PROCEDURAL GENERATION IN 2D METROIDVANIA GAME WITH ANSWER SET PROGRAMMING

John Xu[1], John Morris[2]

[1]Harvard-Westlake High School, 3700 Coldwater Canyon Ave,
Studio City, CA 91604
[2]Computer Science Department, California State Polytechnic University,
Pomona, CA 91768

ABSTRACT

Video game designers often find themselves at a crossroad when designing levels; namely, many have a difficult time balancing the amount of control they want to have over what their levels look like [1]. If too little control is given, like in the case of pure perlin-noise generation, levels can end up with too much variation and unideal generations [2]. Softlock is an example of unideal generation in the case of metroidvania games, if the placement of keys cannot be easily controlled and end up being placed behind gates, the players can get permanently stuck [3]. Developers may usually hand-make all levels in order to try and prevent this from happening, however they risk spending too much effort and time on designing levels, resulting in a general lack of quantity in levels. Objectively speaking, both methods have their strengths and work well in specific genres of games, but limiting oneself to the boundaries of these methods does not fundamentally achieve both quantity and accuracy. This paper proposes an unique solution to this dilemma, providing automated generation of levels while also giving developers much more control over the overall output. Our method uses Answer Set Programming (ASP) to verify generation based on restrictions we place, guaranteeing the outcome to be what we want [4]. To demonstrate our method, we applied our solution to a 2D metroidvania game made in the Unity game engine and conducted quantitative tests to assess how well our method works as a level generator [5].

KEYWORDS

Procedural Generation, Answer Set Programming, Video Game Application

1. INTRODUCTION

Answer set programming excels at searching for multiple solutions (facts) that satisfy constraints (rules) that are desired. Its strengths are very unique and can work extremely efficiently in situations where different solutions that satisfy certain constraints are needed. In other words, ASP is able to declaratively search for its solutions, and thus represent its output as only the rules that are fed into it [6]. There is no need in ASP to individually dictate each solution when the solutions can be represented by the restrictions that imply what they can and cannot be. This lessens the need for manual implementation and control, which makes things a lot more labor intensive and time consuming. In addition, ASP is extremely powerful at solving computationally intensive and difficult search problems (NP-hard). Its efficiency and power lends itself well to real-world applications in topics that would otherwise cause huge headaches. For example, complicated problems like industry-level scheduling and planning can be handled efficiently with

David C. Wyld et al. (Eds): CCSEA, AIFU, EMSA, NLCAI, NCOM, SIPRO, SEA, DKMP, BDML, BIoT, CLOUD - 2023
pp. 123-136, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.130511

ASP. Unfortunately, ASP has not been frequently applied to video game development, which is what this paper focuses on [7]. However, this paper will prove that ASP's powerful and efficient constrained search works well with the delicate generation of video game levels.

The programming language in which the rules are written is called AnsProlog (short for: Answer Set Programming in Logic) [8]. These constraints are solved using answer set solvers, which compute the final answer sets, such as clasp, assat, cmodels, and clingo. Our experimentation with ASP involves the use of clingo as an ASP solver. It was rare for previous endeavors to produce game-related projects with ASP in the first place, and it is extremely rare to see it done with clingo [9]. Many of those who tried ASP as a way of generating levels did not go far with the extent of application, often opting for a proof of concept and very small scaled designs that did not consider larger scale implementation of ASP into games. Many parts of the ASP application into video games remain unexplored and a lot of potential problems in ASP implementations like loading time and loading behavior are not as considered in the rare instances where ASP is used. However, our endeavor strives to create a practical and full-scale game, which would demand our implementation to be efficient in areas previously mentioned. Another point that expands on the lack of practicality is the fact that answer set solvers such as clingo need to go through complicated installation steps on any local machine in which it is used to function properly. Therefore, it becomes difficult for other implementations to work for the wider public. However in our implementation, we have the option for clingo to be hosted in a server so that it is able to send its answer set output to any local machine without the user needing to install anything extra. The online feature will not be the focus of this paper. It is also important to note that our approach to generate metroidvania-style levels is very unique to begin with [10]. Many of previous implementations had their focuses on other aspects, and ASP played varying sizes of roles in these projects

In this paper, we will be using clingo as our ASP solver to solve for rules we can manually input and adjust. The overall level is split into rooms, where each room and its relevant information is determined by the solver first represented by a graph, and then each room is iteratively solved based on the information from the graph. The solution is sent as a dictionary to the game engine, where we've set up scripts to read these dictionaries and build it into an actual level. Compared to a perlin-noise procedurally generated level, our method has strength in its ability to both generate and verify the map structure / geometry in the same step, which guarantees that the level that is generated satisfies any valid constraint we give it. ASP allows for easy prevention of softlock, which traditionally game designers solve by adding work-around solutions, like giving players the ability to selectively destroy or place certain tiles. With ASP, there is no need to create work-around solutions and lessens the burden on game designers to shape game play around the faults of level generation. Lastly, our method also has strength in its flexibility thanks to our incorporation of a powerful game engine. We are able to tweak generation rules and the tiles' sprites and artwork can also be easily modified and changed with a powerful game engine. This flexibility allows for even more potential variation in the levels that can be generated and thus supplying even more interesting combinations of content for players to explore.

