April 2018

General Video Game Level Generation

Spencer M. Beaupre
Worcester Polytechnic Institute

Thomas Grosvenor Wiles
Worcester Polytechnic Institute

General Game Level Generation

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

By:

Spencer Beaupre

Thomas Wiles

Date: 4/26/18

Project Advisor:

Professor Gillian Smith

Abstract

The goal of this project is to employ human design patterns to procedurally generate general 2D arcade-style games in the General Video Game AI (GVG-AI) competition framework. This is achieved by generalizing specific game levels made by humans and using pieces of them as building blocks for new levels of any other game describable in the framework. We produced a constructive and search-based generator to use these design patterns and compared them to the search-based generator of a prior study in a playtesting survey to evaluate their success.

Acknowledgements

This project would not have been possible without the assistance of people that provided us with either guidance or resources to enable our work.

Firstly, we would like to thank Worcester Polytechnic Institute for offering this opportunity to us and providing the necessary resources for completion of this project. Our project advisor, Gillian Smith, provided us with detailed advice throughout all phases of our project and we are grateful to her for her time and dedication.

We thank all the creators of the GVG-AI framework, whose extensive work provided our team with an invaluable starting point in our research.

Finally, we are extremely grateful to Ahmed Khalifa, who not only personally assisted us in providing a code structure for surveying, but whose research was the main inspiration for our project.

Table of Contents

List of Figures

List of Tables

1 – Introduction

Artificial intelligence (AI) is a rapidly growing field of computer science that is being used to create automated solutions to human problems. Commonly this technology is used to solve a domain-specific problem with no need for it to adapt to radically different problem specifications. For example, an autopilot system for an airplane would typically be made for a single airplane rather than being able to perform on any airplane. The next large step in the field of AI is trying to create a general solution to a problem with a much larger scope, referred to as artificial general intelligence (AGI) [1]. AGI systems will become less expensive and able to solve more diverse tasks, potentially overtaking the efficiency of a human on their own [1]. This gives AGI an enormous potential for practical applications in many areas like computer vision, natural language processing, and more.

The General Video Game Artificial Intelligence competition (GVG-AI) is a research competition focused on studying general artificial intelligence methods using game playing agents and level generators [2]. Methods of game AI and level generation are already well-developed in the industry for specific applications, but this competition strives for the advancement of general agents and generators that are capable of functioning for any game describable in the framework rather than for a single game.

Procedural generation is a subset of computing that focuses on creating new content algorithmically instead of requiring human work, allowing for faster and more varied results that are automated. One of the largest and earliest examples of procedural content generation (PCG) is creating video game levels to provide a new experience each time a game is played [3]. Instead of a human creating every individual part of a game level, an algorithm is given the

components that make up the level, and pieces together these components in varying ways to make a completely unique level.

However, most video games are still crafted by human developers that design the levels from scratch. It is difficult to make a procedural generator that can match the quality of World 1-1 in *Super Mario Bros*. (Nintendo, 1985) [4] or Green Hill Zone in *Sonic the Hedgehog* (Sega, 1991) [5] because machines struggle to incorporate creative design practices while also creating playable levels.

The GVG-AI framework is an accessible environment for approaching the problem of AGI as it provides many necessary resources (such as a generally applicable game description language and a large number of existing games) to set up a testbed for competition participants. In this study, we approached the level generation track of the GVG-AI competition by not just trying to procedurally generate functional levels, but by using common patterns of pre-existing human-made levels of video games as the method for building new levels. Most generators do not make use of the wealth of human work available to them and produce levels that are unorganized and clearly not human made. We hypothesized that incorporating human design into procedural generation would provide a meaningful standing in getting closer to making an AGI that outperforms a human. We created a constructive level generator that stores generalized game patterns and picks from these patterns to generate a new game level. We then created a search-based generator that created multiple levels using our constructive generator and evaluated them to produce a more refined result.

In chapter 2, we discuss the background of PCG in games, the GVG-AI competition framework and what resources it provides, detailed descriptions of the two main types of level generators, and our inspiration and approach to level generation. Finally, we detail the success of

a prior study's constructive and search generators in their survey as a solid knowledge foundation for our project [6].

After our background information, we discuss our work process in chapter 3, describing our process of defining and storing patterns from human-made levels. We then outline how we extrapolate these patterns into a general form so that they can be used for any game in the framework and show our process of creating our own constructive and search-based generators.

In chapter 4 we then show the results of our playtesting survey comparing our generators to the prior study's generator and provide metrics expressing the range of levels our generators could produce [6]. Chapter 5 presents our conclusions and discusses the potential of future work.

2 - Background

2.1 Procedural Content Generation in Video Games

As technology advances, there is potential for a wider variety of creating media and content for consumers. In recent years, procedural content generation (PCG) has been recognized as a fast and efficient way of generating new experiences. Although PCG encompasses a large scope of applications in computer science, the earliest and still most prominent usage is in video games. The games *Rogue* (Epyx 1980) [7] and *Elite* (Acornsoft 1984) [8] pioneered the use of PCG in video games in the early 1980's to randomly create new levels and add replayability.

From this point, many games began to use PCG as a way of creating algorithmically generated layouts of levels using the same basic objects in the game, with the benefit being that a player is presented with a new experience each time they play, providing great replayability without the cost of having to manually create more content. However, it is difficult for procedural generators to ensure that levels are both playable and enjoyable. Level generators in the industry are custom built for each game to employ strategies that promote quality. For example, a procedurally generated level in *Spelunky* (Mossmouth 2013) [9] needs to have actual paths to every collectible, enemy placements that present a fair challenge, borders, and most importantly, the playable character itself in the level. PCG is actively researched in academia as it is an approachable medium for solving problems and the value of indefinite amounts of new content is being realized with more applications that PCG can apply to [10] [11].

2.2 The GVG-AI Competition and Framework

The GVG-AI is a research competition focused on studying general artificial intelligence methods using game playing agents and level generators [2]. It can be simple to create a procedural level generator or an AI controller for a single game with fixed rules and objects, the challenge lies in creating methods that can operate for *any* game expressible in the GVG-AI framework. The GVG-AI framework currently supports over 90 different ports or variations of classic 2D arcade games.

In this case, the domain of video games is limited specifically to 2D arcade-style games, to provide a consistent structure and allow for a more limited ruleset, as trying to create AI and generators that expand across multiple dimensions and sizes is out of the scope of this competition. This study focuses on the recently added General Video Game Level Generation track (GVG-LG).

Custom level generators can be made in the GVG-AI framework by creating a new class that inherits from the *AbstractLevelGenerator* class. This involves implementing a constructor that takes in a game description and a timer, a function called *generateLevel* that takes in a game description and a timer and outputs a level as a string, and a *getLevelMapping* function which returns the hashmap of how the level can be decoded with the level mapping.

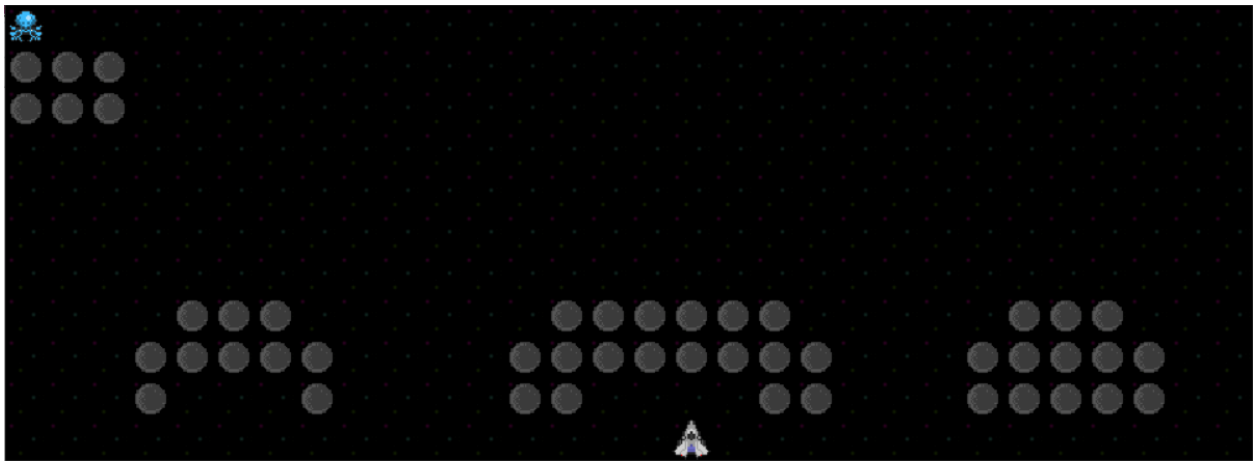


Figure 1 - The first video game in the GVG-AI framework, *Aliens*. A port of *Space Invaders*.

2.3 The Video Game Description Language

The Video Game Description Language (VGDL) is a language that allows for concise definitions of a game's objects and rules in a standardized format [12]. A game description is a short text file listing out the game's objects (sprites), level mapping (the way the sprites are coded in level files), sprite interaction rules, and termination conditions. The level description is stored as a text file composed of equal-length lines with each character representing the contents of a coordinate in the level. The VGDL was originally implemented in Python with py-game by Tom Schaul, but this project, as well as the GVG-AI Competition, uses a Java port of the VGDL [12]. In Figure 2 below, there is an example of a VGDL file describing all the necessary details for a specific game, *Sokoban*. To represent a game in the description language, four pieces of information are needed:

1. Sprite Set - List of all images (i.e. Sprites) used for objects in the game, and where these images are stored.
2. Level Mapping - A list of how to convert a level's contents to game sprites.

3. Interaction Set - Description of how each sprite interacts with one another, giving a command for what happens when these conditions are met.
4. Termination Set - All possible ways of ending the game, whether it is a loss or a win for the player.

```
BasicGame key_handler=Pulse square_size=40
    SpriteSet
        floor > Immovable img=newset/floor2
        hole   > Immovable color=DARKBLUE img=oryx/cspell4
        avatar > MovingAvatar img=oryx/knight1
        box    > Passive img=newset/block1 shrinkfactor=0.8
        wall > Immovable img=oryx/wall3 autotiling=True
    LevelMapping
        0 > floor hole
        1 > floor box
        w > floor wall
        A > floor avatar
        . > floor
    InteractionSet
        avatar wall > stepBack
        box avatar  > bounceForward
        box wall box  > undoAll
        box hole    > killSprite scoreChange=1
    TerminationSet
        SpriteCounter stype=box    limit=0 win=True
```

```
wwwwwwwwwwwww
w........w..w
w...1.......w
w...A.1.w.0ww
www.w1..wwwww
w.......w.0.w
w.1........ww
w..........ww
wwwwwwwwwwwww
```

Figure 2 - Example of a level description file, and a possible level that can be made with this information.

The standardized format of the VGDL allows for convenient implementation and evaluation of general AI systems for both game playing and level generation. Prior game description languages were not designed to have as wide of an expressive range as the VGDL. Examples of this are the PuzzleScript language which allows developers to easily create puzzle-style games [14] or the Extensible Graphical Game Generator (EGGG) which focuses on classic games involving cards or grids, like tic-tac-toe [15]. Languages like these in the past were domain specific to either logic-based games, board games, or text-based adventures. The VGDL enables research of general intelligence systems on a much wider domain because it allows for efficient description of 2D games and provides general objects and interactions in the language

which can then be specified for a wide variety of games. Most simple games that have an avatar and play on a fixed grid of tiles can be described by the VGDL, but many common game genres like board games (e.g. Chess or Go) and side-scrollers (e.g. *Super Mario Bros.* or *Sonic the Hedgehog)* are impossible to be represented in the language.

2.4 Constructive Level Generators

A constructive generator systematically assembles levels, with no evaluation of the quality of the level. The constructive generator produced in the original Khalifa et. al GVG-AI study classifies the sprites in the game description and uses simple heuristics to place them in the level [6]. The generator operates in four main steps: sprite classification, cover percentage calculation, construction, and fixing termination conditions. Game sprites are classified into one of five categories: avatar (player controllable), solid (immovable, no other interactions), harmful (kills avatar or spawns sprites that do), collectable (non-harmful, destroyed upon avatar interaction), and other. A priority value is assigned to each sprite based on the amount of rule interactions it has in the game description. The overall percentage of the level that is covered at the start is proportional to the number of collectible sprites and inversely proportional to the number of harmful sprites in the game description. Each category of sprite is then assigned a cover percentage based on the sum of the priority values for that category. To build the structure of a level, a border is created with a randomly selected solid sprite, then additional solid sprites are placed continuously within it while maintaining level continuity. If an avatar can only move horizontally, like in *Space Invaders*, then it is either placed at the top or bottom. Otherwise, an avatar is placed in a random open location. Harmful sprites are then added in free locations that are distant from the avatar. Collectable and other sprites are then randomly placed to finish the

construction phase. In the final step, extra sprites are added if there needs to be more to accomplish a termination condition.

