Incorporating Required Structure into Tiles.

Cameron McGuinness and Daniel Ashlock

***Abstract*—Search based procedural content generation uses search techniques to locate high-quality content elements for use in games. This study extends an evolutionary-computation based system to generate tiles used to assemble floor plans for levels in a video game as well as evolving assembly plans. The separate evolution of assembly plans and tiles yields a decomposition of the level generation problem that saves substantial time. The tiles used in this study use a more sophisticated fitness function than earlier studies and also permit the user to specify features, e.g. rooms of a particular shape or short or long transit distances between two points in the tile. Evolutionary computation is used as an off-line tool to generate libraries of both tiles and assembly plans. Systems for rapidly assembling tile libraries can then be used to generate large levels on demand with combinatorially huge numbers of level plans available. The new fitness function can be used to control the topology of the tiles and optionally permits the incorporation of enforced symmetry in tiles. Symmetric tiles are found to have a visually striking appearance. Control is achieved by requesting that different pairwise distances between user-specified checkpoints be minimized, maximized, or left free. This study explores the new fitness functions and demonstrates the variety of tiles that can be generated as well as assembling exemplary level plans from those tiles.**

Keywords: Search based procedural content generation, automatic game content generation, evolutionary computation, level generation, scalable content generation, dynamic programming.

I. Introduction

PROCEDURAL content generation (PCG) consists of finding algorithmic methods of generating content for games. Search based PCG [14] uses search methods rather than composing algorithms that generate acceptable content in a single pass. Both sorts of content generation can suffer materially from scaling problems which can be addressed by problem decomposition with an off-line and an online phase to the software.

A *tile* is a piece of a map with a regular shape, such as a square or hexagon. Regularly shaped tiles can be rapidly assembled to cover a large area. This study presents a decomposition of the problem of generating levels for use in games in which a search-based PCG engine is used to generate tiles that can be assembled to rapidly generate very large levels as well as assembly plans. The assembly plan algorithm generates libraries of assembly plans. Once generated, a library of assembly plans and a library of tiles can be used to create a combinatorially large space of maps with certifiable levels of similarity or dissimilarity.

Cameron McGuinness and Daniel Ashlock are at the Department of Mathematics and Statistics at the University of Guelph in Guelph, Ontario, Canada, N1G 2W1. email: {cmcguinn|dashlock}@uoguelph.ca

The authors thank the National Science and Engineering Research Council of Canada for their support of this research.

Automated level generation in video games can arguably be traced back to a number of related games from the 1980s (Rogue, Hack, and NetHack), the task has recently received some interest from the research community. In [12] levels for 2D sidescroller and top-down 2D adventure games are automatically generated using a two population feasible/infeasible evolutionary algorithm. In [13] multiobjective optimization is applied to the task of search-based procedural content generation for real time strategy maps. In [9] cellular automata are used to generate, in real time, cave-like levels for use in a roguelike adventure game. This study continues work done in [3] which introduced checkpoint based fitness functions for evolving maze-like levels. It also extends work from [6] which prototyped tile assembly; this study documents techniques for gaining more control over the character of individual tiles.

Assembly of tiles borrows from the author's earlier research [4] on the creation of dual mazes. A *dual maze* is a maze with multiple barrier types, e.g. stone, fire, and water. If we assume that some agents traversing the maze can swim while others are fireproof then the placement of barriers creates two separate connectivities (and hence two mazes) in the same physical space. One of the dual mazes in [4] took the form of a variable height map in which one connectivity assumed an agent could jump up or down one meter while the other can jump up or down two meters. This sort of dual maze will be used as an assembly plan for tiles. This is a reuse of the original fitness functions used to evolve the dual height maze to yield assembly plans for putting together tiles. The dual maze does not survive into the tile assembly plan, pairs of squares in the height map that are less than two meters apart cause tiles used to replace those squares to have an entrance between them. We note that the full generality of strategies for controlling the character of mazes generated in the earlier studies are available for controlling the assembly plans used in this study. Height maps represent a convenient choice of a source of assembly plans. Many other sorts of assembly plans could be swapped in their place with minimal effort.

A. Dynamic Programming

Dynamic programming [8] is an ubiquitously useful algorithm. It can be applied to align biological sequences [11], to find the most likely sequence of states in a hidden Markov model that explain an observation [15], or to determine if a word can be generated from a given context free grammar [10]. Dynamic programming works by traversing a network while recording, at each network node, the cost of arriving at that node. It is possible to record multiple costs and other factors about the path used to arrive at a node of the network.

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

In this study the network in question is composed of the accessible squares within a tile.

When the cost of a new path is not superior to one that is already found, the search is pruned, otherwise the minimum cost of reaching the node is updated and search continues onward from that node. When multiple costs are being computed, improvement in any of the cost functions merits continuation of the search. In [5], a dynamic programming algorithm simultaneously computes the number of forward moves and turns a robotic agent needs to solve a path planning problem. A single variant of a dynamic programming algorithm is used to evolve tiles in this study; it computes the distance from a checkpoint to other squares within the tile. This is then used to enumerate both pairwise distances between checkpoints and also the number of dead ends or *culs-de-sac* in a tile. A more detailed discussion of the dynamic programming used to evolve assembly plans appears in [4].

