). This way the CPPN can represent a complete image or as we shall see shortly also other patterns like flowers. Another advantage of CPPNs is that they can be sampled at whatever resolution is desired because they are compositions of functions. Successful CPPN-based applications include Picbreeder [9], in which users from around the Internet collaborate to evolve pictures, EndlessForms [3], which allows users to evolve three-dimensional objects, and MaestroGenesis [5], a program that enables users to generate musical accompaniments to existing songs. Figure 9.6 shows some of the images that were evolved by users in Picbreeder, which demonstrate the great variety of patterns CPPNs can represent. The CPPNs encoding these images and the other procedurally generated content in this chapter are evolved by the NEAT algorithm, which we will now examine more closely.

9.4.2 Neuroevolution of augmenting topologies (NEAT)

NEAT [11] is an algorithm to evolve neural networks; since CPPNs and ANNs are very similar, the same algorithm can also evolve CPPNs. The idea behind NEAT is that it begins with a population of simple neural networks or CPPNs that have no initial hidden nodes, and over generations new nodes and connections are added through mutations. The advantage of NEAT is that the user does not need to decide on the number of neurons and how they are connected. NEAT determines the network topology automatically and creates more and more complex networks as evolution proceeds. This is especially important for encoding content with CPPNs because it allows the content to become more elaborate and intricate over generations. While there are other methods to also evolve ANNs, NEAT is a good choice to evolve CPPNs because it worked well in the past in a variety of different domains [9, 5, 11, 4], and it is also fast enough to work in real-time environments such as interactive games.

9.4.3 CPPN-generated flowers in the Petalz videogame

Petalz [7] is a Facebook game in which procedurally generated content plays a significant role. The player can breed a collection of unique flowers and arrange them on their balconies (Figure 9.7). A flower context menu allows the player, among other things, to create new offspring through pollination of a single flower, or to combine two flower genomes together through cross-pollination. In addition to interacting with the flower evolution, the player can also post their flowers on Facebook, sell them in a virtual marketplace, or send them as gifts to other people. An important aspect of the game is that once a player purchases a flower, he can now breed new flowers from the purchased seed, and create a whole new lineage. Recently, *Petalz* was also extended with collection-game mechanics that encourage players to discover 80 unique flower species [8].

The flowers in *Petalz* are generated through a special kind of CPPN. Because the CPPN representation can generate patterns with symmetries and repetition, it is especially suited to generating natural-looking flowers with distinct petals. The basic idea behind the flower encoding is to first deform a circle to generate the shape of the flower and then to colour that resulting shape based on the CPPN-generated pattern. In contrast to the example we saw in Figure 9.5c, we now input polar coordinates $\{\theta, r\}$ into the CPPN (Figure 9.8) to generate radial flower patterns. Then we query the CPPN for each value of θ by inputting $\{\theta, 0\}$. However, instead of inputting θ into the CPPN directly, we input $sin(P\theta)$, which makes it easier for the CPPN to produce flower-like images with radial symmetry in the form of their petals. Parameter P can also be adjusted to create flowers with a different maximum numbers of petals. In the first step of the flower-generating algorithm the outline of the flower is determined, i.e. a radius value r_{max} for each θ value is calculated. In the next step, the RGB colour pattern of the flower's surface is determined by querying

Fig. 9.7: Screenshot from a *Petalz* balcony that a player has decorated with various available flower pots and player-bred flowers. Adapted from [7]

each polar coordinate between 0 and r_{max} with the same CPPN. Finally, the CPPN also allows for the creation of flowers with different layers, which reflects the fact that flowers in nature often have internal and external portions. This feature is implemented through an additional CPPN input L that determines the current layer that is being drawn. The algorithm starts by drawing the outermost layer and then each successive layer is drawn on top of the previous layers, scaled based on its depth. Because the same CPPN is determining all the layers, the different patterns can share regularities just like the different layers in real flowers.

Figure 9.9 shows examples of flowers evolved by players in *Petalz*. The CPPN-based encoding allows the discovery of a great variety of aesthetically pleasing flowers, which show varying degrees of complexity.

9.4.4 CPPN-generated weapons in Galactic Arms Race

Galactic Arms Race [4] is another successful example of a game using procedurally generated content and interactive evolution. Procedurally generated weapon projectiles, which are the main focus of this space shooter game, are evolved interactively based on gameplay data. The idea behind the interactive evolution in GAR, which was briefly discussed in Chapter 1, is that the number of times a weapon is fired is considered an indication of how much the player enjoys that particular weapon.

