Generating Levels That Teach Mechanics

Michael Cerny Green
mcgreentn@gmail.com
Tandon School of Engineering,
New York University
New York City, NY

Ahmed Khalifa
ahmed.khalifa@nyu.edu
Tandon School of Engineering,
New York University
New York City, NY

Gabriella A. B. Barros
gabbbarros@gmail.com
Tandon School of Engineering,
New York University
New York City, NY

Andy Nealen
andy@nealen.net
Tandon School of Engineering,
New York University
New York City, USA

Julian Togelius
julian@togelius.com
Tandon School of Engineering,
New York University
New York City, NY

ABSTRACT

The automatic generation of game tutorials is a challenging AI problem. While it is possible to generate annotations and instructions that explain to the player how the game is played, this paper focuses on generating a gameplay experience that introduces the player to a game mechanic. It evolves small levels for the Mario AI Framework that can only be beaten by an agent that knows how to perform specific actions in the game. It uses variations of a perfect A* agent that are limited in various ways, such as not being able to jump high or see enemies, to test how failing to do certain actions can stop the player from beating the level.

CCS CONCEPTS

• **Theory of computation → Evolutionary algorithms**; • **Applied computing → Computer games**;

KEYWORDS

Super Mario Bros, Search Based Level Generation, Feasible Infeasible 2-Population

ACM Reference Format:
Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, Andy Nealen, and Julian Togelius. 2018. Generating Levels That Teach Mechanics. In *Foundations of Digital Games 2018 (FDG18), August 7–10, 2018, Malmö, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3235765.3235820

1 INTRODUCTION

The prolific use of games as a testbed for artificial intelligence (AI) has brought forth several roles for AI techniques in this setting, such as player, generator and evaluator [40]. A recently proposed role is AI as a teacher, essentially meaning the use of algorithms to automatically generate game tutorials [15]. Tutorials are one of the most important parts of a game, often being the player's first gameplay experience. It can come in the form of text, demonstrations or even gameplay itself. Commercial games often build the tutorial into a level or a series of levels, as exemplified by *Super Mario Bros* (Nintendo, 1985) (SMB) and *Super Meat Boy* (Team Meat, 2010).

As important as tutorials are, they are also frequently relegated to the end of the development process and overlooked in favor of keeping the product's release date or decreasing expenses [28]. More often than not, this is due to avoid constantly changing the tutorial as game mechanics and features evolve throughout development. Thus, the ability to automatically generate tutorial levels could benefit video game designers and developers, decreasing the cost and time required to build a game.

This paper tackles the challenge of automatically generating game tutorials. We hypothesized that if a perfect agent, which knows all game mechanics, wins a level while an agent that cannot perform one mechanic loses or cannot finish the same level, then that level can be used to teach a player that mechanic. Unlike previous work that focuses on the instructional side of tutorials [2], we focused on creating an experience that will teach the player during gameplay, by posing challenges that they can only overcome when using the mechanics we wanted to teach. We used the Mario AI Framework [17] as our testbed. We also used variations of an A* agent, limited in different ways, to evaluate levels generated by an evolutionary algorithm. The catch, however, is that we want to evolve levels that one limited variant **cannot** win, while the others can. Our objective is to evolve levels that can only be beaten by using the mechanic that limits the agent it was tailored to, e.g. jumping high on a level that a Mario agent which can only do the short jump is unable to win. The following sections describe the background of this research, our methods, experiments and results.

2 RELATED WORK

This section discusses frameworks and research relevant to our work. It starts with a description of the Mario AI framework, followed by a brief background on search based level generation and level generation for the Mario AI framework, and concluding with tutorials and tutorial generation.

FDG18, August 7–10, 2018, Malmö, Sweden

ACM ISBN 978-1-4503-6571-0/18/08.
https://doi.org/10.1145/3235765.3235820

2.1 Mario AI Framework

Infinite Mario Bros. (IMB), developed by Markus Persson [27], is a public domain clone of the 2D platform classic game *Super Mario Bros.*. The gameplay of IMB consists of moving on a two-dimensional sideway level towards a goal. The player can be in one of three possible states: small, big and fire. They can also move left and right, jump, run, and (when on the fire state) shoot fireballs. The player returns to the previous state if they take damage, and dies when taking damage while on the small state or falling down a gap. Unlike the original game, IMB allows for automatic generation of levels.

The Mario AI framework has been a popular benchmark for research on artificial intelligence [17]. Based on the IMB, it has been popular ground for AI competitions [17, 37]. It improved on limitations of IMB's level generator, and several techniques have been applied to automatically create levels [29, 33] or to play levels [17].

2.2 Search Based Level Generation

Evolutionary algorithms (EA) are a type of optimization search inspired by Darwinian evolutionary concepts such as reproduction, fitness, and mutation [38]. Evolution can be used within the realm of games for various purposes, including the generation of levels and game elements within them. Ashlock used evolution to optimize puzzle generation for a given level of difficulty [4]. The fitness function measured solution length which was found using a dynamic programming algorithm. Later, Ashlock et al. developed a system which could parameterize this fitness function into *checkpoint based fitness* [6], allowing substantial control over generated maze properties. Ashlock proceeded to build a system which generated cave-like level maps using evolvable fashion-based cellular automata [5], i.e. stylized cave generation. Ashlock also created a system which decomposes level generation into two parts, a micro evolutionary system which evolves individual tile sections of a level, and an overall macro generation system which evolves placement patterns for the tiles [25].

