Decomposing the Level Generation Problem with Tiles

Cameron McGuinness
Department of Math and Stat
University of Guelph, Guelph
Ontario Canada N1G 2W1
cmcguinn@uoguelph.ca

Daniel Ashlock
Department of Math and Stat
University of Guelph, Guelph
Ontario Canada N1G 2W1
dashlock@uoguelph.ca

***Abstract*—Search based procedural content generation uses search techniques to locate high-quality content elements for use in games. This study specifies and tests an evolutionary-computation based system to generate tiles and plans that decompose the problem of assembling large levels. 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 levels available. The study also introduces new fitness functions, generalizing early work on checkpoint based fitness for the evolution of mazes, that is especially well suited for tile creation. Tiles are generated using two different representations that yield tiles with very different appearances. The study demonstrates assemblies of large levels and outlines several directions for extending the work.**

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 [11] 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. 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. The assembly method is itself based on an evolutionary algorithm similar to that used to generate the tiles which generates *assembly plans*. The assembly 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.

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 [9] levels for 2D sidescroller and top-down 2D adventure games are automatically generated using a two population feasible/infeasible evolutionary algorithm. In [7] the authors perform procedural generation using occupancy regulated extension which assembles additional content on demand depending on the position of the user's character. In [10] multiobjective optimization is applied to the task of search-based procedural content generation for real time strategy maps. In [5] 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 [1] which introduced checkpoint based fitness functions for evolving maze-like levels. This study introduces new checkpoint based fitness functions that are more appropriate for the generation of tiles. Checkpoint-based fitness designates checkpoints as distinguished positions within the grid on which mazes are built. A dynamic programming algorithm is then used to compute shortest paths from entrance to exit or between checkpoints. This information is then used to build a variety of fitness functions, each of which yields different styles or types of mazes.

Assembly of tiles borrows from the author's earlier research [2] 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 [2] 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, rather two-meter jumps will become tile sides with lower levels of connectivity between tiles and one-meter jumps will become tile sides with higher levels of connectivity. 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.

The authors thank the Canadian National Science and Engineering Research Council, the Ontario Council of Graduate Studies and the University of Guelph Department of Mathematics and Statistics for their support of this research.

