An Approach to General Videogame Evaluation and Automatic Generation using a Description Language

Chong-U Lim and D. Fox Harrell
Computer Science & Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {culim, fox.harrell}@mit.edu

***Abstract*—In this paper, we present an approach for automated evaluation and generation of videogames made with *PuzzleScript*, a description-based scripting language for authoring games, which was created by game designer Stephen Lavelle [1]. We have developed a system that automatically discovers solutions for a multitude of videogames that each possess different game mechanics, rules, level designs, and win conditions. In our approach, we first developed a set of general level state heuristics, which estimates how close a given game level is to being solved. It is used to adapt the best-first search algorithm to implement a general evaluation approach for *PuzzleScript* games called GEBestFS. Next, we developed an evolutionary framework that automatically generates novel game mechanics from scratch by evolving game design rulesets and evaluating them using GEBestFS. This was achieved by developing a set of general ruleset heuristics to assess the playability of a game based on its game mechanics. From the results of our approach, we showcase that a description-based language enables the development of general methods for automatically evaluating games authored with it. Additionally, we illustrate how an evolutionary approach can be used together with these methods to to automatically design alternate or novel game mechanics for authored games.**

I. Introduction

Videogame designers impart concepts, images, and values for world building and storytelling into games through a variety of game design components, such as aesthetics, mechanics, narrative, conditions for success or failure, and others. The decisions that go into these designs affect how and what players think, and may similarly influence the design of other games and systems. Co-author Harrell states that "designers need to be aware of their own needs and values, the needs and values of users, and how to prioritize them" [2]. Even though a large number of different games exist, there are some basic structural elements that are shared and reoccur. The way that these shared components are defined and structured provides a means to categorize these games, giving rise to things like genre and themes. These shared structures also provides a framework for developing general computational approaches that work on a broader variety of videogames.

We use the term **design evaluation** to refer to the assessment of any subset of components that make up the design of a game to gain quantifiable insight. Some examples of related work include evaluating the quality of level designs [3], feasibility of game mechanics [4], and playability of generated puzzles [5]. Here, **design generation** refers to the procedural content generation (PCG) [6] of games, which may cover any subset of a game's design components. Examples include automatic ruleset design [7], and level designs for platformers and real-time strategy games [6], [8]. However, most approaches are usually game-specific and are not immediately generalizable to a larger variety of games. Our focus here is on developing an approach to design evaluation and generation that is general and applies to to a multitude of different videogame designs and not just a single one.

In this paper, we present results from developing a general approach for automatic design evaluation and generation for games authored using *PuzzleScript* [1], a videogame authoring and description language [9] created by game designer Stephen Lavelle. Its popularity has seen the creation of hundreds of games by both amateur and professional game designers. Thus, a greater collection of games authored by different designers are available for study. Our aim to study a broad variety games implemented by different designers makes *PuzzleScript* an appropriate choice for our research. In our approach, we developed two sets of **general heuristics** for evaluating and generating *PuzzleScript* games. The first set of **level state heuristics** is used for evaluating how close the state of given level is to completion during gameplay. We demonstrate its effectiveness by implementing a general simulation algorithm called `GEBestFS`, which adapts best-first search for our domain of *PuzzleScript* games to automatically find solutions to levels for a multitude of different videogames. We highlight its ability to discover solutions significantly quicker than a what a regular brute-force approach using breadth-first search would take. The second set of **ruleset heuristics** evaluates rules defining a videogame's mechanics and assesses them for playability. We demonstrate the ability to generate playable rulesets from scratch using an evolutionary approach. While the use of *PuzzleScript* certainly constraints the types of games that are being explored, our approach demonstrates the effectiveness of using a description language for performing general design evaluation and generation, and may serve as a guide to further develop description languages and computational approaches for general design evaluation and generation.

The rest of the paper is structured as follows: Section II covers related work that motivates this research and an overview of *PuzzleScript* and selected games. Section III details the level state heuristics and our general simulation algorithm. Section IV details the ruleset heuristics and our evolutionary approach. The experimental setup for both simulation and evolution is outlined in Section V and we present our results and analysis in Section VI. We discuss our findings in Section VII and our conclusions are in Section VIII.

II. Background & Related Work

In this section, we provide an overview of the theoretical framework and motivation for evaluating game designs, along with other work related to description languages, general game playing, and automated content generation and game design. We also provide an overview of the *PuzzleScript* description language and the selected games chosen for this work.

A. Evaluating Videogame Designs

Our motivation for studying computational approaches for evaluating designs of videogames stems from the idea that designing such computational systems goes beyond the technical considerations. An example is Harrell's analysis in [2] (using an approach called morphic semiotics) of the independent game developer Jason Rohrer's *Passage* [10], a retro graphics style side-scrolling game featuring game mechanics, narrative, and designs that encode conventional metaphors for life and death together with its use of culturally specific icons such as hearts and gravestones. Thus, we seek to highlight that importance of the *subjective* and *cultural* considerations that are involved videogame design. In [2], Harrell argues that "computing systems can express values (such as preferences of designers, norms of societies, or traditions of civilization)" and that these values can be "embedded in the data structures of computing systems." Description languages use a common syntax and format to represent such data structures, providing the opportunity to develop approaches to evaluate and compare between different game designs and their components.