We conducted two experiments to prove the function of our method. The first experiment involves measuring the runtime of generation of different types of rooms. All combinations of doors and directionality in rooms were tested, and average runtime was tabulated across a graph of time vs room type using the raw data of each type of room. The types are listed in an increasing order of number of doors, where bidirectional and directional door room types are clearly separated. This was done to see if our method was able to generate levels at good speeds relative to other more popular automated generation methods such as perlin noise generation, therefore determining our method's ability to be applied to more mainstream and real world examples. The second experiment involves testing the traversability of our generated levels to see

how effective they are at preventing softlocking and maintaining playability. A pathfinder was used to determine this across a 2D "letter representation" of the map where all important features of the map are represented by individual characters. The pathfinder would traverse through the level, finding the key first, and then going to the end. The pathfinder will not be able to reach the end if the key is after a gate, thus rendering the end unreachable since the end is always gated by a gate. We used this experiment to see how compatible ASP was with level generation and if it could "replicate" its strengths in a metroidvania level.

There are 5 more sections in this paper, each presenting different aspects of the experiment process and other relevant details. In the next section (Section 2), details are provided on challenges we had during the experiment and designing process. Meanwhile in Section 3, focus is put on the details of our solutions corresponding to these challenges mentioned prior. Section 4 focuses on the relevant details about the experiment we did, followed by related works in Section 5. Finally, Section 6 presents the conclusion remarks as well as pointing out the future of this project.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Directionality in Rooms

One of the important features developed using this ASP method is the implementation of directionality within rooms and determining a viable path in which the player can traverse the level. This specific feature is crucial for our method to generate traversable levels and prevent players from going where they are not supposed to. Though it may seem simple at first, its implementation presents itself as an extremely difficult one since the generation must take in account all the possible moves of the player and determine which way the room can lead players based on those moves. Our solution to this problem was to provide an effective way of determining direction, where we first determined all the possible tile locations the player could be, and then split the room into distinct platforms, which were defined in the code as floor tiles that are connected and within the player's jump height of each other. We then were able to determine directionality of the room based on these platforms but checking to see if the player's possible positions / movements were able to reach these platforms.

2.2. Long Solve Time

ASP is a power tool to be able to use, however we encountered downsides regarding the solve time that the solver is taking in order to generate valid solutions. Generally speaking, ASP solve times are prolonged when there is a big workload or if an excessive amount of constraint is placed. Knowing this, we were able to effectively counteract the factor we believed was giving us the biggest trouble, that being the big workload we were giving to the solver. The most obvious thing that added to the solver's workload was the fact that we were requiring the solver to solve for the entire level, so we came up with a "divide and conquer" solution where we split the generation workload into individual same-sized rooms to be solved iteratively. In this way, each call to the solver will have a room size that is a lot smaller to solve, exponentially decreasing the runtime of a singular room, and therefore decreasing the runtime of the entire level.

2.3. Learning ASP

The syntax and actual writing of AnsProlog was a challenge that was prominent in the early stages of development. Personally speaking, AnsProlog as well as its logic was very unique from other programming languages and difficult to understand at first, and thus it seemed like a far-fetched idea that any playable levels were able to be produced using AnsProlog. The way we set up the solver meant that we had to code the rules as a one whole long string, which were then passed into the solver, meaning that there was no color differentiation, auto-correct nor auto-indentation. This unfortunate circumstance made typos hard to detect, and in combination with our unfamiliarity resulted in a very difficult start to our research. However, we were able to familiarize ourselves with the workings of ASP by testing its behavior tiresomely with a Jupyter Notebook that simulated tile placements until we were able to grasp and utilize AnsProlog for our research.

3. SOLUTION

In this paper, we will be using Answer Set Programming to generate metroidvania-style levels. The levels will contain keys that can unlock their respective gates, where the ASP guarantees the keys to appear before the gate before the player's path. Our method views a single level as separate rooms that have a constant size, and we go through a two step process to ensure the specific room geometry satisfies our parameters and constraints as well as making sure the general pathway is connected throughout the level. The graph contains information on where the room may lead the player (directionality) and whether the room contains a key, a gate or no special objects (traversal rooms). Each room within the graph is iteratively solved based on these information from the graph. To ensure that a constant pathway still remains through the level, the edge geometry of neighboring built rooms were passed in to the solving process of the current room, however this still caused problems because it could be the case that no solution exists under the additional consideration of neighboring edge tiles. For the case of a no solution / unsatisfiable output, an algorithmic fix was applied to this problem involving the removal of neighboring rooms. Our level generator will output a dictionary of all tile, room, key, and gate information after all the rooms are iterated through and all unsatisfiable conflicts are resolved to the Unity game engine, where it is interpreted and built as an actual playable level. Based on the locations of the tiles, the artwork for tiles are determined accordingly to stylize the level.