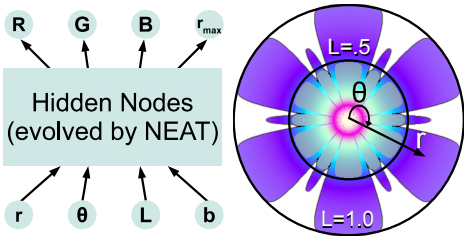


Fig. 9.8: The flower-encoding CPPN in Petalz has four inputs: polar coordinates r and θ, current layer L and bias b. The first three outputs determine the RGB colour values for that coordinate. In the first step of the algorithm the maximum radius for a given θ is determined through output r_{max}. In the next step RGB values of the flower's surface are determined by querying each polar coordinate between 0 and r_{max} with the same CPPN. The number and topology of hidden nodes is evolved by NEAT, which means that flowers can get more complex over time. From [7]

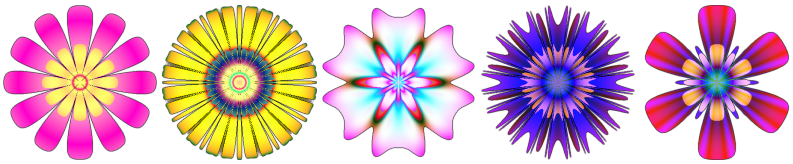


Fig. 9.9: Examples of flowers collaboratively evolved by players in the *Petalz* videogame. Adapted from [7]

As the game is played, new particle weapons are automatically generated based on player behaviour. We will now take a closer look at the underlying CPPN encoding that can generate these weapon projectiles.

Each weapon in the game is represented as a single CPPN (Figure 9.10) with four inputs and five outputs. Instead of creating a static image (Figure 9.6) or flower (Figure 9.8) the CPPNs in GAR determine the behaviour of each weapon particle over time. Each animation frame the CPPN is queried for the movement (velocity in the x and z direction) and appearance (RGB colour values) of the particle given the particle's current position in space relative to the ship (p_x, p_y) and distance d_c to its starting position. After activating the CPPN, the particles are moved to their newly determined positions and the CPPN is queried again in the next frame of animation. Evolution starts with a set of simple weapons that shoot only in a straight line and then more and more complex weapons are evolved based on the NEAT method. By adding new nodes with different activation functions, such as Gaussian and sine, interesting particle movements can evolve and the player can discover a variety of different weapons.

Figure 9.11 shows a variety of interesting weapons with vivid patterns that were evolved by players during the game. Interestingly, different weapons do not just have a different look but also tactical implications. For example, the wallmaker

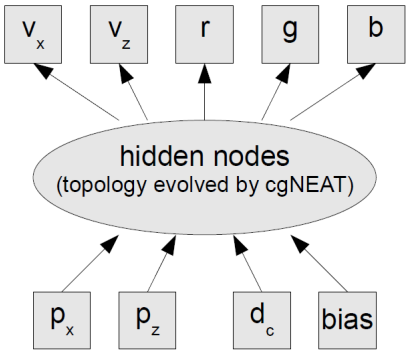
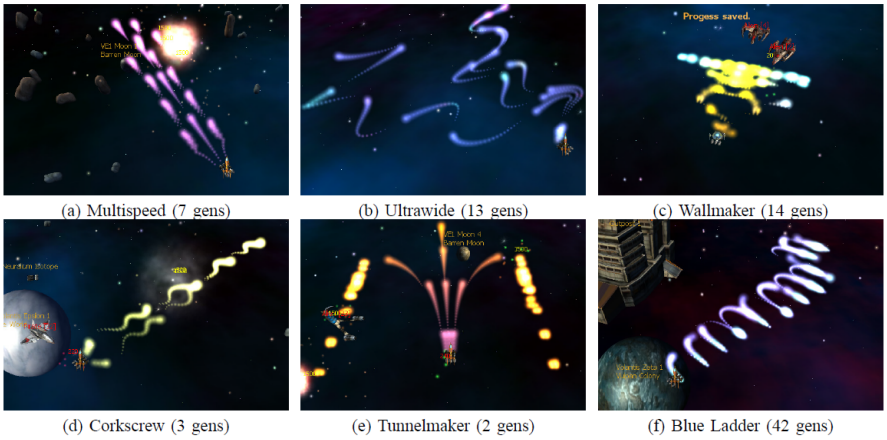