B. General Video Game Playing & Description Languages

Computer science and game AI researchers Levine et al. introduce General Video Game Playing (GVGP) in [11]. The term "general" here comes from the field of artificial general intelligence (AGI) [12], whereby the methods encapsulate a "broader range of environments, and under a broad range of constraints" [11] and are not game-specific or only applicable to a bespoke game or system. GVGP extends General Game Playing (GGP) [13] (which covers traditional, turn-taking games such as *Chess* and *Othello*) to include more complex games like 2-dimensional (2D) arcade games, with the vision of moving into 3-dimensional (3D) games. Subsequently, they motivate the need for a Video Game Description Language (VGDL) for GVGP that is both human-readable and in a format that can be compiled to generate an instance of the game. It should also "support the core mechanics and behavior expected of classical 2D video games and be unambiguous, extensible, and tractable that could provide opportunities for PCG." Python VGDL (PyVGDL) [14] by computer scientist and AI researcher Tom Schaul is the most recent realization of this, providing a simple high-level description language for 2D videogames based on "defining locations and dynamics for simple building blocks and interaction effects when such object collide."*PuzzleScript* bears similarities to PyVGDL in syntax, structure, and implementable games. Our aim to study a broad variety games authored by different designers makes *PuzzleScript* an appropriate choice for our research.

C. Automatic Content Generation and Game Design

Automatic content generation in videogames is an active research area, with approaches to generate components such as levels and maps [15], [16], items [17], game mechanics [4], [18], and entire designs [7], [16], [19], [20]. Computer scientists and game AI researchers Togelius et al. characterizes several features of what they term *generatable games* [21], which are a subset of characteristics originally identified by game designers Elias et al [22]. Our attention focuses on the following identified core features: *heuristics* (i.e., rules of thumb that guide players to do well in the game), *rules* (i.e., game mechanics), *standards* (i.e., commonly accepted patterns that players are familiar with like `WASD` controls), *outcomes* (e.g., number of moves to solve a level), *ending conditions* (i.e., determining game end states), and *complexity tree growth* (e.g., branching factor of actions a player can take from a given state). Several approaches exist for automated design evaluation and generation, such as the use of answer set programming (ASP) to search the design space [23] or simulation-based approaches [4], [24]. The approach we've developed is a general simulation-based approach that is not game-specific, and instead makes use of a description language that has been used by many different game designers to author a multitude of games. The benefits of this over alternative approaches like using ASP are that we may analyze these existing game designs, and that we work directly with the description language without a need to first convert a game's mechanics and rulesets into boolean logic prior to generation [25].

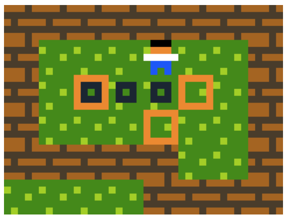
D. PuzzleScript

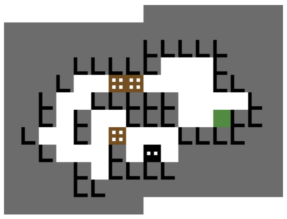
A *PuzzleScript* file is divided into of eight different sections, each for defining the elements for the various game design components. We focus on the `Rules`, `Levels`, and `Win Conditions` sections since they make up the core components of a game's puzzle design, game mechanics, and winning criteria. The `Levels` section defines the levels that the game possesses. Each level is represented using a 2-dimensional array of characters. Each character in the array references defines the objects that are present in that position. The `Rules` section defines the game mechanics of a *PuzzleScript*. Each `Rule` specifies the result of the positioning, movement, and actions of objects in the game. `Win Conditions` specify the requirements for the completion of each level. We provide more detail of the structure and syntax of these *PuzzleScript* components when describing our approach later in Sections III and IV. *PuzzleScript* is used as a case-study here, but the similarities that it has with other videogame description languages [14] allows our approach to be generalizable to fit their respective syntax and structures.

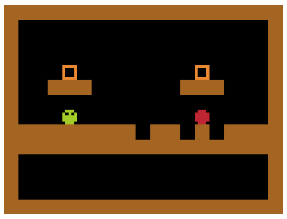
E. Overview of Selected PuzzleScript Games

We illustrate the general applicability of our algorithmic approaches by testing them against four different games created by various authors. Being able to create clones of existing games demonstrates *PuzzleScript*'s capabilities as videogame description language. Figure 1 shows screenshots of them.

1) Microban: A clone of the commercial Japanese videogame *Sokoban* [26], the player has to to push a number of boxes onto designated target areas in each level. *Sokoban* is a provably challenging game [27] (PSPACE-Complete) and has been the focus of various game AI researchers.


(a) Microban


(b) Block Faker


(c) Lime Rick

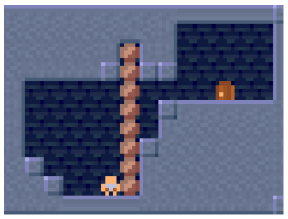

(d) Atlas Shrank

Fig. 1. Screenshots of different PuzzleScript games. Each game possesses different objects, visual appearance, rules, and win conditions.

2) Block Faker: A clone of an original game of the same name that was made for the 48-hour game competition [28], the player must navigate towards an exit (marked in green) in a level filled with both movable and unmovable blocks. Additionally, blocks may be removed whenever they form a continuous row or column of three blocks of the same color.

3) LimeRick: A clone of a popular game of the same name [29], the player must navigate towards a target (marked in red) using a novel mechanic. The snake-like player character can extend its length horizontally and vertically to enables it to navigate across obstacles, including itself to reach higher areas.

4) Atlas Shrank: An original game whereby the player must reach an exit (marked with a door image). The player may pick up boulders in the game either carry them or throw them. Basic physics simulation occurs, allowing crates to fly across mid-air, drop down, and stack on top of each other.