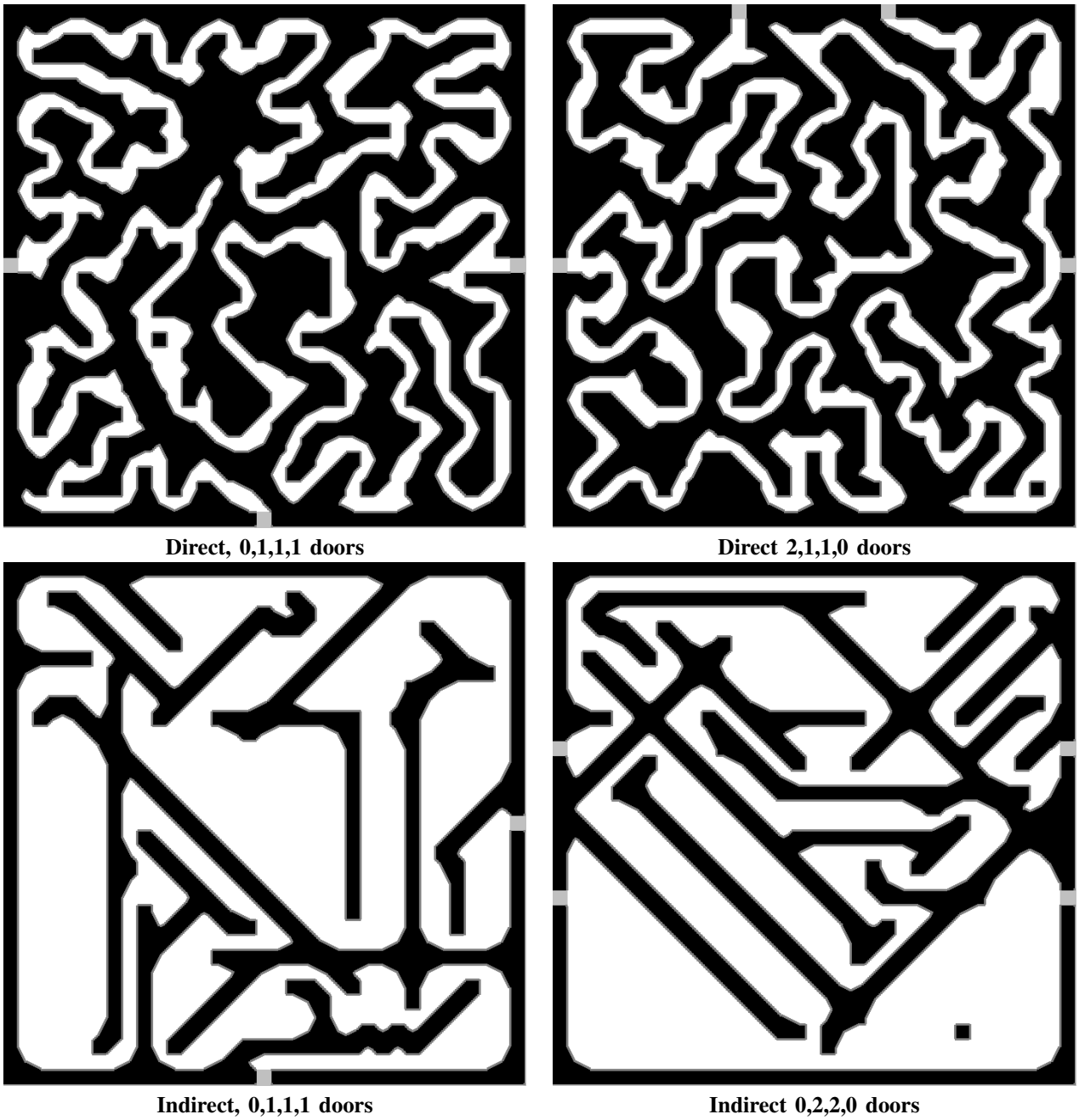

Fig. 1. Examples of tiles done with the direct and indirect representations and a variety of arrangements of entrances. Entrances are shown in gray.

A. Dynamic Programming

Dynamic programming [4] is an ubiquitously useful algorithm. It can be applied to align biological sequences [8], to find the most likely sequence of states in a hidden Markov model that explain an observation [12], or to determine if a word can be generated from a given context free grammar [6]. 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. In this study the network in question is composed of the accessible squares of the maze.

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 [3], a dynamic programming algorithm simultaneously computes the number of forward moves and turns a robotic agent needs to solve a path planning problem. A number of variants of a dynamic programming algorithm are used in this study, each representing a different fitness function that locates a different type of tile or level plan. A more detailed discussion of the functions

used to evolve the tile assembly plans appears in [2].

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

Two representations are used to evolve tiles in this study while a third is used to evolve assembly plans. The representations for tiles are adapted from [1] while the assembly plan is adapted from a representation first appearing in [2]. The first 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 level is built is full or empty. For a tile of size $X \times Y$ this is a string of XY bits. The second representation used for tiles is an indirect representation that specified how and where to put a series of barriers within the level. It is stored as a linear array of integers in the range 0-9999. The integers are used in pairs to specify barriers in an originally empty $X \times Y$ arena with walls at its edge. The first integer is bit-sliced into a one-bit number, a three-bit number, and a remainder R which is taken modulo the larger of X or Y to yield the length of the barrier. The one-bit number determines if a barrier is penetrating or not. The three bit number is interpreted as a direction for the barrier to run from its starting point W, NW, N, NE, E, SE, S, or SW. The second integer i is split as $x = i \ mod \ X$ and $y = ((i \ div \ X) \ mod \ Y)$ to obtain the starting point (x, y) of the barrier on the grid. The operator div represents integer division without remainder. A penetrating barrier runs from it's starting point to it's length or the edge of the arena. A non-penetrating barrier stops when it encounters any other barrier.

The two representations generate very different types of tiles. For both tile representations, fitness is the sum, over all pairs of entrances, of the distances between the entrances. These distances are computed with dynamic programming. As was found in earlier studies, path length maximization yields mazes with few or no closed loops and long winding passages. Examples of evolved tiles for both representations are given in Figure 1. All the tiles used in this study are 33×33 with entrances places symmetrically at position 16 (one door) or 11 and 21 (two doors) in the side.

