5-2015

Procedural Content Generation: Using A.I. to Generate Playable Content

Osler Kendall Moore Jr.

The University of Southern Mississippi

Procedural Content Generation: Using A.I. to Generate Playable Content

by

Osler Kendall Moore, Jr.

A Thesis
Submitted to the Honors College of
The University of Southern Mississippi
in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science
in the School of Computing

May 2015

Approved by

Bikramjit Banerjee, Ph.D., Thesis Advisor
Associate Professor of Computer Science

Andrew H. Sung, Ph.D., Director
School of Computing

Ellen Weinauer, Ph.D., Dean
Honors College

Abstract

This thesis is about using Artificial Intelligence in regards to Procedural Content Generation to help try and avoid a deadlock scenario in a hypothetical dungeon exploring game. The deadlock scenario is essentially having a key being placed in a room that's inaccessible to the player. An example of such a scenario would be if the key to room A being in Room B, the key to room B was in room A, and the player was in room C. The program generates the dungeon world with a random number of rooms inside it, each room also having a random number of generated sub-rooms or no sub-rooms at all, and then will place the rooms' respective keys somewhere in the world, so long as whatever arrangement occurs in the end will allow the player to explore the entire dungeon world if they desire or need to. The entire program was written in Java, and the dungeon world's rooms are arranged into a tree data structure to represent each room's relationship to one another.

Key Terms: Progressive Content Generation (PCG), Artificial Intelligence (AI), Deadlocks, Dungeon Game, Generate Playable Content, Java, Search Tree, and Procedural Generated Content (PGC)

Acknowledgements

I'd like to thank my Advisor Dr. Bikramjit Banerjee for all his efforts mentoring me during this study. I'd also like to thank the entire staff of the School of Computing, the Honors College, and the Office of Disabilities Accommodations for the many things I've learned and the support I've received during my years here at USM.

Table of Contents

List of Figures

List of Tables

Chapter 1: Problem Statement

This thesis is about Procedural Content Generation: Using Artificial Intelligence to Generate Playable Content. What this means is that I will be creating a program that uses Artificial Intelligence (A.I.) to check for deadlocks within a game.

For this project's purpose, the game will be a simple dungeon exploration type of game, where the goal of the game is to collect keys to open up different rooms of the dungeon. Deadlock in the game will occur whenever two or more rooms are impossible to enter and explore because each locked room will have the other room's key, thereby preventing the player from advancing (see Figure 1.1 below.) This deadlock scenario may also occur with more than just 2 rooms (such as 3 or 4 or more rooms), or if the key necessary to enter the next room is in the room itself.

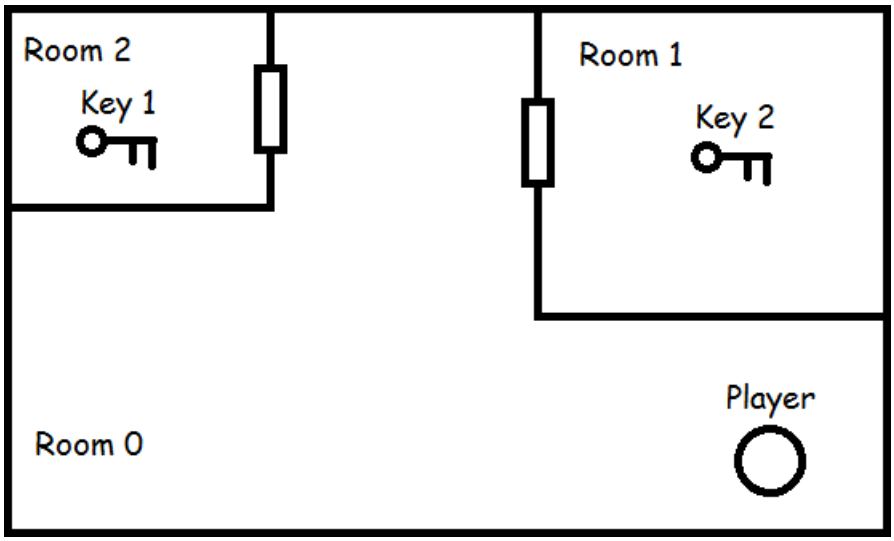

Figure 1.1 – Model of Deadlocked Scenario between two rooms.

Locked rooms already generated whose keys have not been generated yet, or keys already generated whose locked room has not been generated yet is completely fine, as the missing keys or rooms will eventually be generated later in the game. Or the extra keys and locked rooms may not be needed to reach the goal at the end of the dungeon.

There are two possible ways that the A.I. could check content and ensure a deadlock does not occur. One method would be to have an interface or application that checks for any deadlocks while the game or level is being assembled, and will prevent the game from starting if a deadlock is found. Another method is to have the A.I. do consistency tests while the game is actually running. As the player reaches certain points in each level, more rooms will be generated and added onto the original dungeon map. While the rooms are being generated, the A.I. would check each room and the inside of each of these rooms and ensure that the key to the next room the player needs to go to will be accessible to him or her.

Chapter 2: Literature Review

Procedural Content Generation can use a search-based method to generate elements of a game's level to help alleviate the programmer from the work of designing each part of the level by himself or herself as well as allow players some more unique gameplay each time the user runs the game. Artificial intelligence methods have been more recently used in PCG's. Interactive evolutionary computation (IEC), is one A.I. method within PCG's that takes user's actions within the game to help guide the evolution (or changes) of the content within the game, to also help increase the uniqueness of each gameplay based on the user's own actions. (Risi et. al. 2012.)

Togelius et al. (2010) give a few reasons as to why one would prefer PCG over having to hand design each level. First is the concern of memory, and that any content that's procedurally generated can be represented by only a few numbers and will remain undeveloped until it is accessed later in the game. Second is the convenience of not having to manually design each level on one's own. Third is the appearance of a new type of game, not just in terms of being endless, but also the possibility of increasing the replay value by having the game adapt and adjust to a player's style. Their fourth argument is that PCG's could help increase a designer's imagination in regards to what kinds of levels can be developed (Togelius et. al. 2010.)

Procedurally Generated Content has a few distinctions that can be made about it as well. PCG can be done either online, when the game is running, or offline, while the game is still being designed. An example of the online case is when a player enters into a

room and the interior of the room and its content is generated as the player opens the door to enter the room. The offline case would be when the A.I. or another algorithm produces a level layout and then the designer would modify that level as he or she saw fit. (Togelius et. al. 2010.)

Another distinction would be whether or not the content generated is necessary (as in, the player must interact with this content in order to reach a goal) or optional (just bonus content like extra items or alternate paths that are not needed to reach a goal.) PCG algorithms could take in random seeds to determine how the game world is created, or it can take in a "multidimensional vector of real-valued parameters that specify the properties of the content it generates." Whether or not the generation algorithms given the same set of parameters generates the same content is another distinction to be made regarding PCG's. The final distinction is between constructive algorithms and generate-and-test algorithms. The constructive case is that the content is generated one time only, but as it is constructed it tests the content to make sure it is appropriate to put in the level. In the generate-and-test case, the content is generated, then it is tested. If it passes, it stays, but if it fails then a certain amount of the content (be it the entire content or only part of it) is deleted and then generated again. The process of generate-and-test is repeated until the content passes (Togelius et. al. 2010.)

