Evolving Dungeon Crawler Levels With Relative Placement

Valtchan Valtchanov
School of Computer Science
University of Guelph
Guelph, ON, Canada N1G 2W1
valtchanv@gmail.com

Joseph Alexander Brown
School of Computer Science
University of Guelph
Guelph, ON, Canada N1G 2W1
jbrown16@uoguelph.ca or
jb03hf@gmail.com

ABSTRACT

Procedural Content Generation (PCG) is the process of automating the construction of media types for use in game development, the movie industry, and other creative fields. By approaching the process of media creation as a search for content which is evaluated to express desirable features in a well-defined manner, we are able to apply evolutionary techniques such as genetic programming. This can greatly decrease the effort required to bring a project to completion by allowing artists and developers to focus on guiding the creation process. The specific generation process addressed is that of map creation for dungeon crawler video games. The search method proposed allows artists and developers to guide the generation process by specifying a set of tiles that define the composition of each map, and a fitness function that defines its structure.

Categories and Subject Descriptors

1.2.1 [**ARTIFICIAL INTELLIGENCE**]: Applications and Expert Systems—*games*

General Terms

Algorithms, Performance

Keywords

procedural content generation, evolutionary computation, level generation

1. INTRODUCTION

Dungeon crawler (or roguelike) games are a type of role-playing video game where the player leads one or more adventurers to explore vast dungeon complexes composed of dozens or even hundreds of floors. Examples of such games include *Diablo* [5][6], *Hellgate: London* [9], and *Torchlight* [16]. The dungeons in this type of game are usually largely isolated from the game's storyline. This practice allows the game developers to employ random map generation techniques to greatly decrease the development time required to complete the game, and to introduce a level of re-playability that would not be possible otherwise.

The conventional approach to creating a map generator for a game of this type is to develop a procedural algorithm that randomly selects some set of predefined room tiles and then strings them together in a random and somewhat logical way. This approach has two main problems. The first is that it does not offer much room for fitting and tuning of the end result. This means that this method is very difficult to apply to situations where the developer requires the resultant map to fit a complex set of requirements and constraints. The second problem with this approach is that it forces the developer to embed the definition of the map structure into the generation algorithm itself. This makes modifying the behavior of such generators very difficult, and requires a new, and often very different, generator to be created for every map structure.

The evolutionary map generation approach proposed here aims to address both of these issues. It does so by introducing a concept of *fitness*, which allows the developer to define the map structure in a simple and powerful way. It also utilizes proven evolutionary computation techniques to find random maps that closely fit the requirements and constraints defined by the developer. This allows for the use of substantially more complex constraints and requirements, thus granting greater control over the structure of the generated map.

Procedural Content Generation (PCG) using a search for the creation of natural levels is an emerging area of interest for game development. Ashlock [2] looked at the creation of game elements using an evolutionary approach that allowed for the optimization of puzzles to a required difficulty level by using a dynamic programming technique as a fitness function. Ashlock et al. [3][4] created maze-like levels for games which use measures such as the length of path and number of dead ends as fitness criteria. Johnson et al. [10] shows a level design technique using cellular automata that also produces maze-like levels, though it does not give the same measure of fitness based on the properties seen in the tile.

McGuinness and Ashlock [12][13] expanded upon maps generated with the above techniques by using them as tiles in the creation of a larger level. However, the arraignment given was for the generation of square tiles and the placement was made based on a number of doors on the edges of the tiles. The approach proposed in this project goes beyond this by using rooms which can be of any size and shape, not just a set of tiles placed on a gird. This requires the solution of a placement problem which maximizes the use of the space for a given set of tiles. Cook and Colton [8] used evolutionary search-based PCG methods in order to not only provide level tiles but also to set locations of objects and change the rule sets of non-player characters.

The CityEngine is a commercial application of methods used for the creation of individual buildings [14] and entire cities [15]. The generation of such level designs may also have uses in architectural works and the rooms for the generated tiles can also be subject to evolutionary methods. Coia and Ross [7] look at the generation of abstract buildings using single and multiple objective genetic programming. The system works based on the ideas of using a shape grammar that takes a shape and modifies it via the application of a set of rules. This set of rules evolved in a tree structure via genetic programming and the fitness measures selected created abstract and alien buildings. Further, Flack and Ross [9] use a genetic algorithm for the automatic multi-objective generation of architectural floor plans. These methods taken together could be used to create the initial tiles with some level of automation. The problem with these methods is that they are intensive in terms of time and memory resources, and so they are more suited for offline generation. The method proposed in this paper works for online generation of levels, taking the existing tiles and then working on a search for how they should be placed in the space.

The remainder of this paper is organized as follows: Section 2 looks at the evolutionary method used in the creation of automatically generated maps for tile sets. Section 3 shows four experiments for differing requirements placed on the PCG of a tiled map: a small map for on demand creation, a large map for a static game or for a larger world generation where levels are allowed some time for loading, a map in which three event rooms are placed in order to maximize the distance between them, and restrictions on the fitness metric in order to have a map based on the location of event rooms in different regions. Finally, Section 4 provides concluding remarks about the creation of these generated rooms, speculations on where they could be applied in game development as well as future directions in the procedural generation of tile set levels.