The remainder of this study is organized as follows. Section II specifies the representations used to evolve tiles. Section III specifies the experiments performed. Section IV gives and discusses the results. Section V states conclusions and outlines possible next steps.

II. Evolvable Representations

One representation is used to evolve tiles in this study while a second is used to evolve assembly plans. The representation for tiles are adapted from [3] while the assembly plan is adapted from a representation first appearing in [4]. The representation used to evolve tiles is a direct representation that specifies, with a string of bits, if a square in the grid on which the tile is constructed is full or empty. This specification is modified by the placement of required features. Other than re-interpreting it as an assembly plan, the representation used to evolve assembly plans is that given in [4].

The tiles used in this study are constructed on a 33×33 grid. This grid size is chosen because it permits the symmetric placement of a door in the middle of one side of a tile or two doors, at positions 10 and 21, equally spaced along the side, and is large enough to permit interesting structure without being too large itself. Grids with twin entrances on one side are not used in this study but are potentially useful later. A feature specification for a type of tile gives:

1) The number and position of entrances to the tile.
2) The number of required features.
3) A shape for each required feature, including checkpoints.
4) A specification, for each pair of checkpoints, if the distance between them is to be (i) minimized, (ii)left free, or (iii) maximized.

Figure 1 shows a shape specification for a required feature, a spiral passage. This passage appears in the lower-left corner of the map in Figure 4. All required features must be rectangular and filled with four symbols. The symbols have the following meaning: 0 - empty space, 1 - filled space, 2 - empty space that is a checkpoint, 3 - space to be filled in by an evolvable bit later in the representation. This latter symbol permits the designer to place non-rectangular shapes within a bounding rectangle.

```
11 11
1 3 3 3 3 3 3 3 3 3 1
1 0 1 1 1 1 1 1 1 0 1
1 0 1 0 0 0 0 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 2 1 0 1 0 1
1 0 1 0 1 1 1 0 1 0 1
1 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1
```

Fig. 1. Shape specification of a required feature. A spiral in an 11×11 window with a checkpoint in its middle and a north wall that is left free for the bitwise portion of the algorithm to fill or leave open as seems advisable.

The bit string that specifies if grids within a tile are open or full starts with a sequence of integers in the range $0 \leq n \leq 10,000$. One such integer is included for each required feature. These integers n are decoded to positions within the 33×33 tile using

$$x = n \; mod \; (33 - D_x) \tag{1}$$

$$y = (n/(33 - D_x)) \; mod \; (33 - D_y) \tag{2}$$

Where D_x and D_y are the dimensions of the required feature. The required feature is then copied into the tile at that location. There are enough bits, after the required feature position loci, to cover the entire grid. Any location which is already open or filled because of a required feature ignores the bit; otherwise the bit determines if the square will be open or filled. In the event that two required features overlap, or include the grid containing an entrance, a tile is assigned a fitness of zero.

A. Initialization Issues

Two issue arise in initialization of tiles. This first is that, if there are several required features or large required features then there is an excellent chance that a randomly generated gene for a tile will overlap those features, resulting in a fitness of zero. This problem is dealt with by discarding and re-generating tiles in the initial population until all of them have nonzero fitness.

In the earlier studies it was discovered that if the initial population was created by filling in the data structure uniformly at random with full/empty specification bits then almost all population members have bad fitness because no path exists between many pairs of checkpoints. To compensate for this *sparse initialization* is used. In the representation used for tiles, this means that grids in the tile are filled only 5% of the time in the initial population. For the height map used to generate assembly plans, initial heights are chosen with a Gaussian random variable with a mean of 3.0 and a standard deviation of 1.0. In both cases this means that the initial mazes typically have low but non-zero

fitness with ample connectivity. Reducing this connectivity and improving fitness is left to the variation operators and the selection operator.

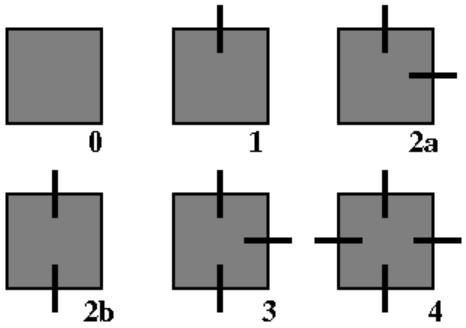

Fig. 2. With at most one entrance per tile-side there are six possible tile types, shown above. The tiles are labeled with their type designator. Any desired pattern of entrances results from a rotation or a flip of one of the above. Hashmarks denote the position of entrances.

III. Experimental Design

All the evolutionary algorithms use single tournament selection of size seven[7]. The evolutionary algorithms for all three representations use two-point crossover. The direct representation for tiles performs two-point crossover of the list of required feature positions and bits while the height map performs two-point crossover of the list of heights. For all algorithms, uniform mutation with probability $p_m = 0.01$ was used. In the direct representation for tiles, a mutated loci was simply a flipped bit and locations for required features were simply replaced with a new location. For mutation of a height while evolving assembly plans, a Gaussian random variable with a mean of 0.0 and standard deviation of 0.5 is added to mutate a loci. Each tile-generating experiment was run 30 times and the most fit tile saved to obtain libraries of 30 tiles.