The first step in our method is to define the parameters of the overall level, those being the size of the level, the type of gates we want it to have, the maximum and minimum number of gates and the size of each room. These parameters are passed in to the solver, which solves based on the ASP rules we've given to the solver as well, and returns to us a graph that contains information about the rooms. The graph shows the rooms' directionality and what they contain (key, gate, or nothing), and based on this graph we are able to iterate through each individual room to generate it. To make the rooms fit together, all the solved neighboring rooms' adjacent tiles to the original room (edge geometry) is mirrored over to the current room-to-build. The above mentioned fix is applied when an unsatisfiable case occurs for a room. The final solution after all rooms have been iterated are outputted as a dictionary to be read in Unity for the actual level to be built in-game.

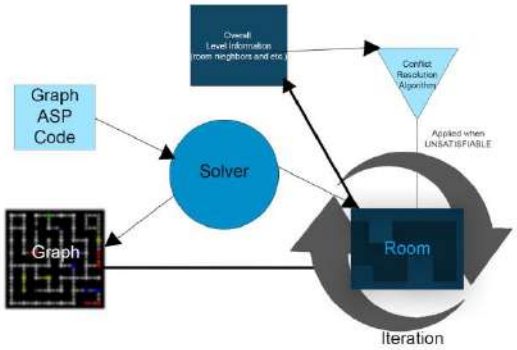


Figure 1. Overview of the solution

The first main step in our method is the generation of the graph that acts as a guideline to the more specific generation later on. The graph is a 2D representation of separate nodes representing rooms and connecting each room are either no arrows, a bidirectional arrow, or an one directional arrow representing the directionality of the room [11]. ASP guarantees solutions that always have a path between any two rooms. The nodes themselves can either be a circular shape, a diamond shape or a rectangular shape. White circular nodes mean that the room does not contain any keys or gates. Gray circular nodes are boss rooms while the green circular node is the start room, which is picked randomly from the first row. Any other colored circular nodes represent areas that are gated by a gate corresponding to its color. The colors of rectangular and diamond nodes represent the type of key/gate that the room contains, and the shape of the node itself determines whether it is a key or a gate. Rectangular nodes mean that there is a gate in that node / room, and the diamond nodes mean that there is a key in that room. The types of key/gates we want to have in our level, how big we want the entire level to be, how big we want each room to be, and the maximum and minimum number of gates we want there to be must all be defined in the code, which will be passed into the clingo solver to be solved. It's important to note that keys are always placed in an area that is before the gates so that there is no way the player can get stuck. It's solution is sent as a dictionary to C# scripts that store this information for further generation.

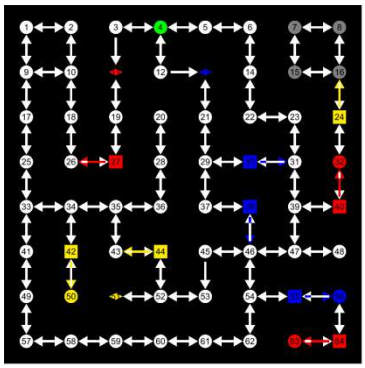


Figure 2. 2D representation of separate nodes

We utilized a mix of C# code and ASP to generate the graph. Different rules that are written in ASP are stored as strings in a separate script, and are combined here to form the complete ASP code. The parameters we pass in are slotted into their corresponding locations in the ASP code, and everything gets passed in to the solver to be solved.

```
public void BuildGraph(int worldWidth, int worldHeight, int gateKeyCount, int maxGatePerKey, int minGatePerKey, int bossGateKey,
int startRoom, int timeout, int cpus)
{
    string aspCode = WorldMap.bidirectional_rules + WorldMap.graph_rules + WorldMap.gate_key_rules;
    string path = ClingoUtil.CreateFile(aspCode);
    ClingoSolver solver = FindObjectOfType<ClingoSolver>();
    solver.maxDuration = timeout + 10;
    solver.Solve(path, $" -c max_width={worldWidth} -c max_height={worldHeight} -c start_room={startRoom} " +
        $"-c key_count={gateKeyCount} -c max_gate_type_count={maxGatePerKey} " +
        $"-c min_gate_type_count={minGatePerKey} -c boss_gate_type={bossGateKey} --parallel-mode {cpus} --time-limit={timeout}");
}
```

