Educational Dungeon Generation Algorithm

Jaskrit Singh
School of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts
jsingh3@wpi.edu

Luke Sanneman
School of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts
lcsanneman@wpi.edu

I. Introduction

Procedural generation is used by many games to generate unique worlds that are interesting to explore without having to manually generate the entire world. While procedural generation is common and well refined for open 2D worlds, it is not as common or refined for more constrained dungeon worlds.

A game currently in development, Daedulus' Dungeon, aims to use a dungeon to embed educational content in the form of a physical space. This project will focus on creating a dungeon generation algorithm to improve the variety of physical spaces present in Daedulus' Dungeon. The current generation consists of a grid of rooms with 1-4 doorways connecting to other rooms. Each door contains a math problem that must be solved to unlock the door as seen in Figure 1.

The purpose of the game is to allow the player to explore different areas of math as a physical space. To fulfill this purpose, the dungeon must always give the player multiple directions to explore in. The idea is that players can decide which area of a subject they want to learn by solving problems in those areas and moving to those parts of the dungeon. Multiple choices are also required to allow the player to avoid problems that are too difficult for their current knowledge level.

It is also desirable for the dungeon to be able to flexibly include custom puzzle rooms in the generation. These puzzle rooms would have a defined start and end location with an interactive puzzle to solve in the room. These rooms can be designed with any arbitrary shape, so it is important that our algorithm can accommodate them.

Our fundamental research is: Can motion planning be used to create a dungeon generation algorithm allowing for custom room shapes and allowing for control over the number of exploration directions available to the user? If we are able to succeed in creating an algorithm with these properties, it could be used in the implementation of Daedulus' Dungeon as well as other dungeon games requiring similar constraints.

II. Related Works

According to [1] dungeon generation can be split into two categories: constructive algorithms and search-based algorithms. An example of a constructive algorithm would be a generative grammar which uses rules to determine how new dungeon primitives can be added to expand the dungeon. Meanwhile, search-based algorithms generate many dungeon candidates and optimize them based on a fitness function that captures some of the desired dungeon properties. These often use genetic algorithms to try to combine good qualities from promising dungeon candidates.

Fig. 1: Example of a puzzle door from Daedulus' Dungeon

Our approach will use a constructive algorithm to generate the dungeon. However, there are certain properties that could benefit from optimization. Using a search-based algorithm to optimize for these properties could be an interesting problem for future work.

One promising generative approach that has been used in several games including *Tiny Keep* is to randomly sample points and create a Voronoi Diagram of the space using the points [2]. The Voronoi Diagram can then be used to make a Delaunay Triangulation which connects adjacent Voronoi cells to form a graph. A subset of these nodes and edges can then be used as the foundation for the generated dungeon, where the edges between rooms are created using A*.

Our approach is similar in that it samples room locations and connects them, but since we are using motion planning, we are able to use custom room shapes as part of the connections between sampled rooms rather than only having corridors generated by A*.

Our implementation will use a Sampling-Based Roadmap of Trees (SRT) [3]. This method involves expanding randomly from some preselected nodes before connecting the different trees together. Although this method is primarily intended for use in motion planning to plan around obstacles in high dimensional space, it offers some benefits for our dungeon generation algorithm. First, it provides a convenient way to generate different parts of the dungeon around each room

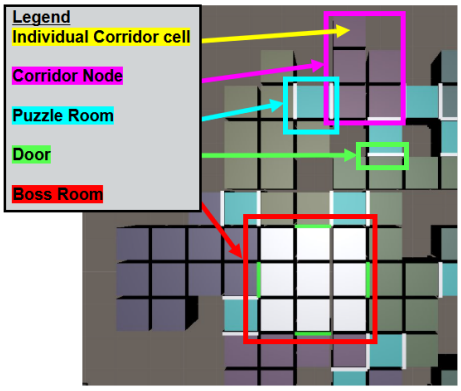


Fig. 2: Diagram of the different dungeon components

before connecting the components together. This ensures that there are many paths to connect to each originally sampled room. Next, using the pre-generated trees as a starting point gives us more ways to connect different rooms. This leads to multiple paths between rooms and it can also be helpful for 3D dungeon generation where the requirement of having certain geometries of stairs can heavily restrict vertical movement. Lastly, SRT would allow for parts of the dungeon generation to happen in parallel, which would increase the performance of the algorithm.

III. Methodology

We define a discrete environment as a grid of cells with doorways connecting adjacent cells. Our dungeon is made up of a set of dungeon components that take up a number of cells and/or doorways as seen in Figure 2. When a cell takes up a doorway, it can either place a door in the doorway to make it passable or it can place a wall in the doorway to make it impassable.