Khalifa et al. used evolutionary search for general level generation in multiple domains such as General Video Game AI [21] and PuzzleScript [18]. In later work by Khalifa et al. [20], they worked on generating levels for a specific game genre (Bullet Hell genre) using a new hybrid evolutionary search called Constrained MapElites. The levels were generated using automated playing agents with different parameters to mimic various human play-styles. Khalifa et al. [19] also offered a literature review of search based level generation within puzzle games.

2.3 Level Generation for the Mario AI Framework

Horne et al. [16] compiled an evaluative list of all Mario AI generators. The Notch and Parameterized-Notch generators write levels from left to right, adding game elements through probability and performing basic checks to ensure playability [30]. Hopper was written for the Level Generation track of the 2010 Mario AI Championship. Much like Notch and Parameterized-Notch, it also designs levels from left to right, adding game elements through probability. However, these probabilities adapt to player performance, resulting in a dynamic level generator [29]. Launchpad is a rhythm-based level generator that uses design grammars for creating levels within rhythmical constraints [32]. The Occupancy-Regulated Extension generator works by placing small hand-authored chunks together into levels [29]. Each chunk contains an *anchor point* to determine how chunks are placed together. The Pattern-based generator uses evolutionary computation to generate levels by representing levels as *slices* taken from the original *Super Mario Bros* (Nintendo, 1985) [9]. The fitness function counts the number of occurrences of specified sections of slices, or "meso-patterns", with the objective to find as many meso-patterns as possible. The Grammatical Evolution generator uses evolutionary computation together with design grammars. It represents levels as instructions for expanding design patterns. The fitness function measures the number of items in the level and the number of conflicts between the placement of these items.

2.4 Tutorials

Most video games contain tutorials in some way, whether they are ingrained within the gameplay or kept separate from it. Green et al. [15] proposed a non-exhaustive list of tutorial types: Instruction-based, Demonstration-based, and a Well-designed Experience. *Instruction-based* tutorials are textual in nature: A pop-up may appear in front of the player during gameplay describing the next step to take, or a board game may come with a booklet explaining the rules in detail. *Demonstration-based* tutorials take control from the player, such as an non-player character acting out the next step. An example can be found in *The Elder Scrolls: Skyrim* (Bethesda, 2011) when the player first learns the *shout* ability. *The Well-Designed Experience* tutorials are the most complex of the three, where the tutorial is built into the level and gameplay itself and not treated as a separate component. Levels in *Super Meat Boy* (Team Meat, 2010) demonstrate this tutorial type as the player learns new game mechanics and navigation techniques while playing.

Sheri Graner Ray wrote about *knowledge acquisition styles* of players could use to divide tutorials into two categories: *exploratory* and *modeling* [28]. *Exploratory tutorials* have the player learn about something *by* doing it, whereas *modeling tutorials* focus on allowing the player to study how to do something *before* doing it.

Often to better understand games, their mechanics, and to create tutorials to teach them, designers create languages with which to model games. Dan Cook [7] described a *skill atom*: the feedback loop through which a player learns a new skill during gameplay. Figure 1 shows a skill atom to learn how to jump. A skill atom can be divided into four separate elements:

- The *Action* the player performs to learn a new skill. This could involve anything from pressing a button or doing a complex series of actions to accomplish an end goal.
- The *Simulation* of that action in game. The player's action somehow affects the world.
- The *Feedback* from the simulation informs the player of the new state of the game, so they know how their action changed the world.
- The *Modeling* the player now performs within their head, mapping the action they just took to the feedback from the

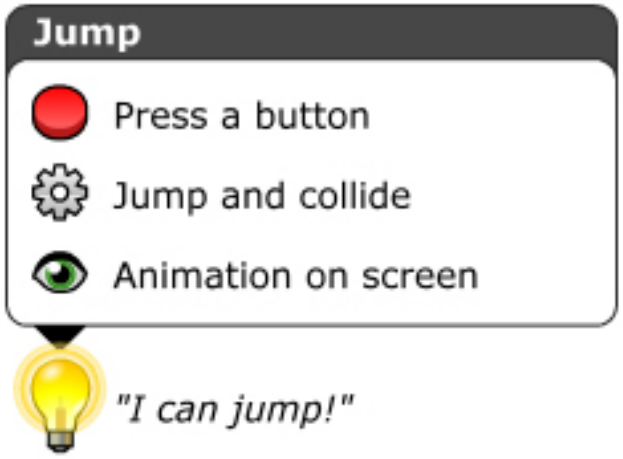


Figure 1: A skill atom for learning how to jump in any generic game, in the order of *action* (button), *simulation* (jump and collide), *feedback* (animation on screen), and *modeling* ("I can jump!")[1]

simulation, e.g. "If I press this button, my character jumps up."

Skill atoms can be linked to other skill atoms to form *skill chains* as shown in Figure 2. Using skill chains, one could model most games that exist.