Search-based procedural content generation (SBPCG) actually follows the generate-and-test distinction of PCG's. Rather than saying a content is good or bad in its test though, it will give each content a numeric grade based on how well the content passes its fitness function. Newer generated content is ideally supposed to have higher fitness compared to the older content. Using an Evolutionary Algorithm (EA) for its

searching mechanism, it will evaluate each generation of content, throw out the lowest scoring content, and then replace the thrown out content with randomly modified versions of the higher scoring content.

Risi et al. (2012) also describe a "variation of artificial neural networks (ANNs) that differ in their set of activation functions and how they are applied" called compositional pattern producing networks (CPPNs) that're used in helping to produce different varieties of flowers in a Facebook game called Petalz. There are a few other differences between CPPNs and ANNs. "CPPNs were used as pattern-generators rather than controllers" (Risi et. al. 2012.) CPPNs include many types of activation functions while ANNs are limited to only sigmoid or Guassian. CPPNs can also be viewed at whatever resolution may be desired while ANNs cannot. CPPNs can be used in more than just a simple 2D image, or 3D images. They can also be used to advance compositions of music, or even be used to help advance or develop better weapons in more action-based games (Risi et. al. 2012.)

The NEAT algorithm (Neuroevolution of Augmenting Topologies) can be used on the CPPNs or even ANNs to help better evolve them and also "is fast enough to run in real time, which is required for an interactive system" (Risi et. al. 2012.) The content can basically become more complex each generation (Risi et. al. 2012.)

Dang and Champagnat (2013) go over a couple of authoring tools that are used for debugging interactive scenarios and explain why some of these other methods aren't as good as the method they are trying to present in their paper that uses Linear Logic. One of these methods happened to be from an earlier paper they worked on with another person in 2011, and mentions the flaw with their previous work is that "this work only

enables users to create scenarios without deadlocks" and that it "is not enough for the notion of "valid scenario" in our approach (where such a scenario has to satisfy many more criteria: no deadlock, no unused modeling elements, not too simple/short, structuralized, etc.)." The next method they bring up, using a KANAL program, is designed to apparently test out military application plans or biology models, thereby not being a good choice for interactive scenarios. "Thus, the current state of the art shows that an authoring tool, designed for normal users, enabling a scenario analysis at the structural level with much important information, is necessary." Dang and Champagnat's authoring tool is to create valid interactive scenarios for users that apply the deduction rules in linear logic. (2013)

Dang and Champagnat (2013) give their authoring tool the following requirements: (see Table 2.1 below)

1	To give users a graphical language to help model a story's scenario even if the user has no knowledge of Linear Logic.
2	Give users graphical language to help establish the scenario's parameters that will be used in analyzing the scenario
3	Automatically build the graph to show all the possible ways the scenario can turn out and then after analyzing this graph return important information regarding the scenario's flaws.

Table 2.1 – Requirements of the Authoring Tool. Dang and Champagnat 2013.

Dang and Champagnat (2013) further clarify that to meet these requirements this authoring tool will need four things: First is a metamodel to help users with no prior knowledge of LL to model their scenarios, second is a metamodel to set up the parameters of the scenario being modeled, third was to make a function that graphically shows the results of the two metamodels, and fourth was to make a program that acts as an analysis module that would execute the two tasks of the third requirement from the table above. They then proceed to go into greater detail regarding the data models they used as well as other examples regarding the tool they have created. Figures 2.1 and 2.2 give Dang and Champagnat's metamodels they use in their authoring tool.

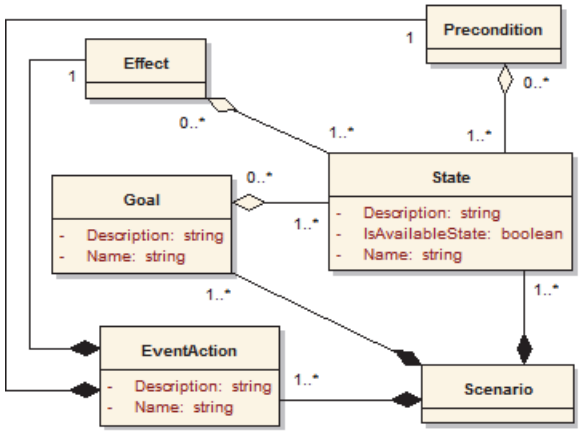

Figure 2.1-Dang and Champagnat's Interactive Scenario Metamodel. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

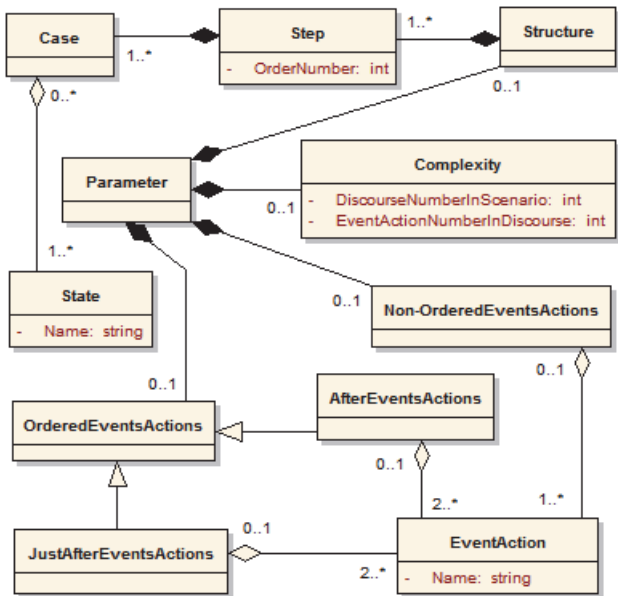

Figure 2.2- Dang and Champagnat's Metamodel of Parameters. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

Dang and Champagnat (2013) give an example as to how their authoring tool works. Their example is an educational game that's supposed to teach the players about household electrical accidents, and give the goal of trying to shock the player. First they define the states of the scenario (see Table 2.2 below), then the list of events and actions (see Table 2.3 below), and then define the goals of the scenario (see Table 2.4 below.)

Name	Is Available State	Description
Pi	True	The player is at the initial position (this state is available)
Pk	False	The player is in the kitchen
Pb	False	The player is in the bathroom
Ik	False	The IS controller starts the strategy of causing the electric shock for the player in the kitchen
Ib	False	The IS controller starts the strategy of causing the electric shock for the player in the bathroom
Pe	False	The player has got the electric shock

Table 2.2 - List of states defined in the example. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

Name	Description	Precondition	Effect
EA1	The player goes to the kitchen	Pi	Pk
EA2	The player goes to the bathroom	Pi	Pb
EA3	The IS controller starts the strategy of causing the electric shock for the player in the kitchen	Pk	Pk, Ik
EA4	The player gets the electric shock in the kitchen	Pk, Ik	Pe
EA5	The player gets the electric shock in the bathroom	Pb, Ib	Pe

Table 2.3 - List of events and actions defined in the example. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

Goal	Description	State
G	The player gets the electric shock	Pe

Table 2.4 - List of goals defined in the example. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

Once this example is tested in the authoring tool, the following graph is produced (see Figure 2.3.)