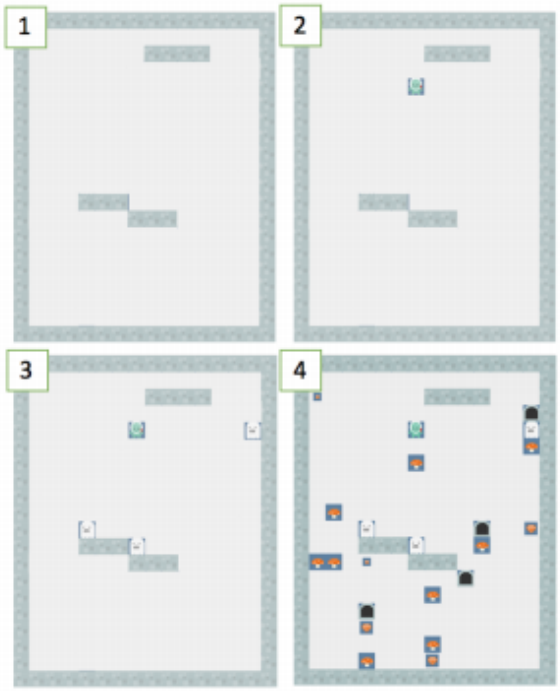


Figure 3 - Step-by-step process the constructive generator takes in building a level [6].

The constructive generator in the framework uses a few very simple rules to make the levels more likely to be playable, but does nothing to attempt to improve the aesthetic, difficulty, or gameplay.

More domain limited constructive generators have been used to greater success because intelligent design decisions can be made much more easily when tailored to a specific type of game, such as *Minecraft* [16] or *The Binding of Isaac* [17], which both use game seeds to construct a level from a random number. The advantage of constructive generators is that the programmer can build in strategies for building levels that are known to result in quality levels for the domain, it can run in a fixed time and with very small amounts of computing resources

and does not require any type of evaluation or iterative generation. However, it is very difficult to use a constructive generator that succeeds in wider domains because level design choices vary across different types of games. Finally, constructive generators provide no intelligent evaluation of how *good* a level is considered to be for a player. Objects are simply assigned values and placed in the level according to their priority, making generating a more refined level much harder.

2.5 Search-based Generator

A search-based generator operates by generating many constructive levels and performing an evaluation on those levels to find the best one. A search-based generator contains several defined constraints to evaluate the level and compares each level using a *fitness* function. The algorithm continuously produces and evaluates levels until either the constraints are met, and the fitness is above a certain percentage, or if the generator runs out of time. The search generator from the Khalifa et. al study uses the Feasible Infeasible 2 Population genetic algorithm (FI2Pop) [6]. This creates populations of levels and evolves them by performing random modifications. This generator mutates levels by swapping, inserting, or deleting individual sprites once per generation of the program. Each level in one generation is called a chromosome. Chromosomes are stored in either the feasible population, the set of levels that meets all constraints, or the infeasible population, the set of levels that fail one or more constraints. Constraints are metrics like completability, number of sprites, etc. The feasible population tries to increase the fitness evaluation of each chromosome while the infeasible population attempts to lower the number of chromosomes that break the constraints. These

populations can evolve on their own and chromosomes can go back and forth between the two populations throughout generations of the program.

Below in Figure 4 is a sample of what metrics the generator uses when performing evaluations. It records the total time the game took to finish (*SolutionLength)*, the amount of time it took for a player that never moves from the starting position to lose (*doNothingSteps)*, the percentage of the total level covered by sprites (*coverPercentage)* and whether or not the best-performing AI controller beat the level. The fitness function uses these parameters to determine the total fitness for each level played, the higher the better.

```
Generator initialization time: 36 ms.
Generation #1:
SolutionLength:236 doNothingSteps:20 coverPercentage:0.09848484848484848 bestPlayer:PLAYER_WINS
    Chromosome #1 Constrain Fitness: 0.9107142857142857
SolutionLength:275 doNothingSteps:42 coverPercentage:0.10606060606060606 bestPlayer:PLAYER_WINS
    Chromosome #2 Constrain Fitness: 0.8010204081632654
SolutionLength:222 doNothingSteps:37 coverPercentage:0.11363636363636363 bestPlayer:PLAYER_WINS
    Chromosome #3 Constrain Fitness: 0.9562499999999999
```

Figure 4 - Output of the first three chromosomes evaluated in a search-based generator.

The genetic generator uses several AI controllers to evaluate the level by comparing controller performance. *Adrienctx,* a controller that previously won the GVG-AI competition in 2014 is used as the *BestPlayer* test which attempts to beat the level and informs the generator if the level can be beaten and the score that it achieved. *Adrienctx* was modified to reduce its superhuman reaction time to a more realistic speed to avoid levels that were impossible for humans. If *Adrienctx's* performance was not reduced, the evaluation would potentially generate levels that needed unreasonable skill or reaction time for a human player to complete. Next, the *OneStepLookAhead* controller greedily searches for the next move from all adjacent tiles and the *DoNothing* controller acts as a way of determining if the level is unfairly difficult, or trivially

easy. If the *DoNothing* controller dies within the first seconds of the game starting, it is considered to be unreasonably difficult, and if the controller never dies at any point, there isn't an appropriate challenge for a human player. As long as the *DoNothing* controller meets expectations, then the game score of the *Adrienctx* and *OneStepLookAhead* controllers are compared. A level that results in a higher score difference between these two controllers is deemed to be a better level, as it shows the level rewards more intelligent play.

2.6 Human Design Patterns in Level Generators

The constructive and search-based generators created for the GVG-AI competition implicitly encode design knowledge. Specific steps and algorithms are used to statistically determine what a good level is, as opposed to subjective analysis from a human player. The automated generation is beneficial because it is fast and can be done by a machine, however it does not guarantee good design. One solution to this problem is the use of design patterns in level generation. To produce a more coherent level, a study by Dahlskog et. al in 2014 used patterns found in the original *Super Mario Bros.* levels as 'building blocks' for procedural level generation as a way of producing varied levels that appear human-made [18]. Our study expands on this idea of using human design patterns from levels in a general 2D domain for the GVG-LG track of the competition.

Two types of patterns were used in Dahlskog et. al's study. Micro-patterns are composed of a single block vertical slice of a level that occurs many times throughout the game. Meso-patterns are combinations of these micro-patterns that create continuous sections of levels. An evolutionary algorithm then pieces together micro-patterns to build levels and searches performing an evaluation on the frequency of meso-patterns found in the generated level.

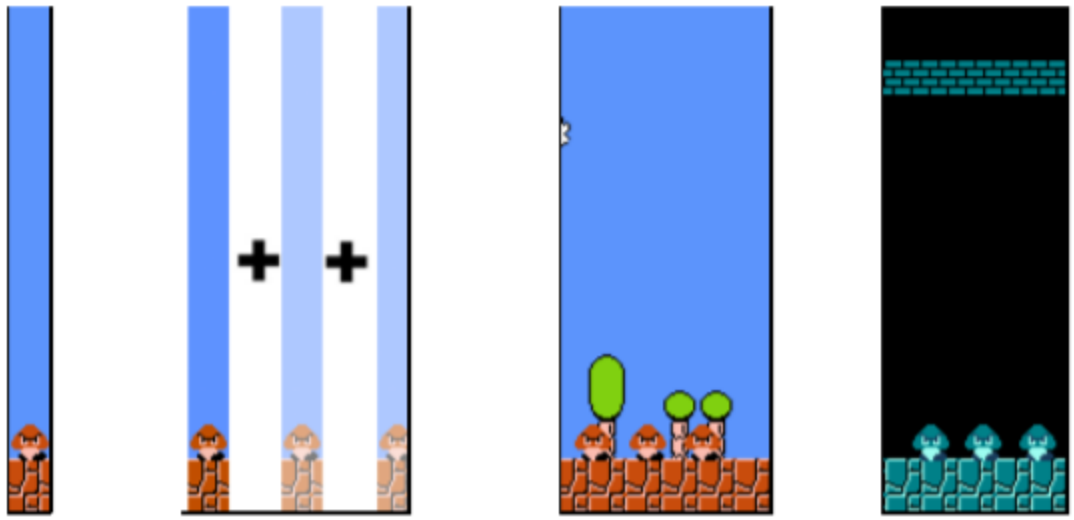


Figure 5 - Visual representation of slicing segments of previous levels to create patterns [18].

The study found that this process could successfully produce playable *Super Mario Bros*. levels that had a consistent flow and were visually appealing. Our study focuses largely on extending this approach to fit the GVG-AI framework and suit general non-platformer 2D games. This involves creating generalizable design patterns from specific game levels and procedurally applying them to levels for different games. Our study focuses on identifying micro-patterns in the GVG-AI levels to ensure that each pattern can work for any different game in the framework.

2.7 Comparing the Success of Generators

Khalifa et. al evaluated the preferences of playtesters to compare the success of the three generators produced in their study: Search-based, Constructive, and Random (which places a few of each sprite in random empty positions and then fills in the level border) [6]. Their playtesting process involved participants playing two levels produced by different generators of the same game for a direct comparison and recording a player's preference of one over the other. Three

generator comparisons were made, Search-Based vs Constructive, Search-Based vs Random, and Constructive vs Random.

	Preferred	Non-preferred	Total	Binomial p-value
Search-Based vs Constructive	23	12	35	0.0447
Search-Based vs Random	21	10	31	0.0354
Constructive vs Random	17	24	41	0.8945

Table 1 - Playtesting results of the Khalifa et. al study [6]

As shown Table 1 above, the study's results showed that players substantially preferred the Search-Based generator over both the Constructive and Random, and players slightly preferred the Random generator over the Constructive. The reasoning for the Constructive performing worse than the Random was that the Constructive couldn't guarantee that at least one object of every type in a game could be placed, whereas Random would place at least one of all the sprite types available. These results confirm that the search evaluation does find better levels than the constructive generator will typically produce on its own. Other methods for surveying players exist such as Likert scales, but could be very ambiguous to a player on how to score and ranked different levels.

There are many qualities that level generators can be assessed on, like difficulty, aesthetics, length of levels, etc., and a more complex feedback system like individual metric ratings would allow for a deeper analysis, but for individual players, these would all be considered differently and could lead to inconsistent data. The direct comparison used in the survey creates less confusion with a simpler response that still manages to answer the question of which generator is better overall.

3 - Methodology

3.1 Approach to Solution

The prior study's generators created levels by inserting calculated amounts of individual sprites into the level [6]. We identified that this strategy typically produces an unorganized and cluttered level that does not feel intentionally laid out. Our study aims to take a different approach for general level generation, focusing on using patterns from human-made levels to use as building blocks for new ones.

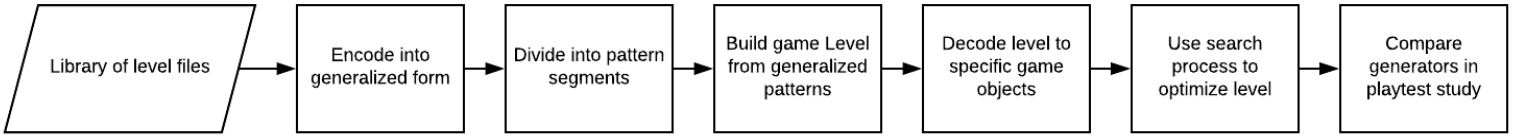


Figure 6 - System architecture for our study.

To use level information from different games, each human-made level was converted into a generalized set of information that could be used to convert into any other game. Then 3x3 segments, which we call design patterns, of these converted levels were stored for use as building blocks for new levels. Levels were then built by assembling combinations of these patterns for the layout, while ensuring that continuity is maintained, and then translating them into the appropriate sprites for a specific game. Once the level was created, it is checked to ensure that all termination conditions can be met so that only playable levels are created.

3.2 Using General Types

Levels are comprised of game-specific information that cannot be directly translated to every other game in the GVG-AI framework [2]. To make use of the level designs for different

games, the game specific information is reclassified into a general type system that can be used to translate sprites from one game into the most similar sprites of any other game.

The GVG-AI framework uses default sprite classifications of: *Avatar*, *Solid*, *Harmful*, *Collectable*, and *Other*. *Avatars* are any sprites directly controlled by the player. *Solids* are sprites that cannot be moved through and have no other interactions. *Collectables* are non-harmful and are destroyed by the player upon interaction. *Harmfuls* either destroy the player or spawn other sprites that do. *Others* are any sprite that does not meet all the qualifications for another category. These classifications allow for an approximate understanding of the role that specific game sprites play and enable the encoding of specific game sprites into a set of generalized information that can be decoded into any other game.

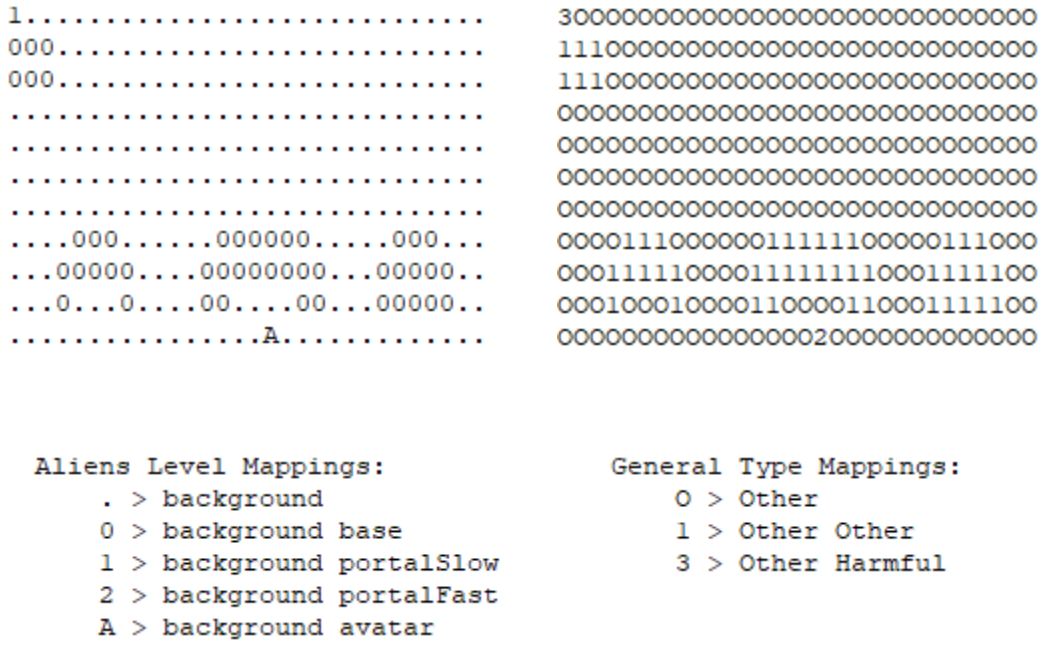


Figure 7 - Process of taking an existing *Aliens* level and encoding each game-specific object into a generalized mapping.