A similar concept to the skill atom is the strategy ladder. Video games could be represented as the strategy required to beat them. Each step in a strategy ladder corresponds to an addition to the strategy of the previous step that makes a noticeable difference in the strength of that strategy. It has been proposed that the depth of a game can be defined as the length of its longest strategy ladder [23]. Then, by reading or interacting with a strategy ladder, one should be able to understand, theoretically, the requirements of a game.

The AtDelfi system uses a graph-based representation to model mechanics in a game [14]. Object nodes, condition nodes, and action nodes are used in unison to describe player abilities, object collisions, scoring, and time-based mechanics. The system creates this graph dynamically after reading a game's rules, which are formulated using the Video Game Description Language (VGDL) [13]. VGDL is a high-level language for 2D arcade games, allowing not only the quick development for these games but also analysis of game rules and events. With this graph, the system can then generate written and visual tutorials demonstrating ways to win, lose, and gain points in the game.

2.5 Tutorial Generation

Previous work has been done in the area of tutorial/instruction generation, such as *TutorialPlan* [24], which generates text and image instructions for new users of AutoCAD. De Messentier Silva et al [10–12] used various search methods to create effective beginner strategies for Blackjack and Poker. Alexander et al. [1] turned *Minecraft* (Mojang 2009) mechanics into action graphs representing the player experience, and created quests and achievements based off those actions. *Game-O-Matic* [39] generates arcade style games

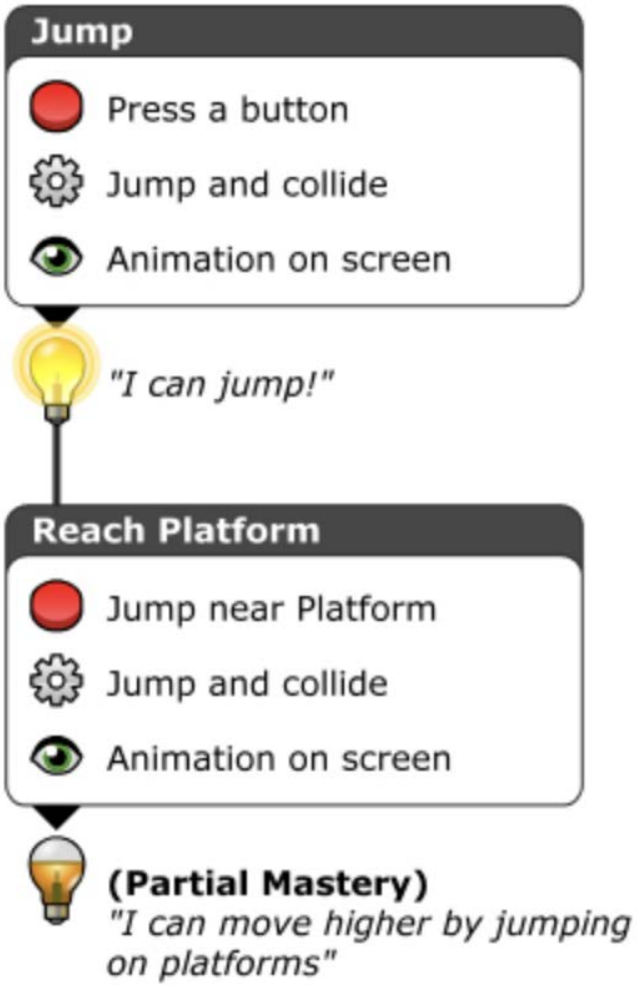
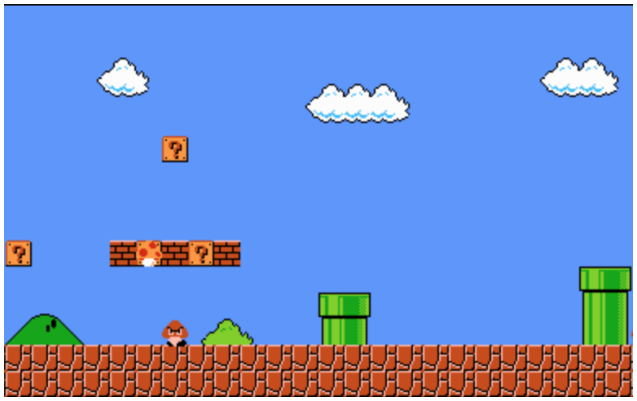


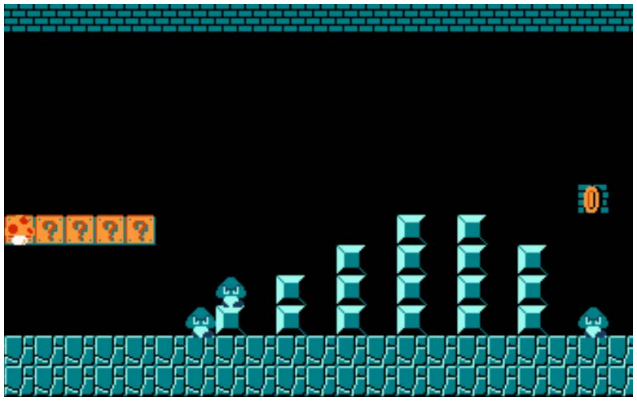
Figure 2: A chain of skill atoms demonstrating the action process through which a player learns platform jumping during gameplay

and instructions using a story-based concept-map. It generates a tutorial page after a game's creation which explains who the player will control, how to control them, and winning/losing conditions, by using the concept-map and relationships between objects within it. *Mechanic Miner* can automatically discover new mechanics using a reflection-driven generation technique using game simulation, and then invert the simulation to produce levels for those discovered mechanics [8].