III. Design Evaluation via Simulation

In this section, we present a general approach to evaluating the various *PuzzleScript* games. We adopt a simulation-based approach that is described as follows. Given a *PuzzleScript* game, we seek to discover the sequence of moves, constrained by the game's set of `Rules`, that results in reaching the game's `Win Conditions` for any given `Level`. We refer to a *PuzzleScript* file's set of `Rules` as its **ruleset**, a game's `Level` as a **level**, and its `Win Conditions` as **win conditions**. The moves considered are the four movement directions `UP`, `DOWN`, `LEFT`, `RIGHT`), and the `ACTION` move. The `ACTION` move is a generic move available for game's to provide interactivity, such as picking up and throwing boulders in the game *Atlas Shrank*. Adopting this approach, we are able to evaluate the games in several ways. First, given a game with feasible rulesets, levels, and win conditions, it acts as a general game solver that produces a **solution** for any given level. The existence of a solution estimates the feasibility of a level's design, while the length of the solution provides a means of evaluating the complexity of a level's design. Additionally, patterns of moves observed in the solution may also provide insight into the elegance of the solution, for example, repetitive movements might be more boring while solutions that are precise and leave no room for error might be deemed more challenging and exciting. Second, imagine modifying a game component while keeping others the same in order to evaluate if alternative designs exist. For example, if we changed the ruleset of a game while keeping the same levels and win conditions, our approach would evaluate whether the modified ruleset still results in a **feasible game design**. A game design is feasible if its ruleset ensures that there exists a sequence of moves enabling the player to reach the win conditions.

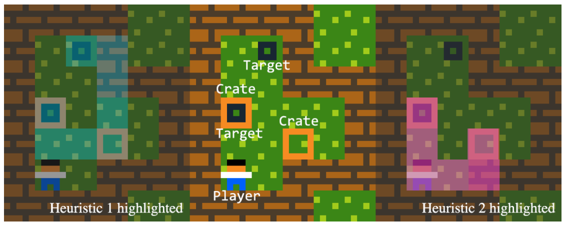


Fig. 2. Screenshots of level state heuristics as applied to a level in *Microban*.

A. General Breadth-First Search Simulation

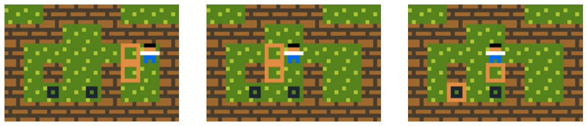
The first step toward a simulation-based general evaluation approach was to implement a brute-force **breadth-first search** (BFS) simulator. This was done to establish baseline performance to compare against. A BFS approach was used by Cook et al. [4] to evaluate if a mechanic for its platformer game was *usable*, that is, "able to overcome an obstacle in order to complete a task, such as reaching an exit." We adapted the BFS algorithm and implemented a **General Evaluation Breadth First Search Simulation** (`GEBFS`), which searches for a sequence of moves to complete a level for any *PuzzleScript* game. Here, a **level state** is a two-dimensional array of all objects in each tile position. Simulating a move results in a new updated state, which may have been previously visited. For each given state, all subsequent states one move apart are evaluate first and added to a queue of states to be evaluated. Moves are simulated one after another, updating the level state until the win conditions are reached.

B. General Best-First Search Simulation

We encountered some shortcomings with the `GEBFS` approach. It adopts a brute-force approach since, upon simulating a move on a state and obtaining several subsequent states, we do not prioritize subsequent level states to be explored in any way. This results in a large search space to be explored in the worst case. In order to overcome these problems, we needed some method of evaluating a **cost** for a given level state. We seek to prioritize one level state over another if it is deemed "closer" (lower cost) to the win conditions. This cost is a linear combination of the following **level state heuristics**.

1) Distance between Win Condition Objects
2) Distance between Player and Win Condition Objects

We made use of the *Manhattan distance* as a measure, which has been shown by computer scientists Dorst et al. to be an effective heuristic for solving games such as *Sokoban* [30]. Figure 2 shows how these heuristics apply to a level state in *Microban*. The first heuristic characterizes level states where `Objects` used in `Win Conditions` (e.g., crates and targets) are closer together. The second heuristic characterizes level states where the `Player` is closer to `Objects` used in `Win Conditions` (e.g., player is closer to crates or targets.)



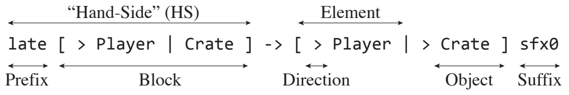
(a) $Cost = 0.244$ (b) $Cost = 0.156$ (c) $Cost = 0.089$

Fig. 3. Screenshots of various points during a solution run for *Microban*. Level states progressively approach a winning state. (Smaller scores are better.)

Intuitively, this means we prioritize states with shorter distances between `Win Condition` objects and the `Player`. These distances are divided by the maximum distance required to be traveled for a given `Level` ($width \times height$) to normalize the `Cost` value between $[0, 1]$. If the `Rule` takes the form: "`NO <object1> [ON <object2>]`", we use $1 - \textsc{Cost}$ to reflect the preference for level states that maximize these distances instead. Figure 3 shows various `Level` states of *Microban* together with their distance costs. It can be seen that the values become smaller as states approach the `Win Condition`. Being able to assign costs to level states enables us to begin searching find solutions using a greedy approach. We adapted the **best-first search** algorithm to implement a **General Evaluation Best-First-Search Simulation** (`GEBestFS`) shown in Figure 4.

```
procedure GEBestFS($state$)                                   ▷ Returns solution
    $Open \leftarrow []$                                       ▷ Queue of $(state, score, actions)$.
    $Closed \leftarrow \{\}$                                   ▷ Indicates $(state, action)$ seen.
    $Open$.enqueue(($state$, Cost($state$), []))
    while $Open.length \neq 0$ do
        ($curState, score, actions$) $\leftarrow$ $Open$.dequeue();
        if IsWinState($curState$) then
            return $actions$
        end if
        for each $action$ do
            Load $currentState$
            $nextState \leftarrow currentState$.apply($action$)
            $nextScore \leftarrow$ Cost($nextState$)
            $actions$.push($action$)
            if !$Closed$[($nextState, action$)] AND ($nextState, action$) $\notin$ $Open$) then
                $Closed$[($curState, action$)]=$true$
                $n \leftarrow (nextState, nextScore, actions)$
                $Open$.enqueue($n$)
                $Open$.sort()                                  ▷ Prioritized by Cost
            end if
        end for
    end while
    return []                                                  ▷ No solution.
end procedure
```