2. METHODS

The search for a map with specific qualities is accomplished via an evolutionary approach known as a *Genetic Program* (GP) [11][1]. The process works by using various aspects of evolutionary biology, specifically the Darwinian idea of survival of the fittest, and of genetics. A population of tree-representations of solutions to a given problem, called *chromosomes*, is manipulated by the genetic operators of *crossover and mutation*.

Crossover is an n-ary operator, normally binary, which combines aspects of the parent chromosomes into child chromosomes. Mutation is a unary operator which makes small changes to a single chromosome. This allows for both *exploration and exploitation* of the search space. The choice of which chromosomes are subject to the operations is decided by a *selection* method which uses a *fitness* score. This fitness score is based on the chromosome's ability to solve the problem. The chromosomes undergo these operations for a number of generations, usually a set number of rounds, until a threshold of time is reached or a solution meets with a specific level of fitness. The given representation does not necessarily have to be a direct encoding for a solution, that is, the genotype might have a *translation step* into the phenotype.

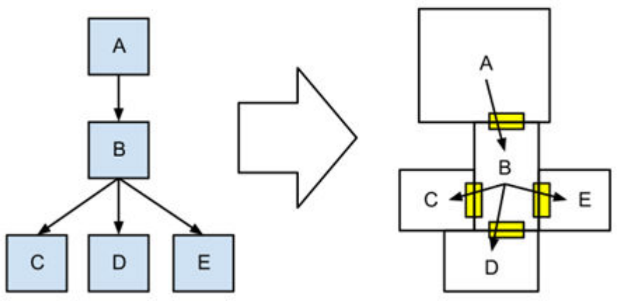

Figure 1: Example translation from tree structure into map.

2.1 Representation and Translation

The chromosome used in this method is a tree structure in which each node represents a reference to the tile to be placed and the door to be used to connect it to its parent. The tile property of each node can reference any one of a pre-defined set of tiles or a special `NULL` tile that is used to indicate that the door is not used. The pre-defined tiles are a set of room blueprints that define the building blocks that the algorithm is allowed to use to compose the map. Each tile defines the size, shape, door positions, and fitness parameters of a room that can be placed.

The genotype translation process used to convert each chromosome into a map phenotype enumerates the nodes in the chromosome in a breadth first fashion, meaning that the tree is processed top to bottom (see Fig. 1). The translation process is initialized by placing the origin room onto the map and then adding the root node, which represents the origin room, to a queue. The enumeration loop then removes the first node from the queue and attempts to place each of its children. The rooms placed are positioned and rotated to attach to the next available (not blocked or connected) door of their parent room. If the translation process is unable to place the current room, it and all of its child nodes are removed from the genotype. This will happen in the case of a collision with a room that has already been placed, or if its parent room has no available doors. If the translation process is successful in placing the specified room, the current child node is added to the back of the queue. When a child node that is referencing the `NULL` tile is encountered, the translation process removes all of its children from the genotype and marks the next available door of its parent's room as blocked.

The map phenotype used in this experiment is a two dimensional grid of cells where each cell can be either empty or occupied by a room segment. Each room segment can have a door facing in every one of the four possible directions. When a cell that a door is facing becomes occupied, the door becomes connected if the new room segment has a corresponding door, and blocked otherwise. All doors that are not connected at the end of the translation process are automatically considered to be blocked.

After placement of a chromosome is complete, the resultant map phenotype is used to generate a graph of the connected rooms. The shortest weighted path from the origin to each vertex is calculated, where the size property of the tile associated with each room is used as the weight of each vertex. Rooms that exceed the maximum allowed distance are removed from the graph and the nodes responsible for placing them, as well as all of their children, are removed from the genotype.

This is a memetic approach which serves two important purposes. The first is that it removes bloat from the trees by not allowing a branch that cannot be placed, or has exceeded the maximum distance, to continually expand. Second, it increases the diversity in the population by not allowing genotypes to accumulate large quantities of unused nodes that can absorb mutations while still encoding the same solution.

2.2 Representation Design Decisions

Three main design decisions influenced the choice of representation used in this approach. The first is the decision to use a set of pre-defined tiles as the building blocks of each map. This decision was made in order to allow game developers to define a set of assets, such as 3D meshes and textures, that correspond to each tile. This makes the process of converting the generated maps into playable game levels a very simple and efficient task. Using a pre-defined tile set also allows for the map phenotype to easily be converted into a graph where each vertex represents an instance of one of the tiles. This allows for the creation of robust and computationally cheap fitness models.