Mappy is a system which takes a Nintendo Entertainment System game and a sequence of buttons presses as input to generate an approximation of a linked map of rooms [26]. *Mappy* attempts to create map understanding from movement mechanics. This is similar to what Summerville et al. created as a part of the *Gemini* system, a logic program that performs static reasoning over game specifications in order to find meaning [34].



(a) Level 1-1



(b) Level 1-2

Figure 3: The first two levels from *Super Mario Bros*. The levels show the difference between overground levels (3a) and underground levels (3b).

3 METHODS

Our system evolves a single screen that teaches a specific mechanic for the Mario AI framework, utilizing several AI playthroughs to find screens that an AI that has full game knowledge can beat, but a limited AI cannot.

In this work, a Mario level consists of a group of scenes, where each scene delivers a specific experience, such as a single jump, killing an enemy, etc [3]. Each scene is represented as a group of vertical slices sampled from the original *Super Mario Bros* (SMB), much like in Dahlskog and Togelius' work [9]. Each slice has a fixed width and height, equal to 1 and 14 respectively. We collected slices from the levels provided in the Video Game Level Corpus (VGLC) [35], excluding underground levels as they differ structurally from the rest of the levels. Figure 3 shows part of the the first two levels from SMB: Level 1-1 and Level 1-2. Underground levels (Figure 3b) have a ceiling on the top of the level. Thus, combining different slices from both levels would generate an inconsistent scene. Additionally, having a ceiling causes problems with the Mario AI framework: the Mario AI framework spawns Mario at the highest solid tile at the beginning of the scene. In an underground level, the framework would spawn Mario on the ceiling instead of on the floor.

Table 1: A* Agent Limitations

Agent	Limitation
B A*	No limitation. Perfect Agent.
LJ A*	Limited jumping capabilities. Cannot jump 'high.'
EB A*	Blind to all enemies. Unable to see enemy collisions.
NR A*	Not able to run. Indirectly limits 'long jump' capability.

3.1 Evolutionary Algorithm

We used the Feasible Infeasible 2-Population (FI-2Pop) genetic algorithm [22] to generate scenes. FI-2Pop is an evolutionary algorithm that uses two populations: one being feasible and the other infeasible. The infeasible population aims at improving infeasible solutions to a certain threshold, when they become feasible and are transfered to the feasible population. The feasible population, on the other hand, aims at improving the quality of feasible chromosomes, If one becomes infeasible, it is then relocated to the infeasible population. After evolving solutions for several generations, our system outputs the scene with the highest fitness.

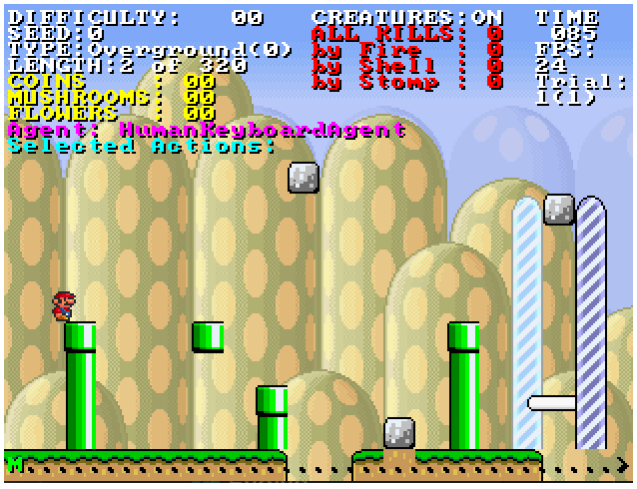
For the purposes of this work, we assumed that a scene is equivalent to one screen in the Mario AI framework, which consists of 18 slices. Therefore, our chromosome consists of a group of 18 vertical slices. We used a two-point crossover and mutation as operators: a two-point crossover switches a group of slices between the two points, allowing the evolutionary algorithm to swap any length from a single slice to the whole scene. For mutation, the algorithm replaces one slice with a random one from the sampled slices.

3.2 Evaluating Scenes

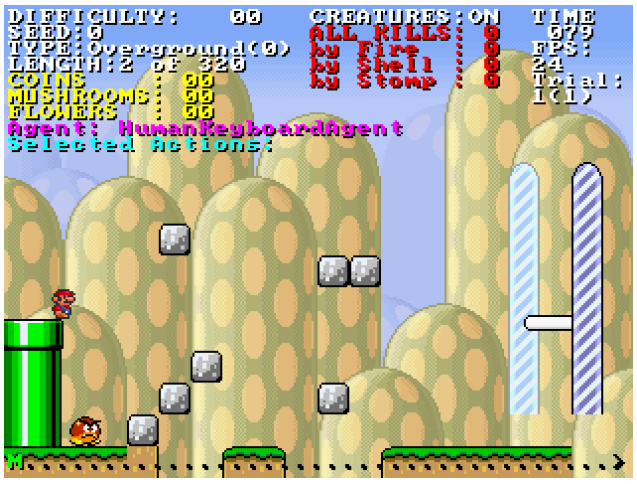
We used two different fitness functions, one for the infeasible population and one for the feasible population. These are described in detail below.