Fig. 4. General Evaluation Best-First-Search Simulation Algorithm



```
late [ > Player | Crate ] -> [ > Player | > Crate ] sfx0
```

Fig. 5. Anatomy of a *PuzzleScript* Rule

IV. Design Evolution with Rulesets

In this section, we present an approach to constructing new `Rule` designs for *PuzzleScript* through evolutionary computation. Figure 5 illustrates definitions for the various components that make up a `Rule`. It has two `HandSides` (`HS`). The `LHS` may have a `Prefix` while the `RHS` may have a `Suffix`. Each `HS` contains one or more `Blocks` and a `Block` may contain one or more `Elements` separated by a "|." An `Element` is made up of a `Direction` and an `Object`, both of which may be omitted. A `Rule` is **valid** when it has the same number of `blocks` and `elements` per block on both its `LHS` and `RHS`. A **null** `Rule` is defined as having the form: "`[ ] -> [ ]`."

A. Ruleset Heuristics

Similar to seeking a scoring mechanism for level states in Section III, here we introduce several heuristics to enable us evaluate a given ruleset. These heuristics have to be game-independent in order to be applicable to any *PuzzleScript* game, and are used to help us identify feasible rulesets that result a game designs with higher playability.

1) Player in ruleset: This heuristic checks each `Rule` in the ruleset for whether the `Player` appears in any of them, irrespective of the block or element position. $\textsc{Score}(\text{“}playerInRuleset''\text{”}) = 1.0$ if so, and 0.0 otherwise.

2) Objects in Ruleset: This heuristic checks each `Rule` in the ruleset for whether each `Object` appears in any of them, irrespective of the block or element position. This implies that rulesets that affect more objects in the game tend to be favored. $\textsc{Score}(\text{“}objectsInRuleset''\text{”})$ = proportion of `Objects` seen out of all defined in the file.

3) Player in LHS: This heuristic checks each `Rule` in the ruleset for whether the `Player` appears on the LHS. This increases the likelihood of a `Rule` specifying that player is able to perform actions, which affect the game world. This is a general pattern observed in most PuzzleScript games. $\textsc{Score}(\text{“}playerInLHS''\text{”}) = 1.0$ if so, and 0.0 otherwise.

4) Player Movement in Ruleset: This heuristic checks each `Rule` in the ruleset for whether player movement is observed (e.g., `< Player`). It increases the likelihood of the player both performing and receiving actions within the game world. $\textsc{Score}(\text{“}playerMovement''\text{”}) = 1.0$ if so, and 0.0 otherwise.

5) Unique Directions per Rule: This heuristics checks each `Rule` in the ruleset for whether the number of unique directional moves per `Rule`. Its score is inversely proportional to this number and favors `Rules` with less erratic directional changes. For example, the `Rule` `[ > player | crate ] -> [ > player | > crate ]` has one unique direction operator, as opposed to `[ > player | crate ] -> [ ^ player | v crate]`. Here, pushing a `Crate` causes it to move sideways while the `Player` moves move orthogonal to it. $\textsc{Score}(\text{“}uniqueDirsPerRule''\text{”}) = \frac{1.0}{\#AvgUniqueDirsPerRule}$.

B. Fitness Functions

Given the heuristics from the previous section, our first fitness function calculates the weighted average of the scores from each heuristic: $Fitness_{Heuristics}(ruleset) = \sum w_i \times \textsc{Score}_i$. These heuristics are evaluated based on the structural definitions of each `Rule` in the ruleset, but do not provide information about how well they suit a given *PuzzleScript* game (i.e., a high-scoring ruleset might not actual be sufficient to solve a given `Level`.) As such, we introduce two additional fitness functions as follows:

1) Feasibility: A ruleset is **feasible** if it is possible to find a solution for a given `Level`
2) Validity: A given ruleset is **valid** if it does not produce runtime errors when solving a given `Level`

Ruleset feasibility is evaluated using the BestFS general evaluation by simulation approach from III. We run the algorithm for 1200 iterations, after which it is assumed that a solution can not be found. $Fitness_{Feasible}(ruleset) = 1.0$

if a solution is found and 0.0 otherwise. Ruleset validity is evaluated using the same approach, except that we check for runtime errors output by the *PuzzleScript* engine as the simulation runs. $Fitness_{Valid}(ruleset) = 1.0$ if no errors are found and 0.0 otherwise. We calculate an overall fitness value by averaging across the three fitness functions described.

C. Mutation

We implemented five **mutation** operators for rulesets. Given a ruleset, each mutator may affect entire ruleset or just individual `Rules` within them. Figure 6 illustrates an example of how a ruleset is modified by each type of mutator, applied in sequence, before resulting in a final ruleset. We next describe each mutator in more detail using this same example.

1) Ruleset Size Mutator: This mutator modifies the size of a given ruleset by incrementing or decrementing the number of `Rules`. It can be viewed as a way of varying the complexity of a game's mechanics. When incrementing, it adds a null `Rule` into a random position in the ruleset. When decrementing, it removes a `Rule` frandomly for rulesets with ≥ 2 `Rules`. In Figure 6, the size of the ruleset is reduced to one.

2) Object Mutator: This mutator modifies a random object `Element` in a `Rule`. It can be viewed as a way of re-prioritizing objects in a game by determining which may be interacted with in the game. Candidate objects are obtained from the *PuzzleScript* file's `Objects` section. The direction operator may also be modified or removed by this mutator. In Figure 6, the `Crate` object is changed into a `Target` object.

