Writeup

Introduction:

The assignment tasked us with developing a program to generate engaging and playable Super Mario Bros. levels using evolutionary algorithms. Our objective was to explore two different encodings of Mario levels and optimize the fitness functions, crossover, and mutation operators to produce diverse and interesting levels. Throughout the project, our aim was to balance traditional gameplay elements, such as level complexity and playability, and more abstract considerations, such as aesthetic appeal and novelty.

Our approach involved modifying the provided codebase, specifically focusing on the ga.py. By implementing custom selection strategies, crossover techniques, and mutation operators for both the Grid and Design Element encodings, we aimed to improve the efficiency and effectiveness of the evolutionary algorithm in generating high-quality Mario levels. Additionally, we explored various fitness metrics and fine-tuned their weights better to capture the desired characteristics of the generated levels.

In this write-up, we detail the changes made to the codebase, the rationale behind our design decisions, and the results of our experimentation. We evaluate the quality of the generated Mario levels and highlight our favorite creations, providing insights into the level-generation process. Through this exploration, we aimed to gain a deeper understanding of evolutionary algorithms and their application in game-level design.

Changes to the Provided Code:

Significant enhancements were introduced to key components of the genetic algorithm, notably the mutation, generate_children, and generate_selection functions, aimed at refining the process of generating Mario levels. The mutation function underwent a substantial overhaul to introduce weighted probabilities for mutating different tile types, allowing for a more controlled and targeted mutation process. By assigning specific mutation probabilities to each tile type, the algorithm could selectively mutate tiles based on their importance in level design. Additionally, various mutation operators were implemented, including random swapping, two-way swapping, becoming a random tile, and becoming an air tile. These operators introduced diversity and novelty into the generated levels, ensuring that mutated individuals retained structural integrity and playability. The mutation rate in the provided code is determined by various weights assigned to different tile types and actions. These weights influence the likelihood of a specific mutation occurring for a given tile type. For example, constants like EMPTY_WEIGHT, WALL_WEIGHT, COIN_BLOCK_WEIGHT, etc., represent the probability of mutating a particular tile.

In the generate_children function, a selective single-point crossover between parent individuals was implemented to improve the diversity and quality of offspring. Instead of blindly selecting

between parent genomes during crossover, specific criteria were applied to guide the selection process. Heuristic constraints were introduced to prevent undesirable level configurations, such as pipes floating in the air or blocks stacking improperly. These constraints ensured that generated children maintained structural integrity and playability, contributing to the overall quality of the generated levels.

Furthermore, the generate_selection function underwent a significant transformation, transitioning from a simple roulette selection to a more sophisticated combination of roulette and tournament selection mechanisms. Roulette selection prioritized individuals based on their fitness scores, with probabilities proportional to their fitness values, introducing a degree of randomness while favoring fitter individuals for reproduction. Tournament selection complemented roulette selection by involving the random selection of individuals for smaller-scale competitions to determine reproductive candidates, mitigating the influence of noise and outliers in fitness scores. Additionally, constraint-based filtering was applied to filter out individuals with zero-length genomes, ensuring that only viable candidates were considered for reproduction. These refinements collectively aimed to improve the efficiency and effectiveness of the genetic algorithm in generating diverse and engaging Mario levels, ultimately leading to the creation of more playable and enjoyable gaming experiences.

In the mutation function, several enhancements were introduced to optimize the process of mutating individual genomes. The traditional approach of uniformly random mutation was replaced with a weighted mutation strategy, where different tile types were assigned specific mutation probabilities based on their significance in level design. This weighted approach allowed for a more controlled and targeted mutation process, ensuring that critical elements such as walls, enemies, and platforms were preserved while still introducing novelty and diversity into the generated levels. Additionally, various mutation operators were implemented to introduce further complexity and variation into the mutation process. These operators included random swapping, two-way swapping, becoming a random tile, and becoming an air tile. Each operator was carefully designed to balance exploration and exploitation, allowing for the generation of diverse yet structurally sound Mario levels. By combining weighted mutation probabilities with diverse mutation operators, the mutation function played a crucial role in enhancing the evolutionary process and driving the generation of high-quality Mario levels.

Several changes have also been made to Individual_DE, aiming to refine the genetic algorithm for level generation in a game environment.

Firstly, adjustments have been applied to the fitness function. The weights assigned to specific metrics, such as meaningfulJumpVariance and linearity, have been altered to potentially better reflect their importance in evaluating level quality. For instance, meaningfulJumpVariance now

carries a higher weight, indicating its increased significance in determining fitness, while the weight of linearity has been reduced, suggesting a decreased impact on the overall fitness score.