Infeasible Fitness: The fitness function of the infeasible population regards only the levels aesthetic. One of the impassable obstacles in *Super Mario Bros* is a green pipe. It can have any height and it takes two tiles in width. Since the chromosome consists of a group of vertical slices where each slice is 1 tile in length, there is a high chance that a half pipe might appear in the scene. The infeasible fitness function makes sure that all the pipe are two tiles wide. Figure 4a shows a chromosome with an infeasible fitness equal to 0, where the pipe parts do not connect correctly. Figure 4b shows a chromosome with an infeasible fitness equal to 1, where all the pipe pieces connect in pairs. Chromosomes that don't have any pipes are considered feasible chromosomes with infeasible fitness equal to 1.

Feasible Fitness: Our system uses agent performance data to gauge the fitness of a level. One of these agents is an A* agent designed by Robin Baumgarten for the first Mario AI competition [36], capable of playing a level almost flawlessly. The heuristic this agent is based on is the time it would take for Mario to move to the end of the level (i.e. the rightmost side of the map), which is admissible because it assumes that Mario is always running at maximum speed. The other agents are variations of Baumgarten's A* agent, limited in different ways. These limitation are summarized in Table 1.



(a) Infeasible Fitness = 0.0



(b) Infeasible Fitness = 1.0

Figure 4: Two generated chromosomes, one from the infeasible population and one from the feasible population. The first chromosome in (4a) has infeasible fitness equal to 0.0, while the second chromosome in (4b) has infeasible fitness equal to 1.0, thus belonging to the feasible population.

Our evolutionary algorithm takes one of the limited agents and the perfect agent to evaluate a level, comparing their success and/or failure. We hypothesized that a level requires the use of an specific mechanic that a limited agent lacks if the perfect agent wins the level but the limited agent fails. Therefore, our fitness function maximized the distance between the limited agent's failure and the perfect agent's success.

4 RESULTS

We ran three experiments, each with a population of 100 chromosomes evolved for 120 generations. The crossover rate was fixed to 70% and the mutation rate was fixed to 30%, and we used rank selection. Rank selection gives each chromosome a rank based on its fitness and then select chromosomes proportionally towards their rank, i.e. higher rank indicate a higher probability of being chosen. We also used elitism of size of 1 between generations to keep the best chromosome.

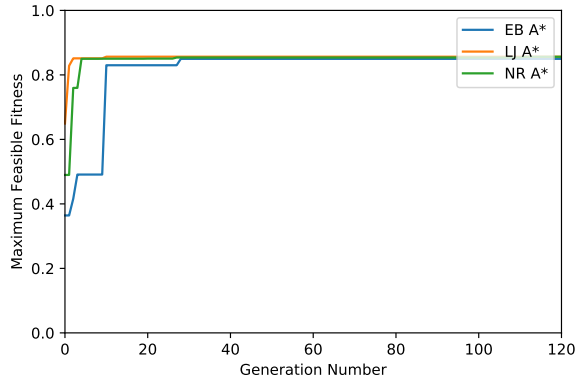


Figure 5: Maximum feasible fitness increases throughout generations

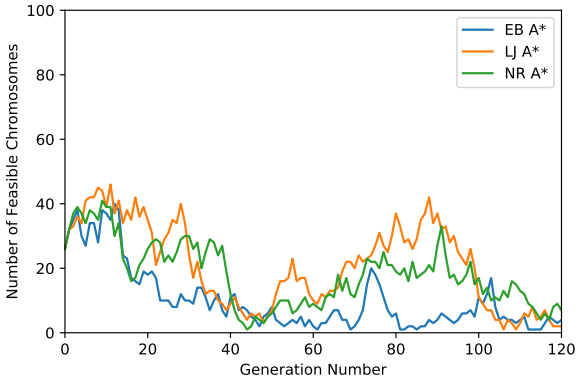


Figure 6: Number of feasible chromosomes throughout generations.

Figure 5 shows the maximum feasible fitness over generations for each different evolution. In every experiment, there was a quick increase in the fitness function in the first few iterations, reaching the highest found value of 0.8 after approximately 15 generations, with no further improvement. We believe that 0.8 is the highest fitness our system can achieve, reflecting that the perfect agent finished the scene (i.e. the scene was 100% traversed) while the limited agent only traversed 20% of the scene before it died or got stuck.

Figure 6 shows the number of feasible chromosomes throughout the 120 generations. Surprisingly, the numbers vary as opposed to only increasing as generations pass. It is possible that, once the evolution finds the chromosomes with the highest fitnesses in the first 20 generations and shows the highest amount of feasible chromosomes, it becomes difficult for the system to find better

ones without separating pipes in feasible scenes, thus making them infeasible.