Using a relative position model within the genotype is another important design decision. Relative position placement means that the actual (x,y) position of the room placed by each node in the genotype is determined during the translation process. The position of each node within the genotype is defined by its index within its parent node and by the door to be used when connecting it to its parent room. This component of the representation was chosen because it offers a substantially smaller search space than absolute positioning. This is because relative room placement restricts the search to only connected maps, which is highly desirable in this case. Another advantage of relative room placement is that it allows for connected sections of the map to be moved, rotated, and copied very cheaply.

Finally, a tree was chosen as the underlying structure for the genotype because it is capable of representing any connected graph and therefore any map that can be composed from the tiles. It is also a relatively easy structure to work with and offers a set of clear mutation and crossover operators.

2.3 Operators

2.3.1 Crossover Operator

The crossover operator used by the algorithm exchanges a random sub-tree between copies of the two parents. One of the very useful properties of this operator is that it has the ability to copy collections of connected rooms to different locations on the map, which has two useful consequences. The first is that it makes it possible for a group of rooms to be moved to a more appropriate location on the map. The second is that it results in a certain amount of symmetry within the generated maps that resembles that of structures designed by humans.

2.3.2 Mutation Operator

The algorithm uses random combinations of three different mutation algorithms, each of which corresponds to one of the three main ways that a tree can be modified. There is some potential overlap between their possible effects; however this overlap is not sufficient to make any of them irrelevant.

- mutation_Grow: This mutation operator selects X random nodes with at least one available door and then adds Y new child nodes to each of them. The new nodes are assigned random door and tile values. The values 3 and 4 for X and Y, respectively, have been found to be effective.
- mutation_Trim: This mutation operator locates a random leaf node in the tree. If the leaf node is not using the `NULL` tile then the operator changes its tile to the `NULL` tile. If the node is using the `NULL` tile already then the operator deletes it from its parent.
- mutation_Change: This mutation operator selects X random nodes and assigns a new random tile and door value to them. A value of 2 for X has been found to be effective.

2.3.3 Combining Mutation and Crossover

Applying crossover and all three mutation operators at the same time has a very high chance of being destructive to currently existing structures. For this reason each operator is assigned a probability to be applied. The following probabilities for each operator have been found effective:

- The Crossover operator has a 70% chance of being applied.
- The mutation_Grow operator has a 50% chance of being applied.
- The mutation_Trim operator has a 50% chance of being applied.
- The mutation_Change operator has a 50% chance of being applied, and is also always applied if none of the other operators were applied.

2.3.4 Selection

The algorithm uses a tournament selection process, which divides the population into randomly assigned groups called tournaments. The selection algorithm sorts the members of each tournament by fitness and overwrites the bottom half of each tournament with the children of the top half. These children are created by applying the crossover and mutation operators to copies of the parents. Increasing the tournament size has been found to yield little benefit to the performance of the algorithm. A tournament size of 4 and a population of 480, giving a tournament count of 120, have been found generally effective.

2.3.5 Fitness Model

The fitness model used in this experiment guides the search toward finding maps that are composed of small, tightly packed clusters of rooms that are connected to efficient paths of hallway. The model also expresses a strong preference for maps that contain up to three special rooms that are placed near the perimeter of the map. The purpose of these rooms is to provide a location for unique enemy encounters and quests.

In order to accomplish this, the fitness model analyzes the map's graph after the translation process is complete and awards incremental rewards or penalties to its fitness based on its structure. Each placed room is viewed as being one of three different types: an event room, a hallway room, or a normal room. The room's type is inherited from the tile used to place it, and specified in the pre-defined tile set.

All connected hallway rooms are grouped together into hallways and awarded fitness based on the number of non-hallway rooms that they connect. The reward is increased for each additional non-hallway room, up to a maximum of five, while no reward is granted to hallways that do not connect at least two rooms together. In order to give preference to maps that contain efficient hallways, the fitness model deals a small penalty for each hallway room placed.

To encourage selection of maps that contain small normal room clusters the fitness model rewards normal rooms that are connected to both hallway rooms and non-hallway rooms. This has the effect of restricting the size of each cluster by requiring each room to also be connected to a hallway.

The strong preference for maps that contain up to three event rooms is accomplished by granting a high fitness reward for each event room placed, and a harsh penalty of setting the total fitness of the map to zero when the maximum allowed number is exceeded. This form of draconian penalty ensures that maps that exceed the maximum number of rooms are never selected. The fitness model also encourages event rooms to be placed near the edge of the map by only awarding them fitness if they are a specific minimum distance from the origin.

The model also awards a fitness bonus for the first few times each of the normal room tiles is used. This encourages the search to find maps that use a diverse set of different tiles.

3. RESULTS AND DISCUSSION

The following experiments were performed to examine the effectiveness of the described method of map generation under a diverse set of conditions. Because the aesthetic value of each map is highly subjective, the experiments here were chosen to focus on examining how effective the algorithm is at finding maps that express the specified fitness criteria.

3.1 Fitness Impact Experiment

The purpose of this experiment was to determine how much impact the fitness function has on the overall structure of the map. For this purpose the generator was allowed to run for an excessively long time and allowed to use more rooms than it would be able to place within a single region of the map.