The encoding process is straightforward because of the direct classification from the type rules in the framework. However, the translation from specific to general results in information

loss. If a game has two sprites that are classified as the same type, like the pellets and fruit of *Pacman*, then the encoded level will lose the distinction between the two and record them both as one type.

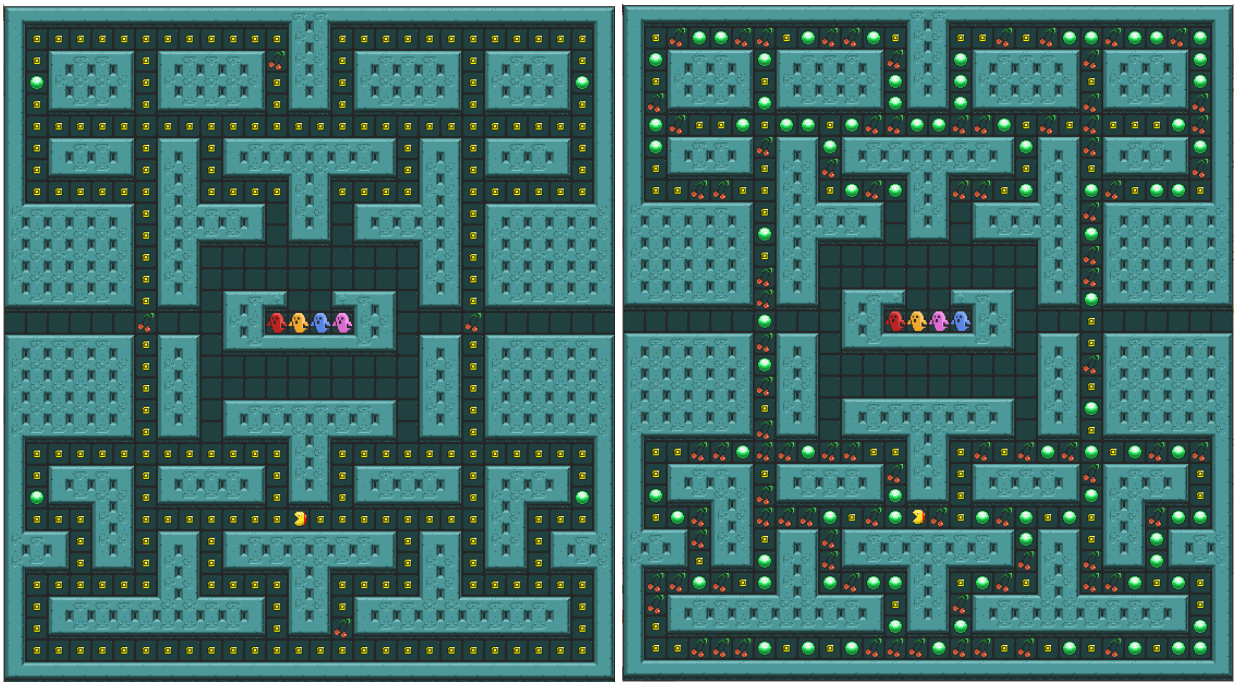


Figure 8 - Original Pacman level (left) encoded into a general form, and then decoded back (right).

Figure 8 shows how a *Pacman* level maintains sprite placement of *solids* and *avatars* but loses the distinction of *other* and *harmful* types after being encoded and decoded. The loss of information results in ambiguity and uncertain outputs. For example, a general *harmful* sprite can be reclassified as any one of the four ghosts in *Pacman* when converted to that game. Overall, the translation process is effective for selecting single sprites to match single generalized types, but games in the GVG-AI framework store levels as matrices of mappings, which can include one or more sprites. This means that to fit a game's intended level format, the process of encoding and decoding needs to be able to operate on translating groups of sprites in the form of

level mappings. In Table 2, *Pacman* is used to show how sprites of that game get classified from our general type mapping. Every space in a game contains at least one 'background' sprite which is purely visual, and any number of foreground sprites, which are the game objects.

Single Type Mappings	
General Type	Pacman Mappings Selected
A (Avatar)	A (Floor, Pacman)
C (Collectable)	o (Floor, Power), f (Floor, Fruit), . (Floor, Pellet)
H (Harmful)	1 (Floor, Red Ghost), 2 (Floor, Orange Ghost), 3 (Floor, Blue Ghost), 4 (Floor, Pink Ghost)
O (Other)	+ (Floor)
S (Solid)	w (Floor, Wall)

Combination Type Mappings	
General Type Mapping	Pacman Mappings Selected
1 (Other, Other)	+ (Floor)
2 (Other, Avatar)	A (Floor, Pacman)
3 (Other, Harmful)	1 (Floor, Red Ghost), 2 (Floor, Orange Ghost), 3 (Floor, Blue Ghost), 4 (Floor, Pink Ghost)
4 (Other, Collectable)	o (Floor, Power), f (Floor, Fruit), . (Floor, Pellet)
5 (Other, Solid)	w (Floor, Wall)
6 (Other, Avatar, Harmful)	A (Floor, Pacman)
7 (Other, Harmful, Harmful)	1 (Floor, Red Ghost), 2 (Floor, Orange Ghost), 3 (Floor, Blue Ghost), 4 (Floor, Pink Ghost)

Table 2 - Translation table from general mappings to *Pacman* mappings

Different games have different combinations of sprite types in their level mappings, thus games will not have a direct translation for every general mapping. To accommodate this for level construction, available patterns to select from could be specifically restricted to only ones with direct translations, but this would reduce the amount and variety of usable patterns. Instead, the most similar level mappings in the game description are chosen when a non-direct mapping translation is needed.

3.3 Creating Patterns

To generate the library of patterns that would be used as the building blocks for construction, the human-made levels from the framework were converted into general mappings and every 3x3 matrix of the converted levels were stored in a text file. In the context of our study, a pattern is any 3x3 segment that can be taken from the existing human-made levels in the GVG-AI framework.

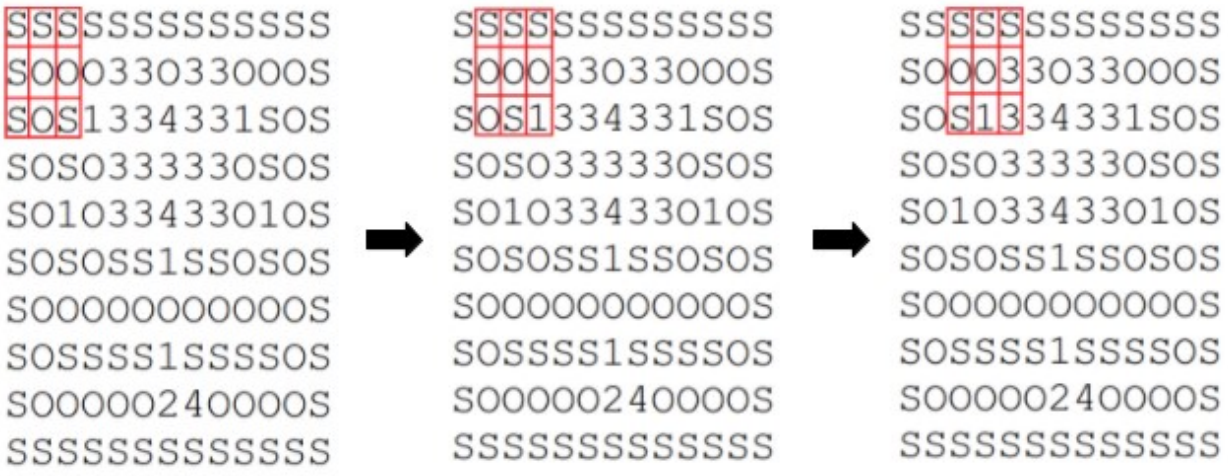


Figure 9 - Process of storing every pattern in a generalized game level. This process repeats until all 3x3 matrices have been written to a text file.

97 games were included for pattern generation, 6 games in the framework were removed (*eggomania, eighthpassenger, jaws, painter, realsokoban* and *thecitadel)* because of errors reading the game description or levels. Every game in the framework has 5 levels and the 97 games total resulted in 485 levels broken down into 114,362 patterns for construction. Most of these patterns were duplicates of the 12,941 unique patterns found.

Figure 10 - Examples of a valid bottom-left corner pattern and top wall pattern respectively, in the general code mapping format. 'S' represents solid sprites that players can't move past, and an 'X' represents any other sprite in the game.

These patterns were then separated into groups based on whether they contained an avatar or had solids around the side and could be used as a wall or edge piece for building the border of a level. Figure 10 above shows examples of border patterns. Patters were grouped into border types so that a level could not be created that an avatar could walk out of. All patterns that contained an avatar were grouped together as well because every level generated in the framework needs exactly one avatar. Due to the larger number of sprites classified as others, most of our patterns contained a majority of 'other' type sprites, as shown below in Table 3.

Five Most Frequent Patterns	
Pattern	Number of Occurrences
OOO OOO OOO	27508
OOO OOO SSS	3155
SSS OOO OOO	2466
SSS SSS SSS	2230
OOO SSS OOO	1449

Table 3 - Most frequently occuring general patterns in the human-made games

3.4 Constructive Generator

Our constructive generator combines patterns from our pre-made library to create the layout of the level and then converts the general types from the patterns into game specific level mappings. The generator uses the game description to make informed decisions about avatar placement, borders, and goal sprites (sprites that play a role in termination conditions). Due to the 3x3 shape of the patterns, levels must have lengths and widths that are divisible by 3. The generator randomly selects widths and heights of 12 or 15 to accommodate this. When building a level, patterns are randomly chosen from their index in our overall pattern file, meaning that the more frequently a pattern occurs, the more likely it is to be chosen.

```
Generate Level

    begin

        get code mapping of specific game

        if game contains solid sprites
            randomly select from border patterns to ensure enclosed level

        else randomly select from all patterns to build border

        randomly select interior patterns from all patterns

        for every pattern placed
            ensure only one avatar pattern exists
            check connectivity of level

        convert selected patterns into game-specific mappings

        ensure termination conditions can be met

    end
```

Figure 11 - Pseudo code for our constructive generator's process of making a level.

If there is no solid sprite, random patterns are selected for the entire level, otherwise a border is created along the edges of the level; a similar process is followed in the Khalifa et. al study [6]. Edge and corner pieces are randomly selected to build the border from the pre-generated lists of patterns. Every time a pattern is selected it is checked to see if it contains an avatar, if so, then no other avatar-containing pattern is chosen for the rest of the construction. In some specific cases, avatars are only allowed to move horizontally, like in *Space Invaders* [19]. For these games, patterns are chosen to ensure the bottom row has open spaces for the avatar to move through. Then, the center of the level is filled with random patterns.

As patterns are selected, they are checked to see if they break the continuity of the level. A simulated level is maintained containing all the selected patterns up until the current point while every unfilled pattern is filled with an empty placeholder. The continuity check finds every non-solid space in the level, real or simulated, and traverses across every connecting non-solid space. In Figure 12 below is pseudo-code for our method of checking the connectivity of a level.

```
Check Connectivity

    begin

        create simulation board of given level

        for every 3x3 chunk on the board {
            if chunk isn't filled with a pattern
                fill with empty spaces
        }

        store list of all non-solid (traversable) spaces

        create empty frontier list

        remove first item from traversable spaces list and add to frontier list

        while the frontier list is not empty {
            remove all traversable spaces adjacent to first item of frontier list
            add these spaces to frontier list
            remove first item from frontier list
        }

        if traversable spaces list is empty {
            return true
        else
            return false
        }

    end
```

Figure 12 - Pseudo code for checking the connectivity of a game level, ensuring a player can reach all points.

If a non-solid space could not be reached, then the level is not continuous, and a new pattern is selected. This process helps to prevent levels from having unreachable areas, while allowing for areas that have not been filled with a pattern yet to connect divided areas during construction. If all the patterns in the level have been selected and there is no avatar, a pattern in the center of the level is replaced with one that contains one. The level is then converted from the general mappings of the patterns to the specific game's level mappings. The last step is for the generator to check if the game can be successfully completed. As shown in Figure 1, every game has a set of termination conditions, defining how the level can be won or lost. If a termination condition is unachievable (i.e. the game cannot finish) or already met (i.e. the game ends

immediately), then the appropriate goal sprites are randomly added to the level to ensure that the game is playable.