Figure 7 displays the evolved scenes from the three combinations of perfect and limited agents. Each level was played by the limited agent used to create it. In each case, the limited agent in question failed to beat the level, thus verifying that the specific mechanic was needed. Each column shows three scenes evolved with one of our different limited agents, and each row shows three evolved scenes that have high (top) to low (bottom) fitness. The last row shows feasible chromosomes with fitness equal to 0.0, meaning that both agents can beat the levels.

It is possible to notice that scenes with higher fitness focuses more on the intended experience. Figure 7a requires high jumps at the first section that cannot be overcome by the Limited Jump agent, Figure 7b has a high amount of enemies in when compared to the other images, and Figure 7c has a wall jump at the beginning that can only be climbed while holding the run button. A member of our team also played the highest fitness scenes to get a subjective human-evaluation. In each case, we observed that they needed to have knowledge of and use the specific mechanic for the given level, although with varying degrees of success. The following subsections further analyze the top three maps shown in Figure 7.

4.1 LJ Agent Scene

Figure 7a shows the level evolved with the LJ Agent, wherein the player had to perform two high jumps to beat the level. If they wanted to test themselves, a third high jump could be done to acquire a coin right before the goal. A mystery block was also included, but wasn't necessary to hit to complete the level.

4.2 EB Agent Scene

For the map evolved with the EB Agent shown in Figure 7b, the player faced three enemies: two goombas and one red turtle. It is an interesting example, as it shows a level that only scores a high fitness function due to the level of proficiency of the A* agent. Due to the rules of the game engine, enemies are capable of falling off cliffs and ledges. Thus, unless the player immediately moved to the right, they would never actually encounter these enemies, as the enemies would fall of the ledge at the very beginning of the game. After observing the EB agent playthrough, we realized that the super-human reflexes of the agent allowed it to die to these enemies and thus fail the level anyway.

4.3 NR Agent Scene

Finally, in the level evolved for the NR Agent (Figure 7c), the player faced an extremely high wall at the very beginning of the map. In order to climb it, the player would have to run at the wall and wall jump right around the dirt tile. This requires incredible precision, which a novice player would probably not have. The following gap requires the player to jump from the first column to the second while running, again requiring precision not found among beginners. If the player failed this jump, they would fall into the gap between the two walls and would have to climb the first wall again.

5 DISCUSSION & CONCLUSION

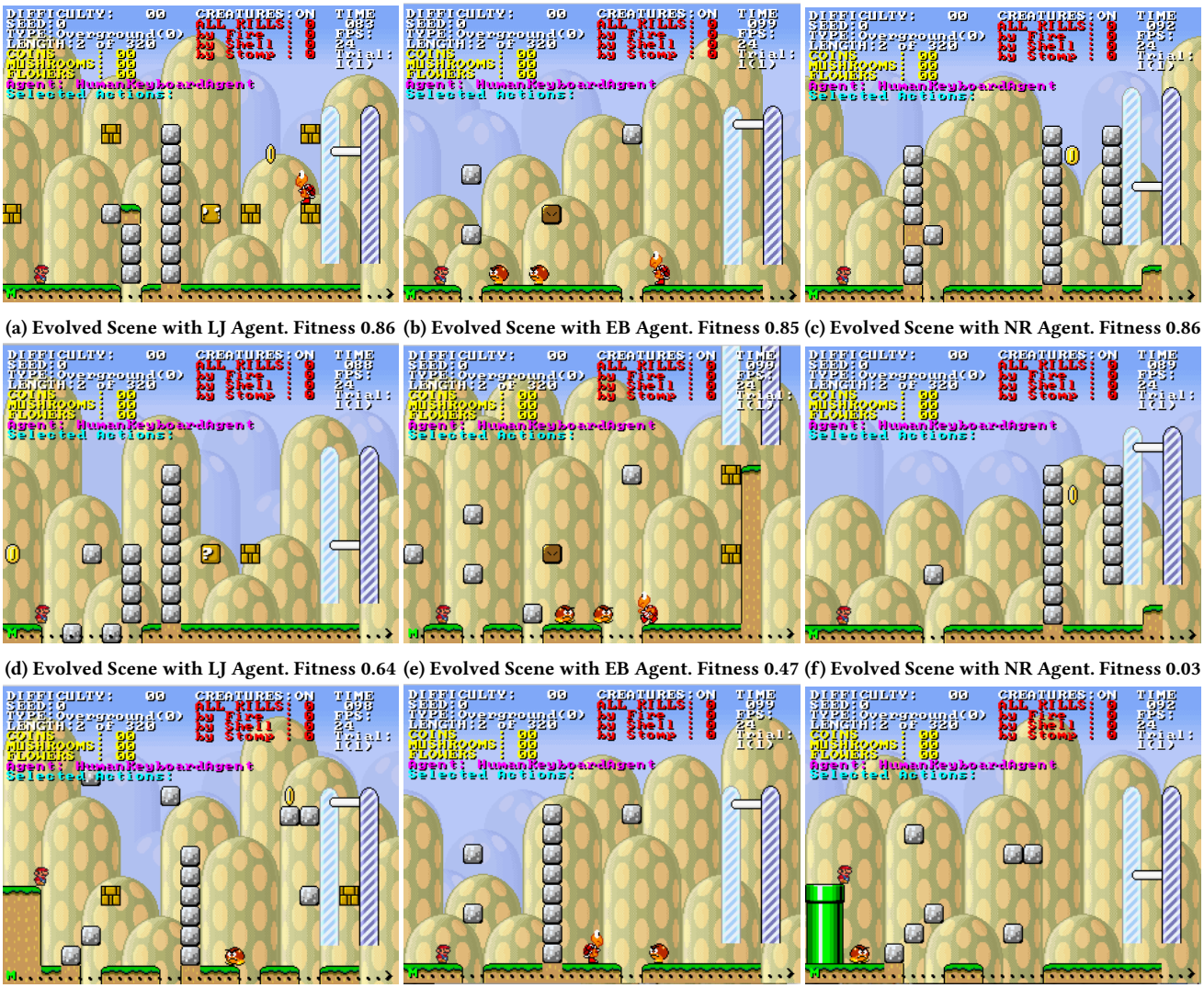
This paper evolved small levels (scenes) for the Mario AI framework that teach specific mechanics. It used a feasible infeasible 2-population evolutionary algorithm, which uses multiple automated playthroughs as the fitness function. We hypothesized that finding levels where a perfect agent (i.e. an agent that has full knowledge of all the game mechanics) wins and a limited agent (i.e. an agent that lacks information about a certain mechanic) dies or gets stuck can teach an specific mechanic. We used three different variants of Robin Baumgarten A* algorithm to communicate three different mechanics: Long Jumps, Stomping Enemies, and Running. The evolutionary algorithm was able to find high fitness scenes in its first 15 generations. The best evolved level in each experiment was subjectively playtested and only possible to beat using a specific mechanic that the agent was missing. However, one of the drawbacks of using Robin Baumgarten A* algorithm is that it has superhuman reflexes, thus the evolved levels require very precise movements that aren't easy to achieve by a novice player.