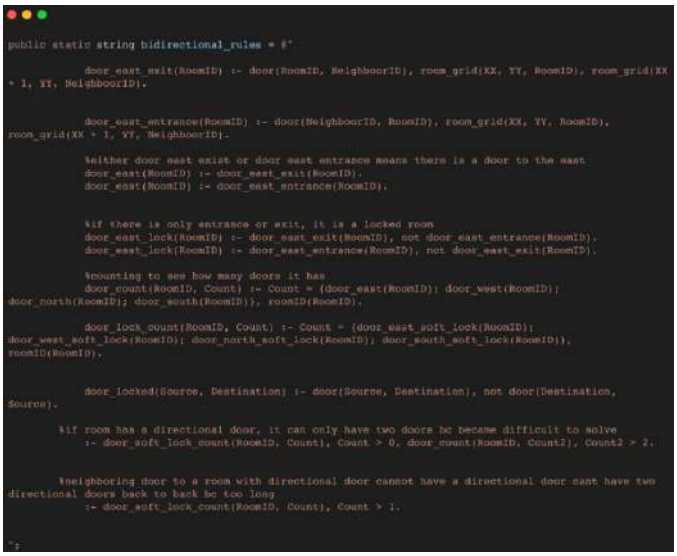
Figure 3. Screenshot of code 1

Below are two examples of ASP rules chunks that are added to the final graph asp code. The following code chunk defines the roomID based on where a room is in the level, and a grid of all the rooms is also defined based on the X position, Y position and a room ID. A door is defined as possibly occurring between any room and its immediate horizontal or vertical neighbors. The start room is also defined as any room in the top row of the level.

```
public static string graph_rules = @"
  #const max_width = 4.
  #const max_height = 3.
  #const start_room = 1.


  width(1..max_width).
  height(1..max_height).
  roomID(1..max_width*max_height).
  1{room_grid(XX,YY, ID)}1 :- width(XX), height(YY), ID = (YY - 1) * max_width + XX.
  room(ID) :- room_grid(XX,YY,ID).

  {door(RoomID1, RoomID2)}1 :- room_grid(XX,YY, RoomID1), room_grid(XX+1, YY, RoomID2).
  {door(RoomID1, RoomID2)}1 :- room_grid(XX,YY, RoomID1), room_grid(XX-1, YY, RoomID2).
  {door(RoomID1, RoomID2)}1 :- room_grid(XX,YY, RoomID1), room_grid(XX, YY+1, RoomID2).
  {door(RoomID1, RoomID2)}1 :- room_grid(XX,YY, RoomID1), room_grid(XX, YY-1, RoomID2).



  1{start(RoomID): room_grid(_,YY,RoomID), YY == 1}1.


";
```

Figure 4. Screenshot of code 2

This next chunk of ASP code is responsible for bidirectional constraints. The code is simplified to only include east doors for the sake of better viewing since all the other doors of other directions all behave similarly. An east door exit and an east door entrance is defined as must being east one room of the original room. A door to the east, whether entrance or exit, is considered an east door to the code. If there is only an entrance or only an exit, the room is considered locked to the east. We then count how many doors the room has as well the number of locks (this would originally contain all four directions). When you can only travel one way through a door, it is directional. When a room has a directional door, there can only be two doors. There can also not be two directional doors back to back in neighboring rooms. Anecdotally, we have observed higher runtimes when these restrictions are not in place, so these restrictions were added for maximum efficiency with solve time.



```
public static string bidirectional_rules = @"

        door_east_exit(RoomID) :- door(RoomID, NeighboorID), room_grid(XX, YY, RoomID), room_grid(XX
+ 1, YY, NeighboorID).


        door_east_entrance(RoomID) :- door(NeighboorID, RoomID), room_grid(XX, YY, RoomID),
room_grid(XX + 1, YY, NeighboorID).

        %either door east exist or door east entrance means there is a door to the east
        door_east(RoomID) :- door_east_exit(RoomID).
        door_east(RoomID) :- door_east_entrance(RoomID).


        %if there is only entrance or exit, it is a locked room
        door_east_lock(RoomID) :- door_east_exit(RoomID), not door_east_entrance(RoomID).
        door_east_lock(RoomID) :- door_east_entrance(RoomID), not door_east_exit(RoomID).

        %counting to see how many doors it has
        door_count(RoomID, Count) :- Count = {door_east(RoomID); door_west(RoomID);
door_north(RoomID); door_south(RoomID)}, roomID(RoomID).

        door_lock_count(RoomID, Count) :- Count = {door_east_soft_lock(RoomID);
door_west_soft_lock(RoomID); door_north_soft_lock(RoomID); door_south_soft_lock(RoomID)},
roomID(RoomID).


        door_locked(Source, Destination) :- door(Source, Destination), not door(Destination,
Source).

    %if room has a directional door, it can only have two doors bc became difficult to solve
        :- door_soft_lock_count(RoomID, Count), Count > 0, door_count(RoomID, Count2), Count2 > 2.


    %neighboring door to a room with directional door cannot have a directional door cant have two
directional doors back to back bc too long
        :- door_soft_lock_count(RoomID, Count), Count > 1.


";
```