Figure 2 shows all the possible patterns of entrances if at most one entrance is permitted per side of a tile. These patterns may require rotation or flipping. Use of such rotation or flipping means that instead of needing 15 tile libraries we only require five. In practice, we developed four different tile libraries of each type using different fitness functions and checkpoint specifications, to yield a greater variety of tiles.

A total of 24 experiments were performed to create tiles. The first 20 consisted of five groups of four for each of the five tile patterns with entrances given in Figure 2. The first two in each group of four had required content consisting of single checkpoints or 3×3 rooms which have little visual impact on the tiles. The second two of each group incorporated one or more rooms with a complex shape that has a substantial visual impact. The last four experiments all generated tiles with four entrances; these first two were coerced to yield tiles symmetric about both the horizontal and vertical central line of grids in the tile; the last two were coerced to yield tiles with four-fold rotational symmetry. In both cases this was accomplished by reducing the length of the gene to cover one-fourth of the tile and filling in the rest of it with the imposed symmetry. A total of six fitness functions were used. If M is the sum of distances between checkpoints whose mutual distance is to be maximized, m is the sum of distances between checkpoints whose mutual distance is to be minimized, C is the number of culs-de-sac and F is the number of filled grids then those functions are:

$$F1 : \frac{M+1}{m+1}$$
$$F2 : \frac{M+1}{m+1} + C/5$$
$$F3 : \frac{M+1}{m+1} + C/4 + F/20$$
$$F4 : \frac{M+1}{m+1} * arctan(C)$$
$$F5 : \frac{M+1}{m+1} * arctan(F)$$
$$F6 : \frac{M+1}{m+1} * arctan(F) + C/5$$

$Arctan(x)$ is a squashing function that reduces the marginal reward for more filled squares or culs-de-sac. These functions were chosen by running several preliminary experiments and choosing the squashed or linear reward that looked best. The details of the experiments are specified in Table I.

The height-map code was run 30 times with a small size, 6×6, to obtain 30 assembly plans. The settings used were, otherwise, exactly those of [4]. Examples of assembly plans are given in Figure 5.

TABLE I
Specification of experimental details. FF is fitness function; Xp is number of pairs of checkpoints for which distance is maximized; Np is the number of pairs of checkpoints for which distance is minimized, T is the tile type, C is the number of checkpoints.

#	T	C	FF	Xp	Np
1	1	4	3	3	4
2	1	4	3	3	5
3	1	5	3	3	4
4	1	5	3	3	4
5	2b	6	3	7	2
6	2b	6	1	4	2
7	2b	4	6	1	3
8	2b	4	6	4	3
9	2a	6	3	7	2
10	2a	6	3	3	6
11	2a	4	6	2	3
12	2a	4	6	2	3
13	3	4	3	3	3
14	3	4	3	4	0
15	3	4	3	3	3
16	3	4	3	3	3
17	4	5	3	8	2
18	4	10	3	12	3
19	4	6	6	2	8
20	4	6	6	2	8
21	4	4	2	4	2
22	4	4	4	0	6
23	4	4	4	2	4
24	4	4	5	4	4

IV. Results and Discussion

An example tile from each of the 24 experiments is shown in Figure 3. Note the substantial variety of tile types and