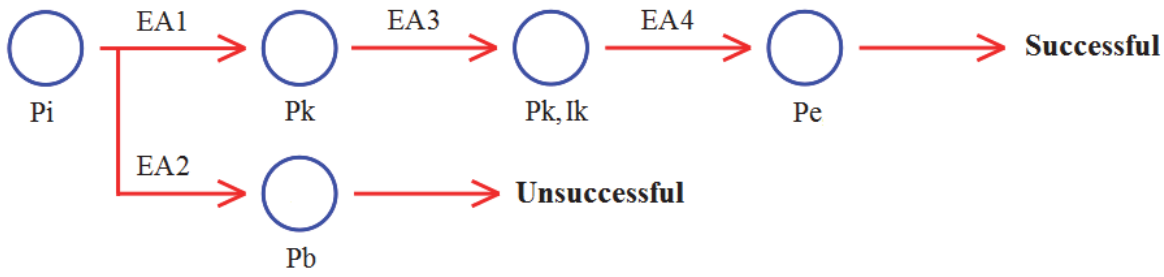

Figure 2.3 – Graphical output of the discourses of the example. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

Because of this graphical representation we can see that a deadlock occurs in Dang and Champagnat's example if the player enters the bathroom (2013). So by adding another Event/Action to the previous list of events and actions (table 2.5 below) and run the tool again, we get a graph as shown by figure 2.4 below, thereby getting rid of the previous deadlock.

Name	Description	Precondition	Effect
EA6	The IS controller starts the strategy of causing the electric shock for the player in the bathroom	Pb	Pb, Ib

Table 2.5 – New Event/Action to add to the previous list of Events and Actions (refer back to Table 2.3). Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

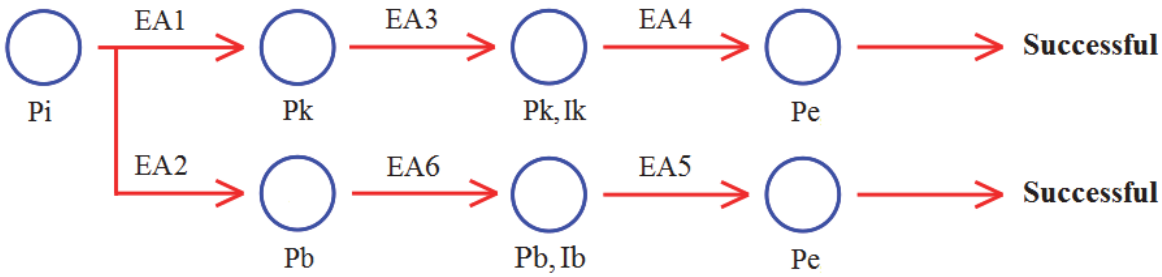

Figure 2.4 – New Graphical output of the discourses of the example. Adapted from "An Authoring Tool to Derive Valid Interactive Scenarios" by Dang and Champagnat in 2013.

With this new version of the scenario of their example, they now have no deadlocks occur (Dang and Champagnat 2013).

Smith et al. (2010) describe a 2D platformer level design aid that can help a level designer reduce the time needed to design a level. By using a set of programmed constraints and reactive planning, the A.I. aid can make appropriate changes to the different components of a generated level every time the designer decides to modify

certain elements (Smith et. al. 2010.) Tanagra uses at least two different Java-based languages. ABL (A Behavioral Language) is used for any sort of rapidly changing world state in a level. Chobo is used more for satisfying constraints, more specifically in Tanagra's case constraints in the level generation. ABL would monitor any actions the designer has done to manipulate the geometry of the different level elements or the beats. In Tanagra, "beats" are the different player actions in a level and help to tie the patterns between different objects or even give the patterns their different sizes. Choco itself takes what constraints to the different level objects that ABL gives it, and then will spit back out solutions for each object to allow it to be generated and allow the level to continue to be generated (Smith et. al. 2010.)

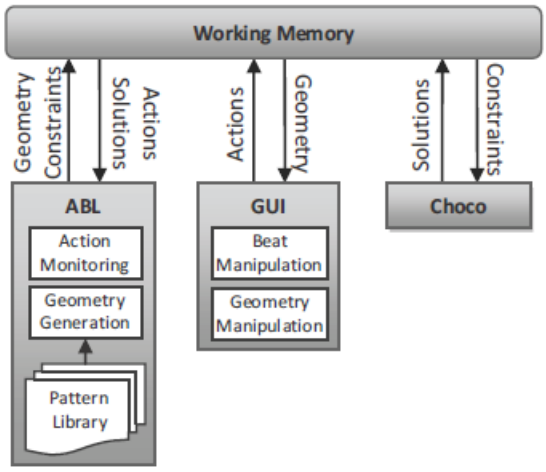

Figure 2.5 – Architecture Diagram explaining the interplay among ABL, Choco, and the GUI of the designer. Adapted from "Tanagra: An Intelligent Level Design Assistant for 2D Platformers" by Smith et al. in 2010.

Van der Linden, et al., suggest a method of using the restraints in a user avatar's gameplay to help generate the content in a PCG game, which could help ensure that there is some consistency in the levels or rooms generated in a game (2013).

Unity can allow components to be made and used using scripts in Java, C#, or Boo and other .NET languages so long as those other .NET languages can compile a compatible DLL (Unity Technologies, 2013.)

Chapter 3: Methods

Data are gathered mostly by trial and error of debugging the programs and personally testing out to see if they behave properly, or within expected parameters. The Program itself is written in Java. The program has at least 8 global variables in the main program, which are listed as follows:

1) maxdepth – an integer value. This variable is set in the main function, and tells us how many levels deep our tree can go. Right now it is hardcoded at a specific value for now, and will most likely be changed later.
2) Nodes – an ArrayList made up of a type Node class. A Node is what constructs most of the tree that makes up the "world" of the game. To be more specific, each Node represents a room in the game.
3) Keys – an ArrayList made up of a type Key class. Keys are normally needed to allow the user accessibility to another room.
4) NodeIDs – a LinkedList of Strings. It is basically a convenient List of Nodes (or Rooms) that still require a key to unlock it.
5) KeyIDs – a LinkedList of Strings, like NodeIDs, which will list all the remaining Keys we have that we still need to place within the world's rooms.
6) NodesPerLevel – a basic ArrayList. It holds how many Nodes there are at each depth of the tree.
7) pendingGoals[][] – A 2D array of Strings. Think of it as being similar to a table. The first column holds different Node ID's. The second column will have either "True" or "False" written in there. I'll go into more detail regarding

pendingGoals' use and importance when I go over the checkAccess function further on down.

8) GoalNode – Right now it is a simple String that tells you what ID the GoalNode (or ending point) of the world will be.

I'll go over the two important custom classes used in this program first before I move onto the many functions the Main Program has, as the Main Program will use these two custom classes for everything it does.

First there's Node. Node is what makes up the individual rooms and sub-rooms of this program, and often it is what makes up how the tree is assembled, representing where the tree may branch or where its leaves (end points that do not branch out into other endpoints) will be located. Each node has only 1 parent, but can have a random number of children, right now the maximum amount allowed being 5. It also will hold a String value that helps act as the node's unique identifier simply called "ID", and a pair of convenient integer values labeled "level" and "nodenum" (short for "Node Number"). It also has a value for how many children it should have. The parent and children are public members, but the other variables are private. The private variables can be set or obtained with the appropriate get and set functions already implemented in the Node class. There is also a print function that acts recursively. The print function simply displays output regarding the node and the tree. It first prints out what level of the tree the node is located, and what node number it has on that level, followed by the number of children it has, the Parent Node's ID, and then it will recursively call the print functions of its children nodes. It also has a pair of constructors.