3.5 Search-Based Generator

The prior study found that playtesters strongly preferred levels from their searched-based generator over those from their constructive generator [6]. This indicates that the search-based approach should produce better levels overall. We adapted the prior search-based generator to use our constructive generator to create initial populations and modified the mutation function to operate on patterns instead of individual sprites and used the same fitness evaluation. The prior search generator had three possible mutations: insert a sprite, remove a sprite, and swap two sprites. Whether or not a mutation would occur was determined by a probability value, outside of the mutate function itself. Pseudo code for our mutate function is provided in Figure 13.

```
mutate

    begin

        if random number is less than mutation probability {

            randomly select 3x3 chunk to replace

            if selected chunk is on the border of the level {
                randomly replace with chunk of same border type
            else
                randomly replace with any other pattern
            }
        else
            swap two non-border 3x3 chunks
        }

        ensure termination conditions can be met

    end
```

Figure 13 - Pseudo code for the mutation function of our search-based generator

To work at a pattern level, our search-based generator has two possible mutations: swap two patterns and replace a pattern. Our study used the default parameters of the search-based generator to compare how our pattern-based approach would affect it [6]. An initial population size of 50 was created, with a crossbreed probability of 70% and a mutation probability of 10%. The generator would keep running until a specified time limit was reached (1 hour by default), where it would then return the level with the highest fitness value.

3.6 Making the Generators

Before winning conditions were accounted for, levels were created to test the basic functionality of creating a new level from piecing together 3x3 matrices. Below is a level purely made from using our pattern logic, with no account for continuity or achievement of goals in the game.

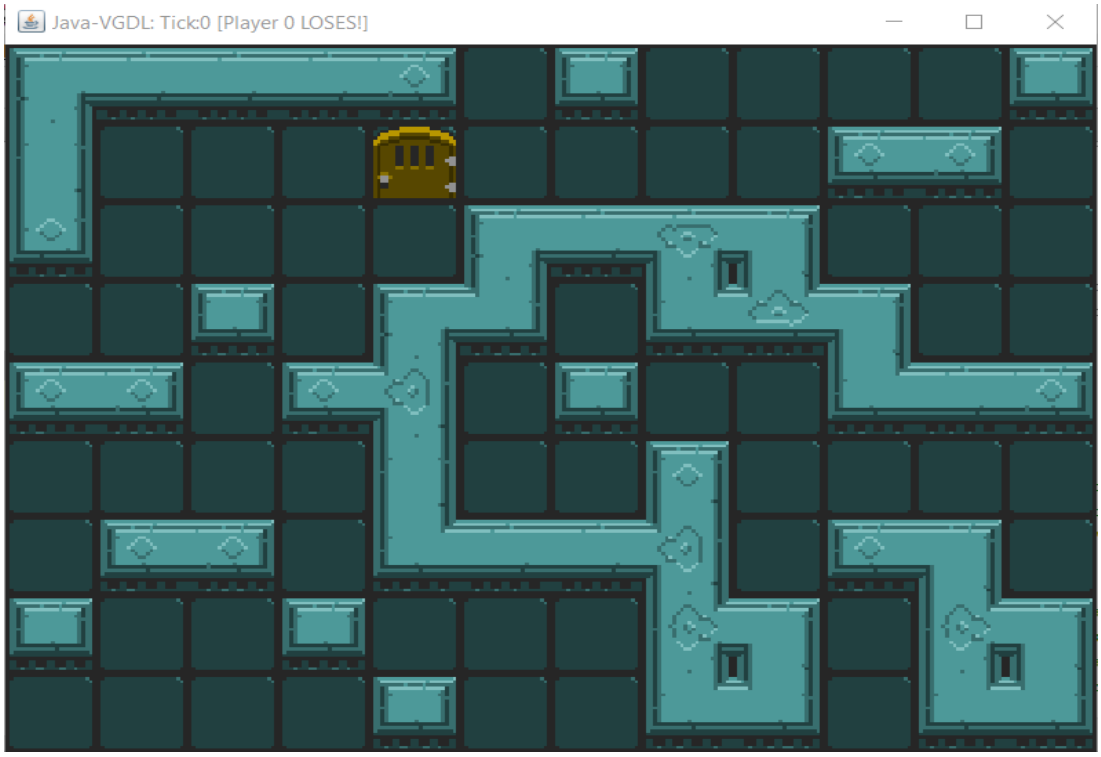


Figure 14 - Later iteration of our constructive generator, testing the use of 3x3 matrix patterns to generate a level. Each matrix was chosen purely at random and inserted in as a solid sprite classification.

Earlier in the process when assuring level mappings, we encountered interesting outputs when some levels were mapped all to one sprite type. Figure 15 is another *Zelda* level before border patterns were introduced, which happened to select no patterns with solids in them.

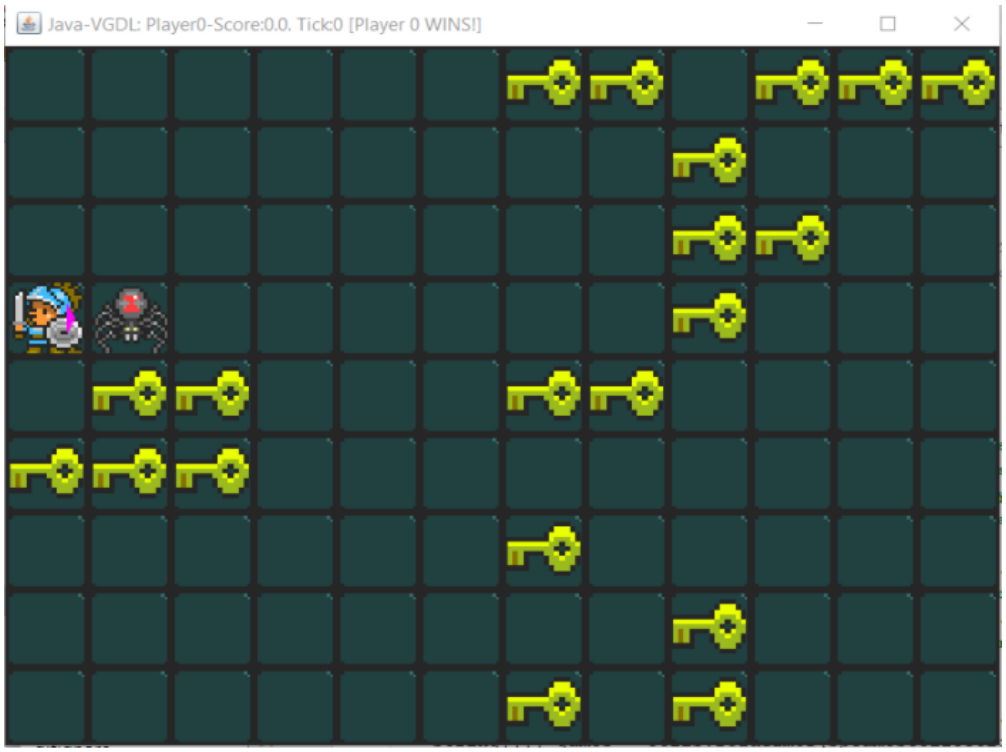


Figure 15 - A constructive generator output that did not yet account for any solid borders, and solid type sprites were mixed with other type sprites, resulting in an interesting layout of keys.

Once our generators had been created, they showed a distinct visual difference from the previous generators, consistently using more of the blank space available in each level.

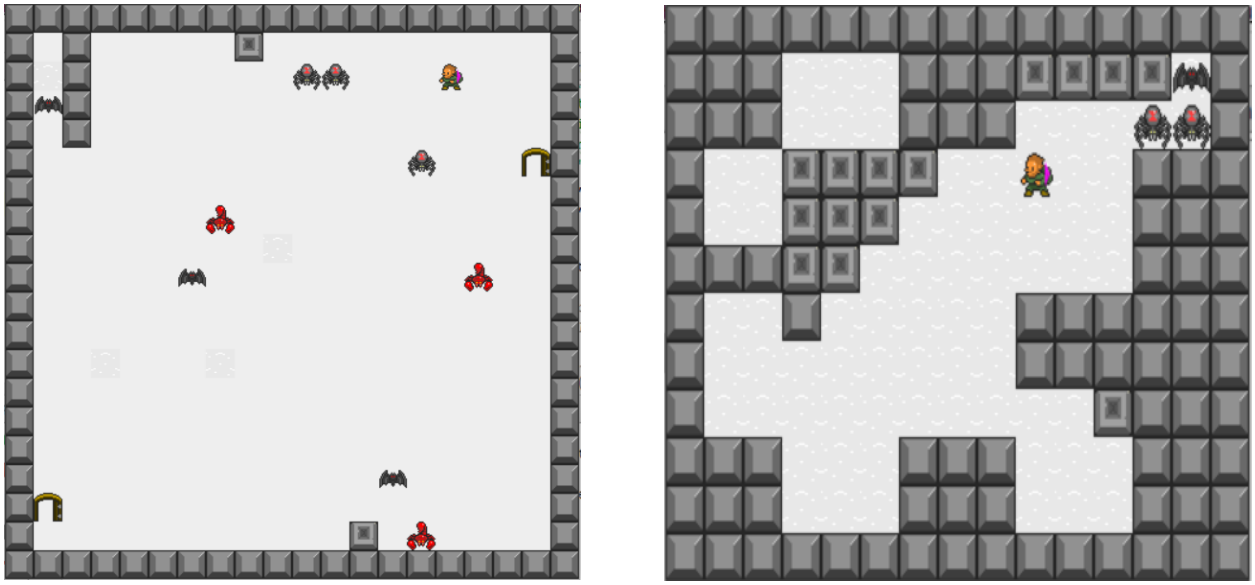


Figure 17 - Comparison of the previous constructive generator vs our constructive generator for *Bomberman*

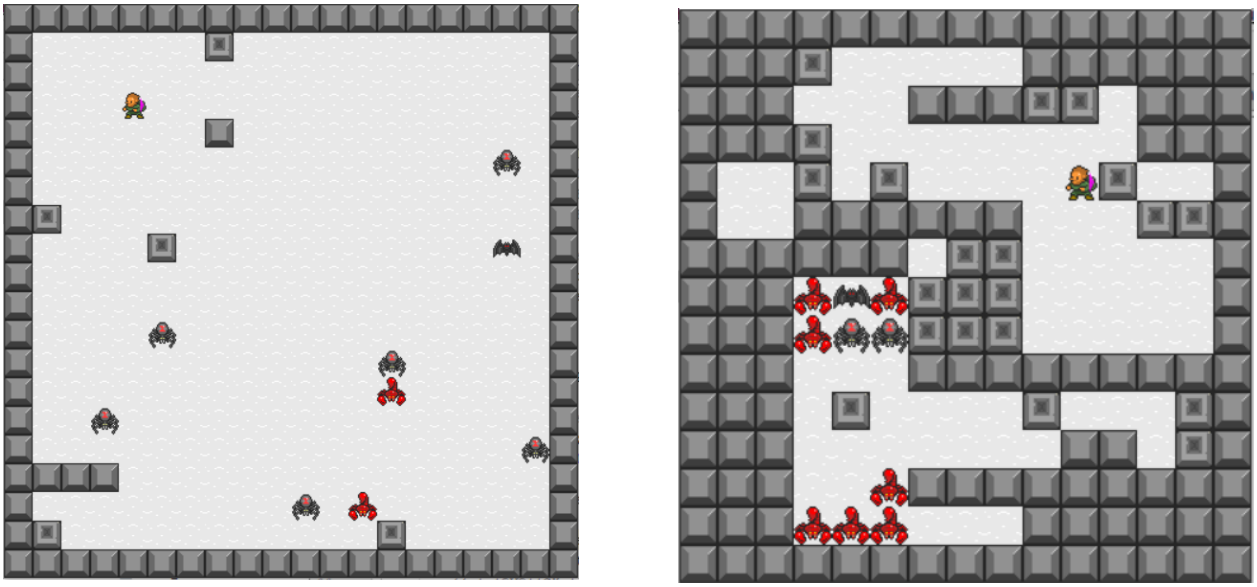


Figure 16 - Comparison of the previous search-based generator vs our search-based generator for *Bomberman*

The previous constructive and search-based generators consistently created levels with a largely minimal environment, including just enough sprites to make the level playable, but lacking structure and flow within the level. Our generators tended produced more objects on average, resulting in more variation per level.

4 - Testing and Results

4.1 Level Preparation

Khalifa et. al's study chose to collect data by having survey participants play three popular games: *Frogs, Pacman,* and *Zelda* [6]. All these games are familiar, short and have easy rules to pick up on. Because of errors with the updated version of *Pacman,* which would break if there was not exactly one ghost of each color generated in any level, we replaced *Pacman* with *Bomberman*, an equally popular game with mechanics varied enough from *Zelda* and *Frogs* to provide our generators with a distinct set of rules to account for to further express generality in our work. Below is a brief description of the gameplay and objectives of each of the games our study used for surveying:

1. *Bomberman* - Port of original *Bomberman.* Move around the level and place bombs that explode in a cross shaped pattern (+) to either destroy dark blocks or kill enemies. The goal is to find all doors which are hidden beneath destroyable blocks.
2. *Frogs* - Port of *Frogger.* The objective is to reach the end goal(s) while avoiding vehicles and water which will kill you on contact. The character can only move across water if they are on a log, otherwise they will fall in the water and drown.
3. *Zelda* - Port of the original *Legend of Zelda* game. The player must first collect a key to then be able to unlock the exit door. The character can attack in the direction they are facing to kill an enemy.