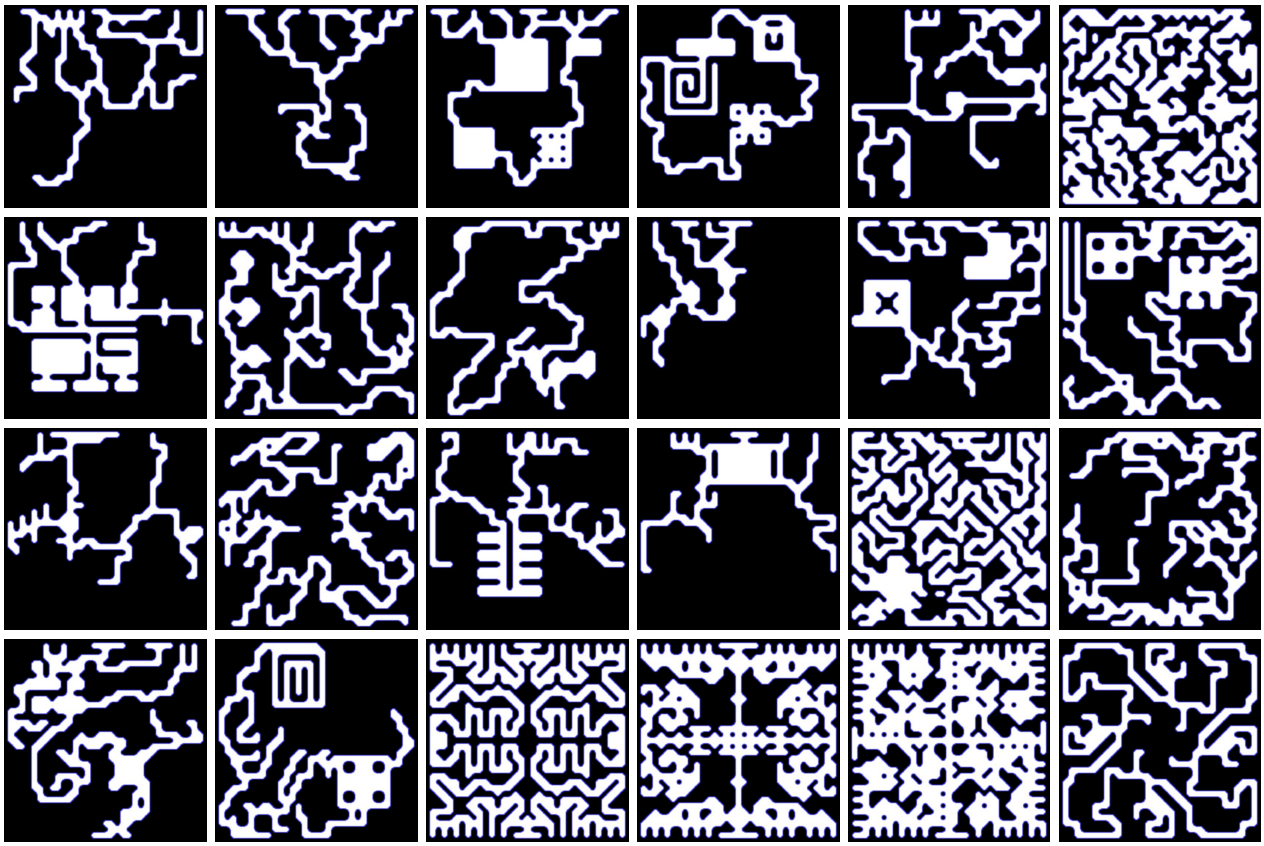

Fig. 3. Example tiles from experiments 1-24 in reading order. The tiles are bordered with black; entrances occur in the middle of a face.

densities that can be obtained with a fitness function utilizing only three types of measurements of a tile (inter-checkpoint distance, number of culs-de-sac, number of filled grids). Rewarding culs-de-sac causes there to be more dead ends in the tile. Rewarding filled grids increases the density of the tiles. Requesting that distances between checkpoints be maximized yields winding connecting paths while asking that they be minimized favors direct paths. Since checkpoints in required features can appear in various locations, the rough location of a feature can be controlled by asking that its checkpoint be far from one entrance but near another. The relative positions of required features representing rooms is also, to some degree, controlled by requests for long distances or short distances between those checkpoints.

A key point is that the pattern of requested maximized, minimized, and free distances between checkpoints gives the designer substantial control over the character of the resulting tile. The control is sufficient to ensure that any tile in a library evolved for a given specification can be swapped in for another without more than minor changes in the connectivity and relative distances within the map built from the tiles. Given an assembly plan with desired properties, it is therefore possible to rapidly assemble vast numbers of maps that realize the assembly. The assembly algorithm is linear in the size of the map, as it consists only of reading data specifying tiles into the assembly plan.

Suppose that a game designer has particular encounters; crypts, temples, dragon's lairs. These can be included in a tile design as required features. The placement of the encounter within the map can be held constant, or localized to a small area of the assembly plan, but the details of the tile containing the encounter can be varied by generating a large tile library for the specification containing the feature.

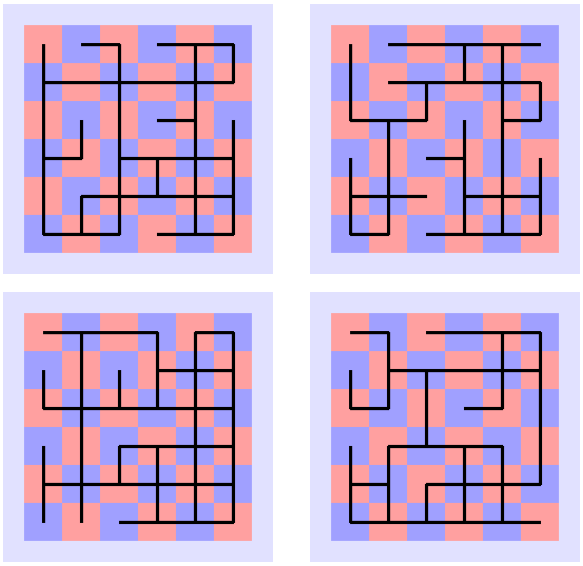

Fig. 5. Examples of Assembly Plans. Where lines cross tile boundaries, adjacent tiles both have an entrance.