We originally set out to explore idea of discovering sections of maps which required the use (and therefore the mastery) of a mechanic to beat them. To that end, we succeeded. Each of the maps we generated demonstrated that a player, AI or human, would have to use the specified mechanic to win. However, the AI had the added benefit of pixel perfect gameplay and inhuman reflexes. Because of this, these maps are too difficult for a human to play and therefore are inadequate for teaching a human these mechanics.

One such example of this is in the EB Agent scene, where all enemies encountered would almost immediately fall off the map at the game's beginning. Unless the player moved to the right, they would never encounter these enemies, and they would therefore never learn what kind of mechanics interacting with these enemies represent. The NR Agent scene, which contains wall-jumps necessary to beat the level, also exemplifies this problem. A more advanced player might be able to perform a wall jump, but a novice player would most likely not be able to do this. We can therefore conclude from this that our experiment, while producing maps that required desired mechanics from an AI viewpoint, did not take human perspective into account.

This work is a stepping stone towards evolving full levels that can teach players the different game mechanics, both in *Super Mario Bros* and other games. A next approach would use an evolutionary algorithm to arrange the evolved scenes to have a full-length game level, similar to Level 1-1 in Super Mario Bros. The scenes have to be arranged in increasing difficulty order, as to not overwhelm new players. Another improvement would be to use human-like agents instead of the perfect A* agent, in order to generate more human like scenes that don't require superhuman reflexes to beat. We also intend on improving generated scenes by running another evolutionary algorithm that tries to simplify the generated scenes, by decreasing the number of used blocks, without decreasing the fitness of the scene as the generated scenes have multiple blocks that do not have a purpose in the playthrough.

Another potential step forward would be to move away from an evolutionary algorithm and use a constraint solving approach. Smith et al's *Refraction* (Center for Game Science at the University of Washington, 2010) level generators use answer set programming



(a) Evolved Scene with LJ Agent. Fitness 0.86 (b) Evolved Scene with EB Agent. Fitness 0.85 (c) Evolved Scene with NR Agent. Fitness 0.86

(d) Evolved Scene with LJ Agent. Fitness 0.64 (e) Evolved Scene with EB Agent. Fitness 0.47 (f) Evolved Scene with NR Agent. Fitness 0.03

(g) Evolved Scene with LJ Agent. Fitness 0.0 (h) Evolved Scene with EB Agent. Fitness 0.0 (i) Evolved Scene with NR Agent. Fitness 0.0

Figure 7: Evolved Scenes Using Perfect vs Limited Agents

to easily control level features [31]. Refraction is an educational puzzle game in which players arrange devices on a grid to construct networks of laser beams. By requiring the player to use beams of different power levels, the game aims to teach mathematical skills. Similar to Smith's work, it might be possible to add mechanics as constraints within a generator in order to require the use of those mechanics to win a level.

ACKNOWLEDGMENTS

Gabriella Barros acknowledges financial support from CAPES and Science Without Borders program, BEX 1372713-3, as well as an NYU Tandon School of Engineering Fellowship. Ahmed Khalifa acknowledges the financial support from NSF grant (Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection."). Michael Cerny Green acknowledges the financial support of the GAANN program.