GENERATOR PARAMETERS:

- Run time of 2000 generations. Fitness gains for this configuration tend to be small and far between after 1500 generations.
- Maximum of 500 rooms, a restriction that none of the maps reached.
- Placing a room after an event room disabled.

The first property worth noting about the results is that the fitness function is very clearly expressed in each of the generated maps. Each map contains the maximum number of event rooms. Long hallways that connect multiple rooms together are common, and rooms are regularly connected to both hallways and other rooms. The structure of each map is also packed together making efficient use of the available space. All of these things suggest that the fitness function is a powerful method for controlling the structure of the generated maps.

The second important observation is that all of the maps are very different. This suggests that the generator does not tend towards a single global optimum. Allowing the generator to run for a longer period of time simply allows it to find maps that express the fitness function better, and not maps that have greater similarity.

3.2 On-Demand Generation Experiment

The purpose of this experiment was to evaluate how well this method of map generation can perform in time constrained circumstances. This is very important for games that generate maps only when they are actually required by the player.

GENERATOR PARAMETERS:

- Run time of 500 generations
- Maximum of 100 rooms, a restriction that each of the maps reached.
- Placing a room after an event room disabled.

This configuration allowed a map to be generated in approximately 30 seconds using a single core of a 2.4 GHz desktop CPU. This time can easily be further reduced by optimizing the implementation of the algorithm.

There are a number of key things worth noting about all of the maps in this set. The first is that despite the relatively low number of generations used, the fitness function is still very pronounced in each of the maps. Hallways are generally used correctly. Rooms are rarely not connected to a hallway. All of the maps contain the maximum number of event rooms. The second thing worth noting is that even with a relatively low number of total rooms, each map still has a very unique internal structure and multiple alternative paths in addition to the ones enforced by the underlying tree structure. The third important thing to note is that the algorithm produces consistent results despite the low number of generations. This is very important for on-demand map generation because failure to generate an appropriate map directly translates into more loading time.

3.3 Alternative Structure Experiment

The purpose of this experiment was to modify the generator to create maps with a different structure. In this experiment the method for calculating the distance of each room was modified to be the shortest distance from either

