ALGORITHMS FOR PROCEDURAL DUNGEON GENERATION*

Nathan Hilliard, John Salis, and Hala ELAarag
Department of Mathematics and Computer Science
Stetson University
DeLand, FL
{nhilliar, jsalis, helaarag}@stetson.edu

ABSTRACT

In this paper, we propose two different algorithms; Span* and Growth, for dungeon generation that could be used in a game. Dungeon generation pertains to the creation of a two dimensional array representing a map of (x, y) coordinates. Each coordinate represents a tile on the map that can be either a floor tile or a wall tile. We can then create a room by grouping sets of floor tiles and create paths between the rooms by connecting each room with a line of floor tiles. Each of our proposed algorithms achieves a different kind of map style. We compared both algorithms in terms of average runtime and number of rooms. Our comparisons show that there are large trade-offs in terms of resulting style, runtime, as well as number of rooms generated between the two methods.

INTRODUCTION

Procedural content generation has been used in producing video game worlds for a long time. More recently, this technique has been popularized in modern game titles like *Minecraft* [1] or *No Man's Sky* [2]. In [1], the map is generated according to a random seed which deterministically generates the same map whereas in [2] the map is generated according to a set of rules.

Many researchers are interested in procedural content generation of dungeons [3, 4, 5]. Some suggest the composition of algorithms where an inner algorithm satisfies lower-level constraints, and an outer algorithm satisfies higher-level properties which can only be approximated by human input or simulation-based testing [6].

---

* Copyright © 2017 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

We can classify content generation into two main categories: Constructive algorithms and generate and test algorithms. Constructive algorithms such as Perlin noise [7], or cellular automata [8] generate content in one attempt but follow a set of operations or rules to ensure that unplayable content is never produced. This method is in contrast to generate-and-test algorithms which may take multiple attempts but include a test phase that scores the generated content based on some criteria. When the test fails, a portion of the content is discarded and regenerated until it passes [9]. Our proposed algorithms fall under the constructive algorithms category.

METHODOLOGY

Span*

Our first algorithm operates by generating a set of points randomly on the map which have to be at least some constant minimum distance from each other. Once the set of points is generated, we use Prim's algorithm [10] to decide how we are going to connect the rooms since it finds the minimum spanning tree for a given set of vertices. To do so, we use the linear distance between the points as the weights for each edge. By applying Prim's algorithm, we can connect each room to the rooms it is closest to which prevents dead-ends or excessively long hallways. We then draw the hallways to give the dungeon a natural feel using the A* search [11] algorithm. A* search [11] finds the shortest path between each point in the spanning tree with specific weights assigned to each tile of the dungeon. This creates nice looking hallways and can be described as generating the "optimal" layout given a set of rooms.

The pseudo code of the Span* algorithm can be seen in Figure 1.

```
Assume input map size, room constraints, tolerance.

Initialize tolerance counter T = 0
Initialize map M according to input map size
Initialize list of points P

While P < Target Room Count and T < Tolerance:
        Attempt to pick a new point on M and store in P
        T = T + 1

Apply Prim's on P
Apply A* on M given P
```

Figure 1: Span* Algorithm

Growth

The Growth algorithm operates by generating a single feature of the dungeon then procedurally growing the dungeon around the area by adding more features. In our case, the only two features are rooms and hallways of varying sizes. The algorithm continuously grows each feature from a list of points until the list is exhausted. Each growth point object contains not only the (x, y) position, but also a direction and the source feature that the point is located on. The direction is a combination of a row direction and a column direction. It is used to control the direction that an individual feature is grown. The source feature is used to decide which new feature should be grown from that point. In our case, growth points that lie on rooms always generate hallways. In this way, rooms will not be side by side. Growth points that lie on hallways can either generate a room or another hallway. The decision is made randomly, and both are equally likely.