The Key class doesn't have any fancy functions, mostly just simple set and get functions along with its constructors. It holds 4 integer values and 2 string values, which are connected to one another. The klvl and knum pair help make up the Key's ID, which tells you what room the Key is supposed to open. Location is made up from locationLvl and locationNum, which tells you what room the key is currently placed in.

The main program has a number of functions, 15 in total when you also include the main function. The functions findNode() and findKey() are pretty simple, as they'll take in a string value (the ID of the node or key you wish to find), and after they search through the Node or Key ArrayLists respectively, they will produce the Node or Key you are looking for. The nodeExists() function prompts you to enter a Node ID you may wish to find or look up, and it will check to see if that node Exists or if it does not. If it doesn't, it will continue to prompt you to type in a new response until an already existing node can be found with whatever ID you typed in. Key has a pair of functions almost identical to this but are slightly different. The keyExists() function behaves like nodeExists(), except of course it is looking for a key. The keyRestrictionExists() function behaves the exact opposite, prompting the user for an ID of a key that does NOT already exist. The function titled functionKey() is a special function that will take in a Node, and then it will give the room that the inputted Node's key is located. ResetKeys() will essentially reset all of the Key's location values back to their default value, the root node.

I'll continue going over the remaining functions as I describe what the main function does. After the main function starts and initializes a good deal of our

variables, the first important thing it will do is create the tree that will be used throughout the rest of the program. It'll do this by first creating a "root" Node, which will be used as the starting point for the tree. Once the root is created, we then add it to a temporary LinkedList called nodesNeedKids. So long as the LinkedList is not empty, it loops through the list multiple times, taking the first Node in the list, and checking to make sure that our currentNode is not at the maxdepth. If it is, it checks to see how many children our current Node is supposed to have, and create those children Nodes for our parent node before adding them to the nodesNeedKids List. At the end of each iteration, whether new nodes were added or not, it then removes the currentNode from the nodesNeedKids list before moving on to the next node in that list. This will essentially help create the tree as well as automatically increment the corresponding item in NodesPerLevel. Each "Object" in the NodesPerLevel collection represents how many Nodes are at a certain depth. Figure 3.1 below represents some of the different tree structures the algorithm can make at a maxdepth = 2.

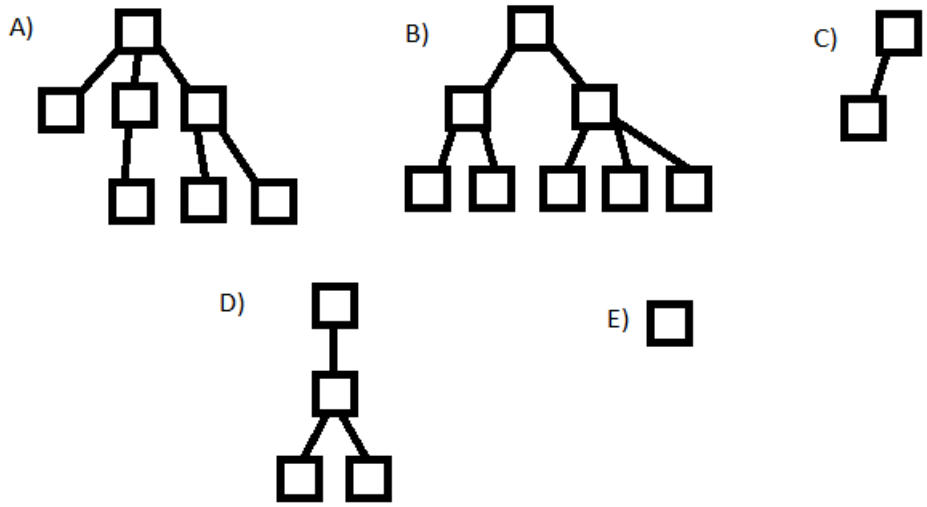

Figure 3.1 – Some examples of different tree structures that can be implemented at maxdepth = 2.

Each Node is represented by a box or square, with the lines representing connections between a parent node and its children. The parent will be the higher node, and the child nodes will be the node or nodes that are immediately below it. Tree A) is made up of 7 nodes, 1 at depth 0, 3 at depth 1 and 3 at depth 2. Tree B) is made of 8 nodes, 1 at depth 0, 2 at depth 1, and 5 at depth 2. Tree C) is made of only 2 nodes, 1 at depth 0, 1 at depth 1, and none at depth 2. Tree D) has 4 nodes, 1 at depth 0, 1 at depth 1, and 2 at depth 2. Tree E) has also occurred on a number of occasions, with only 1 node at depth 0 and none at depth 1 and at depth 2.

Let's go through a step by step example of how it will construct the tree. First it creates the root node, which will always be given the ID "0-0". Then it calls the numkids() function to determine how many kids root will get, passing in what level or "depth" the node is currently at. The numkids function will check the currentNode's depth against the maxdepth. If the currentNode's depth is smaller than the maxdepth, it will give back a random number for how many children the current node will have. Otherwise it will return 0. Below is an example made from an 8 node tree, resulting in it having 1 node at depth 0, 2 at depth 1, and 5 at depth 2. Figure 3.3 shows how this might look as a potential level for the game.