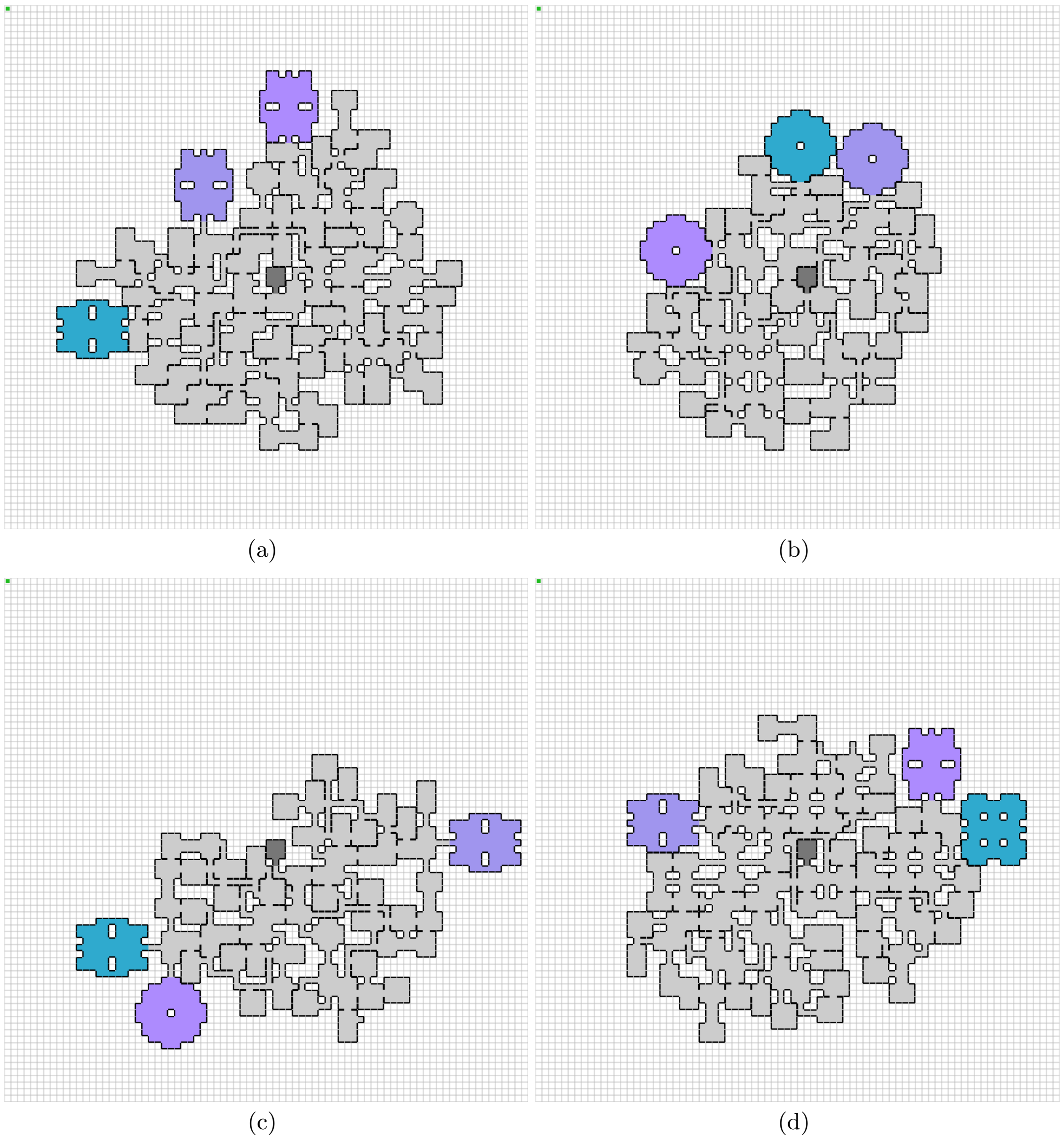

Figure 2: Example maps generated for the Fitness Impact Experiment - Event rooms highlighted

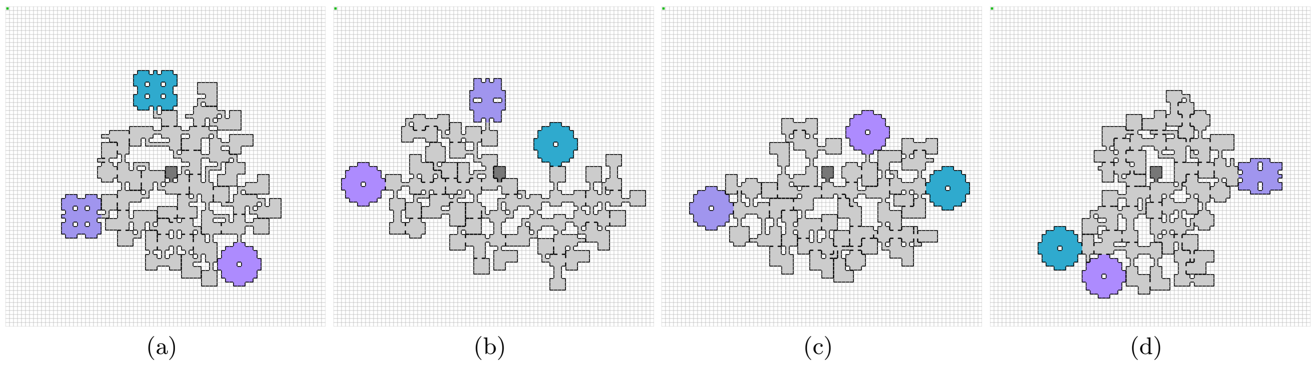

Figure 3: Example maps generated for the On-Demand Generation Experiment - Event rooms highlighted