Fig. 9.10: CPPN representation of weapon projectiles in GAR. The movement of each particle is controlled by the same CPPN, which has four inputs and five outputs. The first three inputs describe the position of the particle (p_x, p_y) and the distance d_c from the location from which it was fired. After the CPPN activation, the outputs determine the particle's velocity (v_x, v_y) and RGB colour value. Adapted from [4]



(a) Multispeed (7 gens) (b) Ultrawide (13 gens) (c) Wallmaker (14 gens)

(d) Corkscrew (3 gens) (e) Tunnelmaker (2 gens) (f) Blue Ladder (42 gens)

Fig. 9.11: Examples of CPPN-encoded weapons evolved in the *Galactic Arms Race* videogame. Adapted from [4]

weapon (Figure 9.11c) can create a wall of particles in front of the player, which allows for a more defence-oriented play. Other guns such as the multispeed weapon (Figure 9.11a) can be used in tactical situations in which a more offence-oriented approach is needed.

9.5 Generating level generators

Our final example of an advanced representation is not a representation of a particular type of game content, but rather of a level generator itself. This example, due to Kerssemakers et al. [6], views the content generator itself as a form of content, and creates a generator for it, a procedural procedural content generator generator (PPLGG). Specifically, it is a search-based generator that searches a space of generators, each of which generate levels for *Super Mario Bros.* in the Mario AI Framework.

As usual, we can understand a search-based generator in terms of representation and evaluation. The evaluation in this case is interactive: a human user looks at the various content generators, and chooses which of them (one or several) will survive and form the basis of the next generation. In order to be able to assess these content generators, the user can look at a sample of ten different levels generated by each content generator, and play any one of them; the tool also gives an estimate of how many of these levels are playable using simulation-based evaluation. Complementarily, the user can see a "cloud view" of each generator, where a number of levels generated by that generator are superimposed so that patterns shared between the levels can be seen (Figure 9.12). Figure 9.13 shows a single level in condensed view, and part of the same level in game view, where the user can actually play the level.

More interesting from the vantage point of the current chapter is the question of representation. How could you represent a content generator so as to create a searchable space of generators? In this case, the answer is that the generator is based on agents (each generator contains between 14 and 24 agents), and the generator genome consists of the parameters that define how the agents will behave. During generation, the agents move concurrently and independently, though they affect each other indirectly through the content they generate.

The genome consists of specifications for a number of agents. An agent is defined by a number of parameters, that specify how it moves, for how long, where and when it starts, how it changes the level and in response to what. The agent's behaviour is not deterministic, meaning that any collection of agents (or even any single agent) is a level generator that can produce a vast number of different levels rather than just a generative recipe for a single level.

The first five parameters below are simple numeric parameters that consist of an integer value in the range specified below. The last five parameters are categorical parameters specifying the logic of the agent, which might be associated with further parameters depending on the choice of logic. The following is a list of all parameters:

- **Spawn time [0-200]**: The step number on which this agent is put into the level. This is an interesting value as it allows the sequencing of agents, but still allows for overlap.
- **Period [1-5]**: An agent only performs movement if its lifetime in steps is divisible by the period.