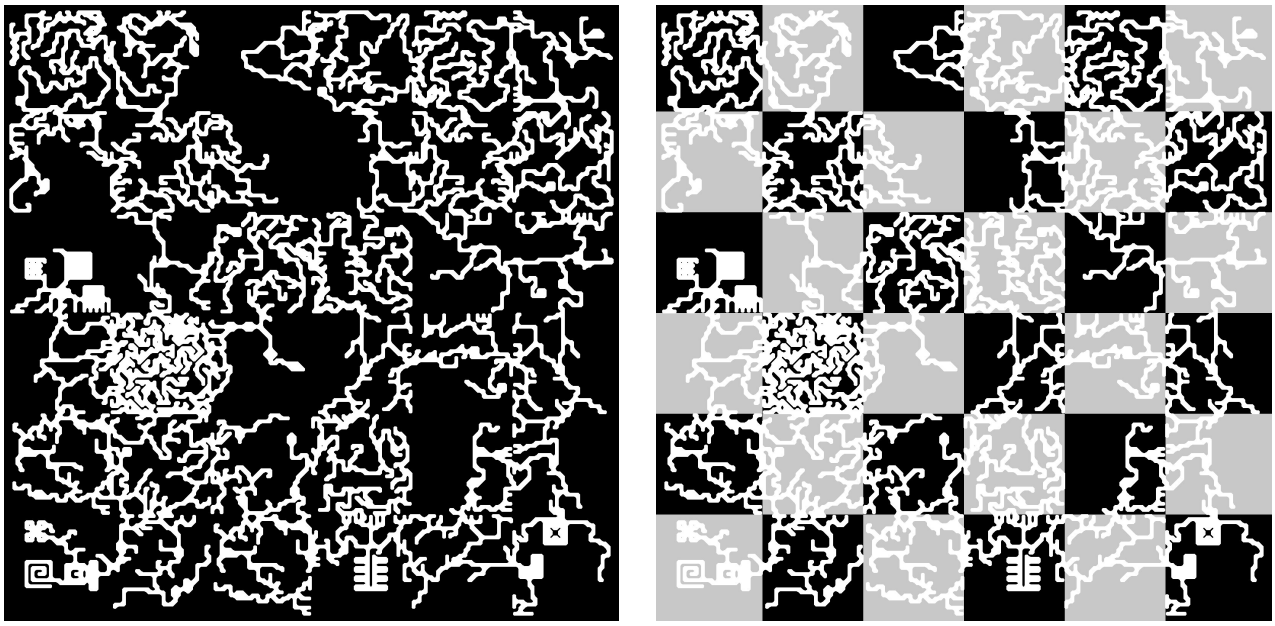

Fig. 4. A map assembled from a collection of 20 libraries of 30 tiles using a randomly selected assembly plan. The version on the left shows the complete map, the version on the right highlights the placement of individual tiles.

Once we have evolved tiles and assembly plans, the next step is assembling level maps. The choice of 6×6 plans in this study follows from a desire to have figures that still look good at the scale of a printed page. Height maps with size 20×20 appear in earlier studies and much larger height maps are not hard to generate. Figure 4 explicitly demonstrates the positioning of tiles within an assembly, using an assembly plan selected at random from the library of assembly plans. Figure 6 shows an assembled level in which the height map used to generate the level is at a constant height (yielding maximal connectivity) and all tiles in the middle 4×4 region have reflective or rotational symmetry. Figure 7 shows a map similar to the one in Figure 6 except the tiles in the middle of the assembly were evolved to reward filled squares and minimal distance between doors.

Choosing to maximize distances between some pairs of checkpoints and minimize distances between others, when the paths that realize those distances live in the same two-dimensional tile, creates conflicting goals that yield a fitness function rich in local optima. The final fitness of the best tile varies substantially within each experiment, suggesting that most of the runs are ending in local optima of the fitness function. The fitness functions themselves were chosen based on intuition about what elements were important to producing good tiles combined with trial and error to find tiles that "looked good" to the authors.

In evolutionary computation the goal is typically to find global optima while escaping local optima. In [2] it is noted that when using an evolutionary algorithm to locate aesthetic images local optima can be a boon. A fitness function that attempt to estimate the aesthetic quality of an image is, in all probability, an approximate guess at a measurement of the quality actually desired. This means that a human judge may rate locally optimal images above globally optimal images. The fitness functions used to evolve tiles in this study have a similar character to those used in evolved art; while there is control of tactical detail, some of the results look better than the others in ways that are difficult to accurately quantify. This means that an algorithm that is not too zealous in the pursuit of global optima simply enriches the tile set substantially.

V. Conclusions and Next Steps

This study has demonstrated that it is possible to use evolutionary computation to create a large variety of different types of tiles that can be used to rapidly create maps of substantial size by assembling the tiles. The assembly plans are drawn from an earlier study and, themselves, have highly controllable properties. The result is a decomposition of the level design problem into two relatively rapid off-line evolutionary computation problems. These yield libraries of tiles and assembly plans that permits a combinatorially large space of levels to be assembled from a relatively modest data object in time linear in the size of the desired map. Control of the properties of the assembly plan and topology of individual tiles mean that the character of the result map can be held within a relatively small part of the design space.

The primary novel contributions in this study are the fitness function, based on controlling relative distances between checkpoints within the tile, which permits a high degree of tactical control over the tiles and the use of required features. A required feature is a space whose shape is directly specified by the designer. The multiple-bay crypts and the spiral passage appearing in Figure 4 are examples of such features. The fitness functions gain additional control over the character of tiles by rewarding (or failing to reward) culs-de-