We define three types of dungeon components: Corridors, Puzzle Rooms, and Boss Rooms.

Corridors are a set of cells that are connected together without any doors between them, meaning the player can freely move through a Corridor.

Puzzle Rooms are a set of cells with a start door and an end door, with walls on every external edge that is not a door. Puzzle Rooms are a challenge to pass through so it is important that the player is never required to solve any Puzzle Room since it might be too hard for them.

Boss Rooms are n x n square of cells with doors coming off in each cardinal direction and walls on every other external edge. There will be several Boss Rooms in the dungeon and it is important that the player has multiple paths to reach each Boss Room.

Our task is to make a dungeon generator with the following properties:

1) The dungeon generator can take in at least 10 different Boss Room coordinates. The generator should be able to create a dungeon that connects to all of these Boss Rooms.
2) The dungeon generator should not connect two sides of a Puzzle Room with a Corridor. This would make the Puzzle Room pointless.
3) The dungeon generator should generate the dungeon in such a way that the player has at least 3 choices of Puzzle Rooms at every location.

The Dungeon Generator will achieve these properties by first using a local planner to generate parts of the dungeon around each Boss Room. The Dungeon Generator will then attempt to join the different pieces of the dungeon together. Meanwhile, the validity checker will ensure that the dungeon that is created follows property 2 and avoids self-intersections.

We can break the Dungeon Generator into 3 sub-algorithms: the Validity Checker, the Local Planner around each Boss Room, and the Global Planner attempting to connect the Boss Rooms.

A. Validity Checker

In order to create a dungeon that avoids self-intersections and follows property 2, we create a validity checker that is called whenever we want to add a new component to the dungeon.

The validity checker first checks that the new component does not intersect any part of the existing dungeon. This means ensuring that none of the new component's cells will be placed on an existing cell and making sure that none of the new component's walls or doors will be placed on an existing door. The new component's walls are, however, allowed to intersect with existing walls.

The next step of the validity checker is to make sure that the new component does not result in a useless puzzle room as described in property 2. This property can be broken in two ways:

- A new Cell can be added which creates a Corridor that connects the two sides of a Puzzle Room
- A new Puzzle Room can be added with both of its doors opening to the same Corridor

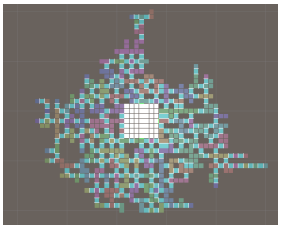
If either case is detected, the validity check will fail.

B. Local Planners

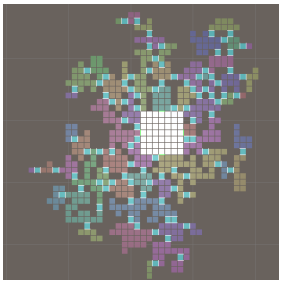
The goal of our local planners is to grow a dungeon around a given Boss Room while attempting to follow property 3 as closely as possible. For the dungeon, we can define our nodes as Corridors and Boss Rooms while our edges are Doors and Puzzle Rooms. This way we can define property 3 as "every node must have at least 3 neighbors."

1) Expansive Spaces Trees (EST) [4]: Considering that we want to prioritize connecting Corridors with few neighbors, EST seems like a logical choice of planner. EST chooses to expand Corridors based on a weight, $w = \frac{1}{1+\#neighbors}$. By expanding Corridors with few neighbors, we are likely to connect them to other Corridors, thereby increasing how many neighbors they have. This should result in the well-connected dungeon that we desire.

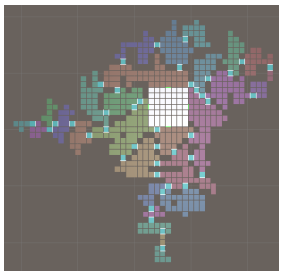
The next step is to determine if the node should expand by adding a Cell or by a Puzzle Room. When the node is small



(a) r=0.5



(b) r=0.05



(c) r=0.01

Fig. 3: Running EST for 1000 iterations with different decay parameters. Colored sections are rooms and the blue and white cells are 1x1 Puzzle Rooms.

and just has a few Cells, we probably want to add more Cells, but when the node gets big, we want to prioritize adding more Puzzle Rooms. We can get this behavior by using the function $P(AddCell) = e^{-rn}$ to determine our probability of adding a cell, where n is the number of cells, and r is a decay parameter that must be tuned.

Finally, we must choose how to expand each node. For this, we just expand from the node in a random valid direction.