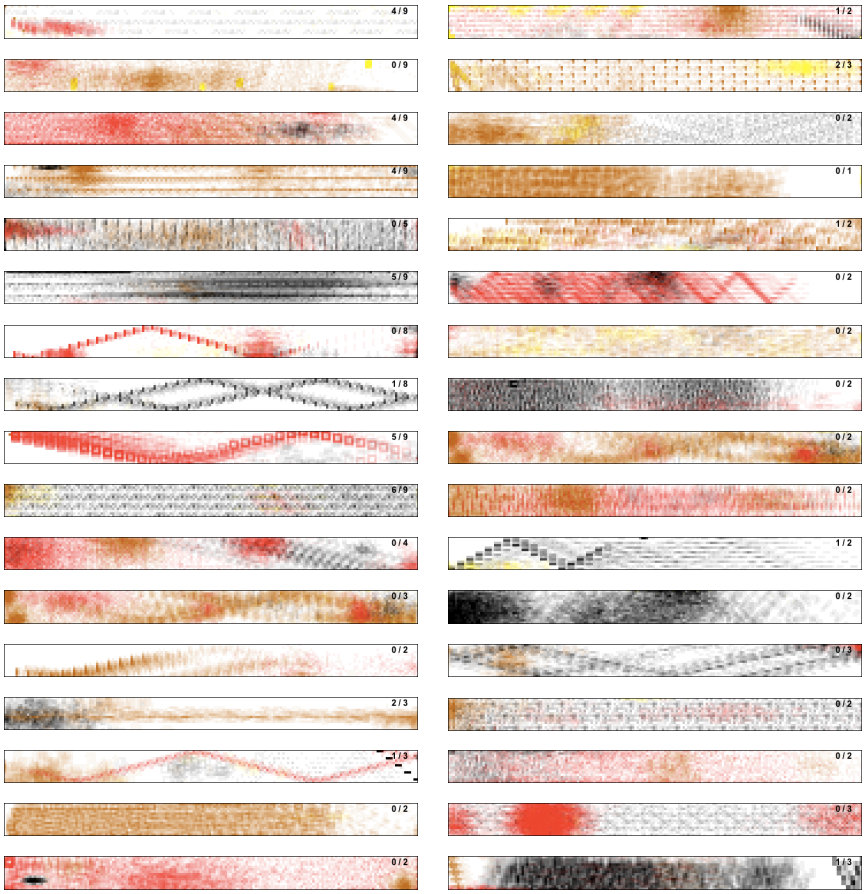


Fig. 9.12: A cloud view of several content generators. Each content generator is represented by a "cloud" consisting of multiple levels generated by that generator, overlaid on top of each other with low opacity. Adapted from [6]

- **Tokens [10-80]**: The amount of resources available to the agent. One token roughly equals a change to one tile.
- **Position [Anywhere within the level]**: The center of the spawning circle in which the agent spawns.
- **Spawn radius [0-60]**: The radius of the spawning circle in which the agent spawns.
- **Move style**: the way the agent moves every step.
  - follow a line in a specified direction (of eight possible directions) with a specified step size.
  - take a step in a random direction.

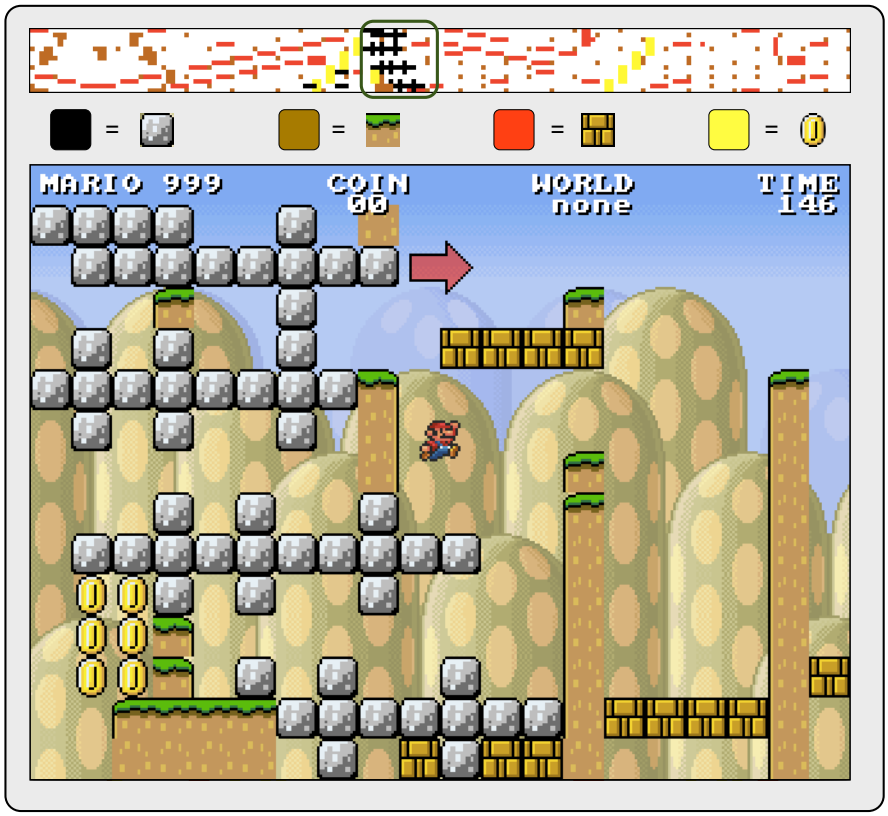


Fig. 9.13: A single generated level, and a small part of the same level in the game view. Adapted from [6]

- **Trigger state**: The condition for triggering an action, checked after each movement step.
  - always.
  - when the agent hits a specified type of terrain.
  - when a specified rectangular area is full of a specified tile type.
  - when a specified area does not contain a specified tile type.
  - with a specified probability.
- **Boundary movement**: The way the agent handles hitting a boundary.
  - bounce away.
  - go back to start position.
  - go back to within a specified rectangular area around the start position.
- **Action type**: The type of action performed if it is triggered.