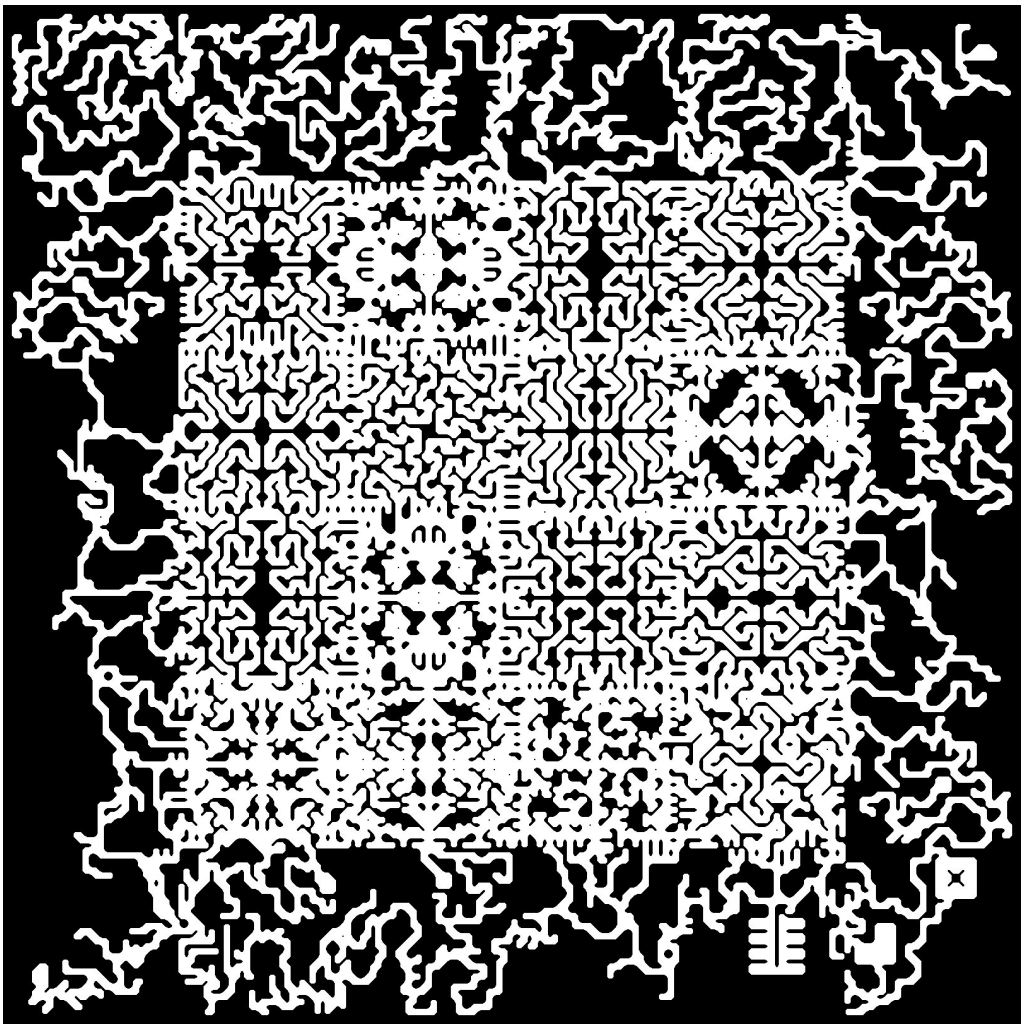

Fig. 6. A map assembled from a collection of 20 libraries of 30 tiles. In this case the center sixteen tiles are taken from libraries where the tiles have some form of symmetry.

sac and the total number of filled tiles in a grid. The central tiles in Figure 7, for example, reward filling grids not needed in passages between the doors.

Another interesting result is the use of imposed symmetry in tiles; examples were created in Experiments 21-24, examples of which appear in the last four panels of Figure 3. These tiles have a visually striking appearance and form the middle 16 tiles of the map in Figure 6. It is possible to impose other symmetries on the tiles and, while none of these tiles have visible required features, it would be possible to symmetrize such features as well. Additional examples of the symmetric tiles appear in Figure 8.

A. Other Representations

In [3] the authors used four representations to evolve maze-like levels. These included close analogs to the representations for both tiles and assembly plans used in this study, but also included a positive generative representation that places barriers in an initially empty map and a negative generative representation that places rooms and corridors in an initially completely obstructed map. This study restricted itself to a single representation for tiles in the name of brevity. The techniques presented here can be smoothly adapted to tiles using these other representations. A pair of examples of the results of such adaptation appear in Figure 9. These tiles incorporate required content in the form of square rooms and used a representation that lays down a set of 60 horizontal and diagonal barriers.

In [1] more exotic representations, such as mazes defined by the placement of chess pieces or a chromatic progression, were defined. This sort of content can also easily be incorporated into tiles. These other representations have been shown to generate tiles with a very different appearance than the ones evolved with the direct representation used here. The interaction of these representations with the new fitness function defined in this study is an early priority for future research.

B. Applications in Games

The next test for the level generation technique should be to incorporate the generated levels into a game. There seem to be three natural venues. The first is an encapsulated,