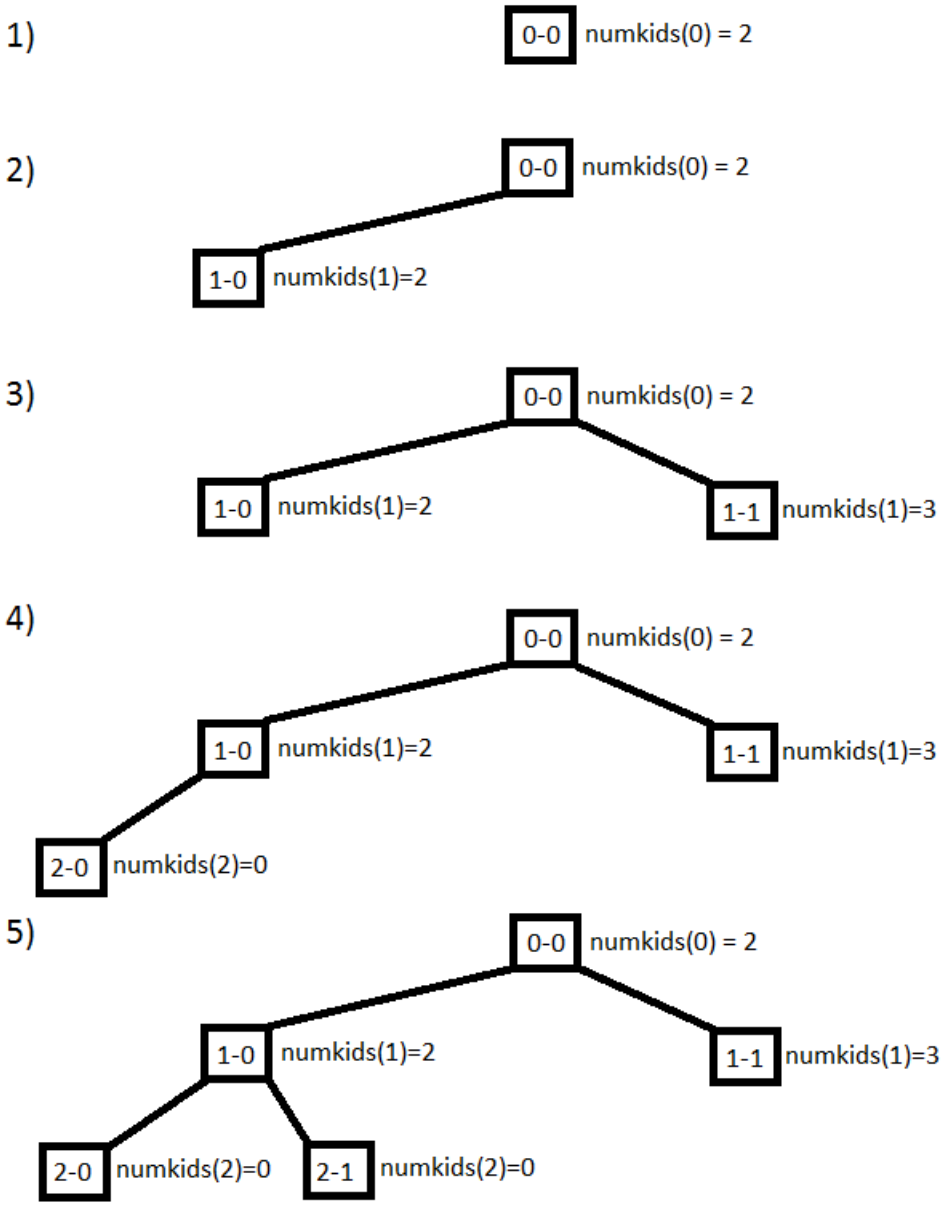

Figure 3.2 – A tree generated with 8 nodes and a max depth of 2 by the program.

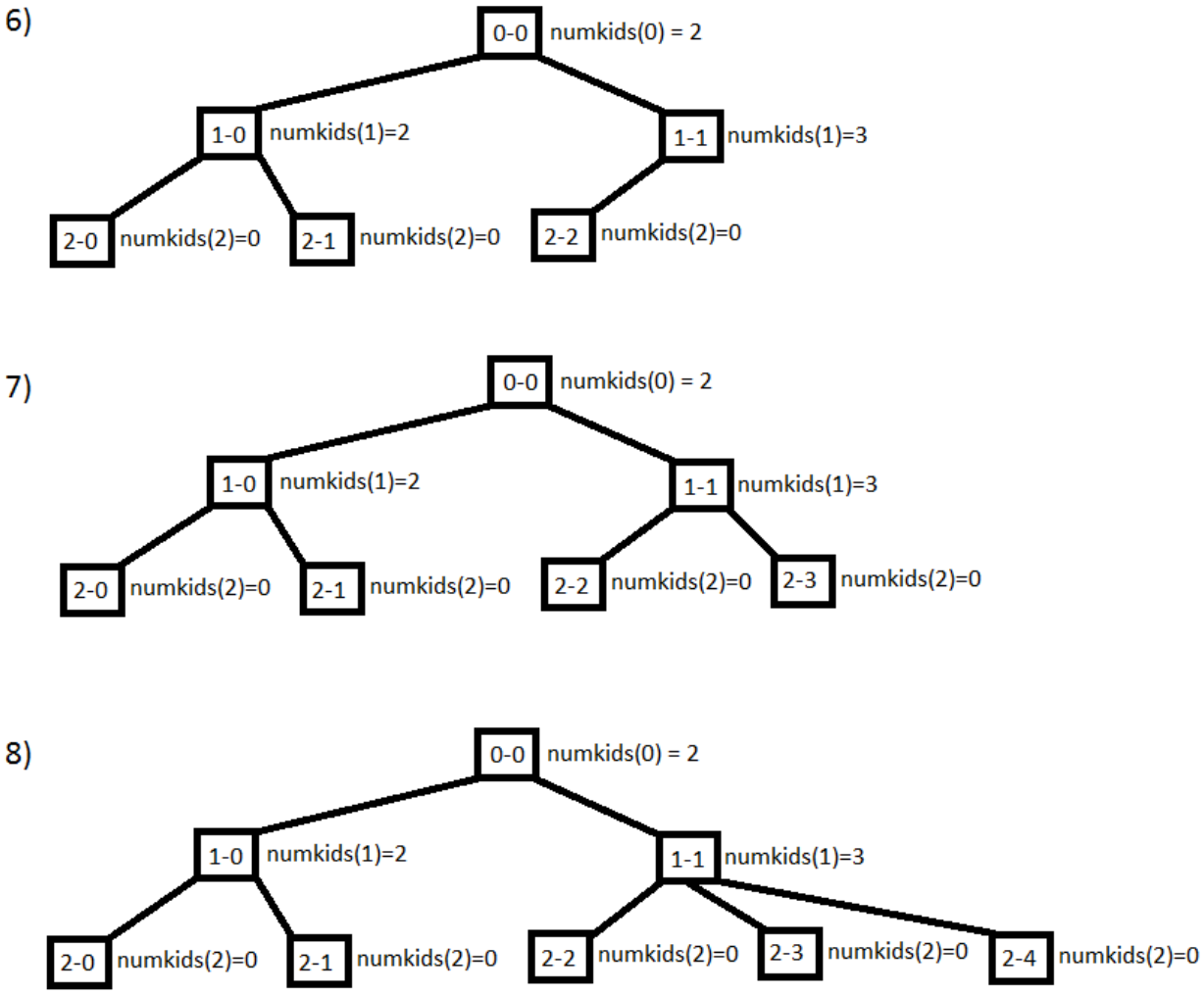

Figure 3.2 – A tree generated with 8 nodes and a max depth of 2 by the program. (continued)