The fitness function used to evolve tiles is to maximize the sum, over all pairs of entrances, of distances between entrances. The tiles that only have one door have to be treated as a special case since the fitness function used for the other tiles will not be defined in this case. With only one door there are no paths between checkpoints to maximize. To generate tiles with a single entrance, internal checkpoints were added along with the checkpoint associated with the door. The checkpoints used are

$$\mathcal{C} = \{(11, 21), (16, 16), (21, 11)\}.$$

The fitness function then becomes the sum of distances between pairs of checkpoints, including the one associated with the door.

The representation for the assembly plan is a grid of heights stored as an array of real numbers. This data structure is called a *height map*. This data structure specifies two kinds of mazes simultaneously. The *height one* maze permits movement between grids that share a face if their heights differ by no more than 1.0. The *height two* maze permits movement between two squares if the heights differ by no more than 2.0. This is a type of structure called a *dual maze*, simultaneously specifying two distinct mazes in a single physical structure. Dual mazes are explored in much greater depth in [2]. The height map has a beginning square in the upper left corner and an ending square in the lower right. Dynamic programming is used to compute the distance from the beginning square to all other accessible squares (a square is accessible if there is a path to it from the beginning square). A *cul-de-sac* is a square for which this distance is larger than all its accessible neighbors. The fitness function used to evolve assembly plans is the geometric mean of the path length from beginning square to ending square for the height one maze and the number of culs-de-sac for the height two maze. This fitness function was used in [2] and examples of mazes evolved with this fitness function appear there.

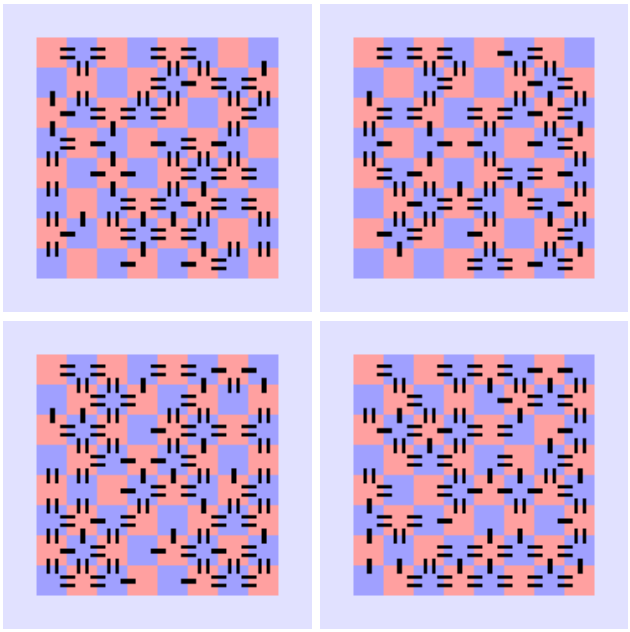

Fig. 2. Four examples of evolved assembly plans. Single hashmarks denote pairs of tiles sharing a single entrance, double hashmarks denote pairs of tiles sharing two entrances.

The height maps used in this study are 8×8. One tile is assigned to each grid. If two adjacent grids have a height difference in excess of 2.0 then no entrances are placed on the corresponding sides of the tiles; if the height difference exceeds 1.0 but is no more than 2.0 then one entrance is used; otherwise two entrances are used. This means that the connectivity of the height one maze realized in the height map becomes the large-scale connectivity of the complete

map. This permits very rapid assembly, from libraries of tiles and assembly maps, of a combinatorially huge collection of level maps. Figure 2 gives examples of assembly plans derived from height maps. It is not computationally difficult nor time consuming to use much larger assembly plans; the resulting levels cannot, however, be usefully viewed in a publication.

A. Sparse Initialization

In the earlier studies it was discovered that if the initial population was created by filling in the data structure uniformly at random then almost all population members have a fitness of zero because there is no path between entrances or from beginning to end of a maze. To compensate for this *sparse initialization* was used. In the direct representation for tiles this means that grids in the tile are filled only 5% of the time. For the indirect representation for tiles, sparse initialization is implemented by choosing the original lengths of barriers in the range 1-3. For the height map, initial heights are chosen with a Gaussian random variable with a mean of 3.0 and a standard deviation of 1.0. In all three 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.

All three algorithms used single tournament selection of size seven. In this model of evolution seven structures are chosen, the two best are copied over the two worst, and then the copies are subject to crossover and mutation. The evolutionary algorithms for all three representations use two-point crossover. The direct representation for tiles performs two-point crossover of the list of bits, the indirect representation performs two-point crossover of the list of barrier specifications, while the height map performs two-point crossover of the list of heights. For all three 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. In the indirect representation for tiles a new integer in the range 0-9999 is generated for a mutated loci. For mutation of a height, a Gaussian random variable with a mean of 0.0 and standard deviation of 0.5 is added to mutate a loci.