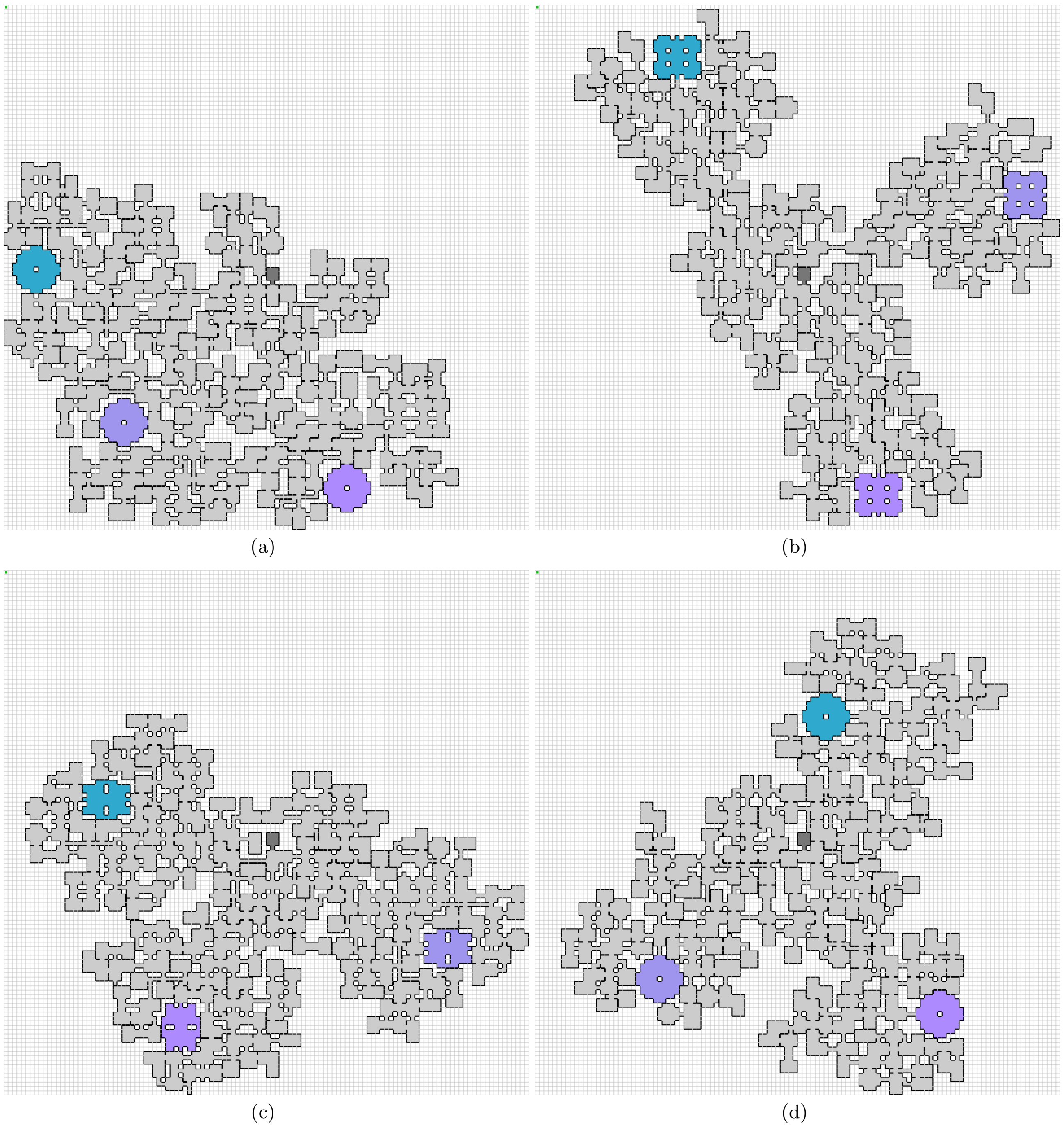

Figure 4: Example maps generated for the Alternative Structure Experiment - Event rooms highlighted.

the origin or the closest event room. The goal of this change was to accomplish two things. The first was to remove the maximum placement distance restrictions from event rooms. The second was to allow a cluster of rooms to be placed around each event room without exceeding the maximum room distance.

GENERATOR PARAMETERS:

- Run time of 2000 generations
- Maximum of 500 rooms, a restriction that each of the maps reached.
- Placing a room after an event room disabled.
- The distance of each room is calculated to be the shortest distance from either the origin or the closest event room.

The maps generated in this experiment clearly reflect the changes in the way distance is calculated. The event rooms are consistently dispersed enough to allow the maximum number of rooms to be placed, achieving maximum fitness. The map structure is divided into loosely connected clusters with event rooms near the center instead of a single cluster with event rooms at its edges.

3.4 Multi-Region Map Experiment

Game developers often need the map to force the player to pass through a specific room before he can enter a region of the map. The goal of this experiment was to evaluate a method of creating such regions within a map generated by this algorithm. The maps generated in this experiment contain multiple regions that are only connected together by a single event room.

GENERATOR PARAMETERS:

- Run time of 2000 generations
- Maximum of 500 rooms. This restriction was often reached by the generated maps.
- Placing a room after an event room enabled. Room distance is reset to zero for all rooms placed after an event room. Such rooms are also prevented from connecting to either rooms in the region owned by the origin, or to rooms in regions owned by other event rooms.

One of the maps generated in this experiment contains an event room that is placed in a way that does not allow for other rooms to be placed after it. The most likely explanation for this result is that the fitness function creates a deep local optimum in such circumstances that the algorithm has a hard time to get past. The easiest solution to this problem is to adjust how event room placement is rewarded to better reflect the desired effect. This can be done by making the reward dependent on the number of rooms placed after it, or by adding a rule that requires available space after the event room for the reward to be granted. The second local optimum shown could allow for other rooms to be placed, it is most likely due to a premature convergence in the population. The chances of adding a new room on to this specific door with the given genetic operators is extremely small.

4. CONCLUSIONS AND FUTURE DIRECTIONS

The results of the experiments suggest that this map generation method accomplishes the goal of providing a simple method for controlling the map structure that is both powerful and effective. Defining the map structure as a fitness function also leaves it decoupled from the underlying search algorithm, making both easy to modify. The method is capable of producing reliable and consistent results that meet a complex set of criteria. It is also capable of performing these tasks in a time constrained environment, making it a suitable tool for its intended purpose. However, the results of the experiments also demonstrate that this method is not a panacea. As demonstrated by the local optima issues shown in section 3.4, it is possible to design fitness and constraint models that result in large local optima that the algorithm struggles to get past. Further research is needed to develop a better understanding of the limitations of this method and how they may best be avoided.

This process has a number of practical applications. It could be used as part of a more fully featured game editor that allows developers to pre-design levels. This technique could also find use as an online method for the generation of random levels with consistency in their difficulty. The level generator would be part of a background process of the game, where as a player moves though the current level, a background process of the game is generating the next level to use. As the generation is based off a structure which is deterministically created interesting levels could be shared by players through saving the chromosome. This can turn into a new aspect in video games — a collectible and transferable level generation.