3) Direction Mutator: This mutator modifies a random direction operator in a `Rule`. It can be viewed as a way of re-defining the game mechanics within a game by modifying the interaction behaviors between objects. Candidate directions are the four relative direction operators. The direction operator may also be removed by this mutator. In Figure 6, the direction operator of the `Player` object in the LHS is reversed.

4) Block Size Mutator: This mutator modifies the size of a random block in a `Rule` by either incrementing, or decrementing, the number of elements in a block. It can be viewed as a way of varying the complexity of a game's mechanics by affecting the interaction behavior of more or less objects during each move by the player. Incrementing adds a random element (direction + object), while decrementing removes a random element. In Figure 6, the number of elements per block is reduced to one for the given `Rule`.

5) Hand-Size Swap Mutator: This mutator swaps the hand-sides of a `Rule`. It can be viewed as re-prioritizing the objects that influence changes in the game. In Figure 6, the block `[ < Player ]` moves from the LHS to the RHS, while the block `[ > Target ]` moves from the RHS to the LHS.

D. Crossover Operators

Crossover operators are genetic operators used to generate children rules from two or more parent rule, which possess traits from both parents. We implemented two **crossover** operators for rulesets. Our condition for crossover was for both rulesets to possess the same ruleset size, number of blocks per rule, and number of elements per block. During **selection** for the crossover operation, pairs of rulesets are continuously

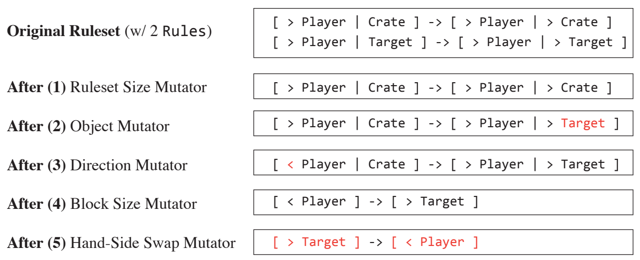
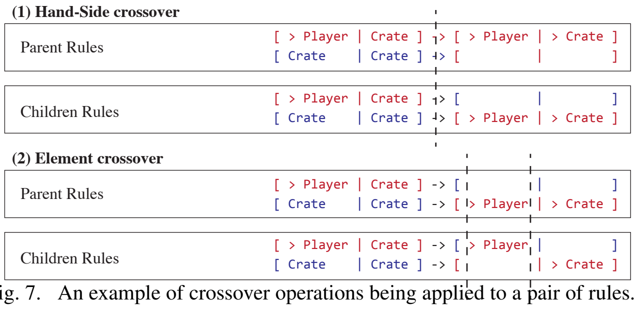

Original Ruleset (w/ 2 `Rules`)
```
[ > Player | Crate ] -> [ > Player | > Crate ]
[ > Player | Target ] -> [ > Player | > Target ]
```
After (1) Ruleset Size Mutator
```
[ > Player | Crate ] -> [ > Player | > Crate ]
```
After (2) Object Mutator
```
[ > Player | Crate ] -> [ > Player | > Target ]
```
After (3) Direction Mutator
```
[ < Player | Crate ] -> [ > Player | > Target ]
```
After (4) Block Size Mutator
```
[ < Player ] -> [ > Target ]
```
After (5) Hand-Side Swap Mutator
```
[ > Target ] -> [ < Player ]
```

Fig. 6. An example of a series of mutators being applied to a ruleset. The red color is used in certain steps to highlight the element being mutated.


(1) Hand-Side crossover

Parent Rules
```
[ > Player | Crate ] -> [ > Player | > Crate ]
[ Crate    | Crate ] -> [          |         ]
```
Children Rules
```
[ > Player | Crate ] -> [          |         ]
[ Crate    | Crate ] -> [ > Player | > Crate ]
```
(2) Element crossover

Parent Rules
```
[ > Player | Crate ] -> [          |         ]
[ Crate    | Crate ] -> [ > Player | > Crate ]
```
Children Rules
```
[ > Player | Crate ] -> [ > Player |         ]
[ Crate    | Crate ] -> [          | > Crate ]
```

Fig. 7. An example of crossover operations being applied to a pair of rules. In Hand-Side crossover (top-image), the LHS of the rules are maintained, while the RHS are swapped with one another. In Element crossover (bottom image), the rule element depicting player object movement in the RHS of the second rule is swapped with the empty element in the RHS of the first rule.

chosen until these preconditions are achieved. Figure 7 shows an example of a sequence of crossover operations on `Rules`.

1) Hand-Side Crossover: In this operation, a single-point crossover occurs resulting in the the RHS of the first parent `Rule` swapping with the RHS of the second parent `Rule`. This is illustrated in the top half of Figure 7.

2) Element Crossover: In this operation, a two-point crossover occurs to isolate a random element in the first parent `Rule` to swap with the corresponding parent of the second parent `Rule`. This is illustrated in the bottom half of Figure 7.

V. Experimental Setup

In this section, we describe the steps that were undertaken to analyze both the general evaluation via simulation approach from Section III and the design evolution approach from Section IV. We implemented a modified version of the *PuzzleScript* engine to extend its features to allow simulating actions and implement the algorithms for simulation and evolution.

A. General Evaluation via Simulation Setup

For design simulation, we made use of the `GEBFS` and `GEBestFS` simulation approaches described in Section III on the 4 selected games representing a diversity of designs from Section II-E. For each game, we ran our simulators on 5 levels to obtain a solution, or until a reasonable iteration threshold was reached. This was repeated 3 times for each level.