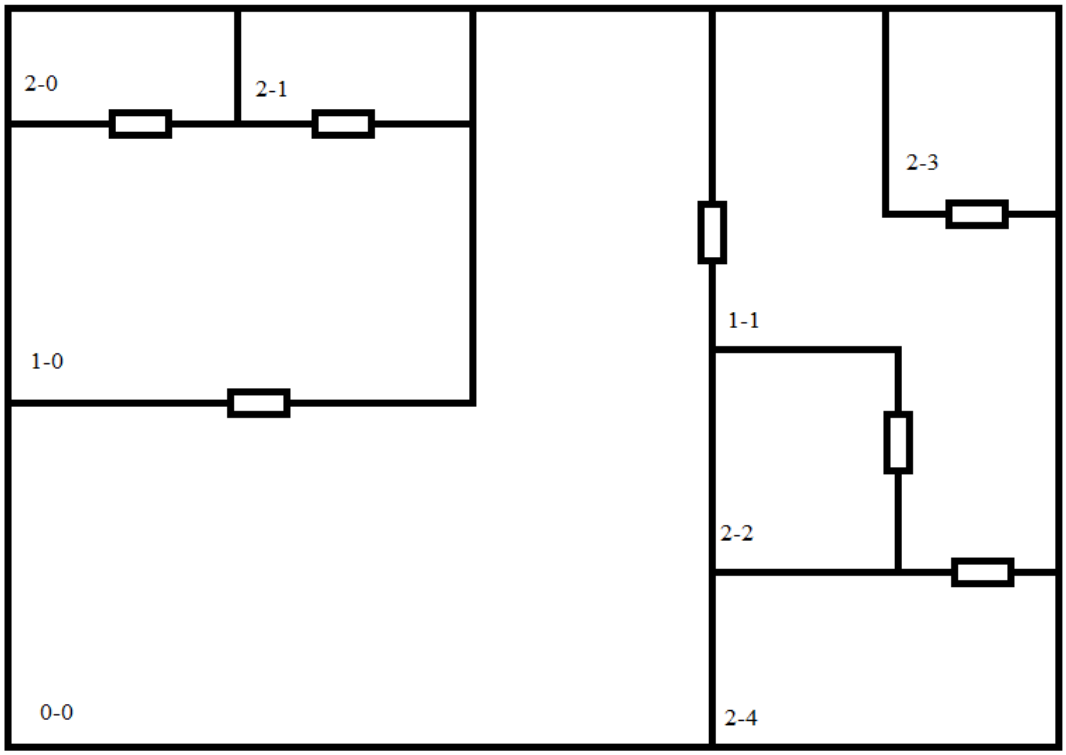

Figure 3.3 – How the tree generated in figure 3.2 may look as a floor plan for a level.

Next we use a function called randomGoal() to randomly determine what our goal node should be. Since we want to go through every room generated, our goal node will most likely be one of our nodes at the deepest level of our tree, so something around depth 2 (aka, any node with an ID like “2-X”). Let’s say it randomly picks out 2-3. I’ll mark our goal node with a star with a “G” underneath it.

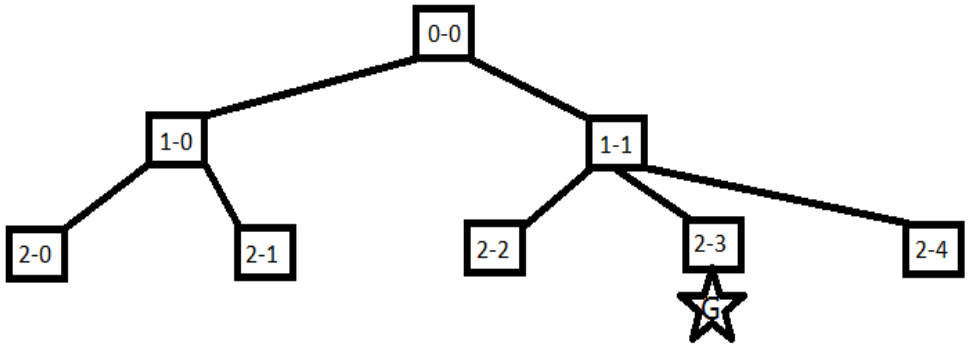

Figure 3.4 – The tree generated with our goal chosen.

Next thing to do is construct the keys to the rooms. As there are 8 rooms, it will immediately construct 8 keys, one for each room, and they will each be initially placed in a “default” position. I have chosen to make the default value the root node, which will always be the room the user will start in whenever the game would be running.

So just to clarify, we have 8 rooms [0-0, 1-0, 1-1, 2-0, 2-1, 2-2, 2-3, 2-4], and 8 Keys [0-0, 1-0, 1-1, 2-0, 2-1, 2-2, 2-3, 2-4] and all the keys will be initially located at 0-0. Our goal is to get to 2-3. But the AI is to randomly scatter the keys in the different rooms and make sure every room and key is accessible to the player. We want to avoid a deadlock, which may occur if a player cannot reach a certain key, and thereby be unable to reach a certain room or the goal.

Below are three possible examples of a deadlock, first being a room with its own key locked inside it, two rooms having each other's keys, and three rooms having each other's keys. Key 0-0 being located inside Node 0-0 is the only allowable exception, as the player will always start in room 0-0 and thus Key 0-0 will already be accessible to him/her, as well as the room itself.

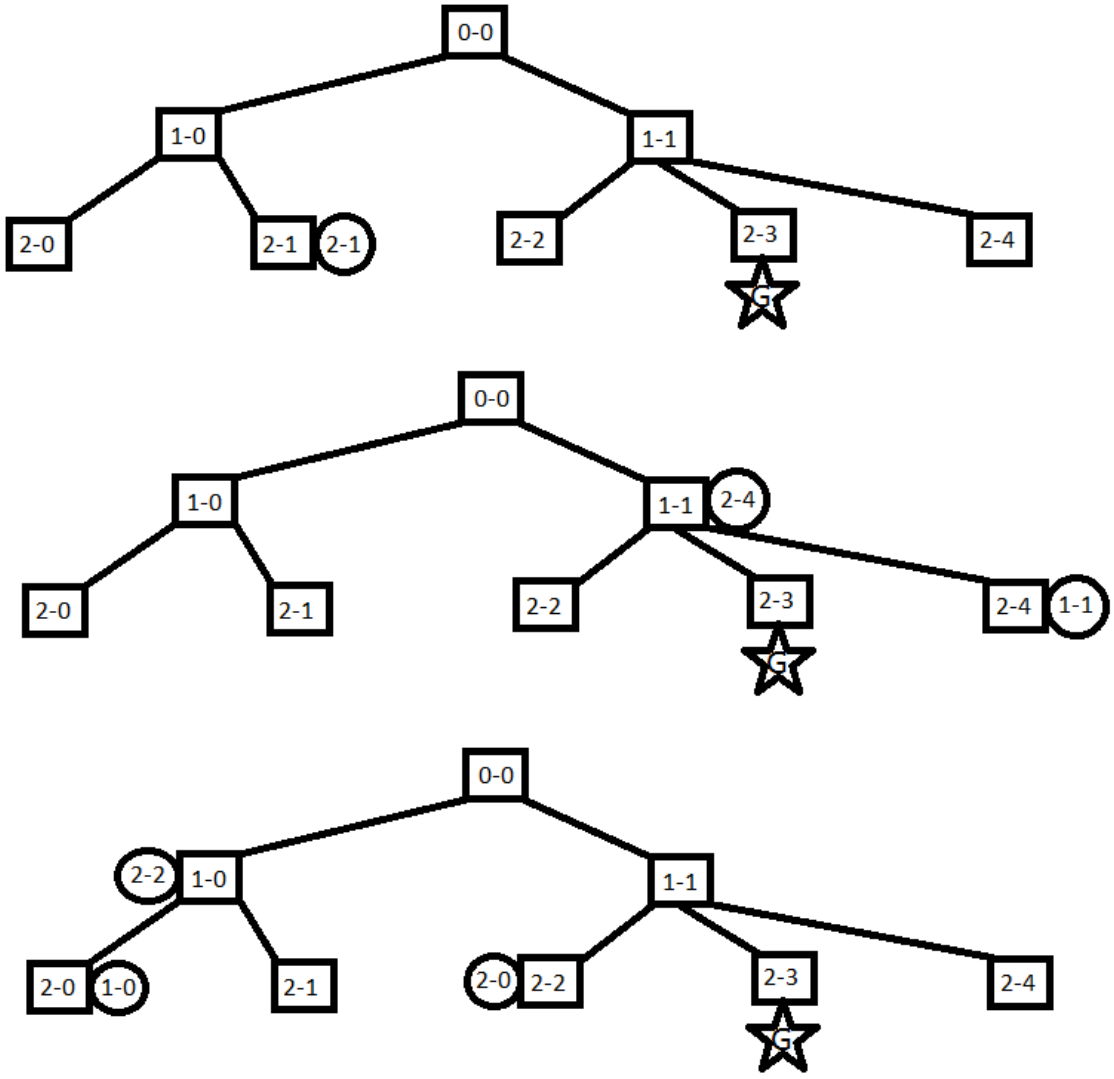

Figure 3.5 – 3 different examples of possible deadlocks that can occur within the program.