B. Polya Theory for Smaller Libraries

If we permit zero, one or two entrances in each side of a tile then the fact that a tile has four sides suggests that there are $3^4 - 1 = 80$ possible types of tiles that need to be saved (the -1 represents tiles with no doors). Notice, however, that it is possible to rotate and flip tiles to match different requirements. All tiles with no doors on three sides and one on a fourth, for example, are rotations of one another. The general case is a little trickier, but the *Burnside-Polya* theorem [13] solves it. Noting that the dihedral group of the square (square tile) is the relevant group of geometric equivalences it is elementary to compute the cycle index polynomial:

$$P(t_1, t_2, t_3, t_4) = \frac{t_1^4 + 2t_2t_1^2 + 3t_2^2 + 2t_4}{8} \qquad (1)$$

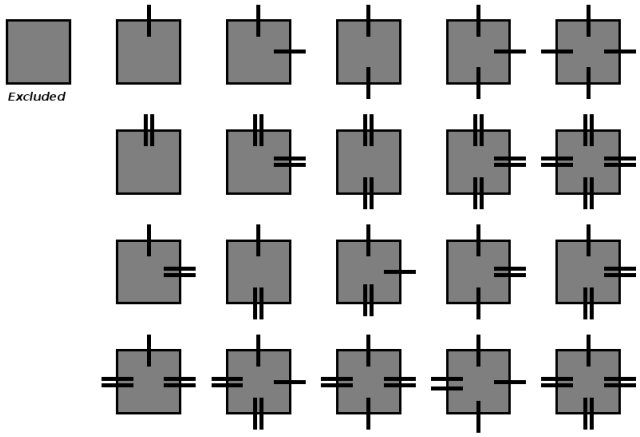

Fig. 3. A systematic enumeration of the fundamental tile types for square tiles with 0, 1, or 2 doors in each side. The tile with no entrances is included for completeness.

Since we have $n = 3$ possible numbers of doors, 0, 1, or 2, we compute there are actually

$$P(3, 3, 3, 3) = \frac{168}{8} = 21$$

fundamental tile types. Systematic enumeration yields a complete list of tile types; the tile with no doors is excluded meaning we need only 20 tile libraries. The enumeration of tile types is given in Figure 3.

III. Experimental Design

For both tile representations 10 replicates of the evolutionary algorithm was run for all 80 tile types. The decision to create all 80 tile types as opposed to the 20 fundamental tile types was based on ease of programming; in a production system using the reduced tile set and edge type matching is likely to be worth the additional effort. Both tile generation experiments used a 33×33 grid while the height map was 8×8. The height map experiment consisted of 30 replicates. For both the tile experiments as well as the height map experiment the population size was 120. It is required that there is a path in the tile from each door to each other door. Any mazes where this is not the case is given a fitness of zero. A *mating event* is one instance of single tournament selection. Experiments with the direct representation ran for 100,000 mating events saving summary statistics every 1,000 mating events. Experiments with the indirect representation ran for 200,000 mating events saving summary statistics every 1,000 mating events. Experiments with the height map ran for 200,000 mating events saving summary statistics every 1,000 mating events. The shorter run-time for the direct representation is based on the results in [1] which showed that the direct representation converged faster.

IV. Results and Discussion

The fitness function used in this study, the sum of distances between entrances as computed with dynamic programming, is considerably easier to understand than those used in earlier studies in this series, and yet it can be used to