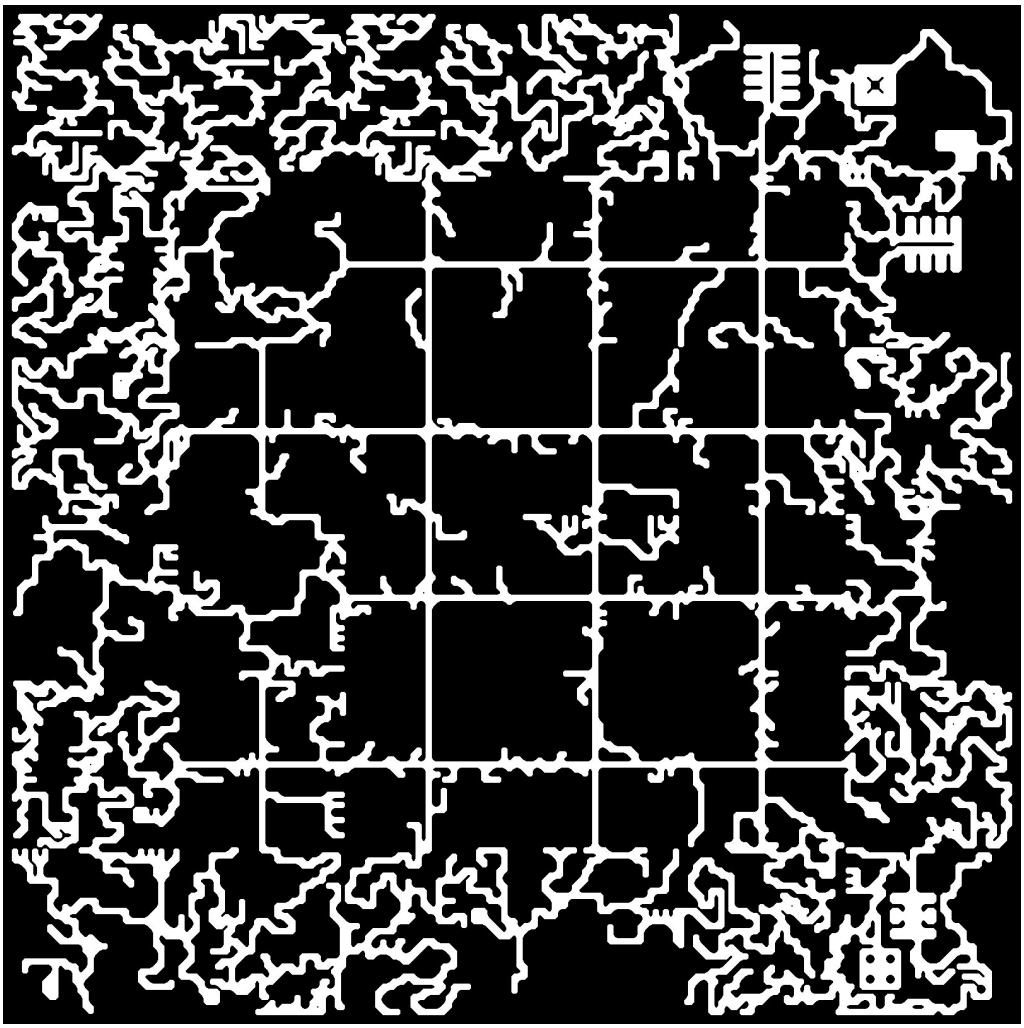

Fig. 7. A map assembled from a collection of 20 libraries of 30 tiles. In this case the center sixteen tiles are taken from libraries where evolution emphasized filling in as much space as possible while also encouraging culs-de-sac.

portable minigame where the tile itself, augmented perhaps with puzzles, would form a self contained unit. The ability to have a minigame that has a different map each time it is activated would improve replayability. In this case puzzles could be drawn from work like that in [1] where barriers, rather than being explicit walls, are implicit in the behavior of traps or environmental features.

The second possible venue is to generate levels for a first person shooter or roguelike adventure game. This is a challenging enterprise requiring either substantial development or integration of the tile library and assembly code into existing platforms. The third is to incorporate the map generation technology into software that automatically generates modules for a fantasy role-playing game. Anything from levels of a traditional dungeon to an endless supply of warehouses and industrial building for a zombie-apocalypse scenario are well within the map generation capabilities of the techniques given here.

C. Landscape Tiles

An unexplored but persuasive area for extending the techniques in this study is the generation of landscape tiles. Impassable terrain would replace stone walls within the metaphor of map interpretation. A landscape tile can also have territory types that are variably hard to traverse or can only be traversed by certain types of agents. If a game had mixed infantry, armor, and standard road vehicles the tile generation technology could be used to create mixed terrain tiles that work with an assembly plan that controls the tactical access of different types of agents. Woods would be infantry-friendly, armor-possible, and truck-impossible. Swamps are infantry-possible, but impassable to armor and trucks. Water would permit the passage of boats, but nothing else. Terrain type contiguity would be an issue, but could be handled automatically by making the underlying structure a height map with a fitness function that penalizes excessive gradients; features of particular types are assigned to particular ranges of height.