The pseudo code of the Growth algorithm can be seen in Figure 2.

```
Assume input map size, room constraints, tolerance.

Initialize tolerance counter T = 0
Initialize map M according to input map size
Initialize list of growth points P

While P > 0:
        T = 0
        Take a growth point G from P
        If G is a room:
                Feature F = hallway
        Else:
                Randomly pick a hallway or room feature F

        Attempt to create feature F at point G
        While unsuccessful and T < Tolerance:
                Retry creating F at G
                T = T + 1
```

Figure 2: Growth Algorithm

IMPLEMENTATION

We implemented both algorithms in JavaScript. In our Span* implementation, as mentioned before, specific weights are assigned to each tile. A non-diagonal move (up,

down, left, or right) is weighted as 10 whereas a diagonal move is 14. This steers the algorithm to tend to prefer non-diagonal motion, which is what makes the paths look more realistic in the Span* algorithm. The *tolerance* threshold represents the maximum number of attempts to pick a new point before giving up. As the number of generated points increases, however, it becomes increasingly difficult to randomly select a point that satisfies the criteria of being at least a certain minimum distance from each other generated point or the edge of the map. Upon a successful placement of a newly generated point, we reset the *tolerance* value prior to the next iteration.

In the implementation of the Growth algorithm we initialize the process by adding the first growth point to the center of the map. When a new feature is added, the used growth point is removed and new growth points are added to the list. The key constraints when adding a new feature is that it must not intersect with existing features and must fall within the range of the map. When this is not fulfilled, the feature generation function returns false and the failed attempt is counted. The algorithm tries again to create the feature at the same point until either it is successful or the number of errors reaches the specified threshold. When this occurs, the growth point is discarded and the process continues to the next one.

When a new room is generated the edge starts at the growth point and extends in the specified direction. Each tile is checked to ensure it is not already a floor tile or adjacent to a floor tile. If the room placement is valid then new growth points are added on each side of the room excluding the side that contained the source growth point. We do not want growth in the opposite direction.

The generation of hallways follows a process similar to rooms. The main difference here is that they are one-dimensional rather than two-dimensional and only one new growth point is added at the end of the hall. The direction of this point is always orthogonal to the source point. In this way, consecutive hallways look more interesting and don't extend in the same direction.

RESULTS

In order to scale the number of rooms appropriately to the map size for our experiments, we set the target room count $R = \left(\frac{M}{S_{max}}\right)*2$ where M is the map size and S_{max} is the maximum size of each room. Unless otherwise stated, we set the minimum and maximum room size to be 10 and 12, respectively and the tolerance value to 20. Figure 3 shows an example of the layouts generated by both algorithms for a map size of 128 units.

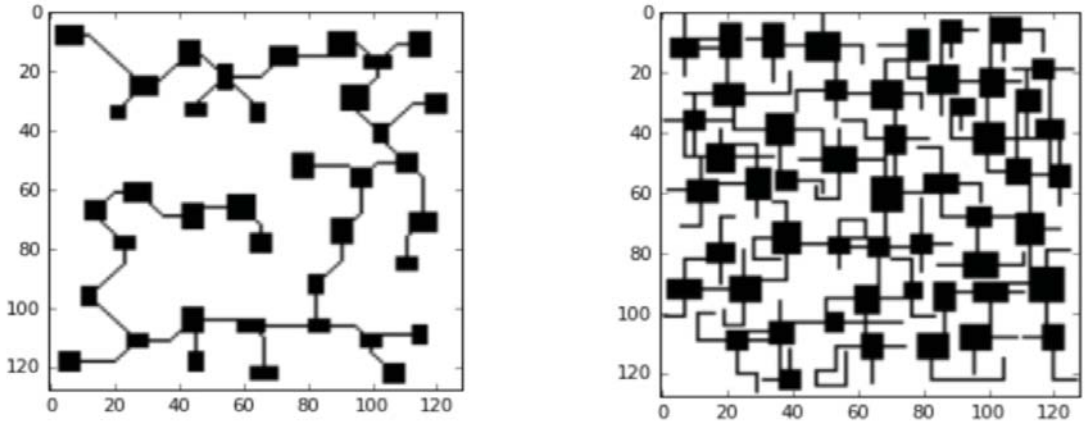


Figure 3: Example layouts from Span* (left) and Growth (right)

We ran experiments to compare the execution time effect vs. Map size. We varied the value of tolerance between 10 - 50. As shown in Figure 4, Due to the cascading computational costs of Prim's algorithm [10] used in conjunction with A* [11], Span* takes orders of magnitude longer to generate dungeons. The iterative properties of the Growth algorithm lend themselves to much, much faster execution times as can be seen in the Figure. The tolerance value has more impact on the performance of the Growth than Span* algorithm.