Figure 5. Room Iteration and Iterative Unsatisfiable Conflict Resolution Algorithm

The graph output dictionary is used to generate each room iteratively based on an ID assigned to it based on its location in the level. The solver is used here again to determine the specific terrain geometry within the room as well as special cases such as implementing the gates and keys into actual tiles. As mentioned earlier an issue of an unsatisfiable case in rooms emerges as we try to fit these rooms back together. Our solution involves removing neighboring rooms in order to get rid of what's causing the unsatisfiable error. Our original approach was to randomly remove neighboring rooms, solve for the current room, then placing the removed room at the top of the stack of all rooms-to-build to be solved next. The random selection of neighboring rooms was not effective enough, because multiple neighboring rooms could be causing this unsatisfiable nature of the current room. Due to this lack of effectiveness, we modified our solution so that it generates all the possible combinations of neighboring tiles from smallest number of neighbors to biggest, and iterated through them to find the minimum set of neighboring rooms that caused the room to be unsolvable. By removing this minimum set, solving for the current room, and putting the removed rooms back to the stack to solve, the unsatisfiable case is able to be completely solved.

The following method gets a random smallest permutation out of the permutation 2D list generated by another method. The code represents its immediate neighbors as a 3 by 3 grid with numberings from 1 through 9, so the permutation returns the to-be-removed neighbors this way.

```
public static List<T> GetSmallestRandomPermutation<T>(List<List<T>> permutations, bool remove)
{

    int smallest = int.MaxValue;
    foreach (List<T> permutation in permutations)
    {
        if (smallest > permutation.Count) smallest = permutation.Count;
    }

    List<int> smallestIndices = new List<int>();
    for (int i = 0; i < permutations.Count; i += 1)
    {
        List<T> permutation = permutations[i];
        if (smallest == permutation.Count) smallestIndices.Add(i);
    }
    int rand = Random.Range(0, smallestIndices.Count);
    List<T> smallestPermutation = permutations[smallestIndices[rand]];
    if (remove) permutations.Remove(smallestPermutation);

    return smallestPermutation;
}
```

Figure 6. Screenshot of code 3

The next piece of code shows how the output of the previous method is used. The list of integer output is stored as the removed Neighbors of the current room, which is later interpreted into the actual room inside the world/level, which can then can be easily removed.

```
if(newRoom.buidStatus == ClingoSolver.Status.UNSATISFIABLE)
{
    List<int> killRooms = newRoom.removedNeighbors;
    foreach(int killID in killRooms)
    {
        Room killRoom = world.GetRoom(killID);
        DestroyRoom(killRoom,newRoom.lastBuildTime, roomID, ClingoSolver.Status.UNSATISFIABLE);
    }
}
```

Figure 7. Screenshot of code 4

This is a cool gif of the iteration algorithm in action.

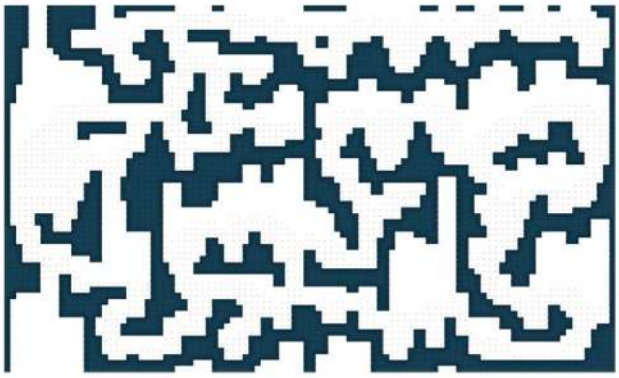


Figure 8. Screenshot of the gif

4. EXPERIMENT

4.1. Experiment 1

Our solution solves the problem because we were able to effectively use ASP to generate rooms in a relatively adequate amount of time. A level generator isn't really effective if it struggles with its runtime, and thus fails to bring a playable map to players within a short period of time. This experiment involves measuring the average runtime in seconds of solving 20 x 20 no gate and no neighbor rooms. The varying runtimes for all combinations of bidirectional and directional passageways are accounted for in this experiment. The fact that our ASP method was able to quickly generate its rooms in most passageway types shows that our method of applying ASP to level generation is viable to create 2D metroidvania maps.

Runtime is an effective way of quantifying map generation, which can otherwise be a very hard subject to measure and compare. Our sample size for this experiment is at least data from 30 generated levels, which could be improved. However, generating large amounts of levels still takes quite a long time, so the sample size is unfortunately limited.