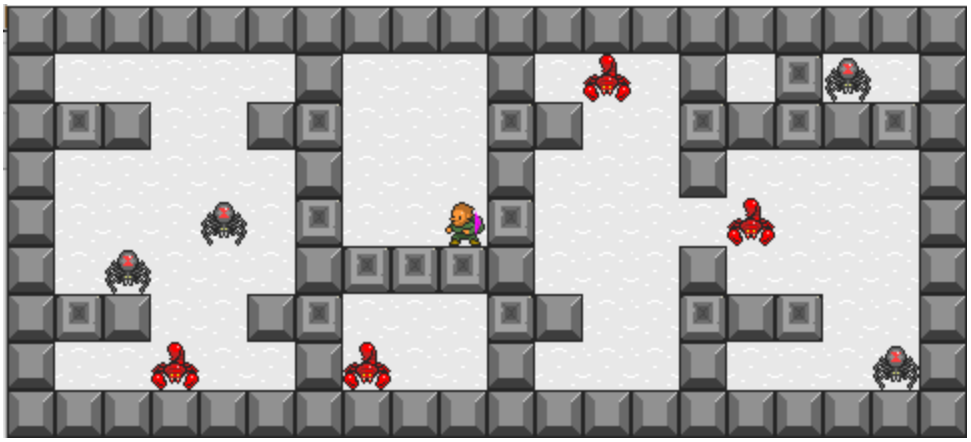


Figure 18 - A human-designed level for *Bomberman*

Figure 19 - A human-designed level for *Frogs*

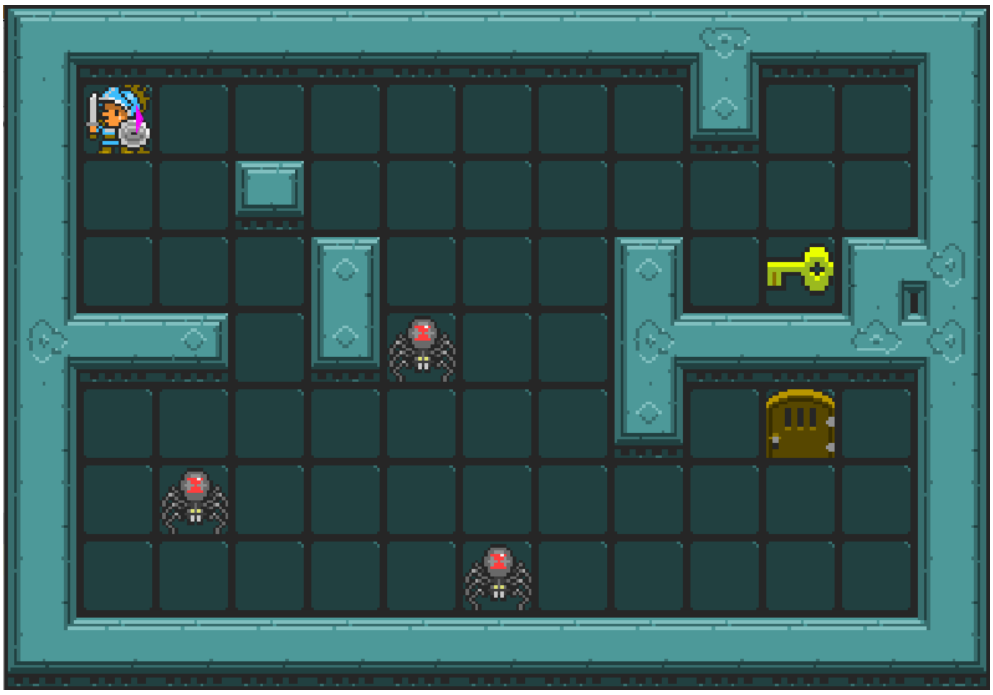


Figure 20 - A human-designed level for *Zelda*

4.2 Survey

After the three games to survey were decided, 5 levels were generated for each game on our study's constructive and search-based generator, as well as the prior study's search-based generator. A total of 45 levels (15 per generator) were created and stored for surveying. No biased selection was made in deciding which levels to test on, to most accurately express the generators.

The survey involved having participants play two levels of the same game but from different generators, repeated for each game to account for all generator comparisons, resulting in each person playing a total of six levels. Each person would play three comparisons:

1. Old Search-based vs New Search-based
2. Old Search-based vs New Constructive
3. New Constructive vs New Search-based.

Khalifa et. al's old constructive generator was not included in our comparison, as it was proven to be inferior to their search-based generator in their results and including another generator would have needed to include several new comparisons, extending the time needed for each participant beyond a reasonable duration. Their random level generator was also not included in our study, as it had no intelligent approach to building a level, only placing the necessary number of sprites for completion of a level in random places [6].

A short program was created to let participants easily play the six levels while randomizing the order of comparisons to ensure stochastic and unbiased results. This program was based off Ahmed Khalifa's survey program used in his initial generator comparison, to which we are grateful for him providing to us [6]. The order of each game was randomized, as well as which comparison and which level out of the five made per generator.

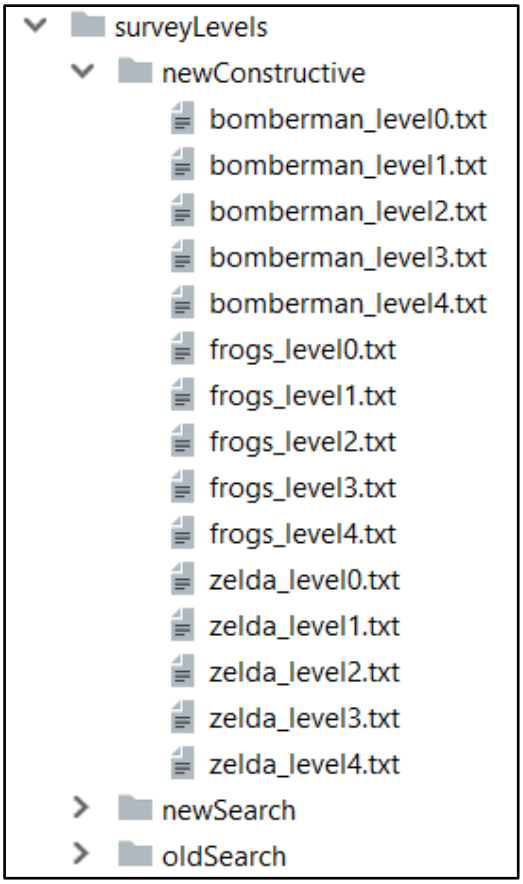


Figure 21 - Folder hierarchy for the game levels chosen for testing. The *newSearch* and *oldSearch* folders also contain 5 levels per game, made with that generator

For each survey taken, a member of our team clearly explained the rules for each game and reassured no personal identifying information would be stored. A player was asked to simply say which level they preferred as well as their gender, age and experience with video games.

4.3 Testing Results

Our hypotheses for each of the three direct generator comparisons would be that our constructive generator would be preferred over the prior search generator and that our search generator would be preferred over both our constructive generator and the prior search generator. Table 6 shows the results of our studies in which our search generator was greatly preferred over our constructive (significant results, $p < 0.05$) and both of our generators were marginally

preferred over the prior search generator. A total of 30 surveys were performed with 20 male and 10 female participants of varying experience with video games.

	A	B	Total	Percent Success	Binomial P-Value
New Constructive (A) vs Old Search (B)	16	14	30	53.33%	0.4278
New Search (A) vs Old Search (B)	17	13	30	56.67%	0.2923
New Search (A) vs New Constructive (B)	20	10	30	66.67%	0.0494

Table 4 - Playtesting results for our generators and the search generator from the prior study

Similarly to the Khalifa et. al study, the likely main reason for our search generator being preferred over our constructive was that the simulation-based evaluation and constraints were effective at producing more playable levels [6]. Particularly that the modified *Adreinctx* agent was able to prevent levels that were too difficult from being outputted. Despite the preference of our search generator over our constructive, both performed similarly when compared against the prior search generator. This could indicate that when levels of the prior search generator were compared against ours, players would evaluate them with slightly different criteria than when they compared the levels of the two pattern-based generators. Table 7 displays how often participants confirmed or contradicted our hypotheses for all three generators, divided by game experience.

	Confirm	Contradict	Percent Success	Binomial P-Value
None	2	1	66.67%	0.5
Limited	6	12	33.33%	0.9519
Moderate	24	12	66.67%	0.0326
Substantial	21	12	63.64%	0.08138

Table 5 - Participant preferences divided by experience with video games

Players with moderate or substantial experience tended to align with our hypotheses about two thirds of the time, while players with low experience did so just one third of the time. This indicates that prior game-playing experience influenced how the players compared levels. Low experience players tend to have a higher learning curve and value a more approachable level that is less difficult and complex, while more adept players can focus more on the level itself rather than the basic mechanics of the game. This makes more experienced players more efficient judges as they require little time to learn the rules and controls of a new game, while newer players are too focused on the game's dynamics to effectively evaluate how the level impacts their experience.

	A	B	Total	Percent Success (Adjusted)	Percent Success (Original)	Binomial P-Value (Adjusted)	Binomial P-Value (Original)
New Constructive (A) vs Old Search (B)	13	10	23	56.52%	53.33%	0.3382	0.4278
New Search (A) vs Old Search (B)	15	8	23	65.22%	56.67%	0.1050	0.2923
New Search (A) vs New Constructive (B)	17	6	23	73.91%	66.67%	0.01734	0.0494

Table 6 - Playtesting results for our generators and the search generator from the prior study, with players of low or no experience removed

Table 8 displays the results of our study for just players with moderate or substantial experience with video games. These results are similar, but with higher confidence values, for the comparison between our constructive generator and the prior search (small preference for our constructive but not statistically significant), as well between our two generators (large, statistically significant preference for our search generator over the constructive). However, for the comparison between our search generator and the prior search generator shows a much greater preference towards ours that approaches statistical significance, with a confidence of about 90%.

4.4 Insights while Testing

In our survey, we only asked for each participant to select which level they preferred overall, yet many people gave interesting reasons for their choices. Reasons for selecting one level over another varied drastically by person, and we found that players who considered

themselves to be less experienced with video games usually chose the old search-based generator over our generators. Often their reasoning was that the levels made using the old search-based generator seemed far easier because it was more open and had less total sprites on screen, making the level seem more approachable. Other people gave unique reasons for their level preference. One participant said that they preferred a level simply because it was much larger and said that contents had no influence for them.

4.5 Level Metrics and Expressive Range

The amount of possible combinations of patterns plays a significant role in both the quantity and quality of possible outputs. For the constructive generator, 4.5 x 10^98 combinations of patterns are possible for non-bordered games like *Space Invaders* [19], for bordered games like *Zelda*, 9 x 10^76 combinations are possible. The number of levels that a generator can produce is only important when it comes to providing unique outputs, a very small amount of levels will result in duplicates or small variations between levels, but a very large amount can result in levels of inconsistent quality.

The expressive range of our generative process is only limited by the continuity of levels and the border, if one exists. This means that the range of possible outputs is not restricted by more sophisticated evaluations of quality, like difficulty or aesthetics. Significant portions of the expressive range are comprised of levels that have extremely high or low difficulty or contain too many or too few objects to be enjoyable. The effect of this was observed in the reactions from study participants. For some levels they would be shocked at large masses of enemies that made it almost impossible to play or confused by a trivially easy level. While still some other levels presented a more moderate challenge that provided an enjoyable experience for players.

These reactions paired with the massive, relatively unrestricted expressive range of our generator indicates that the pattern-based approach can yield successful results but needs to be improved for the consistency of levels.

A simple method of assessing the expressive range of a generator is to count how many of each type of sprite gets placed in levels on average. The raw amount of each object in a level can give a rough estimate of the level's density, difficulty, and length depending on the game. For the games used in our playtest survey, harmful and collectable objects increased the difficulty of the level by adding adversaries or additional goals which must all be reached. The proportion of the level that was filled with solids can also be used as a metric for how much open space exists for the player to traverse. Objects that are classified as *other* are less informative because they are predominantly background tiles that don't impact gameplay. The number of standard deviations away from the mean for each object count can be used to tell how far a level is from the typical level that the generator will produce.

	Solids	Collectables	Harmfuls	Other
Average	98.876	5.793	7.037	194.797
STD	13.585	3.992	5.726	31.828
Max	135	28	33	266
Min	64	2	0	144

Table 7 - Object count metrics for 1,000 levels of Frogs from our constructive generator

Table 4 displays the expressive range for each of the object counts for our constructive generator for *Frogs* across 1,000 levels. The amount of harmful and collectable objects has quite a large range that is a primary cause for the inconsistency of level difficulty. Of the 1,000 levels

generated, they ranged from having no harmful sprites to as much as 33. This type of possible variation makes it very important to assess just how representative the levels used in our playtesting study are of the typical levels produced by our generator.

	Solids	Collectables	Harmfuls	Other
Average	94.8	5	9	187.2
STD	3.493	3.742	5.244	4.324
Max	98	11	16	192
Min	91	2	4	182

Table 8 - Object count metrics for 5 Frogs levels from our constructive generator used in the playtest survey

Table 5 displays the range of object counts represented by the 5 constructive levels used in our playtesting study for *Frogs*. The average amounts of each object type are very close to the averages for the overall generator, except for harmful types being slightly more frequent in the survey levels. The standard deviations of the *solid* and *other* types are much lower in the survey levels indicating that they cover only a small portion of the generator's expressive range. The standard deviations for the *collectables* and *harmfuls* are very close however. As these have a more direct impact on the player, and the averages for all 4 types are relatively consistent with the expected values, we determined that the constructive levels used for *Frogs* in our survey are fair representations of our generators expressive range and that none of the levels used are outliers. Similar conclusions were found for the constructive levels in the survey for *Bomberman* and *Zelda*. The tables for these can be found in Appendix C. Due to time constraints this analysis could not be performed for the search-based generator. The search process takes about half an

hour or more to create a level and was infeasible to generate enough levels to perform a meaningful analysis.