REFERENCES

[1] Ryan Alexander and Chris Martens. 2017. Deriving Quests from Open World Mechanics. *arXiv preprint arXiv:1705.00341* (2017).
[2] Anonymouns. Anonymouns. Anonymouns.
[3] Anna Anthropy and Naomi Clark. 2014. *A game design vocabulary: Exploring the foundational principles behind good game design.* Pearson Education.
[4] Daniel Ashlock. 2010. Automatic generation of game elements via evolution. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on.* IEEE, 289–296.
[5] Daniel Ashlock. 2015. Evolvable fashion-based cellular automata for generating cavern systems. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on.* IEEE, 306–313.
[6] Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 260–273.
[7] Dan Cook. 2007. The chemistry of game design.

[8] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *European Conference on the Applications of Evolutionary Computation*. Springer, 284–293.

[9] Steve Dahlskog and Julian Togelius. 2013. Patterns as objectives for level generation. (2013).

[10] Fernando de Mesentier Silva, Aaron Isaksen, Julian Togelius, and Andy Nealen. 2016. Generating heuristics for novice players. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*. IEEE, 1–8.

[11] Fernando de Mesentier Silva, Julian Togelius, Frank Lantz, and Andy Nealen. 2018. Generating Beginner Heuristics for Simple Texas HoldâĂŹem. In *Proceedings of The Genetic and Evolutionary Computation Conference*. ACM.

[12] Fernando de Mesentier Silva, Julian Togelius, Frank Lantz, and Andy Nealen. 2018. Generating Novice Heuristics for Post-Flop Poker. In *Computational Intelligence and Games (CIG)*. IEEE.

[13] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. 2013. Towards a video game description language. In *Dagstuhl Follow-Ups*, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[14] Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, Tiago Machado, Julian Togelius, and Andy Nealen. 2018. AtDELFI: Automatically Designing Legible, Full Instructions For Games.

[15] Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, and Julian Togelius. 2017. "Press Space To Fire": Automatic Video Game Tutorial Generation.

[16] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. 2014. A comparative evaluation of procedural level generators in the mario ai framework. Society for the Advancement of the Science of Digital Games.

[17] Sergey Karakovskiy and Julian Togelius. 2012. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 55–67.

[18] Ahmed Khalifa and Magda Fayek. 2015. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*.

[19] Ahmed Khalifa and Magda Fayek. 2015. Literature Review of Procedural Content Generation in Puzzle Games. http://www.akhalifa.com/documents/LiteratureReviewPCG.pdf.

[20] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet Hell Generation through Constrained Map-Elites. In *Proceedings of The Genetic and Evolutionary Computation Conference*. ACM.

[21] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 253–259.

[22] Steven Orla Kimbrough, Gary J Koehler, Ming Lu, and David Harlan Wood. 2008. On a Feasible–Infeasible Two-Population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research* 190, 2 (2008), 310–327.

[23] Frank Lantz, Aaron Isaksen, Alexander Jaffe, Andy Nealen, and Julian Togelius. 2017. Depth in strategic games. *under review*.

[24] Wei Li, Yuanlin Zhang, and George Fitzmaurice. 2013. TutorialPlan: automated tutorial generation from CAD drawings. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[25] Cameron McGuinness and Daniel Ashlock. 2011. Decomposing the level generation problem with tiles. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 849–856.

[26] Joseph Osborn, Adam Summerville, and Michael Mateas. 2017. Automatic mapping of NES games with mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. ACM, 78.

[27] Marcus Persson. 2008. Infinite mario bros. *Online Game). Last Accessed: December* 11 (2008).

[28] Sheri Graner Ray. 2010. Tutorials: Learning to play. http://www.gamasutra.com/view/feature/134531/tutorials_learning_to_play.php.

[29] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, et al. 2011. The 2010 Mario AI championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 4 (2011), 332–347.

[30] Noor Shaker, Georgios N Yannakakis, and Julian Togelius. 2011. Feature analysis for modeling game content quality. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, 126–133.

[31] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*. ACM, 156–163.

[32] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. 2011. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on computational intelligence and AI in games* 3, 1, 1–16.

[33] Nathan Sorenson and Philippe Pasquier. 2010. Towards a generic framework for automated video game level creation. In *European Conference on the Applications of Evolutionary Computation*. Springer, 131–140.

[34] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph Carter Osborn, Noah Wardrip-Fruin, and Arnav Jhala. 2017. From Mechanics to Meaning. *IEEE Transactions on Computational Intelligence and AI in Games*.

[35] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. 2016. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487* (2016).

[36] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. 2010. The 2009 mario ai competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE, 1–8.

[37] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. 2013. The mario ai championship 2009-2012. *AI Magazine* 34, 3 (2013), 89–92.

[38] Julian Togelius, Noor Shaker, and Mark J. Nelson. 2016. The search-based approach. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer, 17–30.

[39] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. 2012. Game-O-Matic: Generating Videogames that Represent Ideas.. In *PCG@ FDG*. 11–1.

[40] Georgios N. Yannakakis and Julian Togelius. 2018. *Artificial Intelligence and Games*. Springer. http://gameaibook.org.