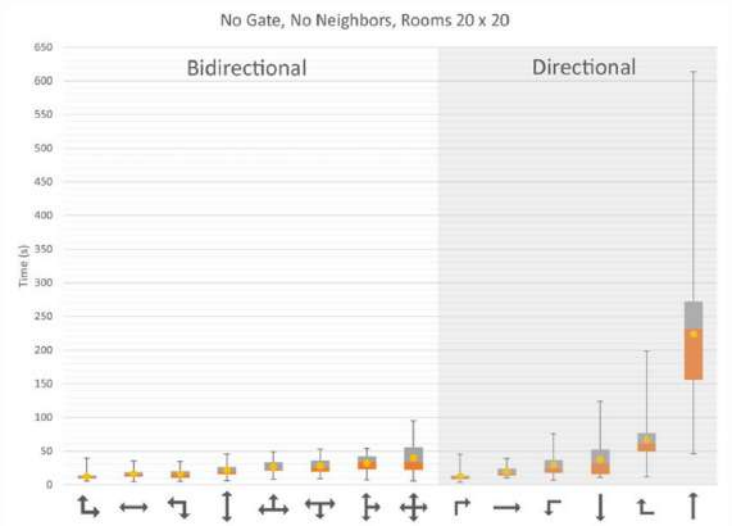


Figure 9. Graph of experiment 1

In a 20 by 20 no gate no neighbor scenario, all passageway types except directionally up rooms solve rapidly (within a minute or two at most extreme cases). It also appears that the amount of doors is directly related to the solvetime for rooms. The first 8 plots in the graph only have bidirectional doors and are ordered based on average runtime, with the first four having 2 bidirectional doors, followed by the 3 door rooms, and then the four door room. The rest of the plots in the graph in the gray section are rooms with one directional door and one bidirectional door. They are ordered from left to right in increasing average runtime. The directionally up room types solve on average 4 times slower than the second biggest average, and go as high as 600 seconds in some cases. Considering the fact that a 20 by 20 room gives plenty of room for players to explore, the generation speed is very solid. By limiting the directionally up rooms in the future, even faster runtime can be achieved.

4.2. Experiment 2

Our methodology is able to not only work efficiently but also work accurately to guarantee the outcomes we want it to achieve. One of the crucial goals for using ASP in level generation is to guarantee the player's path from point A to point B, whether that be to a key or to a boss area. We are able to prove that our solution completely guarantees the player to reach the key before a gate that leads to an endpoint by using a pathfinding algorithm not designed but modified by us.

We used around 100 maps to guarantee that the map is traversable everytime. More could be used for testing because the level was represented as a 2D map consisting of ascii characters that represented different objects and tiles in the level. The pathfinding algorithm was also able to work quickly through multiple files and what we were looking for was simple: a true or false for traversability.