- place a specified tile at position.
- place a rectangular outline of specified tiles and size around position.
- place a filled rectangle of specified tiles and size around position.
- place a circle of specified tiles and size around position.
- place a platform/line of specified tiles and size at position.
- place a cross of specified tiles and size at position.

Given that the starting position of agents implies a large amount of randomness, and a number of other behaviours imply some randomness, the same set of agents will produce different levels each time the generator is run. This is what makes this particular system a content generator generator rather than "just" a content generator.

9.6 Summary

This chapter addressed the issue of content representation within search-based PCG. How content is represented affects not only how effectively the space can be searched, but also biases the search process towards different parts of the search space. This can be illustrated by how different ways of representing a dungeon or maze yield end products that look very different, even though they are evolved to satisfy the same evaluation function and reach similar fitness. Representations can be tailored to extend the search-based paradigm in various ways, for example by providing "required content" that cannot be altered by the variation operators of the search/optimisation algorithm. More complicated representations might require a multi-step genotype-to-phenotype mapping that can be seen as a PCG algorithm in its own right. For example, compositional pattern-producing networks (CPPNs) are a form of neural network that maps position in some space to intensity, colour, direction or some other property of pixels or particles. This is an interesting content generation algorithm in itself, but can also be seen as an evolvable content representation. Taking this perspective to its extreme, we can set out to evolve actual content generators, and judge them not on any single content artefact they produce but on samples of their almost infinitely variable output. The last example in this chapter explains one way this can be done, by representing Mario AI level generators as parameters of agent-based systems and evolving those.

References

1. Ashlock, D., Lee, C., McGuinness, C.: Search-based procedural generation of maze-like levels. IEEE Transactions on Computational Intelligence and AI in Games **3**(3), 260–273 (2011)
2. Ashlock, D., McGuinness, C., Ashlock, W.: Representation in evolutionary computation. In: Advances in Computational Intelligence, pp. 77–97. Springer (2012)

3. Clune, J., Lipson, H.: Evolving three-dimensional objects with a generative encoding inspired by developmental biology. In: Proceedings of the European Conference on Artificial Life (2011)
4. Hastings, E., Guha, R., Stanley, K.: Evolving content in the Galactic Arms Race video game. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp. 241–248 (2009)
5. Hoover, A.K., Szerlip, P.A., Norton, M.E., Brindle, T.A., Merritt, Z., Stanley, K.O.: Generating a complete multipart musical composition from a single monophonic melody with functional scaffolding. In: Proceedings of the 3rd International Conference on Computational Creativity, pp. 111–118 (2012)
6. Kerssemakers, M., Tuxen, J., Togelius, J., Yannakakis, G.N.: A procedural procedural level generator generator. In: Proceedings of the IEEE Conference on Computational Intelligence and Games, pp. 335–341 (2012)
7. Risi, S., Lehman, J., D'Ambrosio, D.B., Hall, R., Stanley, K.O.: Combining search-based procedural content generation and social gaming in the Petalz video game. In: Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (2012)
8. Risi, S., Lehman, J., D'Ambrosio, D.B., Stanley, K.O.: Automatically categorizing procedurally generated content for collecting games. In: Proceedings of the Workshop on Procedural Content Generation in Games (2014)
9. Secretan, J., Beato, N., D'Ambrosio, D., Rodriguez, A., Campbell, A., Folsom-Kovarik, J., Stanley, K.: Picbreeder: A case study in collaborative evolutionary exploration of design space. Evolutionary Computation **19**(3), 373–403 (2011)
10. Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines **8**(2), 131–162 (2007)
11. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10**(2), 99–127 (2002)
12. Togelius, J., De Nardi, R., Lucas, S.M.: Towards automatic personalised content creation for racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp. 252–259 (2007)