Furthermore, penalties within the fitness function have been recalibrated. Notably, the penalty for the presence of excessive stairs ("6_stairs") has been intensified, shifting from a previous penalty value of -2 to a higher penalty of -3. This modification suggests a stronger discouragement against generating levels with an abundance of stairs, emphasizing the aesthetic and gameplay considerations.

The mutate() function has undergone significant refactoring to enhance modularity and readability. Introducing a dictionary named mutation_strategies, each design element type is now mapped to a corresponding mutation strategy. This restructuring streamlines the mutation process, making it more organized and easier to extend with additional mutation strategies in the future. Additionally, certain mutation strategies have been adjusted to refine the mutation behavior. For example, mutations for "4_block" and "5_qblock" now primarily modify the y-coordinate instead of the x-coordinate, potentially resulting in more varied and interesting level designs.

Despite these adjustments, the core logic of the generate_children() function remains unchanged. It still randomly selects crossover points from parent individuals and combines their genomes to produce offspring. This approach to generating children ensures diversity in the population while preserving desirable traits from the parent individuals.

Overall, these changes collectively aim to fine-tune the genetic algorithm, potentially leading to improved level generation by prioritizing key metrics, refining mutation strategies, and reinforcing penalties for undesirable design elements. Such adjustments reflect an ongoing effort to optimize the algorithm's performance and enhance the quality of generated game levels.

Rationale and Design Decisions:
Firstly, in the mutation function for grid, a weighted mutation strategy was implemented to ensure that different tile types were mutated with varying probabilities based on their importance in level design. Each tile type was assigned a specific mutation probability, such as EMPTY_WEIGHT, WALL_WEIGHT, COIN_BLOCK_WEIGHT, etc. This approach allowed for a more controlled mutation process, preventing uniform randomness and ensuring that critical elements like walls, enemies, and platforms were preserved while introducing novelty and diversity into the generated levels.

In the generate_children function for grid, a set of specific heuristics and logical constraints were applied to guide the crossover process and ensure the structural coherence of the generated Mario

levels. The process began by selectively inheriting tiles from the second parent only if they did not compromise the integrity of the level. For example, tiles representing empty spaces were included in the offspring with a probability of 50%, balancing diversity and preservation of essential level elements like walls and platforms.

Furthermore, the crossover strategy varied based on the location of tiles within the level grid, distinguishing between sky and ground sections. Additional constraints were applied to tiles in the sky to prevent floating platforms or disjointed pipes. For instance, enemies were inherited only if positioned on solid ground, and pipes were included if properly supported, enhancing level of realism and playability.

To maintain coherence, constraints were imposed to prevent unrealistic stacking of ground tiles, particularly in platform sections. Offspring inherited ground tiles from the first parent with a higher probability if consecutive ground tiles were present in both parents, reducing the likelihood of excessive stacking and preserving level balance.

Moreover, pipe placement was optimized to ensure correct alignment and structure. Pipes were inherited from the second parent only if adjacent to ground tiles and forming a continuous structure. Pipe tops were aligned with corresponding segments, contributing to navigability and aesthetic appeal.

Lastly, in the generate_successors function, the selection process for generating successor populations was refined using a combination of roulette selection and tournament selection methods. Roulette selection was initially employed to create a candidate pool of individuals based on their fitness values, which were then subjected to tournament selection to determine the fittest individuals for generating offspring. Furthermore, individuals with empty genomes were filtered out to prevent their inclusion in the successor population, thereby optimizing the selection process and improving overall performance.

Overall, these modifications to the mutation, crossover, and selection processes played a crucial role in enhancing the efficiency, diversity, and quality of the Mario level generation using the genetic algorithm approach. By incorporating weighted mutation, guided crossover, and refined selection strategies, the GA was able to produce more coherent, challenging, and engaging Mario levels that closely adhered to predefined design constraints and objectives.

Experimentation and Results:

The experimentation phase of the level generation process involved a comprehensive approach to optimize the quality, diversity, and computational efficiency of the genetic algorithm (GA). Parameter tuning played a crucial role, with extensive adjustments made to parameters such as population size, mutation rates, crossover strategies, and selection mechanisms. This iterative

process aimed to identify the optimal parameter settings that strike a balance between level quality and diversity while minimizing computational overhead. Heuristic evaluation was another key aspect, focusing on assessing the effectiveness of implemented heuristics in guiding crossover and mutation processes. Various heuristic adjustments, including weighting tile selection probabilities and introducing additional constraints, were explored to enhance the coherence and complexity of generated levels.