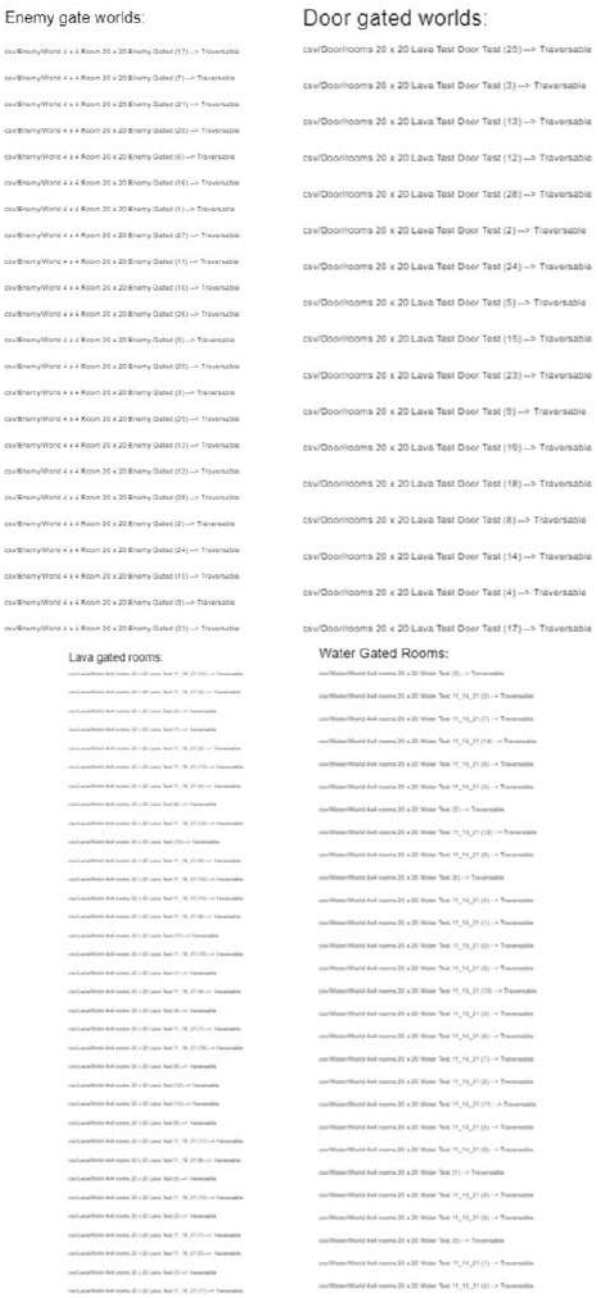


Figure 10. Result of experiment 2

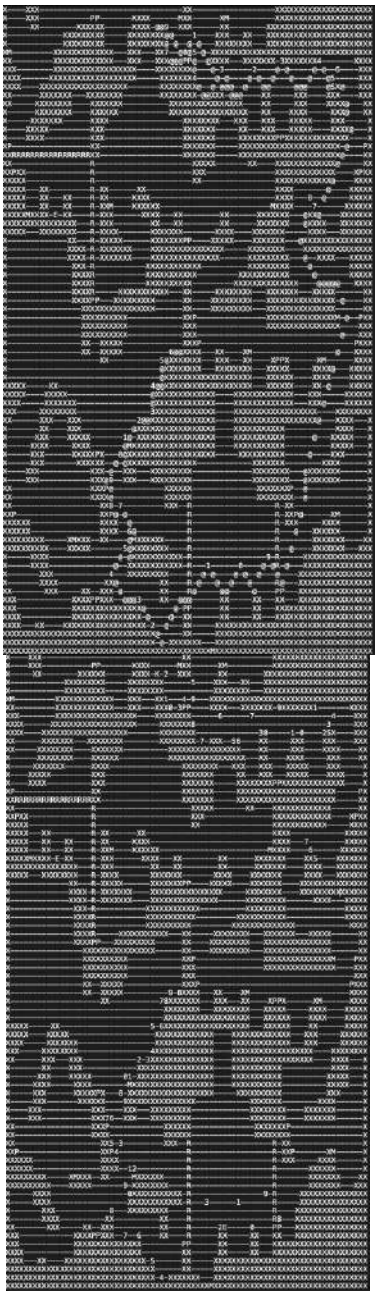


Figure 11. Example Ascii Maps for an enemy type of gate for better reference

The first list is very long, however it represents the guaranteed traversability of all the tested worlds. The pathfinding does its calculations and only outputs a “traversable” if the map that it traced was traversable, otherwise only the name of the map would appear. To visualize what the pathfinder is doing, a specific example is provided to show how the pathfinder starts at its start point (represented by ‘S’) traverses itself through the map to reach the end (‘B’). There are two maps that have either tiles (‘X’) or empty (‘-’) as its base structure, one representing a graph that has not been traversed yet, with all the key, start, and end visible. The other represents the pathfinder’s path (‘@’), and it is only able to go through the enemies (‘R’) when it has the key. The pathfinder always prioritizes getting the key first, so we see that the pathfinder

gets the key first before backtracking to get to the end in the shortest distance. Other ascii keys such as numbers as other letters are for previous testing that will not be covered in this paper.

From the first experiment, we can tell that the runtime for rooms does not usually exceed a one minute to two minute generation at most, meaning that generation for rooms will usually happen fast, thus levels in general will not take a long time to generate. There are limiting edge cases to the runtime performance, so the solution still needs to be better refined to produce more consistent results for overall level generation. In general, the room generation runtime is not bad for so many restrictions we put on it, however there is definitely room to grow. The second experiment can be considered a bigger success since the traversability of the levels are proven to be working for all test cases. This meets the demand of having a level generator where game designers can guarantee the playability of a level, or in more general terms, the ability to fully constrain certain parts of generation.

Considering how many restrictions we have put on the code for the solver to solve in addition with the iterative conflict algorithm, solve time was not expected to be too low. The actual solvetime of rooms did match our expectations, and proved to be an overall okay speed. We had high hopes of traversability on the other hand, since the guaranteeing aspect of ASP was one of the strong points we looked forward to utilizing. After testing, the traversability of ASP is indeed guaranteed in all cases, taking more time to solve for a more complete and accurate solution.

5. RELATED WORK

This paper focuses on using Computational Tree Logic to prevent softlocking in Super Metroids [12]. Path determination is also studied closely in this paper as the basis to solving softlocking as well. Overall there are many similarities between this paper's approach and our approach since we also view path determination as a crucial part of determining possible softlocks. This paper mostly differs in the aspect of what specific tool we chose to use to prevent softlocking, we used ASP while they used CTL.

This paper provides a very in depth "skeleton" of how to generate correct key-gate structures in video games using Answer Set Programming [13]. The paper does not specifically apply to metroidvania games, but games in general. Both our paper and this paper uses answer set programming as its tool of generating the placement of keys and gates since both want to guarantee keys before their corresponding gates. This paper differs from ours in the sense that we had more practical application to games since we used the output solution to go one step further and generate real 2D metroidvania levels. In other words, both papers' approach to generate the overall structure of the map is similar but differs in the actual implementation of the level.