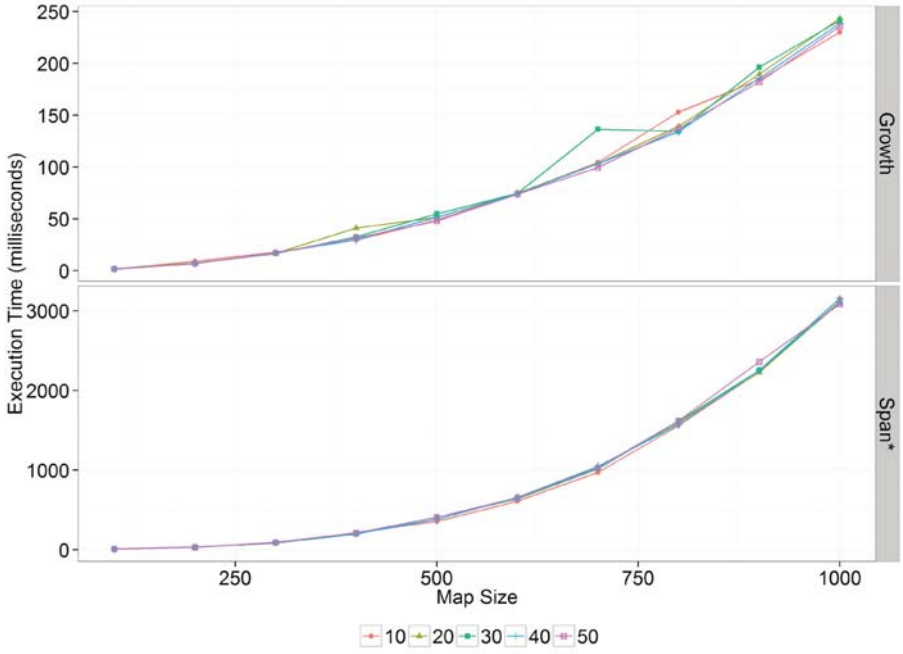


Figure 4: Average Execution Time vs Map size for Tolerance values 10-50

In Figure 5, we compare the execution time of both methods vs Map size with different room sizes. We varied the room size between 4 - 20. Similar to Figure 4 Span * still takes order of magnitudes more than Growth to execute. The size of the room has more noticeable effect on the execution of Span* than Growth, especially with smaller room size of 4.