Areas of improvement and use upon this technique in the future may include: creating a fitness model that can be parameterized to generate a variety of common map structures, generation based on a frozen ancestor or already given map, creation of better initial tiles, changes in the representation to be based on graphs, a multi-step map generation for larger levels, and the inclusion of utility on the tiles such as items. The frozen ancestor creation is based on beginning with a currently designed map and using the GA to finish the construction with random elements. This would allow for designers of levels to have control of the path a character must take while allowing for interesting game-play through random features. Using a graph representation would allow each child to have multiple parents. Each node would be placed only once by the parent closest to the start node. The goal of this experiment would be to see if this change improves the stability of the representation and the overall performance of the algorithm. The multi-step map generation is a process which may prevent the problems seen with the multi-region maps. By re-centering the start room to an ending event room and running the GA from this location, these regions may have better separation. The addition of utilities to the room in the form of locations of quest givers, traps, monsters, and boss encounters would allow for a fully featured and more domain specific map generator.

5. ACKNOWLEDGMENTS

The authors would like to thank Dr. Daniel A. Ashlock for his comments and the National Science and Engineering Research Council of Canada (NSERC) for its support of this

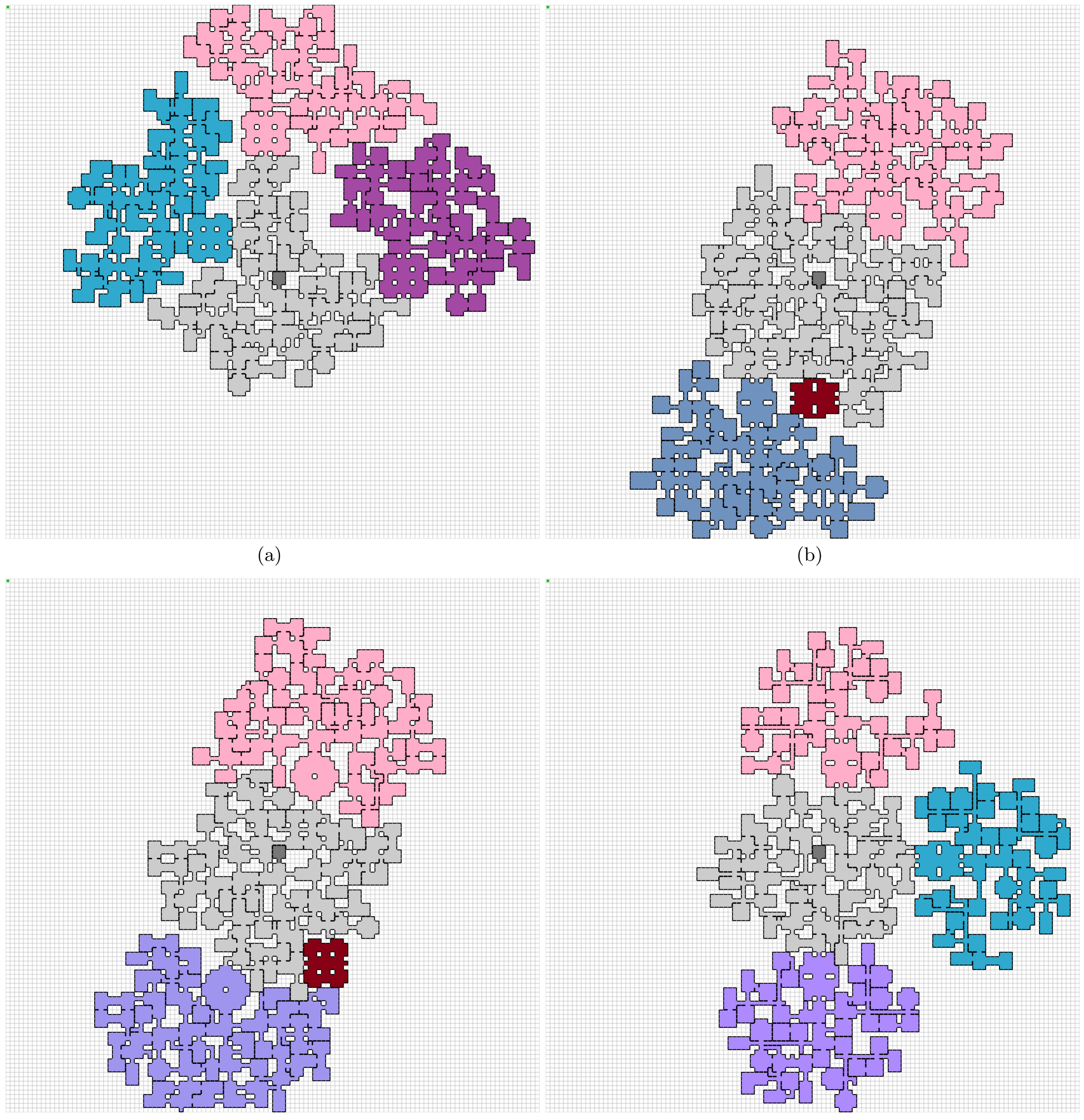

Figure 5: Example maps generated for the Multi-Region Map Experiment - different regions colored. Note that 5(b) and 5(c) have converged to a local optimum.

research.