In Figure 3 we can see what it looks like to run EST for three different decay parameters. In Figure 3a we see that $r = 0.5$ is probably too large of a decay parameter, creating rooms with just a few cells in them. Meanwhile, looking at Figure 3b and Figure 3c, we see rooms of a more reasonable size. These larger rooms also do a better job of connecting to multiple other rooms, giving the player lots of options.

However, looking at the generated dungeons, we see a problem. All of the dungeons are very tightly clustered around the center, leaving no open space. It would be desirable for the generation to leave some holes to limit player motion so they have to make large scale navigation decisions on which way they want to go.

2) KPIECE [5]: In order to try to create dungeons that explore the space better, we can use a simplified version of KPIECE. In this case our high-level projection is the Corridor we decide to expand while our expansion is the single Cell or Puzzle Room we add.

In this case, we can consider KPIECE as an extension of EST since we still have weights that are inversely proportional to the number of neighbors a Corridor has. For KPIECE we just have to add a few extra terms. We can say that our expansion weight for each Corridor is, $w = \frac{\log(i)}{sn}$, where

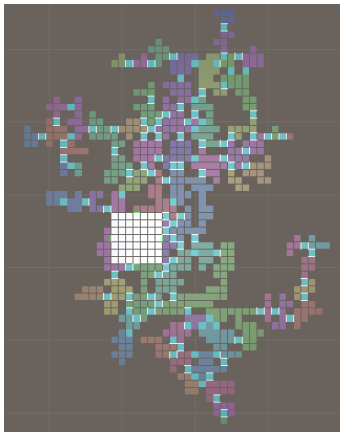


Fig. 4: KPIECE dungeon generation after 1000 iterations with r=0.05

i is the iteration the first cell in the Corridor was created, s is the number of times the Corridor was expanded, n is the number of neighbors the Corridor has. For the number of expansions, s, we also count expansions that fail due to the validity checker, this way if a Corridor fails to expand too many times, we start to ignore it.

Just like EST, we expand in a random direction and use a decaying exponential to determine the probability of adding a Cell vs a Puzzle Room.

Looking at Figure 4 we see that KPIECE did better than EST at leaving some space in the dungeon, thereby making a more interesting dungeon to explore. It also did a good job at maintaining a lot of connections for each Corridor.

3) Rapidly-exploring Random Tree (RRT) [6]: EST and KPIECE do well at maintaining a high number of connections between Corridors, but even KPIECE struggles to really explore the 2D space. It could be desirable to have a dungeon that expands to fill a larger area without filling in everything on the way. RRT is well suited for this purpose.

To implement RRT, we start by sampling a random point within our generation area. We then find the Corridor that is closest to the sampled point and extend it by one Cell or one Puzzle Room in the direction of the point. Finding the nearest Corridor instead of the nearest cell allows up to speed up the process of finding a nearest neighbor. Additionally, when expanding from a Corridor, we can use the same decaying exponential function we used in EST and KPIECE to determine if we should add a cell or a Puzzle Room.

Looking at Figure 5 we can see that RRT does a much better job of exploring the space than EST and KPIECE. However, this comes at the cost of node neighbor connectivity. Most Corridors appear in a line of Corridors and there are no loops in the dungeon.

4) Mixing Planners: We can mix RRT with either EST or KPIECE in order to get Corridors that connect to a lot of neighbor Corridors while also exploring the space well.

In Figure 6 we can see a generation run where each iteration there was an 80% chance to expand the dungeon

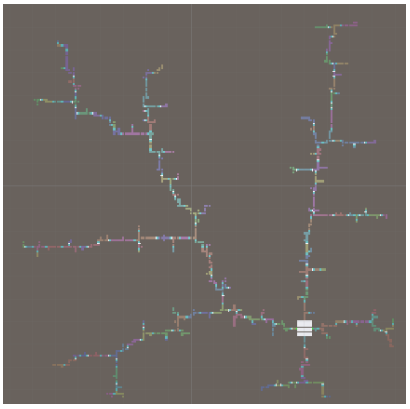


Fig. 5: RRT dungeon generation after 1000 iterations with r=0.05

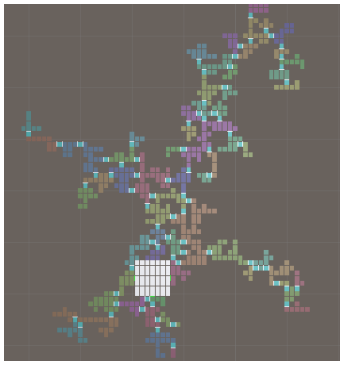


Fig. 6: Dungeon made with a mixed KPIECE and RRT planner. Each iteration there was an 80% chance to use KPIECE and a 20% chance to use RRT.

using KPIECE and a 20% chance to expand the dungeon using RRT.