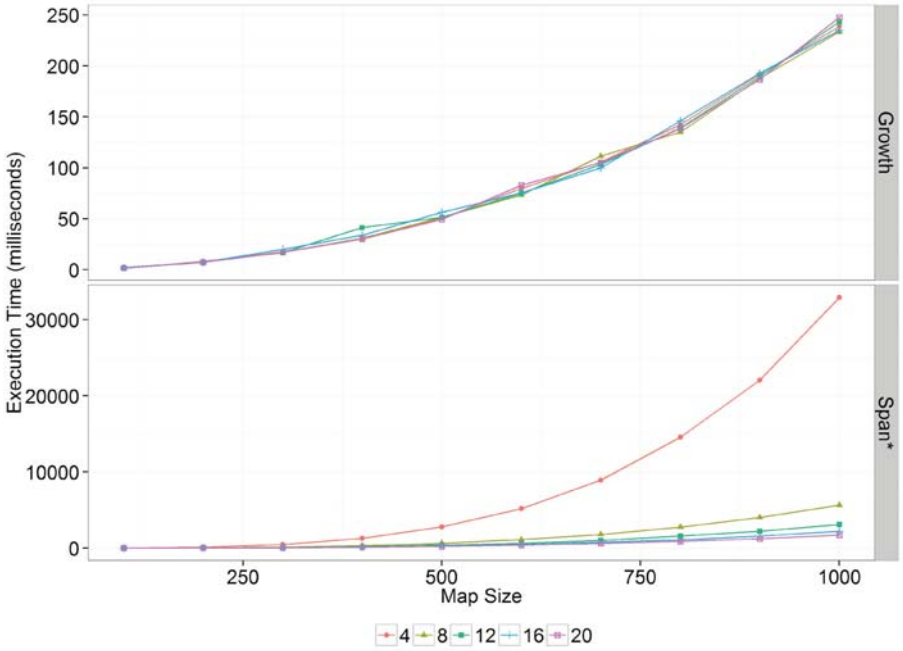


Figure 5: Average Execution Time vs. Map size for Room Sizes 4-20

In Figures 6 and 7, we present the numbers of rooms generated by each algorithm vs. map size with different tolerance values and room sizes, respectively. Due to the iterative process of the Growth algorithm, it is able to achieve much higher room counts whereas the Span* algorithm lags behind. Span* seems to struggle to find points on the map that satisfy the criteria described in our implementation. In both methods, the impact of modifying the tolerance constant is negligible as illustrated in Figure 6.

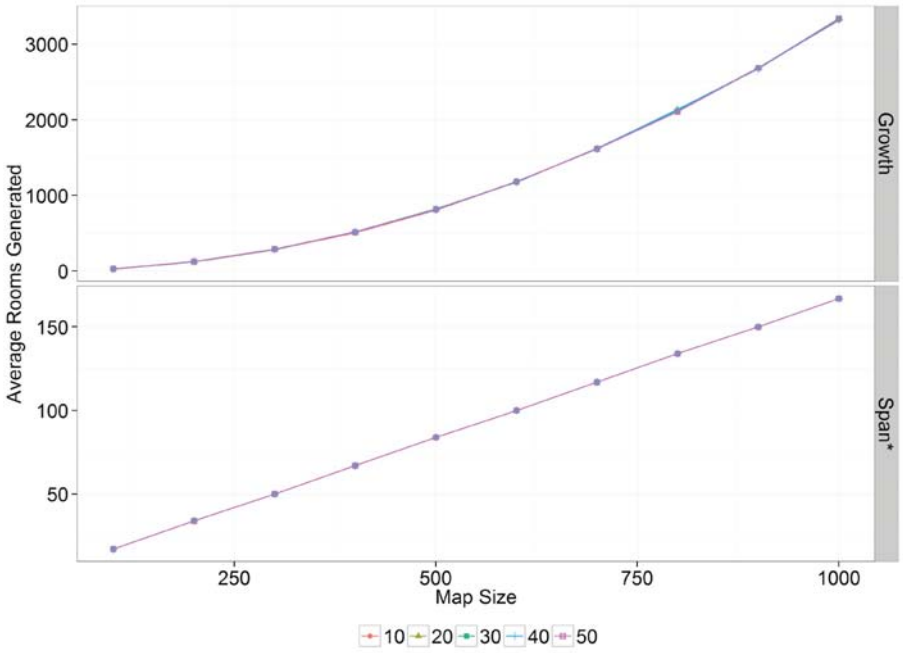
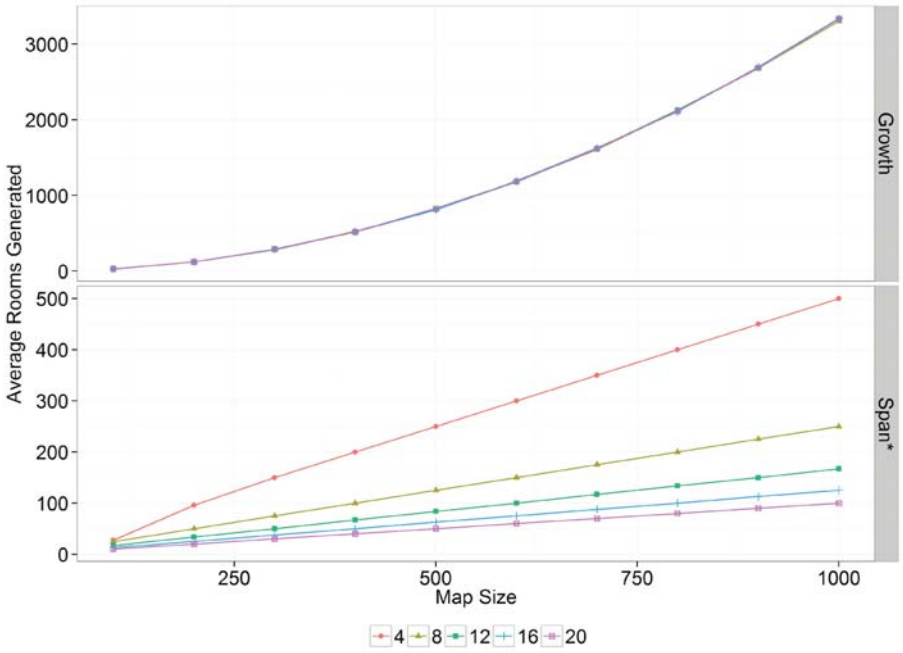