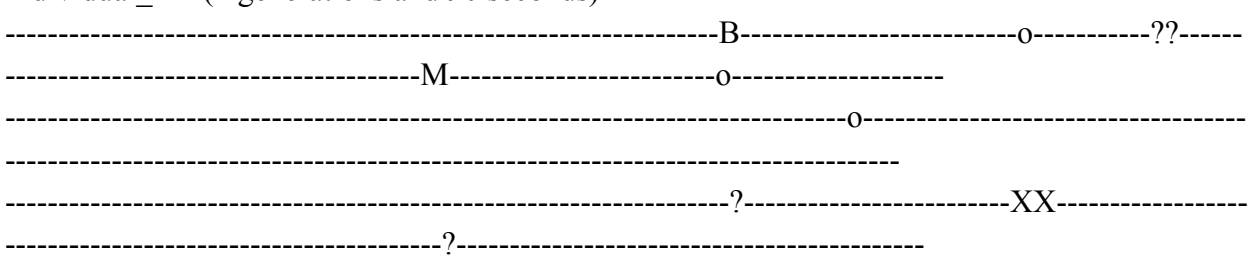
Quality metrics were employed to quantitatively evaluate the generated levels, assessing factors such as solvability, path length, enemy distribution, and aesthetic appeal. These metrics provided insights into the gameplay characteristics and design fidelity of the levels. Human evaluation complemented quantitative metrics by providing subjective assessments of factors like level layout, challenge progression, visual aesthetics, and overall enjoyment. Additionally, comparative analysis compared the performance of the proposed GA against baseline methods or alternative algorithms for Mario level generation, highlighting strengths, weaknesses, and areas for improvement.

Sensitivity analysis was conducted to examine the robustness of the GA to variations in input parameters and heuristics. By systematically varying individual parameters while keeping others constant, the sensitivity of level generation outcomes to specific factors was assessed. The experimentation also focused on refining the algorithms responsible for creating new levels within the genetic algorithm framework. Various strategies were explored to enhance the level generation process, such as fine-tuning the tile placement algorithms, optimizing enemy distribution patterns, and refining the overall layout coherence.

Through iterative experimentation and refinement, the level generation process aimed to strike a delicate balance between creativity and playability. By leveraging insights from parameter tuning, heuristic evaluation, quality metrics, human evaluation, comparative analysis, and sensitivity analysis, researchers iteratively improved the level generation algorithms to produce high-quality Mario levels that offer compelling gameplay experiences for players. This iterative refinement process aimed to continually enhance the quality, diversity, and playability of the generated levels.

Favorite Level Selection:

Individual_DE (2 generations and 90 seconds)



--o-----------------B---
--o-------------------------XXXXXXXX---

------????---
-------------------------------------X---X----------------
---X-------------------------------------
---X---------o-----------------------------------

---XX-----T---------------------------------X--
-------------------------------B---v
----o-------X---XXX-----|---
---f
---o---------XXXX----X|---------T-----------------------
----B---f
-----????-------------------------B-------------------------XXXXX----X|---------|--------M----------
----------------X---f
---XXXXXX----X|X-------X-|---------------------
---T---f
---?--------T-----XXXXXXX----X|XX------XX|--------T-------
-----------------------------BBBB------------|-----------------------------o------------f
------------------------------X------------?---------|----XXXXXXXX----X|XXX-----XX|--------|-----
-----------------------------?-------------------|----------B--------------------------o------f
m---------E-------------o----EXX--------------------E-|-E-XXXXXXXXX----X|XXXX----XX|X---
----|----X----------------------E-------------------------|-------------------------------E-----E---E--X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX--XXX--XXXXXXXXX----XXXXXXXX|--X
XXXXXXXXXXXXXXX|XXXXX-XXXX|XXXXXX--|--XXX--XX------XXX-------XXXXXX
XXXXXXXXX---XXXXXXXXXXXXX|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX-XX
The reason this is my favorite map is due to the consistency and pattern structure of the map.