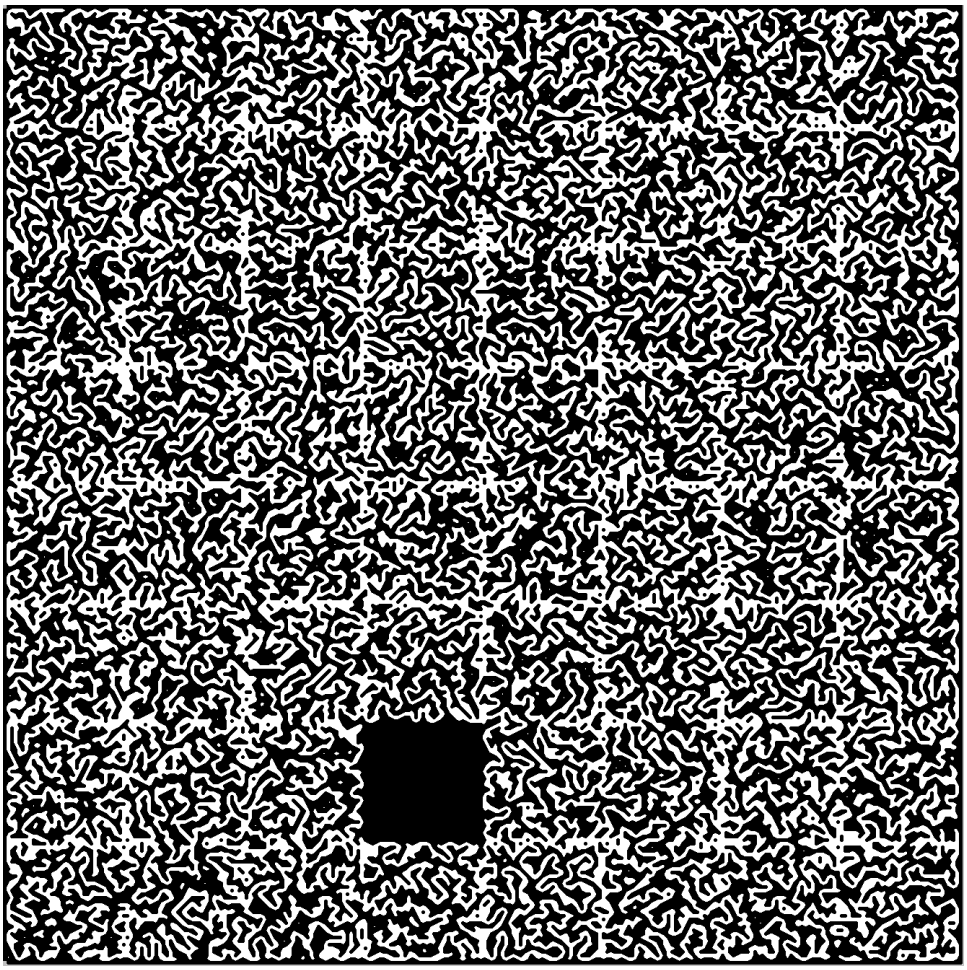

Fig. 4. An assembled 8x8 map consisting of direct representation tiles. The black square represents an inaccessible grid in the assembly plan.

generate libraries of tiles off-line in an efficient way. Since assembly plans are fairly small dual-height mazes they can be generated rapidly. These two tasks decompose the problem of level assembly for large levels in two off-line tasks. Assembly itself is an algorithm with time linear in area of the final assembled level making real-time assembly of level maps from pre-generated tile libraries an easy task.

Figure 1 shows examples of tiles generated for both the direct and the indirect representations. Similar to findings in previous work the direct representation tiles have an intestinal character while the indirect representation tiles have much more open space. The tiles in this study are exemplary; a great deal can be done to yield different appearances.

Figures 4, 5, and 6 all show different assemblies of tiles using the same assembly plan. Figure 4 uses only direct representation tiles. This creates a vast catacomb of winding tunnels. Figure 5 uses only tiles generated with the indirect representation tiles. This yields a map with more open spaces and distinguishable rooms. Figure 5 has a band of direct representation two deep and is filled in the center with indirect representation tiles. This shows that you can make maps that mix tile types, permitting substantial freedom of design.

All of these maps have one noticeable feature in common. White grid-like structure, like grout, is visible between the tiles. This is the result of a tendency of the evolutionary algorithm to place corridors at the doors instead of just a single open spot. This makes the connectivity of the door to the rest of the tile more robust. This could easily be corrected by adding in a fitness penalty for open spots near the edges that is proportional to distances from doors. This would probably still yield small corridors near the doors, but further away from the doors would be more likely to be walls. Notice that the "grout" is not entirely open; barriers force an agent traversing the maze into the interiors of tiles.

V. Conclusions and Next Steps

This study demonstrates that the task of creating large maps can be decomposed into the subtasks of tile generation and assembly plan generation. The pre-computation of libraries of tiles and assembly plans can be used to create a combinatorially large space of levels. Since the properties of both the tiles and the assembly plans can be tuned to a high degree by use of checkpoint based fitness functions, it is possible to certify that the entire space of final level designs have particular properties. This in turns creates the

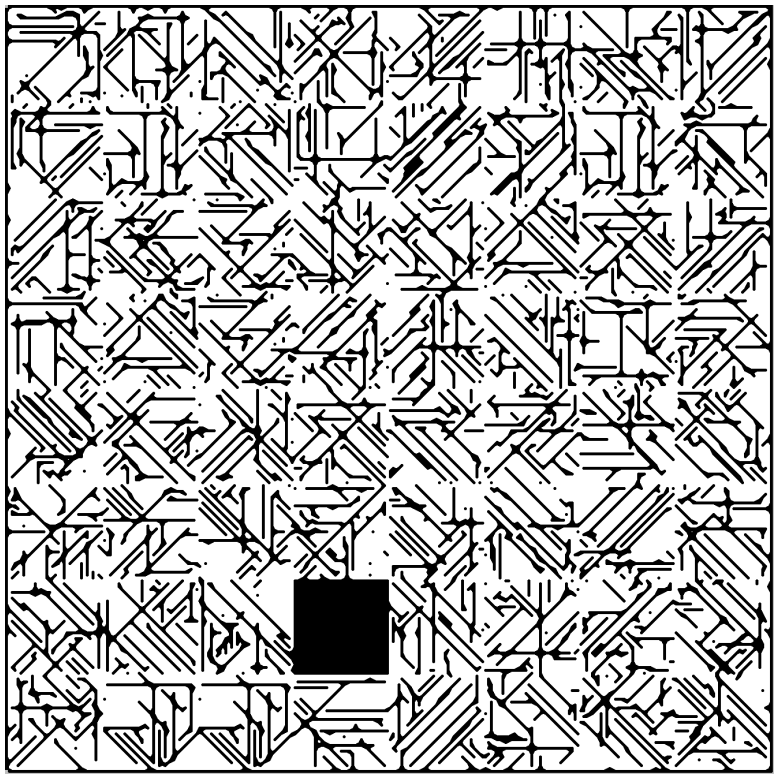

Fig. 5. An assembled 8x8 map consisting of indirect representation tiles. The black square represents an inaccessible grid in the assembly plan.

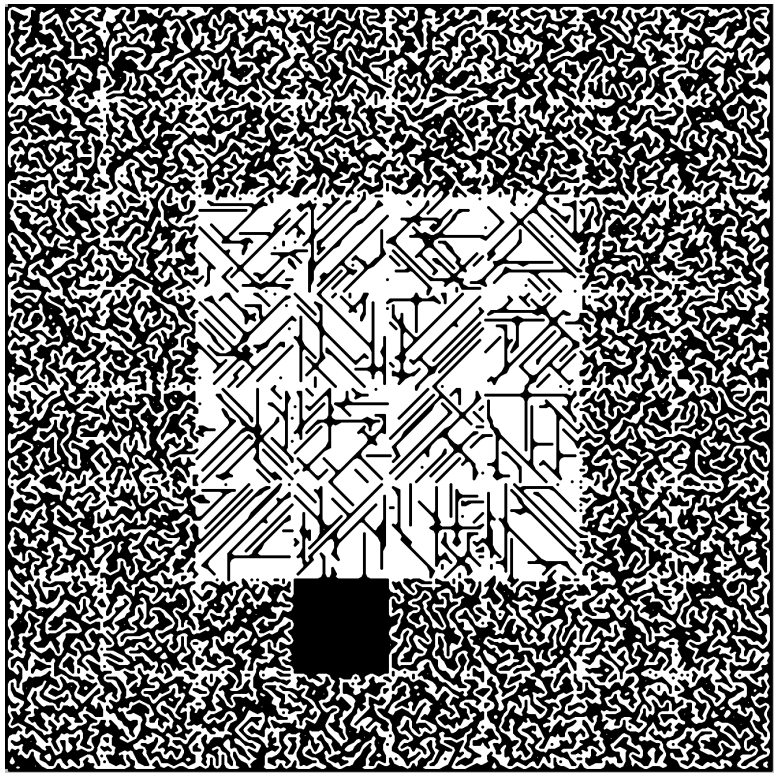

Fig. 6. An assembled 8x8 map consisting of direct representation in the outside two rows/columns and indirect representation tiles on the inside. The black square represents an inaccessible grid in the assembly plan.

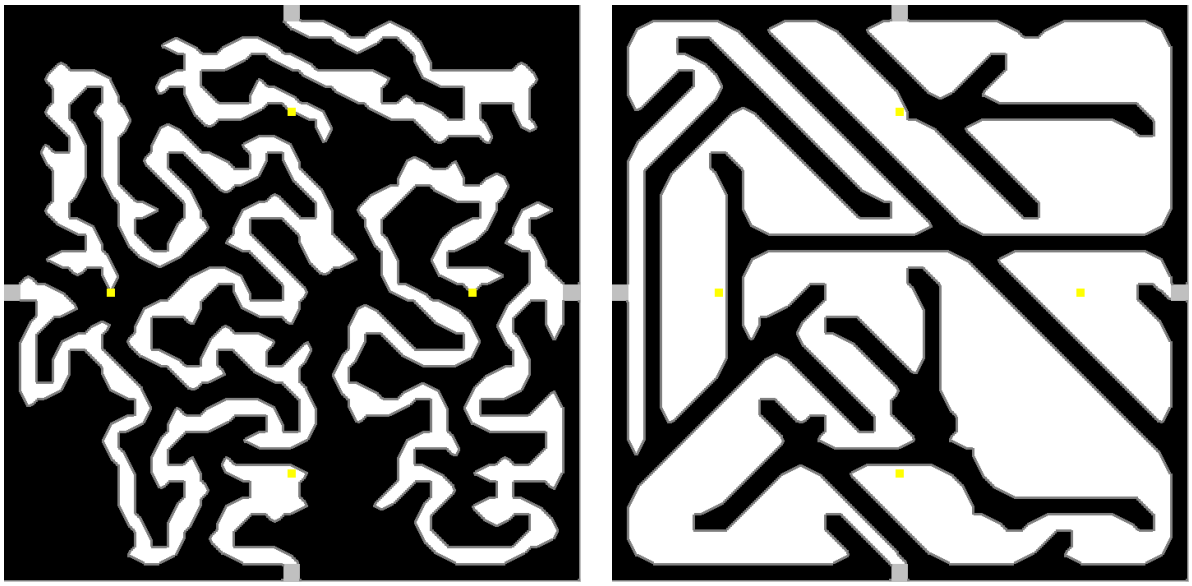

Fig. 7. Examples of dead tiles done with the direct and indirect representations. Entrances are shown in gray, internal checkpoints are shown in yellow.

potential to increase the replayability of a game. The novel features of this study are the decomposition of the level generation problem and the use of a fitness function based on the distance between entrances of a tile.

It is interesting to note that the tile libraries used in this study are $33 \times 33 \times 10 \times 80 = 871200$ bits. While the height map library is $8 \times 8 \times 64 \times 30 = 122880$ bits for a total of 994080 bits, or 121 kB. With this size of library we can create roughly 3×10^{64} different maps. It is clear that with even a small sized library we can create a gigantic and diverse collection of maps.

A. Dead Tiles

A *dead tile* is a place in the assembly plan requiring no entrances. One such tile appears in the assembly plan used for Figures 4, 5, and 6 and it creates a somewhat jarring visual feature. There are a number of options to deal with dead tiles. One way would be to just alter the assembly plan to make that tile accessible on all four sides and use the current tile library. This would substantially change the connectivity of the assembly plan and is undesirable. Another, perhaps better, way would be to create tiles that have one door on each side but with no paths between any two doors. The same technique of using internal checkpoints that solved the problem of tiles with one door can be used to create a small corridor or corridor system attached to each door. This would avoid disrupting the structure of the assembly plan. Examples of four entrance dead tiles evolved with both representations can be seen in figure 7.

B. Tile Variation

The tiles in this paper were all generated using only a sum of path lengths fitness function. Tiles can be generated using other fitness functions, a large number of which are given in [1]. Using a cul-de-sac counting fitness function it would generate tiles in which the paths through the tiles were shorter but there were far more branching paths. Another option would be to use the techniques shown in [2] to create dual maze tiles that have different wall types. Such tiles, with the potential to create tactically complex puzzles and traps, could be added into the general tile library.

C. Required Content

An extension of this and previous work would be to have content (rooms, corridors, etc.) that is required to be in a tile. This can also be extended to include object placement, such as monsters, and treasures. For instance, a good location for a valuable item could be in a cul-de-sac tile of the large map. It would even be possible to place the item within a cul-de-sac inside that tile if the tile was created using the cul-de-sac fitness function.