6. REFERENCES

[1] D. Ashlock. *Optimization and Modeling with Evolutionary Computation*. Springer-Verlag, 2006.

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

[3] D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, 2011.

[4] D. Ashlock, C. Lee, and C. McGuinness. Simultaneous dual level creation for games. *IEEE Computational Intelligence Magazine*, 6(2):26–37, 2011.

[5] Blizzard Entertainment. Diablo, 1996.

[6] Blizzard Entertainment. Diablo II, 2000.

[7] C. Coia and B. Ross. Automatic evolution of conceptual building architectures. In *the proceedings of the 2011 IEEE Congress on Evolutionary Computation (CEC)*, pages 1140–1147, 2011.

[8] M. Cook and S. Colton. Multi-faceted evolution of simple arcade games. In *the proceedings of the 2011 IEEE Conference on Computational Intelligence in Games*, pages 289–296, 2011.

[9] Flagship Studios. Hellgate: London, 2007.

[10] L. Johnson, G. N. Yannakakis, and J. 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 : ACM, 2010.

[11] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[12] C. McGuinness and D. Ashlock. Decomposing the level generation problem with tiles. In *the proceedings of 2011 IEEE Congress on Evolutionary Computation (CEC)*, pages 849–856, 2011.

[13] C. McGuinness and D. Ashlock. Incorporating required structure into tiles. In *the proceedings of the 2011 IEEE Conference on Computational Intelligence in Games*, pages 16–23, 2011.

[14] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. In *the proceedings of SIGGRAPH '06*, pages 614–623. New York, NY, USA: ACM, 2006.

[15] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *the proceedings of SIGGRAPH '01*, pages 301–308. New York, NY, USA: ACM, 2001.

[16] Runic Games. Torchlight, 2009.

APPENDIX

A. TILE SET

The following is the tile set that was used for this project. Black lines indicate solid walls and red lines indicate door connection points.

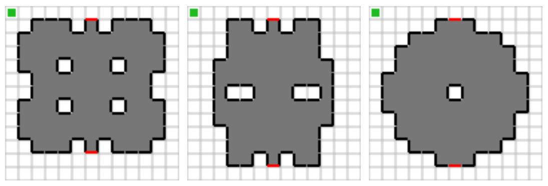
(a) Size 1 (b) Size 1 (c) Size 1

Figure 6: Event Rooms

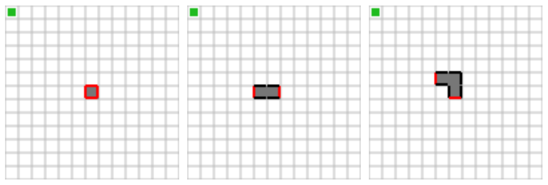
(a) Size 1 (b) Size 2 (c) Size 2

Figure 7: Hallways

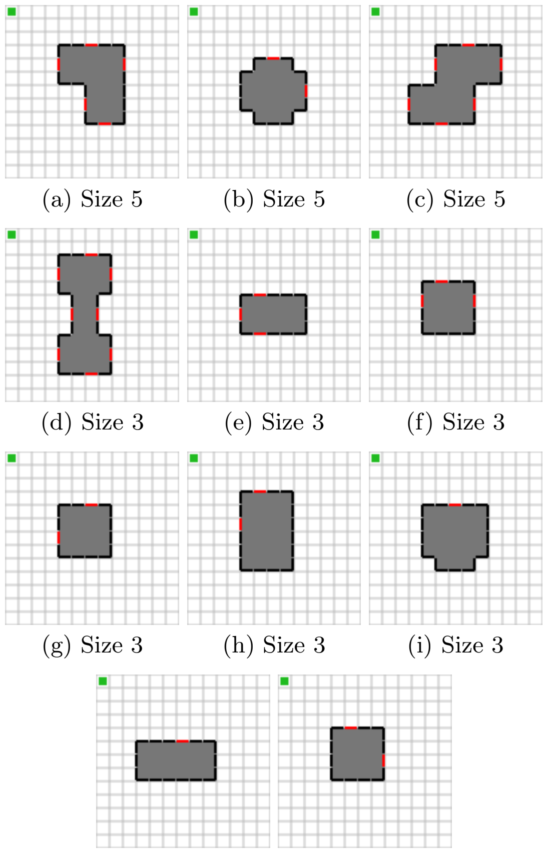
(a) Size 5 (b) Size 5 (c) Size 5
(d) Size 3 (e) Size 3 (f) Size 3
(g) Size 3 (h) Size 3 (i) Size 3
(j) Size 3 (k) Size 4

Figure 8: Normal Rooms