Figure 6: Average Number of Rooms Generated vs. Map size for Tolerance 10-50

In our experiments, we found that the Growth algorithm was also indifferent to the room size whereas Span* was much more reliant on finer tuning of the parameter as illustrated in Figure 7.



CONCLUSIONS

In this paper, we proposed two algorithms to create procedurally built dungeons, Span* and Growth. Our Span* algorithm creates a more useful dungeon for games, while the Growth algorithm creates a dungeon quicker and fills the map more completely. However, Growth creates dead ends while Span* doesn't since it creates hallways between the rooms and tries to fill all available space. In addition, Span* creates a dungeon that is more aesthetically pleasing. In an application where the map is pre-generated, the Span* algorithm would be more preferable if a cleaner looking map is required.

REFERENCES

[1] Mojang, Minecraft, 2011, https://minecraft.net/en-us/, retrieved July 2017.

[2] Hello Games, No Man's Sky, 2016, https://www.nomanssky.com/, retrieved July 2017.

[3] Hendrikx, M., Meijer, S., Van Der Velden, J., Iosup, A., Procedural content generation for games: a survey, *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9 (1), 2013.

[4] Shaker, N., Liapis, A., Togelius, J., Lopes, R., Bidarra, R., Constructive generation methods for dungeons and levels, *Procedural Content Generation in Games*, Springer International Publishing, 31-55, 2016.

[5] van der Linden, R., Lopes, R., Bidarra, R., Procedural generation of dungeons, *IEEE Transactions on Computational Intelligence and AI in Games*, 6 (1), 78-89, 2014.

[6] Togelius, J., Justinussen, T., Hartzen, A., Compositional procedural content generation, *Proceedings of the Third Workshop on Procedural Content Generation in Games*, 2012.

[7] Perlin, K., An image synthesizer, *ACM Siggraph Computer Graphics*, 19 (3), 287-296, 1985.

[8] Johnson, L., Yannakakis, G. N., Togelius, J., Cellular automata for real-time generation of infinite cave levels, *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010.

[9] Togelius, J., Yannakakis, G. N., Stanley, K. O., Browne, C., Search-based procedural content generation: a taxonomy and survey, *IEEE Transactions on Computational Intelligence and AI in Games*, 3 (3), 172-186, DOI: http://dx.doi.org/10.1109/TCIAIG.2011.2148116, 2011.

[10] Prim, R. C., Shortest connection networks and some generalizations, *The Bell System Technical Journal*, 36 (6), 1389-1401, DOI: http://dx.doi.org/10.1002/j.1538-7305.1957.tb01515.x, 1957.

[11] Hart, P. E., Nilsson, N. J., Raphael, B., A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, 4 (2), 100-107, DOI: http://dx.doi.org/10.1109/TSSC.1968.300136, 1968.