Examples of tiles generated with required content appear in Figure 8. Required features could be fixed into position in a tile or can simply be specified as a sub-square within the tile to be placed by the evolutionary algorithm. These tiles are evolved with a simple modification of the software used elsewhere in the paper.

D. Automatic Population of Levels

A clear next step is to populate the level with traps, treasures, opponents, and other features of interest. The vector of distances from the doors, or other checkpoints, yields a coordinate system that would make it easy to write automatic level-populating algorithms. Treasures, for example, should be farther from as many doors as possible as guardians and traps intended to protect them.

Similarly, the placement of traps is facilitated by dynamic programming information. The algorithm can be modified

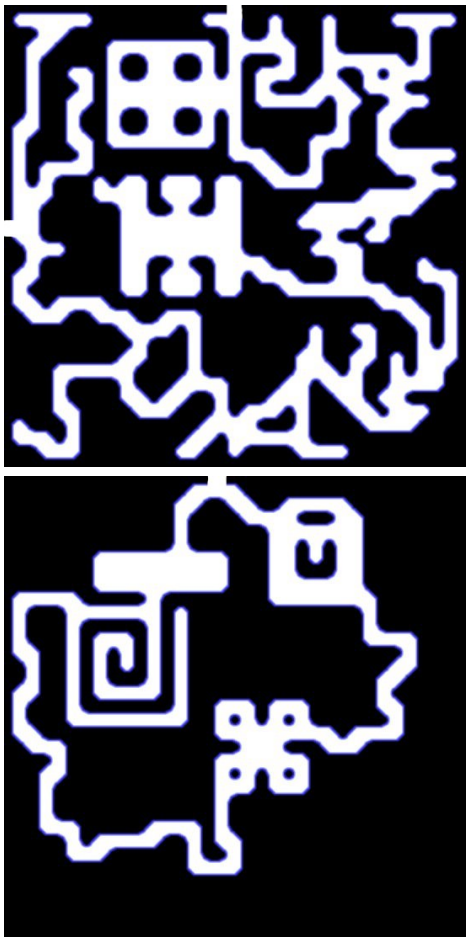

Fig. 8. An example of tiles with required content. A specification file is used to require rooms with a specific shape be added to the tiles above. The fitness function maximizes distances between doors while minimizing or maximizing, at users direction, distances between doors and checkpoints in the center of each room.

to locate points that are on all paths between two doors or checkpoints and so place traps that players cannot avoid or place traps that leave only a single safe path to or from a given objective.

This sort of algorithm would be especially easy to implement in tiles that have a single door since there is only one door to maximize distances from. These tiles essentially create a long cul-de-sac in the overall map and therefore make excellent places for treasures and other features of interest.

For features that are very important to a game-designer's plot arc such as the grand treasure for a level or a boss, placement could be incorporated into required content features. This permits the game designer to place complex ideas into the automatic level generation system in a transparent fashion.

E. Automatic Room Identification

When populating a level it is helpful to have a discrete list of large open spaces, enclosed by walls, that do not intersect. This list gives the *rooms* available within a level. With a list of non-intersecting rooms available, placement algorithms would operate on representative squares within the rooms rather than on every square of the grid, yielding potential efficiency. Potential rooms could be identified by trial placement and enlargement of rectangles. Such a list would then be filtered for intersecting rooms. This filtration could be done with a simple greedy algorithm or treated as an optimization that attempts to maximize area within rooms. An ordered gene evolutionary algorithm, that evolves the order in which the rooms were considered before presenting them to a simple greedy algorithm, would be a natural choice for such an optimizer.

F. Game Design

An obvious next step would be to incorporate the techniques from this paper and the earlier studies into an actual game. The ability to generate level maps on the fly from relatively compact libraries would add to the replayability of the game, since the maps and item placements would be different every time. This is an early priority for future research.

REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

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

[7] P. Mawhorter and M. Mateas. Procedureal level generation using occupancy-regulated extension. In *Proceedings of the 2010 IEEE Confrence on Computational Intelligence in Games*, pages 351–358, Piscataway NJ, 2010. IEEE Press.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] R. M. Wilson and J. H. van Lint. *A Course in Combinatorics*. Cambridge University Press, New York, NY, 2001.