So what happens if a deadlock occurs? How can it be avoided? As we are inserting keys into the nodes, we will check to make sure that having the keys inserted at such places will allow them to be accessible.

First we insert a key, say Key to room 1-0, into room 2-2. Next, we set every pending Goal's value for each node equal to False. After that, we run a checkAccess() boolean function, passing in Node 2-2 as the parameter. checkAccess will first check to see if the node passed in is the default value (0-0) and if so will return true. If that doesn't work, it will check the Node's corresponding value in the potentialGoals 2D array. If its value is already set to True, then we have the function return false. However, if the potentialGoals value was instead marked false, we temporarily change that value to true, then call up a path List for the node we are currently checking. A path list essentially comprises of the node, its parent, its grandparent, its great-grandparent, etc. and recursively runs the checkAccess function on the key locations to each value in that list one at a time. In other words, checking the Accessibility of the node's own key, its parent's key, its grandparent's key, etc. If any of the values return false, then the whole function will return false. If none in the path return false, then the whole check function will return true.

Now it starts checkAccess(2-2). It will set 2-2 in potentialGoals to true, then recursively cycle through the checkAccess for each key in Path(2-2). Key[2-2], Key[1-1], and Key[0-0] all equal one node currently, 0-0. So each of the checkAccess being recursively stepped through right now will all become checkAccess(0-0), which returns true in all three situations, so checkAccess(2-2) returns true, and it can keep Key[1-0] in its present place, Node 2-2.

If it tried adding Key(0-0) to node 2-2 however, it would get false returned from the checkAccess function, and as it returns failure, it sets Key(0-0) back to its previous value (default 0-0) and move on to another node.

While the program was generating Nodes and Keys, it also fills up both the NodeIDs and KeyIDs lists. NodeIDs is just a convenient list of strings of all the rooms the program currently has in the tree. KeyIDs help keep track of any keys that have not yet been placed somewhere. It removes Key 0-0 from the list as the program is to assume that the agent (the player) is to have already have Key 0-0 at the start of the program. After each successful checkAccess, it removes a key from our KeyIDs list. It checks to make sure that whatever node ID and key ID currently set are not equal to each other. It will remove the GoalNode from the NodeIDs because the GoalNode does not need a key placed in it. It will place the GoalNode's key last to make sure the Goal is the last place the user will visit in the tree. Any time some condition isn't met, it will randomly select a new Node or Key depending on the situation.

Chapter 4 Results, Discussion, and Conclusion:

Running the code multiple times has led to a variety of different trees that could be generated. In addition to this, it has also lead to a successful possible arrangement of keys for each randomly generated tree.

Here is the output of a tree that was successfully generated and had all its keys successfully placed. First it generates the following Tree:

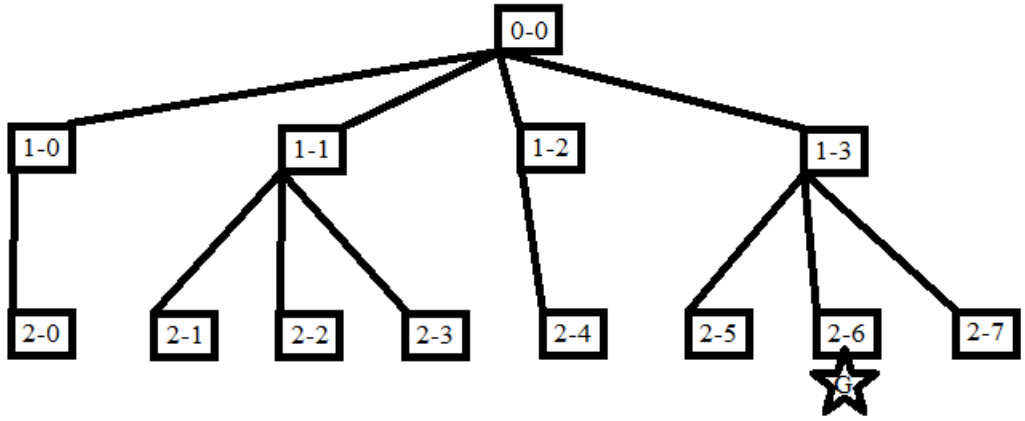

Figure 4.1-Tree diagram of a tree generated by the program

Now that the tree is generated, it begins generating the following output as it randomly picks different keys and different rooms to place them in, as well as checking to make sure that placing a key at such a location is possible.

Testing check on Root Node true

Number of Keys needing to be Placed Left: 12

Testing check Key (2-4) in room 2-3 true

Number of Keys needing to be Placed Left: 11

Testing check Key (2-7) in room 2-1 true

Number of Keys needing to be Placed Left: 10

Testing check Key (1-1) in room 2-0 true

Number of Keys needing to be Placed Left: 9

Testing check Key (2-3) in room 2-0 true

Number of Keys needing to be Placed Left: 8

Testing check Key (2-2) in room 2-1 true

Number of Keys needing to be Placed Left: 7

Testing check Key (1-2) in room 0-0 true

Number of Keys needing to be Placed Left: 6

Testing check Key (2-0) in room 2-1 false

So now it has come across an error. In order to get to room 2-1, the player would need to go to room 2-0 to get the key to 2-1's parent node 1-1. But the player wouldn't be able to get into room 2-0 if its key was already in 2-1. A room is accessible mainly if both its parent node and its key are accessible. It cannot go to room 2-1 without both the key to room 2-1 and the ability to access the parent. The parent 1-1 can only be accessed after

visiting room 2-0 to get its key. 2-0 cannot be accessed however without its key. Thus this makes a circular deadlock with rooms 1-1, 2-1, and 2-0. As 2-0's key is being placed in a room it cannot currently access, the test fails. So now the program attempts to test on another node until it succeeds in placing a key in a room without causing a deadlock.

Testing check Key (2-0) in room 2-4 false

Testing check Key (2-0) in room 2-7 false

Testing check Key (2-0) in room 1-1 false

Testing check Key (2-0) in room 1-2 true

Testing Key 2-0 being placed in room 2-4 fails because the Key to room 2-4 is located inside room 2-3, but the key to 2-3 is currently in room 2-0. This would cause a circular deadlock involving rooms 2-0, 2-3, and 2-4. The only way the Key to room 2-7 can be accessed is by visiting room 2-1, but the only way to access 2-1's parent 1-1 is by visiting 2-0. Thus a deadlock occurs involving 2-0, 1-1, 2-1, and 2-7. Attempting to place Key 2-0 into room 1-1 would simply cause a circular deadlock between the two nodes (2-0 and 1-1) simply because room 2-0 already has the key to 1-1. When it places Key 2-0 in room 1-2 however, no deadlock occurs mainly because not only is its parent 0-0 already initially acceptable, room 1-2's key is also already in 0-0, thus allowing for the key to room 2-0 to be placed with no issue at all.