5 - Conclusions and Future Work

Our study sought to explore the potential of procedurally generating game levels that felt more human-made than ones from more basic generators. Overall, we have shown that a pattern-based approach is a viable method of level generation for 2D arcade-style games. The levels produced by our generators had a more organic and flowing structure that created a positive gameplay experience. Our generators produced levels that were far more varied in style and gameplay compared to the previous generators, and the majority of survey participants expressed that the levels were considered to be fun and a challenging experience.

The major weakness of our generators was the inconsistency of level difficulty. Some levels would place the player directly next to an enemy at the start or would require the player to path through a region filled with a massive number of enemies. Meanwhile, other levels could have no enemies or place them in such a way the player was unlikely to need to interact with any. The prior study's constructive generator partially addresses this by using a relatively consistent amount of each object in a game and placing harmful objects distant to the avatar, however this does not work well universally across different games [6]. For example, a water tile in *Frogs* is not nearly so dangerous as the alien spawner in *Space Invaders* and a quality level generator would treat them differently. The more promising approach is to examine how the level can be played out using an AI player, because the evaluation can be performed regardless of what type of game it is. The search algorithm developed by Khalifa et. al partially addresses this by evaluating the performance of the modified *Arienctx* AI compared to a greedy AI, but this does not directly assess how difficult the level is, only if more intelligent play will lead to a better score [6]. If given the opportunity to continue our work, we would have developed a method of evaluating difficulty with AI play. Numerous approaches could be taken for this

including an AI which attempts to lose the game as fast as possible to evaluate if the level could end too abruptly or a decision tree analysis that estimates how many lines of play result in victory vs the amount that result in a loss. The game specific information in the game description is very difficult to use directly to understand what effect objects will have on the difficulty of a level.

The pattern-based approach was fairly successful at creating flowing levels with multiple varied sections. However, these sections were not always combined in optimal or natural ways because the patterns were grouped and selected randomly. The Dahlskog et. al study working on pattern-based approaches for *Super Mario Bros.* addressed this issue in their search generator that scores levels by how many meso-patterns, or combinations of patterns, that appear in the original levels that the patterns were generated from [18]. If given the opportunity to continue our work, we would have combined this strategy with the search evaluation we used. This would involve abstracting the process used by Dahlskog et. al out to a 2D general domain to incorporate both vertical and horizontal combinations of patterns that occur in many different games.

The survey process we used to evaluate the different generators was effective, but could be expanded in future work to include direct comparisons for more specific metrics like difficulty, playtime, or aesthetics. The player's overall level preference is important and effective for providing a holistic understanding of which generator is more successful, but similar questions about more specific traits would allow for a more detailed analysis of the strengths and weaknesses of a generator.

This project took the pattern-based approach for a single linear game and expanded it to work on a much wider domain of 2D games. The success that our work has had so far indicates that pattern-based approaches could be expanded to even wider domains of 2D games or even

3D games. The VGDL is very effective for working on a subset of 2D arcade-style games, but if our approach were to be used on a different type of game it would require a description language capable of standardizing game definitions for the new domain. Side scrolling games like *Super Mario Bros.* and games that don't have an avatar like board games are common types that cannot be expressed in the VGDL, but if given a strong description language, generalized human design patterns should be a valid approach for level generation.

Generalizable object types are the key to this approach working for a non-specific domain. A domain that spans widely different games would require a type system that can effectively classify all objects and could pose a problem if design patterns from extremely different games cannot be used effectively. This issue did not arise working with the VGDL but could be addressed with an evaluation of what design patterns could be most pertinent to a game, or with a search process that involves AI player evaluations like the one used in this study.

The GVG-AI framework and VGDL provided an effective environment to research PCG once the nuances of how they work were understood. Some games descriptions have errors in them that break the game and result in malformed levels. Additionally, the framework is continuously being updated which can impact work made on previous versions. The prior study's generator suffered from this because backgrounds had not been introduced yet, so combinations of objects in a single space of the game were not common, leading to the use of a less sophisticated level mapping system that did not handle backgrounds properly in the outputted levels.

The GVG-AI framework remains an active base of research for general AI principles, and our study provided a distinct approach towards the level generation track. By employing the extensive body of work from human developers, implicit design information can be combined

with the efficiency of PCG to produce consistently fun and unique content over many different games.

Bibliography

[1] - B. Goertzel and C. Pennachin, *Artificial General Intelligence*. Berlin: Springer, 2007.

[2] - *The GVG-AI Competition*. [Online]. Available: http://www.gvgai.net/. [Accessed: April 23, 2018].

[3] - N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[4] - Nintendo. 1985. Super Mario Bros. Nintendo Entertainment System. Shigeru Miyamoto, Takashi Tezuka.

[5] - Sega. 1991. Sonic the Hedgehog. Sega Genesis. Hirokazu Yasuhara (designer).

[6] - A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General Video Game Level Genration", 2016. [Online]. Available: http://julian.togelius.com/Khalifa2016General.pdf. [Accessed: April 23, 2018]

[7] - Michael Toy, Glenn Wichman. 1980. Rogue (video game). Atari 8-bit.

[8] - Acornsoft. 1984. Elite. BBC Micro. David Braben, Ian Bell.

[9] - Mossmouth, LLC. 2008. Spelunky. Microsoft Windows. Derek Yu.

[10] - J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in Proceedings of EvoApplications, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2010.

[11] - G. Smith. "Understanding Procedural Content Generation: A Design-Centric Analysis of the Role of PCG in Games." In *Proceedings of the 2014 ACM Conference on Computer-Human Interaction*. Toronto, Canada. Apr, 2014.

[12] - T. Schaul. "An extensible description language for video games". *Computational Intelligence and AI in Games, IEEE Transactions*, 2014.

[13] - J. Togelius and G. N. Yannakakis, “General general game AI,” in *Proceedings of the Computational Intelligence and Games Conference*. IEEE, 2016.

[14] - S. Lavelle. *PuzzleScript*. [Online]. Available: http://www.puzzlescript.net/. [Accessed: April 23, 2018].

[15] - J. Orwant, "EGGG: Automated programming for game generation," *IBM Systems Journal*, 2000.

[16] - Mojang. 2011. Minecraft. Microsoft Windows. Markus Persson, Jens Bergensten.

[17] - Edmund McMillen. 2011. The Binding of Isaac. Microsoft Windows. Edmund McMillen, Florian Mimsl

[18] - S. Dahlskog and J. Togelius, "Procedural Content Generation Using Patterns as Objectives," in *Proceedings of EvoGames, part of EvoStar*., A. I. Esparcia-Alcazar, Ed., 2014.

[19] - Taito. 1978. Space Invaders. Arcade. Taito.

Appendix A - General Code Mappings Table

General Code Mapping Spreadsheet		
Character(s) Present	Sprite(s) Represented	Code Mapped
A	Avatar	A
C	Collectable	C
H	Harmful	H
O	Other	O
S	Solid	S
O, O	Other, Other	1
O, A	Other, Avatar	2
O, H	Other, Harmful	3
O, C	Other, Collectable	4
O, S	Other, Solid	5
O, A, H	Other, Avatar, Harmful	6
O, H, H	Other, Harmful, Harmful	7
A, S	Avatar, Solid	8
S, H	Solid, Harmful	9

Table 9 - Process of mapping characters to general types. In the first column, every combination of general character types is accounted for within the current framework. Secondly, a description of what classified sprites are represented

Appendix B - Pattern Frequency Table

Pattern Frequency	Number of Unique Patterns with That Frequency
20,000+	1
10,000 - 19,999	0
1,000 - 9,999	6
500 - 999	18
250 - 499	14
100 - 249	86
50 - 99	101
25 - 49	200
10 - 24	583
2 - 9	4821
1	7111

Table 10 - Frequency chart for unique patterns

Appendix C - Expressive Range Metrics

	Solids	Collectables	Harmfuls	Other
Average	93.047	4.814	6.909	101.714
STD	12.822	3.742	5.244	22.880
Max	141	22	30	170
Min	62	1	0	52

Table 11 - Object count metrics for 1,000 levels of *Bomberman* from our constructive generator

	Solids	Collectables	Harmfuls	Other
Average	89.6	4.6	10.2	98.4
STD	21.090	3.847	8.319	9.044
Max	121	11	12	112
Min	65	1	3	87

Table 12 - Object count metrics for 5 *Bomberman* levels from our constructive generator used in the playtest survey

	Solids	Collectables	Harmfuls	Other
Average	94.603	4.917	6.915	99.136
STD	13.709	4.001	5.578	22.674
Max	134	23	30	173
Min	59	1	0	52

Table 13 - Object count metrics for 1,000 levels of *Zelda* from our constructive generator

	Solids	Collectables	Harmfuls	Other
Average	106.2	3.6	5	109.8
STD	17.852	2.966	4.528	24.222
Max	133	8	11	147
Min	88	1	0	84

Table 14 - Object count metrics for 5 *Zelda* levels from our constructive generator used in the playtest survey

	Solids	Collectables	Harmfuls	Other
Average	98.876	5.793	7.037	194.797
STD	13.585	3.992	5.726	31.828
Max	135	28	33	266
Min	64	2	0	144

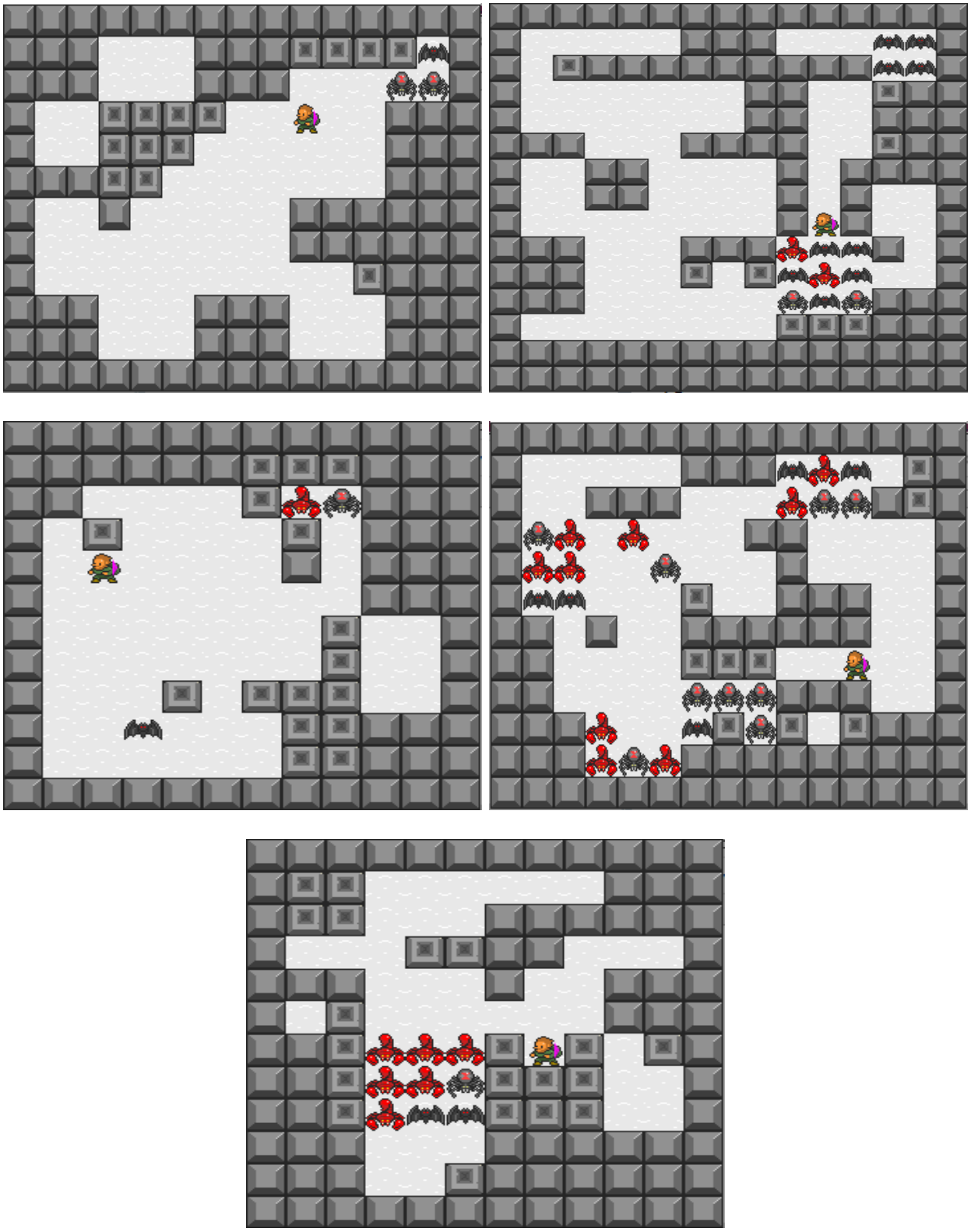
Table 15 - Object count metrics for 1,000 levels of *Frogs* from our constructive generator

	Solids	Collectables	Harmfuls	Other
Average	94.8	5	9	187.2
STD	3.493	3.742	5.244	4.324
Max	98	11	16	192
Min	91	2	4	182