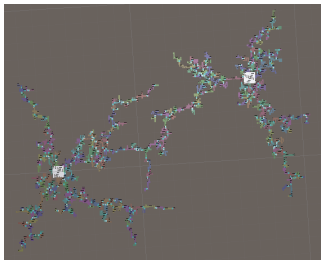
C. Global Planner

We can handle multiple boss rooms by having each Boss Room grow a dungeon around it until it collides with the dungeons formed by the other boss rooms. This process can work with the mixed planner described in the last section, but it is inefficient when it comes to connecting boss rooms together in a large space. Therefore, we need to modify our planner to encourage the Boss Room dungeons to grow towards each other.

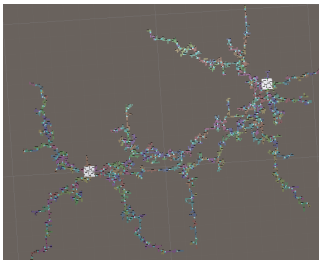
We can do this by adding a chance to sample centers of Corridor Nodes from nearby Boss Room Dungeons as the points to expand towards.

Looking at Figure 7 we can see that adding the sampling points on the other sub-dungeon helps to increase the number of paths between the two sub dungeons.

Finally, once we have a dungeon that adequately connects the Boss Rooms together, we need to trim some of the extra Corridors that only have 1 or 2 neighbors to make sure that we are following property 3. We must do this trimming iteratively

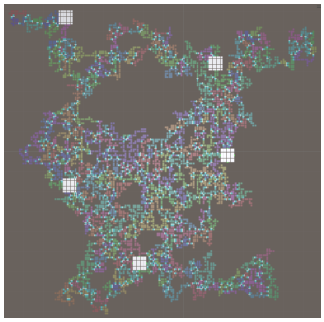


(a) 0% chance to sample other dungeon node

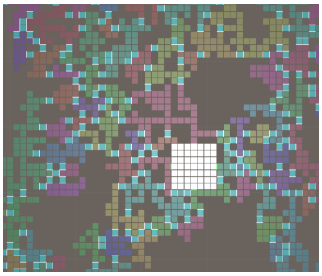


(b) 10% chance to sample other dungeon nodes

Fig. 7: KPIECE-RRT with and without chance to sample nodes from the other sub-dungeon



(a) Full Dungeon View



(b) Closeup

Fig. 8: RRT-KPIECE run for 20000 iterations followed by trimming of under-connected Corridors

to ensure that no Corridors remain without an adequate number of neighbor Corridors.

Looking at Figure 8 we can see that after running RRT-KPIECE for 20000 iterations and then trimming Corridors with less than 3 neighbors, we are left with a suitable dungeon. From Figure 8a we can see that all the Boss Rooms are connected together in multiple ways and from Figure 8b we can see that all Corridors are connected to multiple other Corridors, giving the player multiple options. The dungeon also has a lot of holes, making for a more interesting dungeon to explore.

IV. Platform and Evaluation

A. Platform

All code was implemented in C# and Unity. The code can be run as follows:

1) Ensure that Unity-Hub and Unity 6.0 are properly installed on the test machine.
2) Locate the C# package containing the files for the Dungeon Generator and add them as a project in Unity Hub.
3) Launch the project by clicking on the project's name.
4) Navigate to the 'SampleScene.unity' file within the 'Assets' folder, and click on it.
5) Press the play button on the top of the screen, and click on the 'Scene' tab on the top left.

B. Evaluation

We generate 5 dungeons with 5 Boss Rooms each on a 200 by 200 grid using RRT-KPIECE. We allow the dungeon to expand for 25,000 iterations and then we trim away Corridors that are connected to less than 3 Corridors. We visually inspect each dungeon to ensure that the generated dungeons follow the following properties:

1) Boss rooms and origin are all sufficiently spaced out
2) No cells overlap
3) No doors overlap
4) No doors open into the side of a Puzzle Room
5) Check all Puzzle Rooms to ensure that both ends do not connect to the same Corridor
6) Check that all Boss Rooms are connected together with multiple paths

During the evaluation, we found that almost all parts of the 5 dungeons matched the above criteria. There are just a few rare cases where both sides of a Puzzle Room end up connecting to the same Corridor. Considering how rare this phenomenon is, it is unlikely to effect the playability of the dungeons in general. However, the fact that it exists at all suggests that there is a bug in our validity checker. This bug could likely be eliminated in future work on the project.

Generation of the dungeons took an average of around 25 seconds to execute. This is an adequate amount of time considering that these dungeons were on the same scale as 8a which would take many hours of play-time to complete. This generation cost could also be incurred incrementally if a faster loading time is desired.

The only other error that was found during generation was the occasional occurrence of floating islands that do not connect to the rest of the dungeon. These islands do not seriously negatively impact the dungeon and they could be removed with a with a simple algorithm if needed.

V. Limitations and Future Work

This project is meant to be used as a base from which further development of the dungeon game can be built on. As such, the algorithm does not generate a playable dungeon, it only generates a visualization that shows that the algorithm is working as intended. Additionally, the project is limited to 2-dimensional (2D) generation. While the objects on the map will appear 3-dimensional (3D) to the player, they are 3D objects placed on top of a 2D map, and the routes taken to the bosses will all be 2D. Next, while the block framework used for the Corridor allows for easy collision detection, it also limits the number of directions that can be directly traveled to 4, and while it is possible to travel in a diagonal direction, it would create a jagged Corridor. Future work outside of the scope of the project could be done to optimize and smooth diagonal movement and turns. Lastly, the dungeon currently has to be a pre-allocated size. The nature of the generation algorithm allows would allow for incremental generation as the player moves around, but this would require a fair amount of additional work.

After the scheduled work for the project was completed and an MVP was created, there are some additional limitations that were presented based on the implementation of the project. Currently, the dungeon creator has to manually tune several parameters for their dungeon, such as how many iterations the algorithm should run in order to ensure each boss room is connected. Another parameter that has to be tuned is the RRT to KPIECE probability ratio. If the map is really large, it might be more ideal to have a larger probability of RRT being called. Future work could automatically pick ideal parameters for a given map size and number of dungeons.

There are also several validity check anomalies with the current implementation. Occasionally, a Puzzle Room will be placed somewhere where both ends connect to the same Corridor, making it pointless. However, steps have been taken to reduce the occurrence of this bug so that it only occurs very rarely. While not ideal, the bug does not break anything else so it is acceptable. Lastly, there are some limitations to trimming. Over-trimming can result in some parts of the map becoming isolated islands from the rest of the map. Future work could fully delete isolated islands.

VI. Contributions

Many of the tasks were done collaboratively, but the work was roughly broken down as follows. Jaskrit developed the initial IComponentGeometry framework and environment set up. He also made the sub-dungeon structure around each Boss Room and made the KPIECE, EST, and mixed planners. Meanwhile, Luke focused on the graph representation of the map, the RRT planner, and boss room generation. Other tasks including the implementation of Corridors and the validity checker were completed collaboratively during team coding sessions using Visual Studio Code Liveshare.

VII. Conclusion

We have created a novel dungeon generation approach that uses motion planning. This approach allows us to control the connectivity of each Corridor in the dungeon through the use of EST and KPIECE. It also allows us to control how expansive the dungeon is through the use of RRT. Compared to other dungeon generation methods, our approach allows for greater flexibility in having custom dungeon components that appear between the randomly selected points for Boss Rooms. Finally, since our dungeon generator was implemented in Unity, it is ready to be added to Daedulus' Dungeon and it is also ready to become an asset on the Unity Asset Store.

References

[1] B. M. F. Viana and S. R. dos Santos, "A Survey of Procedural Dungeon Generation," 2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), Rio de Janeiro, Brazil, 2019, pp. 29-38, doi: 10.1109/SBGames.2019.00015.

[2] A. Santamaria-Ibirika, X. Cantero, S. Huerta, I. Santos and P. G. Bringas, "Procedural Playable Cave Systems Based on Voronoi Diagram and Delaunay Triangulation," in 2014 International Conference on Cyberworlds (CW), Santander, Cantabria, Spain, 2014, pp. 15-22, doi: 10.1109/CW.2014.11.

[3] E. Plaku and L. E. Kavraki, "Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning," Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005, pp. 3868-3873, doi: 10.1109/ROBOT.2005.1570711.

[4] D. Hsu, J. . -C. Latombe and R. Motwani, "Path planning in expansive configuration spaces," Proceedings of International Conference on Robotics and Automation, Albuquerque, NM, USA, 1997, pp. 2719-2726 vol.3, doi: 10.1109/ROBOT.1997.619371.

[5] I. A. Sucan and L. E. Kavraki, "A Sampling-Based Tree Planner for Systems With Complex Dynamics," in IEEE Transactions on Robotics, vol. 28, no. 1, pp. 116-131, Feb. 2012, doi: 10.1109/TRO.2011.2160466.

[6] L. S, "Rapidly-exploring random trees : a new tool for path planning," Research Report 9811, 1998, Available: https://cir.nii.ac.jp/crid/1573950399665672960