Required features within landscape tiles could include

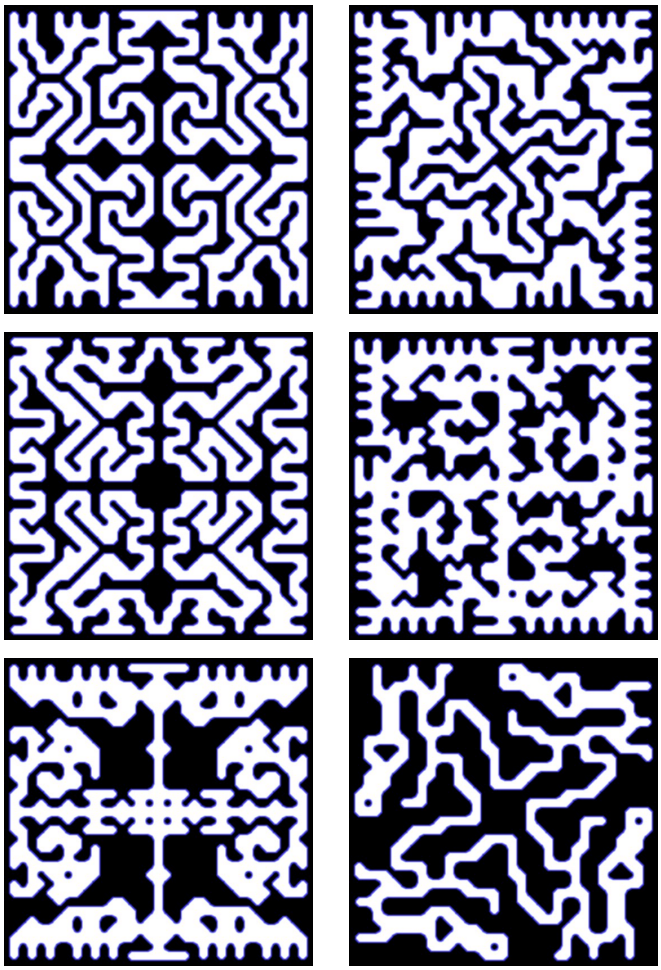

Fig. 8. Additional Examples of Symmetric Tiles. Those on the left are symmetric about the vertical and horizontal axis while those on the right have 4-fold rotational symmetry. These tiles all have entrances at all four corners.

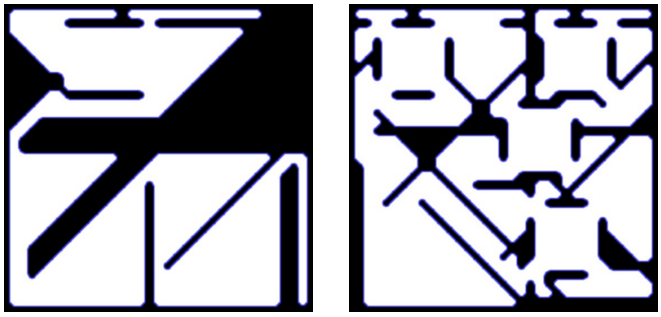

Fig. 9. Example tiles generated using a positive generative representation. Required content in the left tile consists of a single square 9×9 room while the right tile contains four such rooms.

bases, supply dumps, towns, mineral resources, and other important objects. The placement of these objects could be seconded to the software that evolves assembly plans while the layout of individual tiles, with or without strategically important desired features, is handled by a tile library generator. The resulting decomposition permits simple, high level strategic layout with multiple instances combined with tile libraries that efficiently decompose the problem of creating strategically equivalent maps with varying details.

D. Visibility and Lines of Sight

The issue of *lines of sight* yields another possible direction for this research. Fitness functions that reward the presence or absence of positions within a tile that give lines of sight to a large number of grids would permit designed control of another important tactical element. Another potential factor that could be incorporated into tile design is level of illumination. Agent types might have no night vision, adequate night vision, or perfect darksight. With these different illumination types, soft barriers could be defined by level of illumination. At this point the maze might be the streets of a city, with a representation that places buildings. A sparse, safe (well lit) network of major streets could form a core portion of the map while a poorly lit but much larger set of areas would comprise the back alleys and abandoned lots.

REFERENCES

[1] D. Ashlock. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence in Games*, pages 289–296, Piscataway NJ, 2010. IEEE Press.

[2] D. Ashlock and B. Jamieson. Evolutionary computation to search Mandelbrot sets for aesthetic images. *Journal of Mathematics and Art*, 1(3):147–158, 2008.

[3] D. Ashlock, C. Lee, and C. McGuinness. Search based procedural generation of maze like levels. Accepted to the IEEE Transactions on Computational Intelligence and Artificial Intelligence in Games, 2010.

[4] D. Ashlock, C. Lee, and C. McGuinness. Simultaneous dual level creation for games. Accepted to Computational Intelligence Magazine, 2010.

[5] D. Ashlock, T. Manikas, and K. Ashenayi. Evolving a diverse collection of robot path planning problems. In *Proceedings of the 2006 Congress On Evolutionary Computation*, pages 6728–6735. IEEE Press, Piscataway NJ, 2006.

[6] D. Ashlock and C. McGuinness. Decomposing the level generation problem with tiles. Submitted to CEC 2011, 2011.

[7] Daniel Ashlock. *Evolutionary Computation for Opimization and Modeling*. Springer, New York, 2006.

[8] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[9] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, New York, NY, USA, 2010. ACM.

[10] D. E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison-Wesley, New York, NY, 1997.

[11] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 3(48):44353, 1970.

[12] N. Sorenson and P. Pasquier. Towards a generic framework for automated video game level creation. In *Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications)*, volume 6024, pages 130–139. Springer LNCS, 2010.

[13] Julian Togelius, Mike Preuss, and Georgios N. Yannakakis. Towards multiobjective procedural map generation. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, New York, NY, USA, 2010. ACM.

[14] Julian Togelius, Georgios Yannakakis, Kenneth Stanley, and Cameron Browne. Search-based procedural content generation. In *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 141–150. Springer Berlin / Heidelberg, 2010.

[15] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2), 1967.