Individual_Grid (4 generations and 97 seconds)
Bo-----E--E-E--Eo-----B-E-E----------o-------E--M-oE---E--M-----B--M----E---B-B-M------?-----
E-o---E----------o?o--E-E------?-EE-----o----E-------------B---o----BB---------E----E--E----------
B-M-?
|-------E------ME-------o----BM-B-M----------o-o?----o-B--E-B-E-B--o--?-E----------o---M-----?
BE----?----o-------E-----B---------o------M----E---------E--E---oMBM-------E-E--o---------EE----o-
oEo--
E-B----?---B---o-Eo----E-B--E-o--o---B-------o------o-----E---EE------E---E-----o-----B---E-B---E
o----------E?B-----M--EE---?-E-M---------B----BE-------Eo-?-oM?--Eo?-----B---E??E--------B---
-Bo---E?

B----?---Bo----------oM----M-oE------M-?----o----E-----E-----E--------o----o-----M---B--------E--
-?--B-o----------o-------E--o-o---------E---?-----------E---o----o-E-?Bo-M-o--------M-Eo--o-o--M-
?M
--------Eo-E----------------BE-E?-------E--B-oM--E-B--B-ME?---E----oM--M-M-oB---o-Eo-?----------
------?--o--E----E----------------M--------o------------E--B------------------E-o-------------------ME-
-?
M--EEB-E--BB--E---o----Eo--B--M--E--o----o-----??-----E----E-BBE---o-M------?-E--------E---?-
------------EE---ME-EE---?----E--E-----BEB-E----oB---o--------B-M-B------o?--EE--E-?---?-E-----
-E--------ET
---B?-o--------BE--?o-o-------o------------?--------E-E-----o--E---?Eo------oB-------------oM-M---?
-M-M---B--E---o-o--E----------------M-o-----E?-E-----E--M--------o-o-----o----------o-Mo?EM-----
-E--
o-o-Eo---------E-o---o-E---?---o---B----B------------E?E----------?------E--------------?--E--------o
B------?o--E----------B-----o--o---------o----E----------??--o-M--------o--M-----oo?--o--------v
o-MM------------o---?o-o-B--o------o---E----?BE?----M-----o----E--oo-BoM--o---o-o-o---------o-B
--o--E-E--------E---?EB-----oB----E?------o--oE?---EEB-E--------M----Eo------EB---E--------E-?-?
--------B
|B---------BBE--------B-?o---E---------M----E-oo-E-------o-------??-oo-------?------E----o----------
---?-?-E?-?------------M-o----oM----oo-----M-E--M--M-----M-?--------B-----?----?o-M-E----------
X
X--o-B---EE-o-----?----------o----o-M-----o--------o--------?----o----oEMo--M--B------B------B--o-
EE------B----?-------ME----MEM-?---------?B--------------------o---------o----------EooE----------o
X
X---------E-MoM----E-o-E---EEB----------------------M---o-------------oo----o-----------------MBB
--o-----------------BE-----E----o------E---------o--o-o---------o---------?--------M-E------------X
----------------T?E-----|----?--TTT----T-T-TT-T|T------M---T---|--TX-T?--------?T--T--o-----T--T-T
T|----|T-T-X---T--T|--X--T--|T--|MM-----T-?-T-?---oXT-X---|TMT---T--E--T--T-T--M-T-|-T-T--T-|-
------M
o--------||----|-X-X--TT-TT--------T|------|----TT------|?|----o||-|M------XM?---||T-TT-?|T-----T--||----
To-|T--X--X|-T------o-||-T|M----B--X-TT-|-----|--B?-TT---T---TM-----T-||-----T|E|--T--T---TTB-B
m--------TTB--||---B--TT-T||-T|-?--TTT--TT-T-T--T|-ET-TTT-|-TT--TT-TBB-TM-T--ET-?||?T|-XB|
T-ET?--o-TT-|--T-ET-||-B--T-T---TX|-MTT--T-B|TT--T--TT-T---T-T-----T-----MT|-oTXTT|||TT-?-|
-TTMT--TT-TX-E|T-|B
XTXXXXTXXTTXTXTTXTXXTXXXTTTTTXTTXTTXTXTXXXTTTXXTTTXXXTXXTX
TTXXXXTTXTXXTXXXXXXXXXXTTXXTXXXTXTXXXXTXTTTXTTXTTXTXTTXXT
XTTTTTXXXTXXTTXXTXXTTXTXXXXXXTTXXXXXTXXXXXTTXTTTXXTTXXTXX
TTTTTTX-TXTTXTXXXXXTTXXTXTTX
The reason I like this is map is due to it’s unpredictability, and the multiple variables involved. By having many components, there are more than one ways to beat this map which makes it alot more fun to play and beat.