Number of Keys needing to be Placed Left: 5

Testing check Key (1-0) in room 1-0 false

Testing check Key (1-0) in room 0-0 true

Placing the key to room 1-0 inside itself fails as such a situation would be no different than if a person locked their keys in their car by mistake. Placing any key in room 0-0 will be fine, as room 0-0 will always be accessible to the player.

Number of Keys needing to be Placed Left: 4

Testing check Key (1-3) in room 2-3 true

Number of Keys needing to be Placed Left: 3

Testing check Key (2-1) in room 2-5 true

Number of Keys needing to be Placed Left: 2

Testing check Key (2-5) in room 1-3 true

Number of Keys needing to be Placed Left: 1

Testing check Key (2-6) in room 2-1 true

So now the program has given one possible solution in regards to randomly placing keys in such a way that no deadlock occurs. In the figures below is the tree with its keys labeled at each node, as well as a possible floor plan for such a tree.

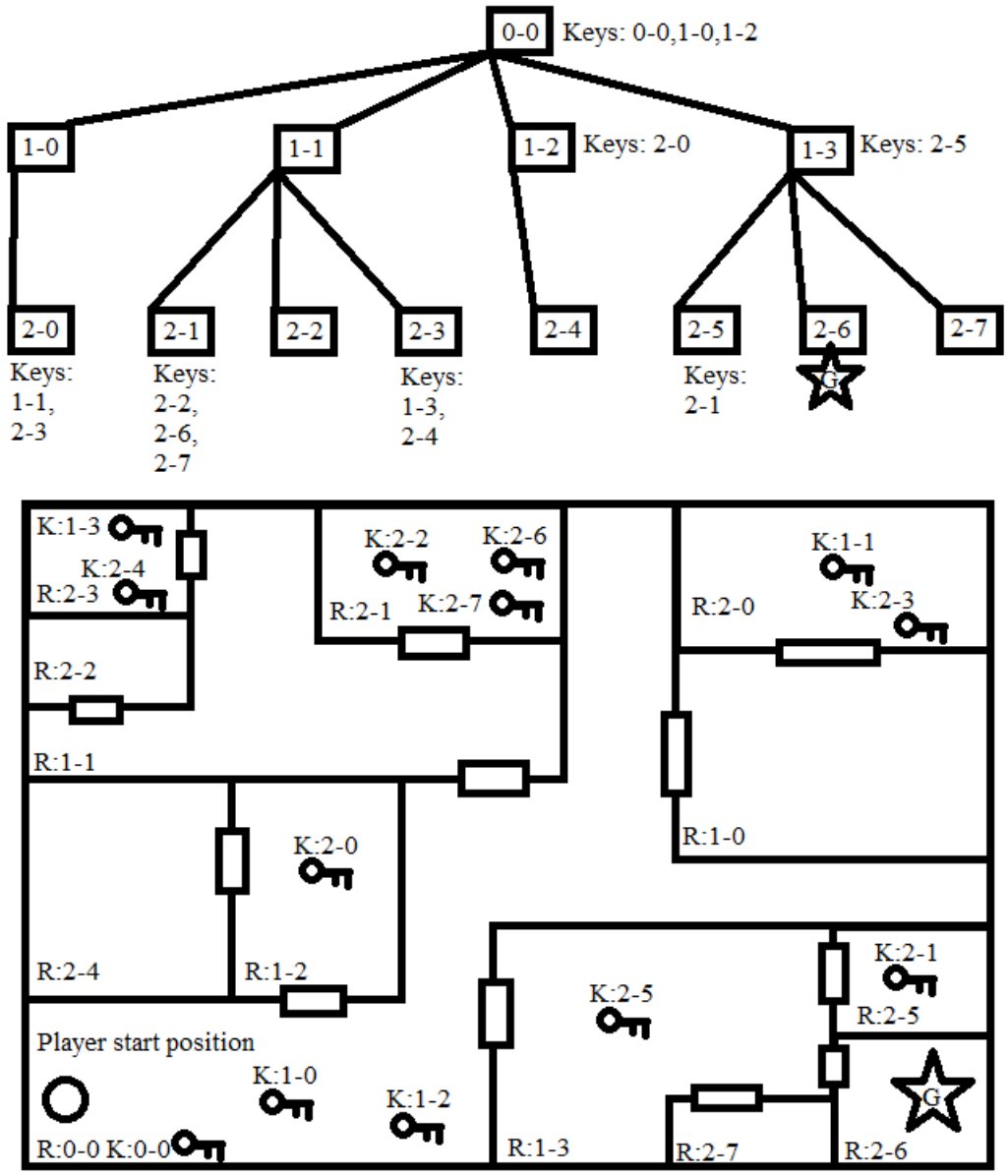

Figure 4.2-Tree generated by the program with keys properly placed, and potential floor plan layout of solved tree.

With this it's now easier to see how the world might look after the keys have been sorted in the world. The player would start out in room 0-0 and instantly obtain keys to allow access into rooms 0-0, 1-0, and 1-2. The player could then proceed to room 1-2 to

get the key to room 2-0, then proceed to 1-0. From 1-0, the player can now access 2-0 and get the keys to rooms 1-1 and 2-3. Going to rooms 1-1 and then 2-3 will give the player the keys to rooms 1-3 and 2-4. Going into room 1-3 the player will obtain the key for room 2-5, and then from 2-5 the player can get the key to room 2-1. From room 2-1 the player gets the keys for rooms 2-2, 2-6, and 2-7, thus allowing the player to now reach the room with the Goal, 2-6.

The limitations so far with this program besides its currently set max depth and the leaf nodes having anywhere from 0 to 5 children is that this project has mostly been focusing on the idea of a game. The logic with deadlock checking could definitely be used in other projects or for other purposes such as multi-thread programs. This program can be built upon and further modified to the point that it could one day be actually made into a dungeon exploring video game with uniquely generated levels.

Literature Cited

"Creating and using Scripts." Unity Technologies. 2013. Web. 27 Apr. 2014. <http://docs.unity3d.com/Documentation/Manual/CreatingAndUsingScripts.html>

Dang, Kim Dung, and Ronan Champagnat. "An Authoring Tool to Derive Valid Interactive Scenarios." Association for the Advancement of Artificial Intelligence. 2013. Web. 05 Feb. 2014. <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/viewFile/7427/7645>

van der Linden, Roland, Ricardo Lopes, and Rafael Bidarra. "Designing Procedurally Generated Levels." Association for the Advancement of Artificial Intelligence. 2013. Web. 05 Feb. 2014. <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/viewFile/7450/7641>

Risi, Sebastian, Joel Lehman, David B. D'Ambrosio, Ryan Hall, Kenneth O. Stanley. "Combining Search-Based Procedural Content Generation and Social Gaming in the Petalz Video Game." Association for the Advancement of Artificial Intelligence. 2012. Web. 10 Mar. 2014. <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/download/5449/5698>

Smith, Gillian, Jim Whitehead, and Michael Mateas. "Tanagra: An Intelligent Level Design Assistant for 2D Platformers." Association for the Advancement of Artificial Intelligence. 2010. Web. 8 Apr. 2014. <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewFile/2126/2574>

Togelius, Julian, Georgios N. Yannakakis, Kenneth O. Stanley, Cameron Browne. "Search-Based Procedural Content Generation." EvoWorkshops. 2010. 27 Apr. 2014. <http://eplex.cs.ucf.edu/papers/togelius_evogames10.pdf>