This paper focuses on a very well-rounded approach to the procedural generation of metroidvanias [14]. Similar to our paper, this paper recognizes the difficulty of automating level creation, and strives to outline their process of creating an entire metroidvania, however it includes much more details than our paper in details such as player controls, animations and other visualizations while we tended to focus on the level generation aspect of the entire game. The paper also has a different approach to determining the key gate order, opting for a branching approach where node position represents its priority in the overall order of gates.

6. CONCLUSIONS

We have proposed a method to generate 2D metroidvania levels with ASP that prevents softlocks and guarantees traversability [15]. We generated an outline structure of a level first, then were

able to iteratively solve and render each room in the level. Each room and its neighbors were checked for their ability to connect together and any possible unsatisfiable errors were accounted for in the process using our iterative conflict resolution algorithm. We made sure that only the minimum permutation of rooms causing unsatisfiability was removed. To quantitatively test that our method works consistently and effectively, we applied our method to two experiments. The first being an experiment measuring the runtime of different types of room passageway generation. The second solution being an experiment measuring the traversability of each level by using a pathfinding algorithm. Runtime turned out to not be extremely fast, but in most cases short enough for level generation to take place in a brief time. Traversability was guaranteed in all cases in which levels were tested, showing that ASP can be very effective at verifying the playability of levels. Though it may take longer times to solve, ASP has complete insurance of the level working. This “trade-off” for verification over time is a good enough cost to address the issue of many issues plaguing traditional procedurally generated metroidvania levels.

One of the glaring problems with our method of applying ASP is the fact that actual users need to install clingo on their local computers before being able to actually generate maps with our method. This makes the idea of applying ASP to games a bit counter-intuitive. In addition, the runtime also seems to be a big issue in some cases, so more constraints have to be placed to avoid long solve time spikes for certain rooms that could prolong the generation of the entire level, leaving players to wait a long time to start playing a level.

We plan to solve the issue of installing clingo by hosting clingo in an online server, so any local computer may just request a solution from the server through local scripts setup in the game. Solve time can be reduced if we limit the amount of up directional passageway rooms that can exist in the map, which could speed up runtime significantly.

REFERENCES

[1] Dondlinger, Mary Jo. "Educational video game design: A review of the literature." Journal of applied educational technology 4.1 (2007): 21-31.

[2] Jakes, David, et al. "Perlin noise generation of physiologically realistic patterns of fibrosis." BioRxiv (2019): 668848.

[3] Guo, Yunfan, et al. "Soft-lock drawing of super-aligned carbon nanotube bundles for nanometre electrical contacts." Nature Nanotechnology 17.3 (2022): 278-284.

[4] Brewka, Gerhard, Thomas Eiter, and Mirosław Truszczyński. "Answer set programming at a glance." Communications of the ACM 54.12 (2011): 92-103.

[5] Oliveira, Bruno Pinheiro, et al. "A framework for metroidvania games." Proceedings of SBGames (2020): 836-844.

[6] Erdem, Esra, Michael Gelfond, and Nicola Leone. "Applications of answer set programming." AI Magazine 37.3 (2016): 53-68.

[7] McAllister, Graham, and Gareth R. White. "Video game development and user experience." Game user experience evaluation (2015): 11-35.

[8] Mellarkod, Veena S., Michael Gelfond, and Yuanlin Zhang. "Integrating answer set programming and constraint logic programming." Annals of Mathematics and Artificial Intelligence 53.1-4 (2008): 251-287.

[9] Gebser, Martin, et al. "Multi-shot ASP solving with clingo." Theory and Practice of Logic Programming 19.1 (2019): 27-82.

[10] Morris, John. DESIGNING A PROCEDURALLY GENERATED METROIDVANIA STYLE VIDEO GAME USING ANSWER SET PROGRAMMING. Diss. California State Polytechnic University, Pomona, 2021.

[11] Hou, Ji, et al. "Pri3d: Can 3d priors help 2d representation learning?." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021.

[12] Mawhorter, Ross, and Adam Smith. "Softlock Detection for Super Metroid with Computation Tree Logic." Proceedings of the 16th International Conference on the Foundations of Digital Games. 2021.

[13] Smith, Thomas, Julian Padget, and Andrew Vidler. "Graph-based generation of action-adventure dungeon levels using answer set programming." Proceedings of the 13th International Conference on the Foundations of Digital Games. 2018.
[14] Stalnaker, Trevor Wayne. "Procedural generation of metroidvania style levels (thesis)." (2020).
[15] Claiche, Benjamin Elliot. Haiku, the Robot. Sistema de mapas para 2D Metroidvania. Diss. Universitat Politècnica de València, 2021.