B. Design Evolution Setup

For design evolution, our aim was to see if using our ruleset heuristics and simulation approach enabled us to design new rulesets that were both feasible and valid for a given game and level. Thus, our approach first involves modifying an existing *PuzzleScript* game by removing all its `Rules`.

TABLE I. COMPARATIVE SUMMARY OF THE MEAN NUMBER OF ITERATIONS EACH ALGORITHM REQUIRED TO SOLVE THE GAME-LEVELS

	Game	Level	BestFS	BFS	p.value
1	atlas shrank	1	6	15	0.09331
2	atlas shrank	2	314	3062	0.00031
3	atlas shrank	3	1195	1466	0.02000
4	atlas shrank	4	6373	5802	0.90799
5	atlas shrank	5	3005		
6	block faker	1	108	220	0.00016
7	block faker	2	62	136	0.00005
8	block faker	3	30	2452	0.01424
9	lime rick	1	30		
10	lime rick	2	295	747	0.03095
11	lime rick	3	3926		
12	lime rick	4	5731		
13	lime rick	5	265		
14	microban	1	1059	1311	0.00163
15	microban	2	284	2129	0.00401
16	microban	3	2466	3944	0.00136
17	microban	4	17453		
18	microban	5	7479		

Then, working from an initial population of rulesets, we sought to evolve newer rulesets that enabled our simulator to solve a given `Level`. We used *Microban* as a case-study for this. To generate the **initial population**, we initialized 50 rulesets to contain only a single null `Rule`. Next, we performed an initial round of mutations using all 5 mutators to generate a population of 50 random rulesets. We made use of **fitness proportionate selection** using the overall fitness function described in Section IV-B. The **mutation rate** for each generation was set to 10%. The probability of a mutator being chosen during mutation was set to 50%. The **crossover rate** was varied between 60%–70%, and we settled on 65%.

VI. RESULTS & ANALYSIS

In this section, we present the results obtained from the experimental setup described in Section V. We first present and analyze the results for the general evaluation via simulation procedure, followed by the results from the design evolution.

A. *Simulation Results & Analysis*

For each of the four *PuzzleScript* games, we ran both simulators for five levels for a total of twenty **game-levels**. To illustrate the difficulty of some of these game-levels and the complexity of their solutions, we encourage readers to view a gallery[1] containing animated playthroughs of each game-level obtained by `GEBestFS`. We used the following measures to compare the performances of `GEBestFS` and `GEBFS`.

1) Solved Game-Levels: With a total of 4 games and 5 levels each, we had a total of 20 game-levels to test. `GEBestFS` was able to find a solution for 18/20 (90%)) and `GEBFS` was able to find a solution for 11/20 (55%) of them.

2) Number of Iterations: In both `GEBestFS` and `GEBFS`, each iteration of the algorithm selects a node from the queue to be evaluated. In `GEBFS`, neighboring nodes, which are one action away, are added to this queue and evaluated in the order that they are added. In `GEBestFS`, neighboring node are added and prioritized based on the cost-heuristic, seeking more promising nodes ahead of others to find a solution faster. Table I shows a summary of the average number of iterations each approach took to solve the game-levels. Overall, `GEBestFS` finds a solution significantly quicker than `GEBFS`.

[1]Gallery of solved game-levels: http://imgur.com/a/AkiMv

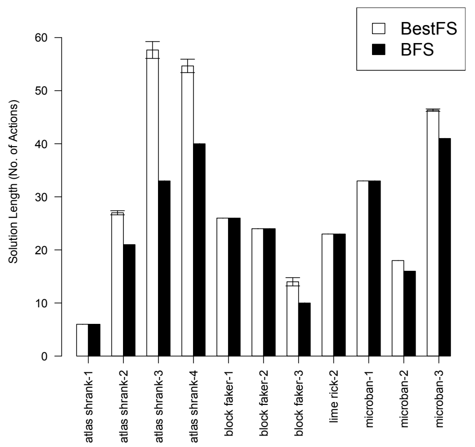
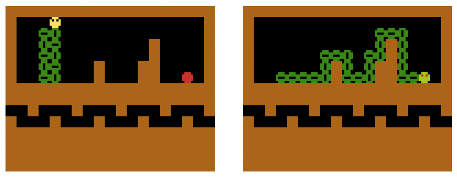
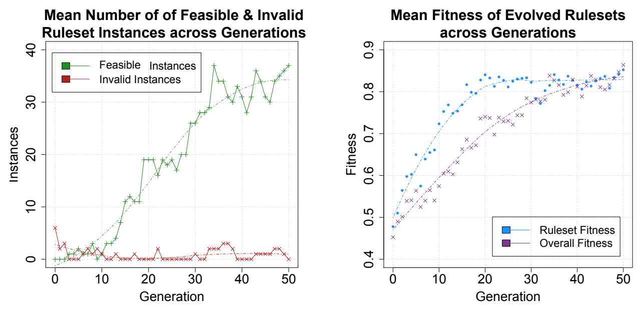


Fig. 8. Mean solution lengths obtained by both GEBFS and GEBestFS for game-levels that were solvable by both simulators. GEBFS always finds solutions of the shortest length, while GEBestFS found longer solutions (between 10%–70% longer). The trade-off between the solution length found and time-taken to discover a solution made GEBestFS preferred as an approach.



(a) GEBFS Simulation (b) GEBestFS Simulation

Fig. 9. Screenshots comparing simulation approaches in *Lime Rick*, whereby the player character needs to reach the exit (colored red) in each level. GEBFS is significantly slower as it keeps finding valid moves that extend the player character by a single space, shown in Figure 9(a). GEBestFS prioritizes a path that leads toward the exit as shown in Figure 9(b).



(a) Feasible & Invalid Counts (b) Ruleset & Overall Fitness

Fig. 10. Scatterplot of evolved ruleset performance across generations.

The reason lies in the heuristics in `GEBestFS` that seek to move *closer to* objects that satisfy the win conditions of a level. Most games involve the player moving, or manipulating objects, towards a target, and `GEBFS` is ineffective when the mechanics provide freedom for the player to explore the level-space in games such as *Lime Rick*, as illustrated in Figure 9.

3) Solution Length: Each game-level solution is a sequence of moves and actions that are executed in order. A shorter solution thus requires fewer moves. Figure 8 compares solution lengths for game-levels solvable by both `GEBFS` and `GEBestFS`. `GEBFS` always returns the shortest solution, while `GEBestFS` does not necessarily give the shortest solution due to its greedy approach. We feel that the trade-off of a longer solution length but obtained in a much shorter time (#iterations) makes GEBestFS preferable over `GEBFS` overall.

B. Evolution Results & Analysis

With an initial population of 50 randomly generated rulesets for *Microban*, evolution continued for 50 more generations. We used the following measures for evaluation:

1) Feasibility & Validity: In Figure 10(a), we observe that the number of feasible rulesets in the population (i.e., rulesets that enable the game-level to be solved) increases while the number of invalid rulesets (i.e., rulesets that cause errors) decreases. Thus, starting with a design with no game mechanics defined, our evolutionary approach designed playable rulesets that *increasingly satisfied feasibility and validity*

2) Fitness: In Figure 10(b), we observe that the ruleset fitness ($Fitness_{Heuristic}$) increases over the evolved generations. The overall fitness, which combines $Fitness_{Valid}$ and $Fitness_{Feasible}$ thus exhibits a similar increasing trend. This illustrates that our chosen fitness functions are effective in *modeling better ruleset designs*, and that they work well both individually and collectively for evolving designs.

C. Evolved Designs

During the evolutionary process, certain generated rulesets, while feasible and valid, resulted in mechanics and solutions that were deemed "unplayable." We suspect that this behavior was a result of overfitting to the single game-level of *Microban* that was used for ruleset evolution. However, we observed several evolved rulesets that resulted in interesting and playable game designs. We describe each of them next.

1) Evolved Design #1 – Crate Pull: The evolved ruleset enabled mechanics for a `Player` to pull a `Crate`. Figure 11 shows this mechanic in action whereby the `Crate` is pulled several times to be positioned over the `Target`.

2) Evolved Design #2 – Morphing: The evolved ruleset enabled mechanics for a `Player` to "morph" into another object when touching a `Target`. In Figure 12(a), the `Player` morphs into a `Wall`, reducing the number of `Targets` to one, which happens to already satisfy the `Win Condition` with the `Crate` on it. In Figure 12(b), the `Player` morphs into a `Crate`, satisfying the `Win Condition` for both `Targets`.

3) Evolved Design #3 – Spawning: The evolved ruleset enabled mechanics for a `Player` to "spawn" other objects in the level. In Figure 12(c), the `Player` is able to spawn additional `Crate` objects at empty tile positions. Spawning a `Crate` on the `Target` solved the level.

VII. Discussion

We have described an approach that uses a videogame description language for developing general computational approaches to enable the evaluation a multitude of different games and to generate alternate and novel game designs from scratch using an evolutionary approach. Here, we address the broader implications of this research pertaining to the subjective aspects of videogame design like the expression of the goals, identity, and values of both designers and players. As different level state heuristics returned different solution sequences (and lengths), general heuristics may be used to develop models of different play styles in a "top-down" manner or used to inform other approaches like empirically

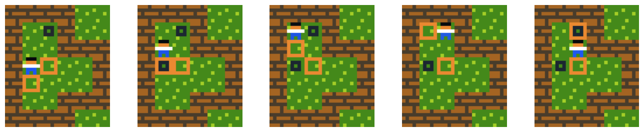
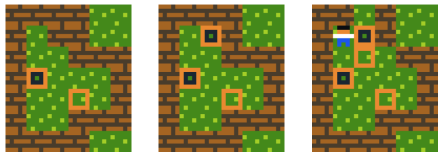


Fig. 11. Screenshots showing the "Crate Pull" evolved ruleset design.


(a) Wall Morph (b) Crate Morph (c) Crate Spawn
Fig. 12. Solution states obtained via "Morphing" and "Spawning."

derived player models, such as learning curves [16], but with the added benefit of being generally applicable to a broader set of games. We may map them to different player motivations and styles [31], and generate a level requiring a solution that involves the execution of a difficult move to cater to achievement-driven players. The same applies for ruleset heuristics for the purpose of designing alternative or novel game mechanics. A more social impact-driven aim would be to adopt more a value-driven design approach to videogame design, addressing components that are often deemed subjective such as aesthetics (e.g., avatar representation, customization, asset replacement during mutation). General approaches to addressing these issues may allow us to computationally evaluate and compare games and criticallly assess the underlying values held by designers, enabling us to seek better heuristics and generate alternative designs that reinforce more positive values.

A. Future Work

Several extensions to this work would make progress toward some of these aims. The first is developing additional general heuristics for both level states and rulesets and testing their general applicability on more games. Second, to prevent overfitting, we could perform design generation across more game-levels or perform co-evolution of rulesets and levels. Third, we may use different heuristics and fitness functions to not just generate valid and feasible rulesets, but rulesets that increase difficulty. For example, a fitness function that prefers longer solution lengths might generate rulesets that only allow non-trivial action sequences. Fourth, we can extend these approaches to apply to other game design components such as level design or visual design through *skinning*. Computer scientists Mike Treanor et al. notes that "a thoughtfully applied skin produces a more meaningful experience by coherently matching a games mechanics with its contents theme." [32]. Fifth, we could use a formal approach to evaluating game designs for playability. For example, in [2], Harrell uses algebraic semiotics [33] and morphic semiotics [34] as a precise language to "design systems to reflect users' values" by considering the precise descriptions of such structures, termed *semiotic spaces* [34], [35], and their mappings from one to another, termed *semiotic morphisms* [35]. Comparison between games may be done by analyzing the structure-preserving mappings between their design and implementation [34]. We may then use this to guide the evaluation of the game designs to determine those that have more desirable game mechanics.

Finally, although we focused on *PuzzleScript* in this paper, our hope is that the approach will generalize to other description languages, including the much more general morphic semiotics. One challenge in this area is that the description language approach, while successful for simple 2D games, may not generalize well to other types of videogames with greater complexity. Morphic semiotics provides a formal (universal algebra and category theory-based) approach to describing user interfaces (including game) that is promising in this regard. We believe it would be an effective way to extend our approach and other related work to cover more different types of games.

VIII. Conclusion

We have presented a general approach for the automated evaluation and generation of videogames authored for the description language *PuzzleScript*. We developed general heuristics for both level states and rulesets that are applicable to multiple videogames, each possesssing different game designs. Our approach effectively evaluates game designs by obtaining solutions to puzzles automatically and is used together with an evolutionary approach to automatically generate playable rulesets from scratch. We believe that these results will help to inform the future development for related research areas and seek use to address value-driven design needs of videogames.

Acknowledgment

The authors would like to thank Rajeev Rikhye for the useful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 1064495.

References

[1] S. Lavelle. PuzzleScript. [Online]. Available: http://puzzlescript.net

[2] D. F. Harrell, *Phantasmal Media: An Approach to Imagination, Computation, and Expression*. MIT Press, 2013.

[3] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a generic method of evaluating game levels," in *Proceedings of the AAAI Conference on Artificial Intelligence for Interactive Digital Entertainment (AIIDE)*, 2013.

[4] M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic miner: reflection-driven game mechanic discovery and level design," in *Applications of Evolutionary Computation*. Springer, 2013, pp. 284–293.

[5] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[7] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Computational Intelligence and Games (CIG)*, 2011.

[8] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi *et al.*, "The 2010 mario ai championship: Level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, 2011.

[9] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language," *Artificial and Computational Intelligence in Games*, p. 85, 2013.

[10] J. Rohrer. Passage. [Online]. Available: http://hcsoftware.sourceforge.net/passage/

[11] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, T. Thompson, S. M. Lucas, M. Mateas *et al.*, "General video game playing," *Artificial and Computational Intelligence in Games*, vol. 6, pp. 77–83, 2013.

[12] B. Goertzel and C. Pennachin, *Artificial general intelligence*. Springer, 2007.

[13] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI magazine*, vol. 26, no. 2, p. 62, 2005.

[14] T. Schaul, "A video game description language for model-based or interactive learning," in *IEEE Conference on Computational Intelligence in Games (CIG)*, 2013, pp. 1–8.

[15] J. Togelius, R. De Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," in *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006, p. 61.

[16] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *IEEE Symposium On Computational Intelligence and Games (CIG)*. IEEE, 2008, pp. 111–118.

[17] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009, pp. 241–248.

[18] M. J. Nelson and M. Mateas, "Towards automated game design," in *AI* IA 2007: Artificial Intelligence and Human-Oriented Computing*. Springer, 2007, pp. 626–637.

[19] C. Browne and F. Maire, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1–16, 2010.

[20] M. J. Nelson and M. Mateas, "Recombinable game mechanics for automated design support." in *Proceedings of the AAAI Conference on Artificial Intelligence for Interactive Digital Entertainment (AIIDE)*, 2008.

[21] J. Togelius, M. J. Nelson, and A. Liapis, "Characteristics of generatable games," in *Proceedings of the Fifth Workshop on Procedural Content Generation in Games*, 2014.

[22] G. S. Elias, R. Garfield, and K. R. Gutschera, *Characteristics of games*. MIT Press, 2012.

[23] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[24] M. Shaker, N. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2013.

[25] A. M. Smith and M. Mateas, "Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games," in *IEEE Symposium on Computational Intelligence and Games*. IEEE, 2010, pp. 273–280.

[26] T. Rabbit. Sokoban. [Online]. Available: http://www.sokoban.jp

[27] J. Culberson, "Sokoban is PSPACE-complete," in *Informatics 4 Fun With Algorithms*, vol. 4. Carleton Scientific, 1999, pp. 65–76.

[28] Droqen. Block Faker. [Online]. Available: http://www.ludumdare.com/compo/ludum-dare-21/?action=preview\&{}uid=1552

[29] T. Tuovinen. Lime Rick. [Online]. Available: http://www.kongregate.com/games/KissMaj7/lime-rick

[30] M. Dorst, M. Gerontini, A. Marzinotto, and R. Pana. Solving the sokoban problem. [Online]. Available: http://forum2.fragfrog.nl/papers/solving_the_sokoban_problem.pdf

[31] N. Yee, "Motivations for play in online games." *Cyberpsychology & behavior : the impact of the Internet, multimedia and virtual reality on behavior and society*, vol. 9, no. 6, 2006.

[32] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "Proceduralist readings: How to find meaning in games with graphical logics," in *Proceedings of the 6th International Conference on Foundations of Digital Games*. ACM, 2011, pp. 115–122.

[33] J. Goguen, *Algebraic semantics of imperative programs*. MIT press, 1996.

[34] D. F. Harrell, "Shades of computational evocation and meaning: The griot system and improvisational poetry generation," in *Proceedings of the Sixth Digital Arts and Culture Conference (DAC)*, 2005, pp. 133–143.

[35] J. Goguen, "An introduction to algebraic semiotics, with application to user interface design," in *Computation for metaphors, analogy, and agents*. Springer, 1999, pp. 242–291.