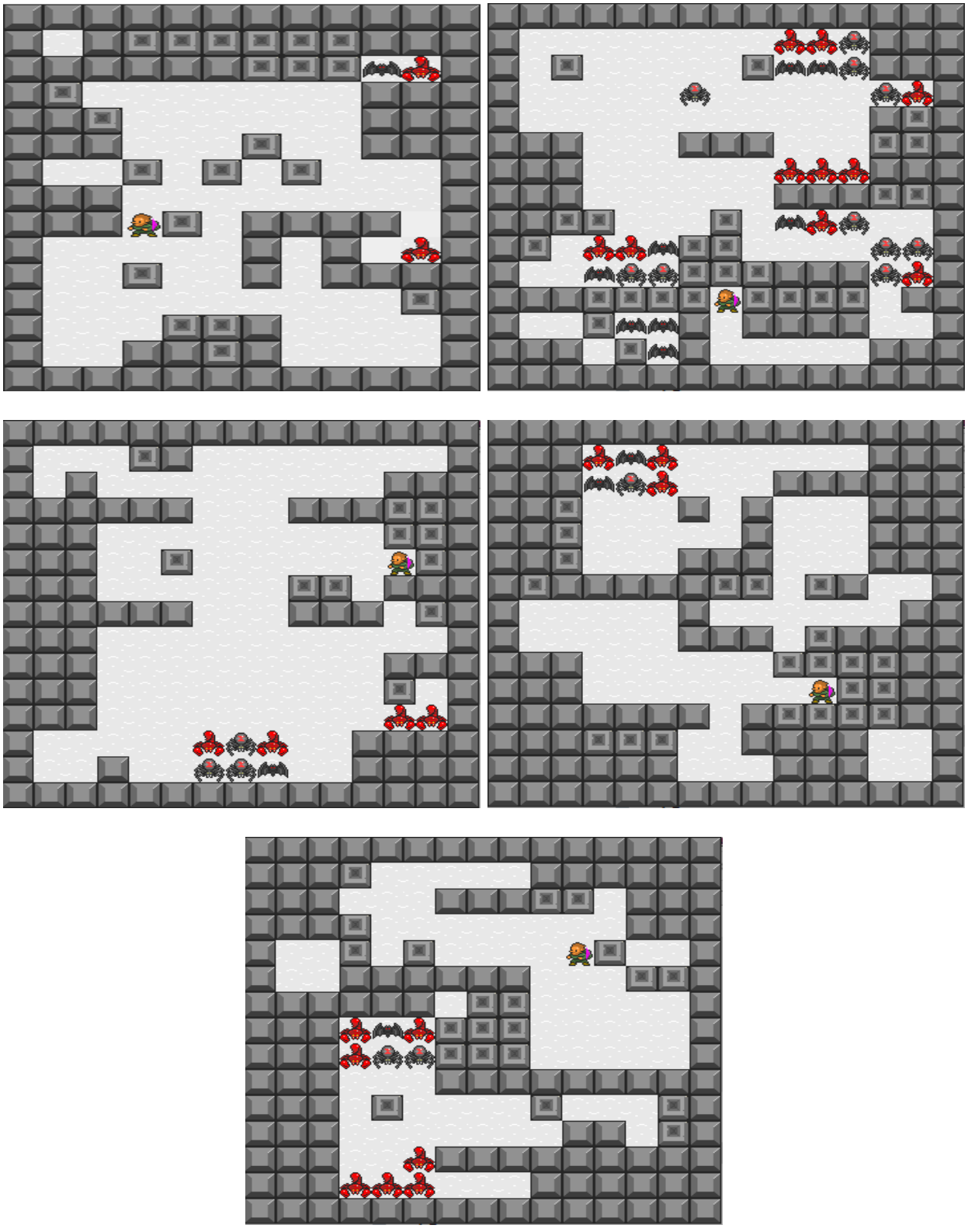
Table 16 - Object count metrics for 5 *Frogs* levels from our constructive generator used in the playtest survey

Appendix D - Survey Levels

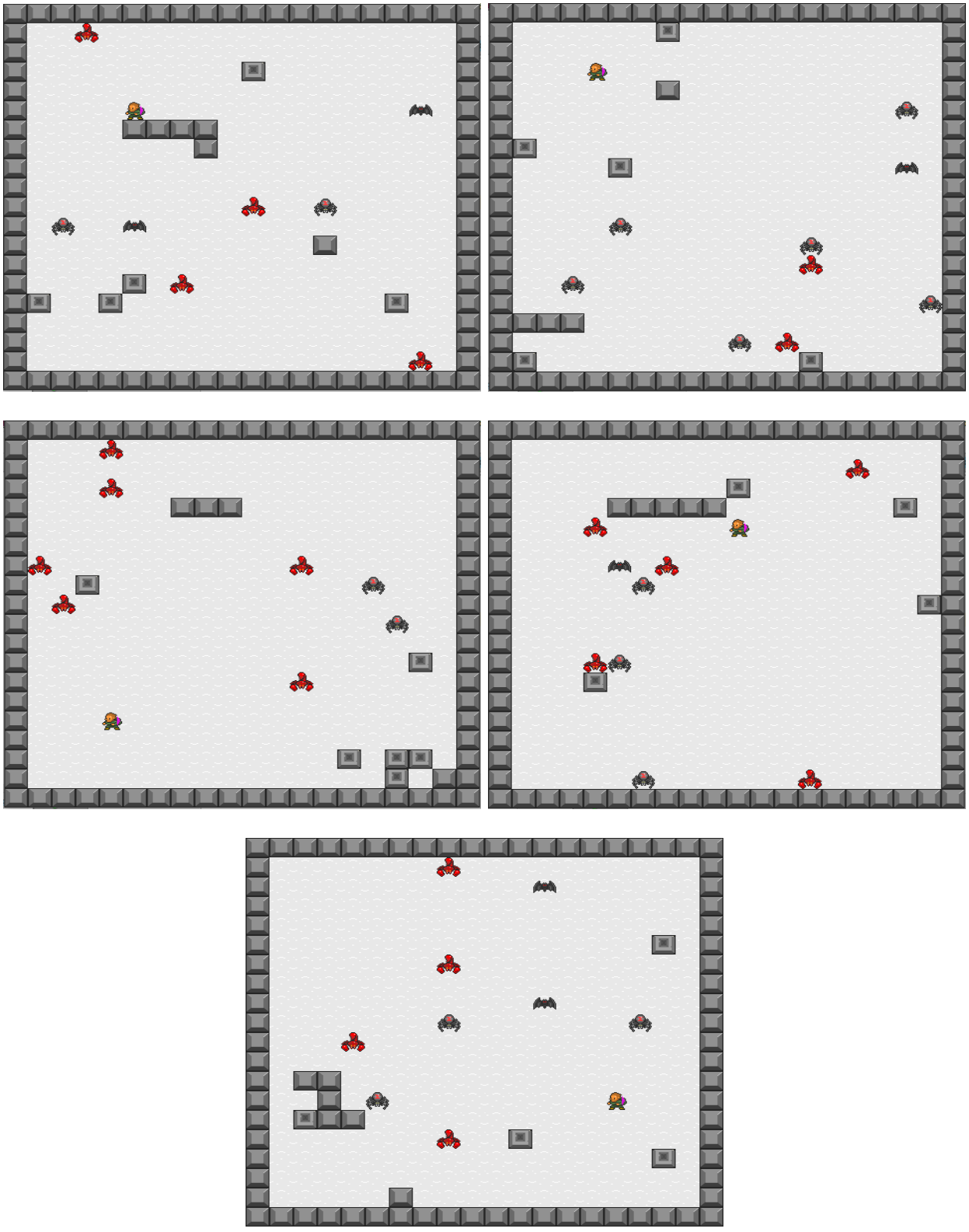
Bomberman New Constructive Levels:



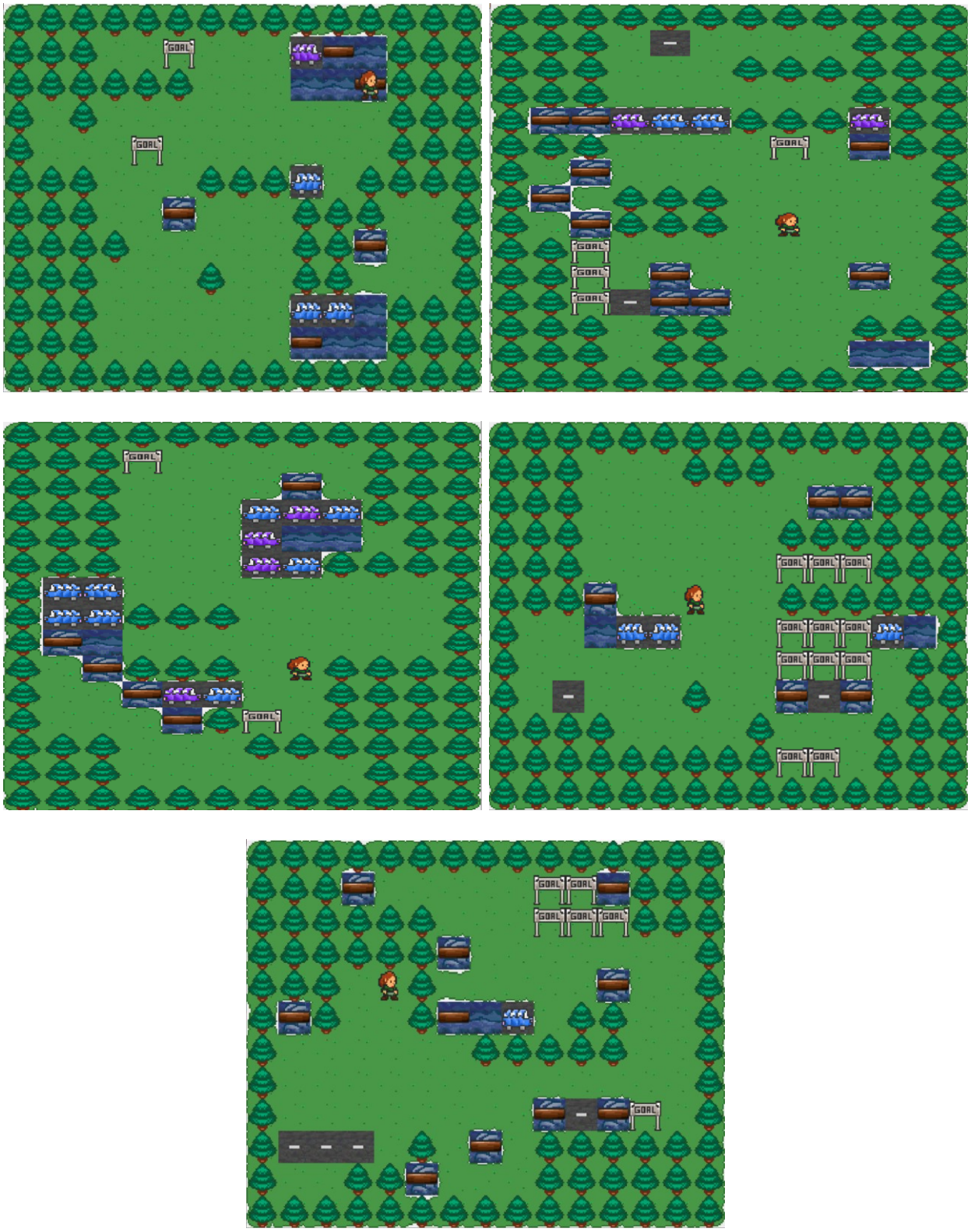
Bomberman New Search Levels:



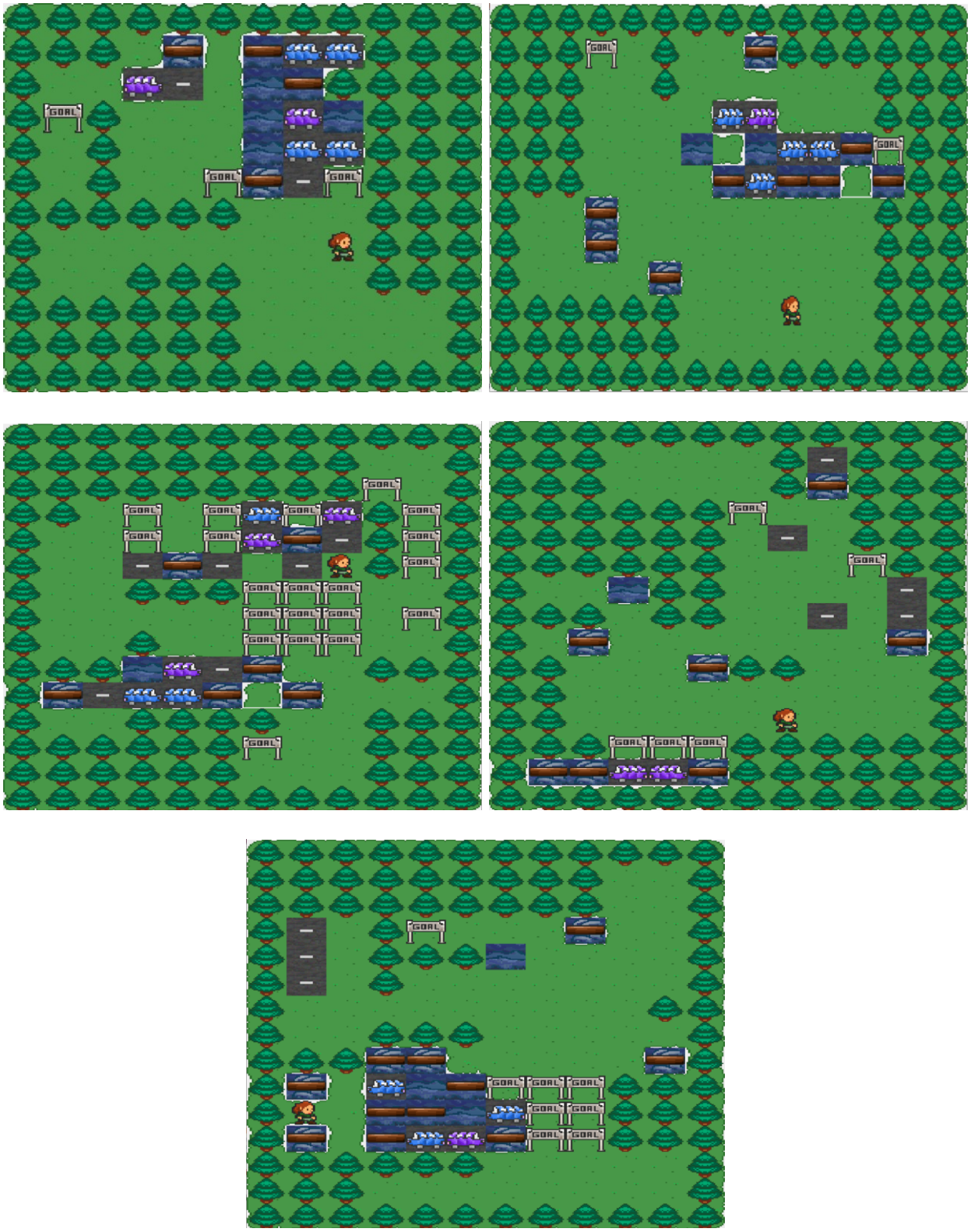
Bomberman Prior Search Levels:



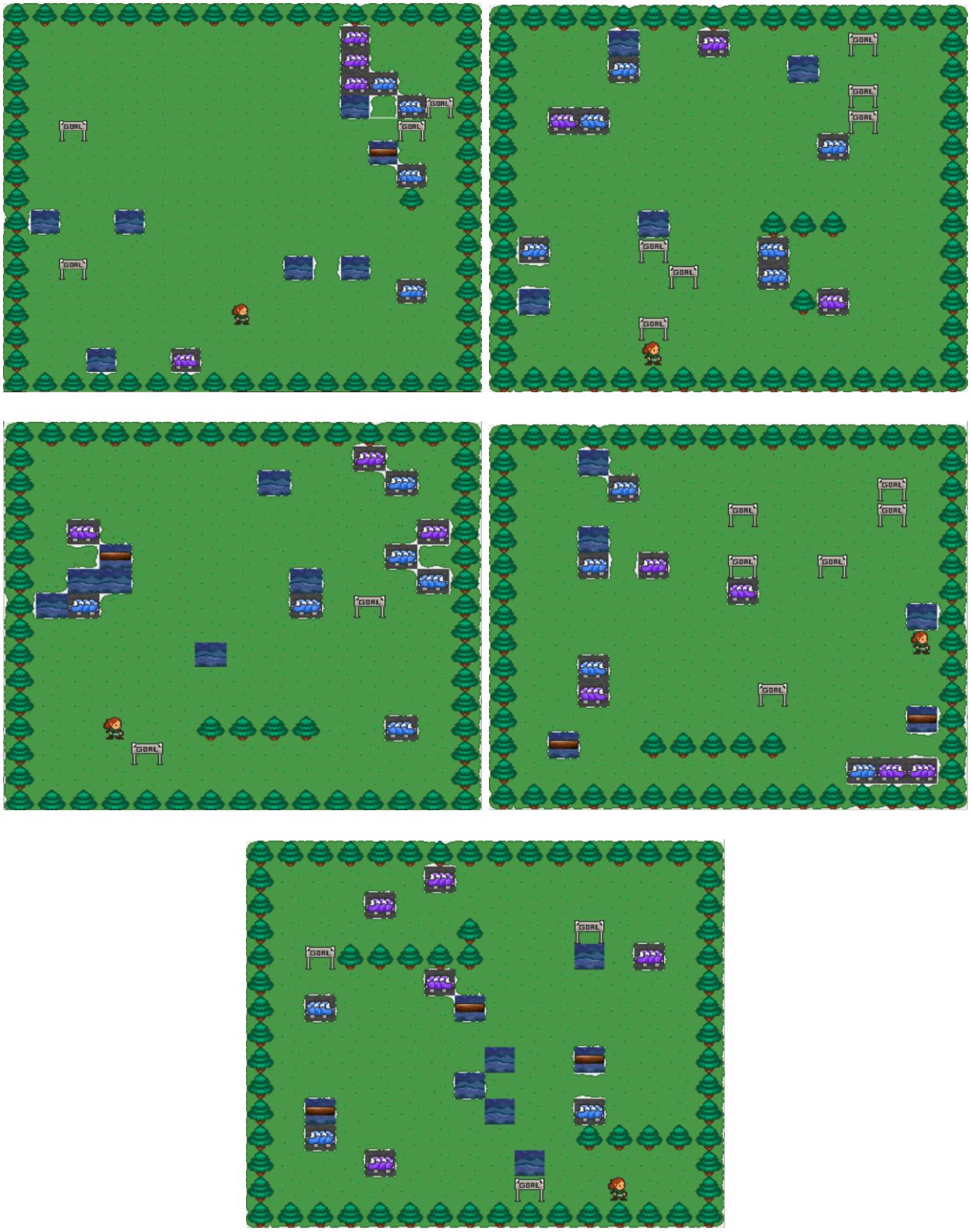
Frogs New Constructive Levels:



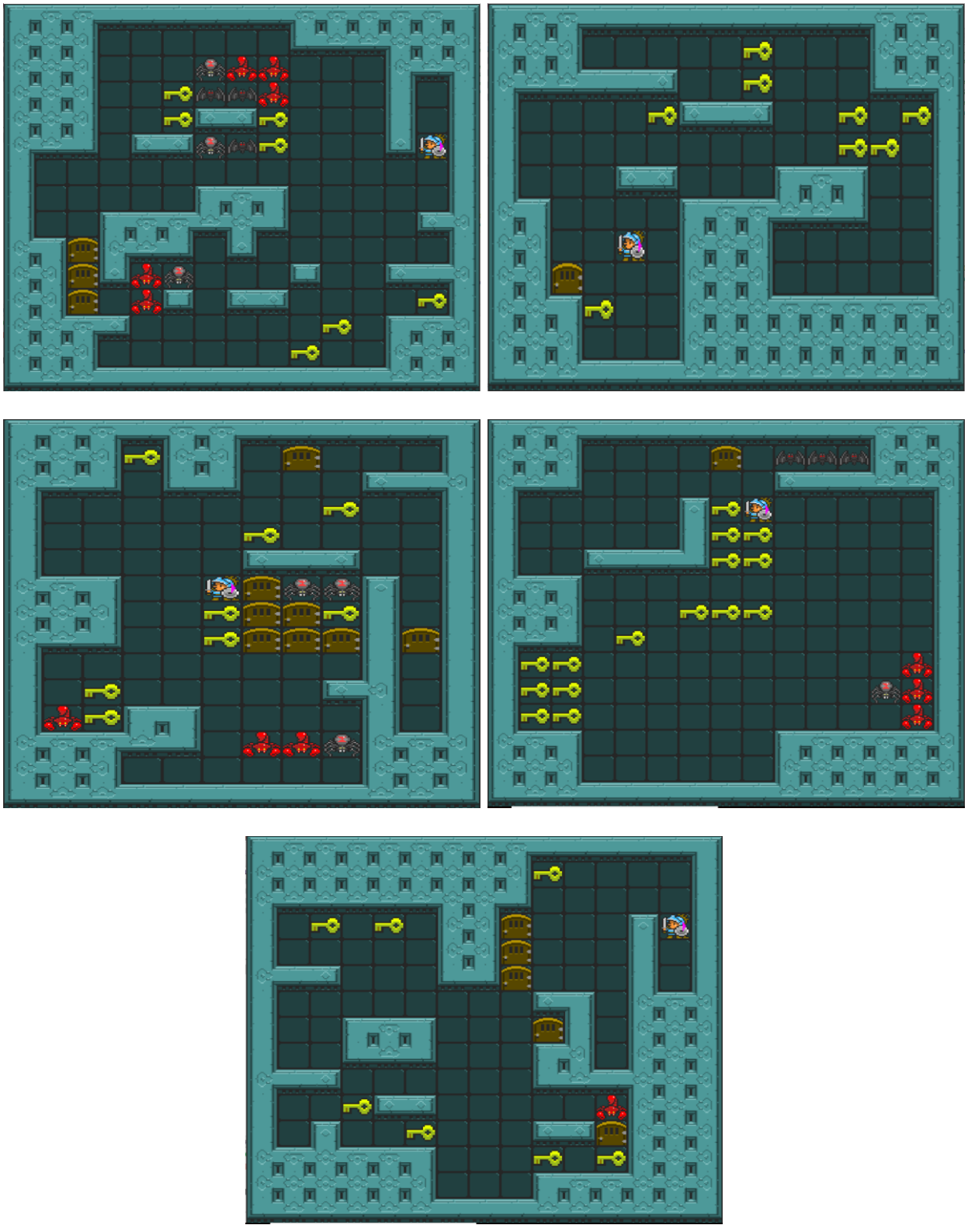
Frogs New Search Levels:



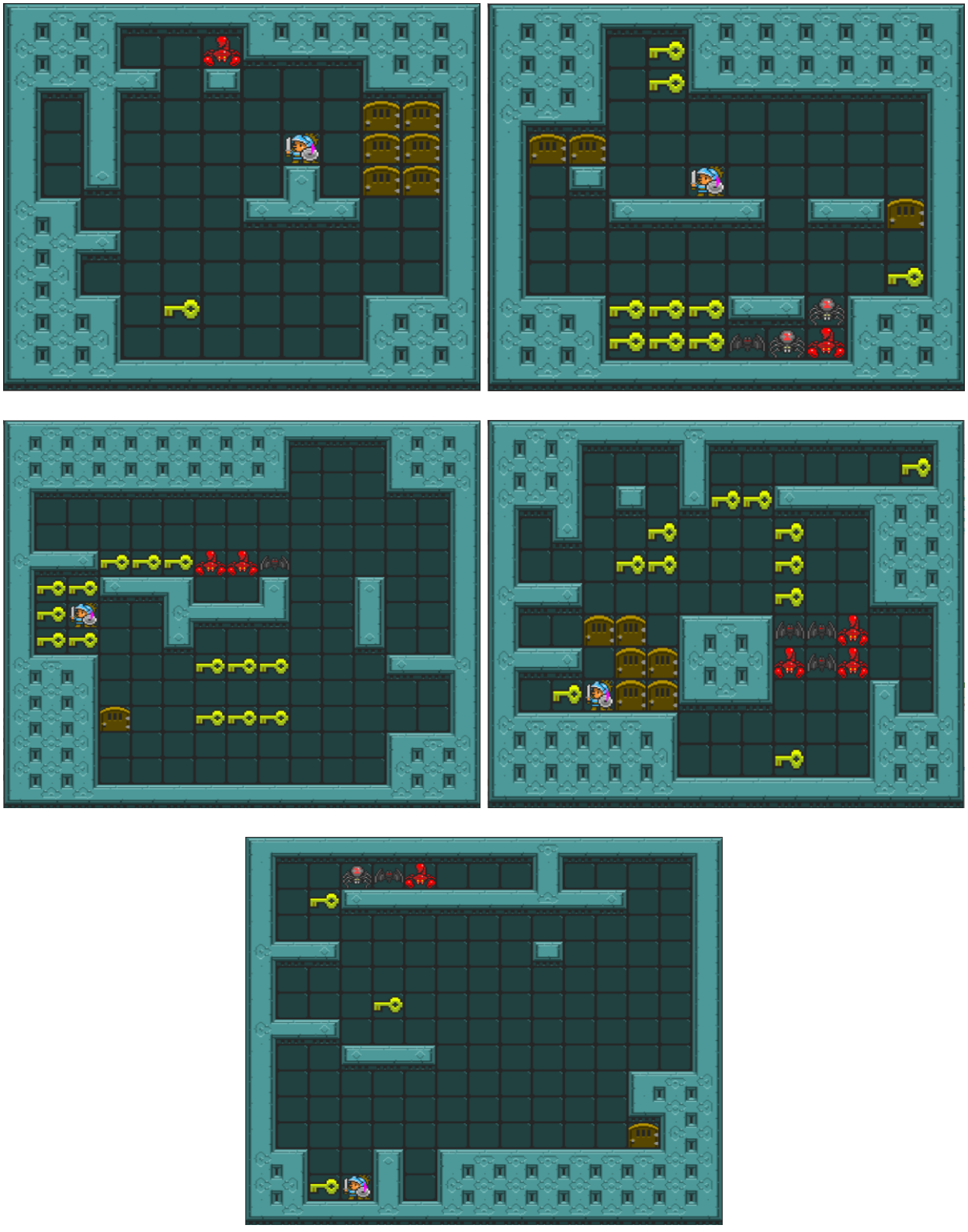
Frogs Prior Search Levels:



Zelda New Constructive Levels:



Zelda New Search Levels:



Zelda Prior Search Levels: