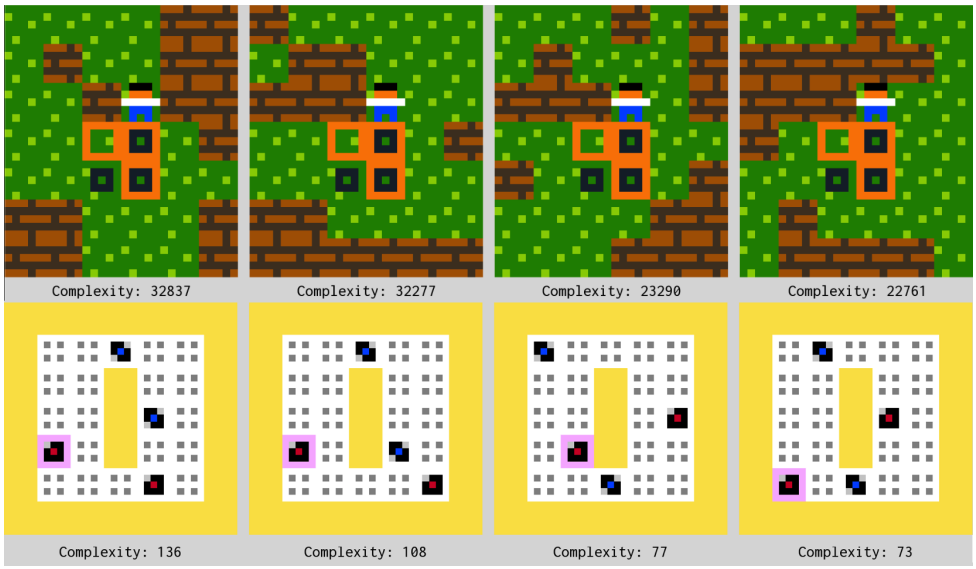
Mixed-initiative methods for designing Sokoban-like puzzles



Kevin De Keyser

Bachelor Thesis
July 2019

Supervisors:
Dr. Anna Maria Feit
Prof. Dr. Otmar Hilliges

ETH zürich

Advanced Interactive Technologies

Abstract

Puzzle games have successfully been designed entirely by hand or completely through computer generation. Can value be derived from combining both strategies towards puzzle design and, if so, in what way?

To address this question, we developed MixedAim, a tool with which designers can interactively co-create Sokoban-like puzzles through direct manipulation and steering passively presented level suggestions. We conducted a think-aloud user study followed by a structured interview with experts on the subject, identified conventional design approaches, and analyzed in which ways interactions with MixedAim supported these approaches. Furthermore, we analyzed how our participants adapted to the MixedAim suggestions, identified that the perceived usefulness is dependent on the role of MixedAim in the design process and noticed that it was particularly good at identifying interesting mechanics in small levels with many possible variations.

In conclusion, we found that MixedAim benefitted puzzle designers as another tool in their toolset for creating puzzle games but received concerns that the tool can make it easy to create complicated and bad levels which adds to the responsibility of the designers planning to incorporate MixedAim in their workflow.

Acknowledgement

I want to thank my supervisor Dr. Anna Feit for her continued support during the weekly interviews, for her input concerning the preparations and execution of the user study, for frequently sending me papers on the topic and giving me feedback on the wording and structure of the thesis. I would also like to thank Prof. Dr. Otmar Hilliges for enabling this passion-driven BSc. thesis. Lastly, I would like to thank all participants for taking their time and providing valuable insights without which this thesis would never have been possible.

Contents

List of Figures

List of Tables

1

Introduction

A puzzle in the context of video games is conventionally any problem that the player has to solve inside the video game to progress, be it a visual, an auditorial or a dialogue-based problem, and should not be confused with 'Jigsaw puzzles' which are just one possible puzzle type.

Most puzzle games serve as a recreational activity, although they are increasingly used for other purposes, such as for the gamification of jobs [Mohammadi 2014], for reframing educational problems as puzzle games [1] (also [Lee et al. 2014]) or for using them as training grounds for general artificial intelligence [Perez-Liebana et al. 2016].

To get a better picture of why people play puzzle games, we conducted a need-finding survey A.1 where we asked participants, amongst other questions, why they like to play puzzle games. We received the following responses:

"to feel smart", "experience aha-moments", "increased spatial reasoning skills", "to get into the state of flow", "because it is fun".

Prior literature found similar reasons. In [Kangas 2017], Kangas analyzed the pleasure of puzzles in the context of adventure games and highlights Csikszentmihalyi's work on flow [Csikszentmihalyi 1990], also referencing flashes of insights and a cycle of suspense and relief as key reasons for player engagement. If the puzzles are too difficult, the player gets frustrated, when the puzzles are too easy, the player gets bored. It is only when the difficulty is just right that the player can get immersed in the challenge and derive pleasure from playing the puzzle. Kangas particularly mentions that puzzles can also be used as part of a broader experience and can help with immersing the player inside the game world by putting them in a state of flow.

[1] `https://www.euclidea.xyz`

1.1. Sokoban

Video games allow for puzzles that cannot easily be played on pen-and-paper (contrary to say Sudoku) since they can have an interactive element to them. A video game could, for example, hide information from the player and only reveal it at a later stage, change the game state while the player tries to solve the level or make it a lot easier to trial-and-error solutions.

One of the arguably first puzzle video games to take advantage of this was Sokoban invented by Hiroyuki Imabayashi in 1981. Inspired by warehouses, in Sokoban, the player controls a person whose job is to push crates/boxes onto storage locations (see image). Since its inception Sokoban has spawned hundreds of successors [2] and has inspired countless others. It has also been a popular choice for research, a quick search on Google Scholar reveals roughly 2'040 papers that mention Sokoban at the time of writing, hinting at its popularity.

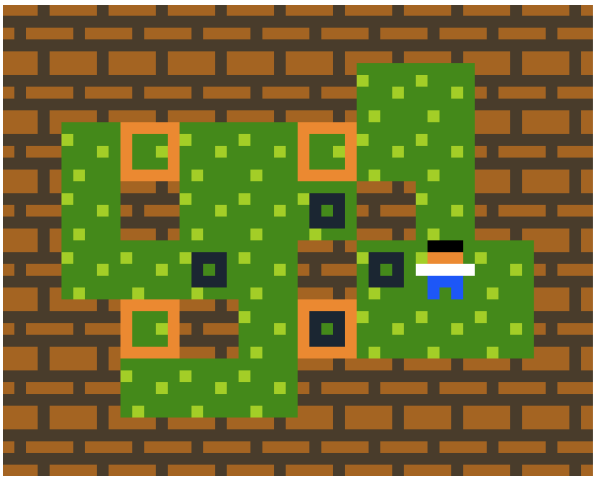


Figure 1.1.: *A Sokoban level taken from [Radek 2011]*

Solving a Sokoban level requires much trial-and-error, making it less suited to be played on pen-and-paper. To get a sense of the difficulty, according to [Radek 2011], the puzzle in Figure 1.1 already takes a median solving time of 43 mins.

For this reason, it only seems natural for designers to correspondingly use tools outside of pen-and-paper when designing Sokoban levels. From our need-finding survey A.1 we know that in practice puzzle designers use a diverse number of approaches and tools. Commercial games have likewise been made with a diverse set of approaches ranging from fully-automated approaches (like Donnantuoni's Dis Pontibus[3]) to approaches which are almost exclusively based on pen-and-paper (like Blow's `TheWitness`[4]). Our need-finding survey also, however, showed that designers are open to trying out new interactive methods to add to their design process.

One such tool for designing puzzle games is PuzzleScript[5] from Stephen Lavelle, a tool all 5 participants from the need-finding interview use, which comes with a level editor (a graphical

[2] `http://www.onlinespiele-sammlung.de/sokoban/list-of-sokoban-games.php`
[3] `https://marcosd.itch.io/dispontibus`
[4] `http://the-witness.net`
[5] `https://www.puzzlescript.net`

tool to place the blocks easily) and a run mode (a mode to play-test the designed level). Apart from that it does not provide further interactions to create puzzles for puzzle design.

1.2. Aim of the study

This brings us to the aim of this study, which is to develop interactive tools that bring 'genuine value' to the process of puzzle game design, by which interactions are meant, which the designer could not easily attain through direct manipulation (say by using a level editor). For this purpose, we created a mixed-initiative creative interface (MICI) for PuzzleScript called *MixedAim* that allows a tight interaction between the designer and the tool when creating Sokoban-like puzzle games. The reason we choose PuzzleScript was so we could support a broader range of games without giving up much functionality.

[Koch et al. 2019] have shown that computer tools which are both controlled by the human and by the machine, so-called mixed-initiative tools, can be employed successfully to creative tasks like designing moodboards. Moodboards are a collection of images meant to communicate a theme and inspire people who see it. The tool suggests images based on the already designed moodboard, and the user can steer the suggestions using three buttons 'more like this', 'not this one' and 'surprise me'. Similarly, puzzle games are a collection of levels intended to be fun and designed to make the player better at solving them. Our tool MixedAim gives suggestions on how to change a level and allows designers to change the type of suggestions via buttons like 'modify this level more' or 'change only the walls of this level'. Furthermore, it allows experienced designers to specify their own ways of steering suggestions explicitly. MixedAim then tries to suggest interesting levels based on these constraints to the user. But what makes a good suggestion?

1.3. Formal theory of fun

The puzzle games we concern ourselves with are comparable to formal systems, the keyword being *formal* meaning that every state and action inside the game is *well-defined*. In this analogy, the player starts from a well-defined starting configuration/state (theorem), applies inputs to the puzzle game which lead to well-defined state changes (derivation rules) to achieve a possible well-defined goal configuration (axiom). Usually, puzzle games can have multiple goal states, but they only have a single starting configuration.

In the literature of puzzle games [Salen and Zimmerman 2004] these well-defined state changes (derivation rules) are called *operational mechanics*, i.e., the mechanics which are programmed explicitly into the game to make it function (hence the term operational). While playing the game, the player identifies *constituative mechanics* and starts using them in addition to *operational mechanics* to find a solution. For example, in Sokoban, as soon as a crate is pushed next to the right-most wall, there is no way of pushing the crate back from the wall.

Usually, people concern themselves with formal systems because solving a formal problem (proving a theorem) has useful implications in the real world. For puzzle games, on the other

hand, as we have mentioned, it is not very clear why people solve them and which puzzles are more interesting than others.

The best explanation we have found so far comes from Jurgen Schmidhuber's formal theory of creativity, fun, and intrinsic motivation [Schmidhuber 2010]. In a nutshell, the human player is modeled as a reinforcement learner, and the learning process (in this case identifying constituative mechanics and creating a smarter process for solving a level based on these mechanics) is the human player's fun. This also matches with some of the responses of our need-finding study: *"because it is fun.", "experience aha-moments"*.

In more detail, the human player needs to utilizes a certain amount of energy (both mentally and through interaction with the system) to find a goal state. We call this the *subjective difficulty* or correspondingly the *subjective simplicity*. Measuring the *subjective difficulty* quantitatively is difficult since some players stare at a level for a long time before solving it in one go while others start using pen-and-paper to solve a puzzle. It is a lot easier however to measure the energy/the number of operations of a computer program (or as in Schmidhuber's case a reinforcement learner) has to expend in order to find a solution. Schmidhuber also calls this measure the *subjective momentary simplicity* or *compressibility* or *regularity* or *beauty* depending on the problem/context.

Next, while playing a single level, the human discovers constituative mechanics and new strategies of solving levels with these mechanics. This level of discovery is the *subjective interestingness* of a level. Notice that if a player plays the same level over and over the level will start to be less and less interesting to the player. Again quantifying this is very difficult for humans but can be done easily for computer programs (specifically for learning programs). Schmidhuber measures this by subtracting the *subjective difficulty* before solving the level and after solving the level. This way he quantifies how much the program (usually a reinforcement learning program) has learned from playing a level. Schmidhuber also calls this measure *novelty* or *surprise* or *aesthetic reward* or *aesthetic value* or *internal joy* or simply **fun**, again, depending on the problem/context.

Unfortunately, reinforcement learning algorithms have not been successfully applied to Sokoban, and, to our knowledge, the best Sokoban solvers are not in any shape or form 'learning'. Still, we can measure the difficulty of a level by the number of states an algorithm has to explore in order to find a solution. Since all fun puzzle games feature difficult levels, we had a hunch that difficulty was a useful measure to design fun games.

Based on this, our final research question is: How can interactive tools, specifically tools which focus on providing more difficult level suggestions, aid the user in designing puzzle games?

2

Related Work

In creative tasks, such as puzzle design, designers do not always know from the start what they are trying to create. This makes it a challenging task to automate fully (blindly optimizing some difficulty measure can reduce the perceived fun, for example).

Generating Sokoban Levels

Arguably the first group to have written on Sokoban generation is [Murase et al. 1996]. Here they generate the puzzle game from a preset prototype level which is modified using templates (2×2 or larger sequence of blocks) into levels. They then check the solvability of the level using a breadth-first-search and evaluate them based on a few criteria: length of solution, number of changes in directions of pushing & number of detours in a solution sequence. This approach of generating, solving, and then curating is at the heart of most other puzzle generation we encountered in the literature and is also reflected in our method for designing puzzle games interactively. [Radek 2011] did a human case study on Sokoban problem solving and suggest their own measurement of difficulty, a state-space bottleneck. They also show that humans make use of problem decomposition and that decomposability is a useful difficulty metric and generalize their results to other games [Jarušek Petr 2011].

Another approach for automatically creating Sokoban levels is from [Taylor and Parberry 2011]. Here they use a reverse search method and try to find the state farthest away from the goal position of a random solved level. This is elegant because it does not require more time than it would take to solve the level using breadth-first-search. [Kartal et al. 2016] use a very similar approach but start from a Sokoban level filled only with crates and walls (without the targets for the crates) and then start to let the player move the crates to find a difficult starting position and put the targets at the place where they moved the crates. Instead of finding the state farthest

away, they use a Monte Carlo Tree Search method to find such a level. Both of these methods are quite domain-specific and do not necessarily generalize well to other types of games without some effort.

Generating Puzzlescript games was successfully attempted by [Khalifa and Fayek 2015] using a genetic approach. [Chong-U Lim and Harrell 2014] took this a step further to not only generate levels but entire PuzzleScript rules for the games. They also use a genetic approach but apply the mutations on the ruleset starting from an initial population of empty rules.

Solving Sokoban Levels

Solving a Sokoban-like game, to begin with, is already a vast research topic. In theoretical computer science, E. Demaine analyzed what he calls 'pushing blocks' games and proves that many are NP-hard in [Demaine et al. 2003] and similarly together with Hearn proves what he calls 'sliding block puzzles' are PSPACE-complete in [Hearn and Demaine 2005]. Sokoban specifically has even been proven to be PSPACE-complete [Culberson and Culberson 1997]. [Viglietta 2012] and [Aloupis et al. 2015] proved that many other puzzle games and video games with agents are computationally hard to solve. [Williams 2017] features a particularly compact construction showing that the simple game of MazezaM (another Sokoban-like) can require an exponential number of moves in order to be solved.

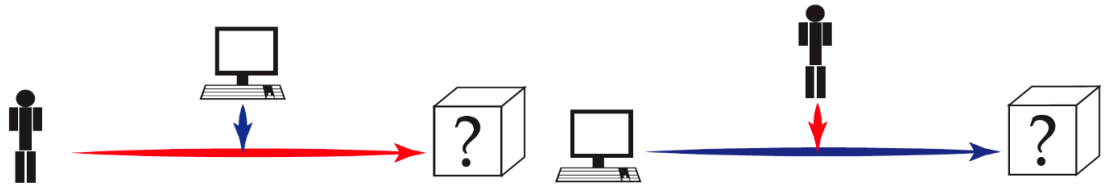
In general, no free lunch theorems apply for single-agent search problems, meaning that no algorithm works best on all search problems. If one has an algorithm that performs well for a specific subset of problems, it must do worse than random search in another subset. This result has not stopped researchers from trying to incorporate domain-specific knowledge towards specific search problems. To our knowledge, the best-known solution for solving Sokoban levels is still based on Junghanns Ph.D. thesis [Junghanns and Schaeffer 1999], a conclusion shared by [Froleyks 2016].

Solvers based on Monte Carlo Tree Search (MCTS) have not performed very well on Sokoban because Sokoban levels involve long-term planning, something that MCTS is still very bad at. A recent work [Racanière et al. 2017] goes into a promising direction by enhancing a reinforcement learning algorithm with imagination and is worth mentioning here.

Mixed-initiative systems

The term 'mixed-initiative interaction' is an umbrella term for works where a human interacts together with an agent to achieve some goal. In [Horvitz 1999], Horvitz lies down principles on how one should approach designing such a system. Mixed-initiative methods have been employed to address such problems with uncertain objectives (like Puzzle design) by presenting multiple solutions to the user and prompting the user for a decision during optimizing (see below for examples specific to game design).

[Liapis et al. 2016] provide a good overview of various systems which have been employed in research/practice to generate game content in a mixed-initiative setting. They make a useful distinction based on the agency of the tool [Figure 2.1]: Either the human takes the design lead, in which case the computer tries to help shape the idea, or the computer takes the design lead,



(a) Computer-aided design: Humans have the idea, the computer supports their creative process

(b) Interactive evolution: The computer creates content, humans guide it to create content they prefer

Figure 2.1.: *Two types of mixed-initiative design, taken from [Liapis et al. 2016]*

in which case the human guides the system towards solutions. Mixed-initiative tools can land anywhere on this scale. Our tool can be used in both ways as we will see in the user study.

Mixed-initiative systems have been used for generating various components in games. We are aware of the Sentient Sketchbook for strategy game map generation [Liapis et al. 2013], the Tanagra system for 2d platformer designs [Smith et al. 2011] and the dungeon creation system from [Baldwin et al. 2017].

Furthermore, for puzzle games in specific, M. Shaker et al. created a mixed-initiative system called Ropossum used for creating Cut The Rope levels [Shaker et al. 2014] & [Shaker et al. 2013], in which they pioneered such interactive tools for physics-based puzzle games. Similarly to the automatic approaches, they also distinguish between the generator, the solver, and the curator, with the difference that the generator uses a Grammar Evolution technique that prompts the user to guide it.

Additionally, [Butler et al. 2013] made a mixed-initiative system for Refraction, which is a puzzle game that lends itself well to a logical formulation. In their system, they were able to formulate some concepts that the player should learn in order to solve a level. From there, their tool generates progressions of levels that contain more and more of these concepts. The mixed-initiative aspect involved tweaking this progression by letting the user choose which concepts should appear in which level of the progression.

All of these mixed-initiative tools target a particular set of problems, which make their methods hard to generalize for other games. This has also been noticed by [MacHado et al. 2018], who also mentions all these mixed-initiative systems. They complain about a *lack of generalisability* in current results and a *lack of empirical justification* mentioning that there is a dearth of literature on the human factors of mixed-initiatives systems designed for game development tasks, something we try to address in this study.

To address *lack of generalizability* they have created the game description language VGDL. On top of it the created Cicero, a tool that supports the following key features: agent-based testing for automatic gameplay testing, replay analysis for storing and playing back gameplay sessions, play trace aggregation to visualize where players and agents tend to move on the game map and a mechanics recommender which retrieves mechanics from a large variety of games and Kwiri which is a tool to analyze why certain actions happen which is useful for debugging

moves. Another one of these broader frameworks is [Osborn et al. 2013] who has created the language *Gamelan* inspired by board game rules, for modeling, among others, card games like Dominion. Also, *BIPED* was yet another mixed-initiative system developed by [Smith et al. 2008] for creating prototypes more quickly. Both of these frameworks have to our knowledge unfortunately not been used outside of their respective study. In the end, I think we singled out a right balance of generality by focusing on grid-based puzzle games with agents in this research.

Regarding *lack of empirical justification* for using such tools to our knowledge only the recent work by [Guzdial et al. 2019] has done a think-aloud user-study on their mixed-initiative system *Morai maker* for creating Super Mario Bros. levels. We will highlight shared insights within our study. Their study was carried out from a sample of students of which only a small subgroup has designed game levels before, whereas our study is exclusively done with people who have spent a considerable amount of time with puzzle game design. They mentioned that Morai Maker was either used as an unintentional inspiration source or an intentional means of getting over a lack of ideas.

[Nelson and Mateas 2009] gathered a requirement analysis of video game design tools. They mention Sutherland's claim of "it is only worthwhile to make drawings on the computer if you get something more out of the drawing than just a drawing", a claim we hope to make about this tool as well. They continue to mention [Lawson and Loke 1997]'s five roles for a mixed-initiative system in the design conversation: learner, informer, critic, collaborator, and initiator. In this study, we mostly focus on the aspects of informing, criticizing, and collaborating, although all five aspects are present in one form or the other.

Furthermore, [Nelson and Mateas 2009] mention the term *window dressing* which is the act of removing redundant solution paths and the distinction between *operational mechanics* and *constitutive mechanics*. They also mention the lack of a design vocabulary and that having one would make further research clearer and more approachable. For this reason, we try to frame our results in their terms. Finally, they also address [Giaccardi and Fischer 2008] suggestion that tools ought to support *problem framing* as well as *problem solving*. This is in line with the research on *creative constraints* that suggest that limiting the possibilities can minder the *paralysis of choice*. We will see concrete ways on how designers constrain themselves in the user study chapter.

3

MixedAim: The Mixed-Initiative System

MixedAim is the tool we developed with which designers can interactively design PuzzleScript levels. The binaries and source code for MixedAim is published at `https://dekeyser.ch/mixedaim`.

3.1. PuzzleScript

The programming language with which designers can implement puzzle games in MixedAim is PuzzleScript[1] developed by Stephen Lavelle (also known as increpare). In the appendix Figure A.2, we have published a full Sokoban implementation in PuzzleScript for reference.

PuzzleScript is grid-based, and each PuzzleScript game consists out of the following seven sections.

Objects In here, the objects of the grid and their style are declared. For Sokoban, one would have to declare the following objects *Background, Walls, Player, Crate, Target*. *Player* and *Background* are special objects and always need to be declared.

Legend In the legend, three kinds of names can be specified.

1. Define a synonym for an object.
   Example: `# = Background`
2. Define an aggregate, which is a symbol for referencing a tile that includes multiple

[1] `https://www.puzzlescript.net` @ PuzzleScript

```
========
OBJECTS
========
Wall
BROWN DARKBROWN
00010
11111
01000
11111
00010
```

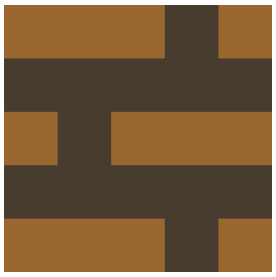
⟹ 

Figure 3.1.: *Objects section*

objects. It is not possible to create an aggregate from objects which belong to the same collision layer.
Example: `@ = Target and Background`

3. Define a property, which references any tile that contains any of the objects included in its definition.
Example: `Obstacle = Wall or Crate or Player or Target`.

It is worth mentioning that it is not possible to define aggregates from properties or vice-versa.

Sounds can be used to add chip-tunes generated with Bfxr [2].

Collisionlayers define on which layer which objects are stored. In PuzzleScript a level is rendered by multiple two-dimensional layers layered on top of each other. In this way one can specify that both a *Player* and a *Wall* cannot both occupy the same tile while a *Crate* and a *Target* can.

```
================
COLLISIONLAYERS
================
Background
Target
Player,Wall,Crate
```

⟹ 

Figure 3.2.: *Collisionlayers section*

Rules The rules are deceptively simple and are structured in the following form `Match -> Replace`. *Match* consists of either rows or columns that have to be matched and *Replace* must be of the same size stating with what the objects will be replaced.

Every object can come with force attached to it (highlighted in green by the examples). Forces are used to record player inputs. When the player presses the right button, all *Player* objects get the right force assigned to them. In the end, when all (non-late) rules have been applied, each object will try to move in the direction of their force. This will succeed provided there is no object of the same collision layer in that direction.

[2] `https://www.bfxr.net`

The rules of Sokoban can be written with a single rule:

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

This rule is implicitly expanded in 4 directions, so it works both on columns and rows:

```
RIGHT [right Player | Crate] -> [right Player | right Crate]
+ UP [up Player | Crate] -> [up Player | up Crate]
+ DOWN [down Player | Crate] -> [down Player | down Crate]
+ LEFT [left Player | Crate] -> [left Player | left Crate]
```

To match multiple rows/columns at once, one can use multiple boxes. Take the following rule as an example. If the player is on top of a key and there is a door on the map, then remove the key and the door. `[Player Key] [Door] -> [Player] []`

Lastly, every rule can execute *commands* if the rule is executed, such as playing a tune defined in the sound section, stopping the current moves (*cancel*), replaying the rules another time (*again*) & more.

Additionally, one can use properties and aggregates within rules, use the keyword *No* to make sure no object/aggregate/property matches, use an ellipsis for arbitrary distances, build groups of rules, make these groups rigid & a lot more. For more details please consult the PuzzleScript manual[3].

We also published a BNF grammar for a single syntactically correct PuzzleScript rule (without rule grouping) in the appendix Figure A.3.

Winconditions In this section, one can add multiple win-conditions which all need to be satisfied in the level state simultaneously to 'win' a level.

The rules can be any of the following: *No X, Some X, No X on Y, Some X on Y, All X on Y* where X and Y can be objects or properties and do what one would expect them to do.

Levels is the section where one can specify the puzzle problems, also referred to as levels. The previously defined objects/synonyms/aggregates with a single letter can be used to create a level.

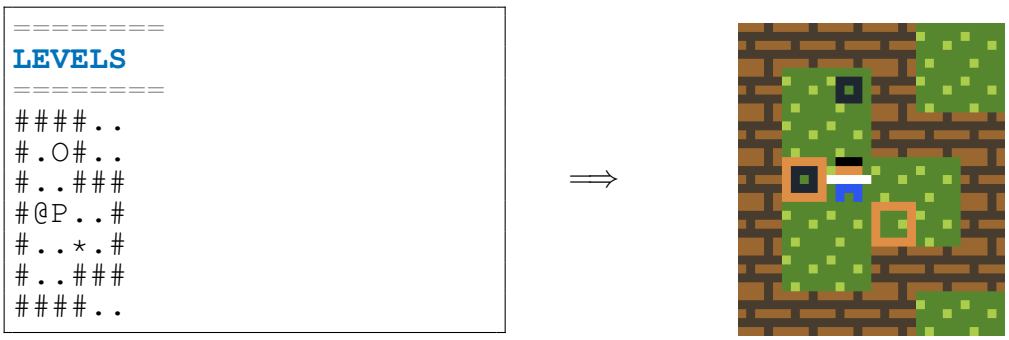


Figure 3.3.: *Levels section*

[3] `https://www.puzzlescript.net` @ PuzzleScript

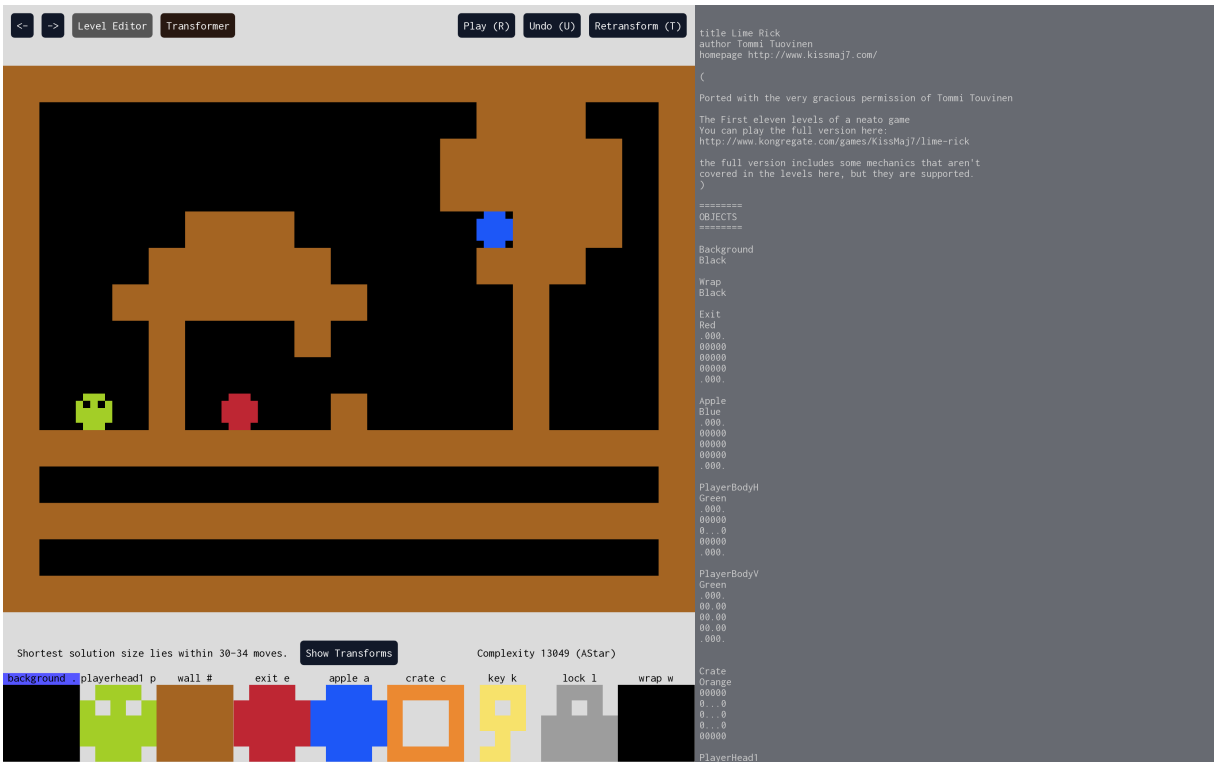


Figure 3.4.: *Level Editor Mode*

3.2. User interface

The mixed-initiative system we developed to create PuzzleScript levels is comprised of three modes of operation: The *level editor mode* [Figure 3.4], where the designer can place the blocks on the level, the *playtest mode* [Figure 3.5], where the player can test the level and see if it is solvable from a certain position and lastly the *transform mode* [Figure 3.6], where the designers can steer the passive suggestions which are shown both in the *level editor* and in the *transform mode*.

Information about the current level is passively shown at all times to the designer [Figure 3.7] and includes information like whether or not a level is solvable, and, if it is, in how many steps at least. Additionally, it also displays the difficulty of the level. This allows designers to quickly see whether a modification on a level they are working on is still solvable without needing to solve it themselves. In the next chapter, we will discuss in which ways designers exploit this when designing a level.

3.3. Solver & Difficulty

To check solvability, we employ three different types of solver: Breadth-first search, A* search, and Greedy best-first search. All of these search algorithms are complete, meaning they will find a solution eventually since the search state space of PuzzleScript is finite. The breadth-first search algorithm guarantees to find the smallest solution but is often too slow due to the size of

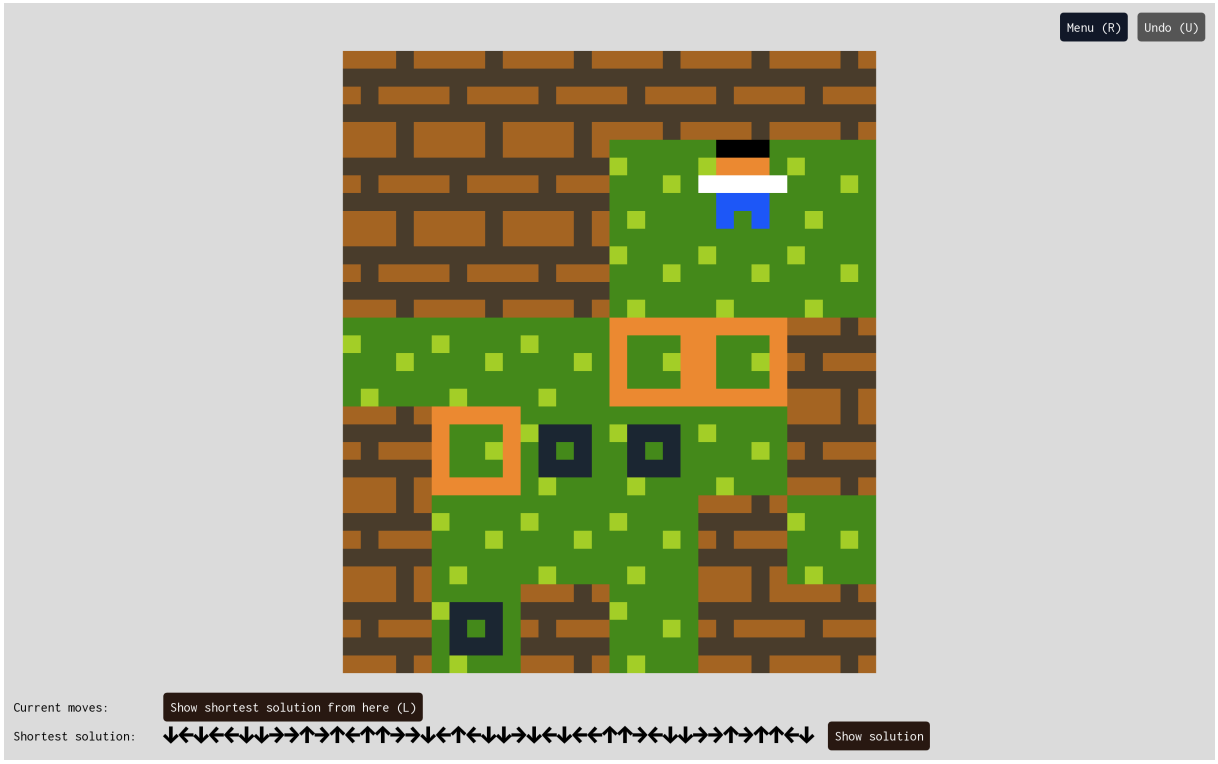


Figure 3.5.: *Playtest Mode*

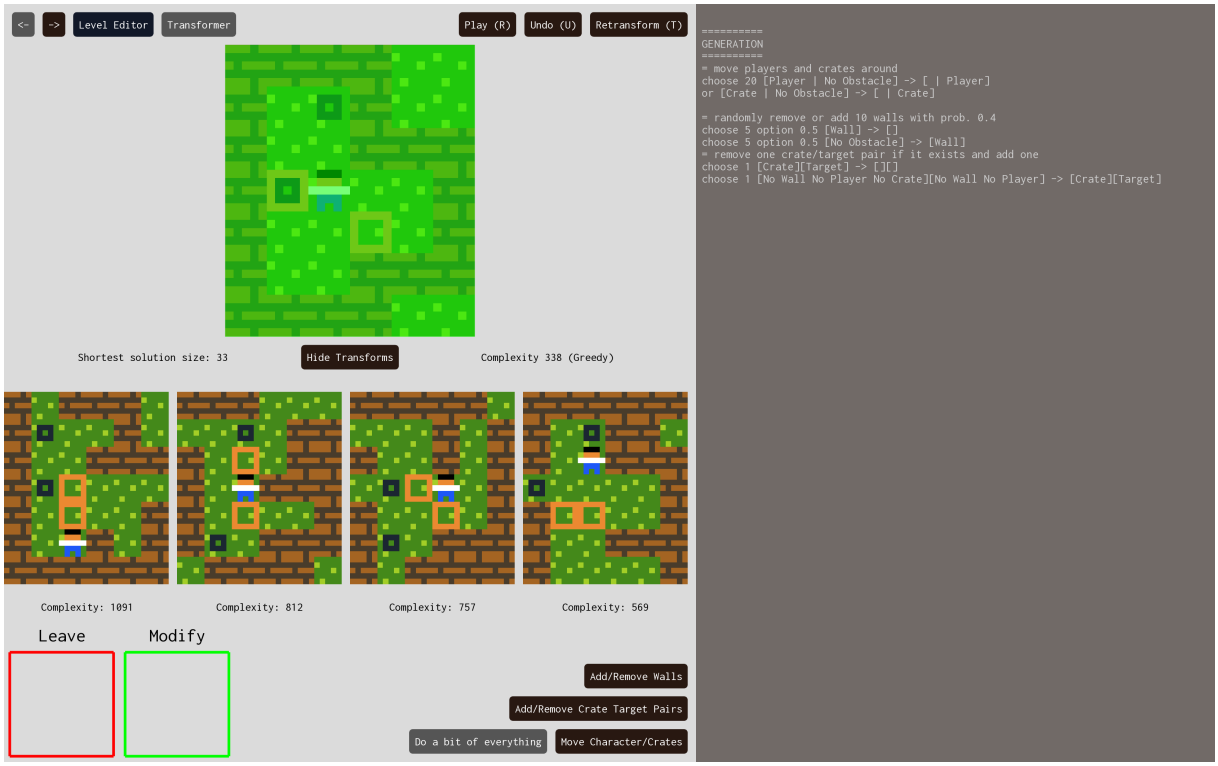


Figure 3.6.: *Transform Mode*

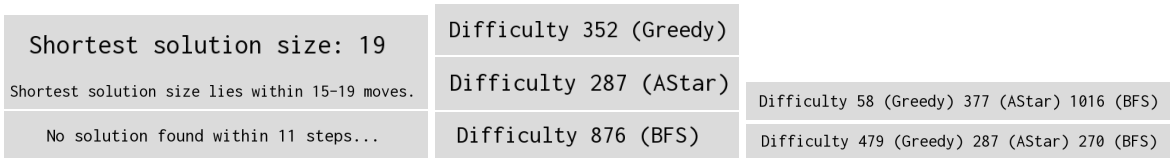


Figure 3.7.: *Passive information displayed to the user*

the PuzzleScript state space.

Both A* and Greedy search need a heuristic which, given a state, approximates the number of steps the state is away from a solution. Ideally, this heuristic never overestimates the number of steps, making it admissible 3.1. That way any solution found by A* will, like breadth-first search, require the least number of moves. If we only care about finding any solution, the greedy best-first search usually is the fastest.

$$\forall s_i \in \text{STATES} \,.\, h(s_i) \leq \text{min-cost-to-goal-from}(s_i) \tag{3.1}$$

It is a popular misconception that A* performs optimally under a heuristic h, in the sense that it expands the fewest number of nodes, compared to other search algorithms that also only rely on h and only search from the start. [Holte 2010] elaborates this in more detail, but this is not true if we store which nodes were already expanded (referred to as a closed set). In many puzzle games, like Sokoban, it is possible to end up in the same state via multiple paths. For example, moving up or moving left, up, right leads to the same state if no obstacles block the path. This might lead A* to reopen a state. If h additionally is consistent/monotone 3.2 however, then this claim is valid because A* does not need to re-evaluate already visited states / states in the closed set.

$$\forall n_i \in \text{NEIGHBOR}(s_i) \,.\, h(s_i) \leq d(s_i, n_i) + h(n_i) \tag{3.2}$$

Notice that not all puzzles profit from carrying a closed set. For example, it is not possible to reach the same state via different paths in a puzzle where one moves an ever-expanding snake. In this case, keeping a closed set is just a memory overhead. Further techniques like iterative deepening A* have been employed to address the cost of keeping such a closed set, but we have not implemented these methods.

In practice, the greedy best-first search outperforms A* in many cases and BFS in some other cases (tested games include Sokoban, LimeRick, Sokobond). When Greedy performs better than A*, then usually BFS performs even worse and when BFS performs better than A* then usually Greedy performs even worse making A* our default solver for the suggestions (see next section).

The heuristic we use for the A* and Greedy solver is a generalized version of [Junghanns and Schaeffer 1999] Sokoban heuristic. Since every crate has to be pushed on a target (and two crates cannot be on the same target) Junghanns observes that the minimal matching between crates and targets (the cost of matching a pair being the Manhatten distance) is an admissible heuristic for Sokoban.

The win-condition of Sokoban implemented in PuzzleScript is: `All Crate on Target`. Notice that there can be more targets than crates, so we want to find the minimal-cost maximum matching.

A PuzzleScript file can contain multiple win-conditions that all need to be satisfied. We compute the heuristic $h(s)$ for state s as the sum of all of the following win-conditions:

No X: As soon as X is nowhere on the level state this win-condition is triggered. We add +1 cost for every X appearing on the state s.

Some X: As soon as X is somewhere on the level state this win-condition is triggered. We add +1 cost if no X is found in the state s.

No X on Y: We add +1 for every tile that has both X and Y on it.

Some X on Y: We compute the closest X, Y pair in terms of their Manhattan distance and add their distance towards the cost function.

All X on Y: We compute the minimal-cost maximum matching on the Manhattan distance of all X, Y. Although such matchings can be computed in polynomial time, computing these still bring a large overhead. Instead of using this, we use a maximal matching, which is a matching obtained by greedily taking the X, Y pair with smallest Manhattan distance. It is well-known that a minimum cost perfect maximal matching is at worst twice as large as a minimum cost maximum matching. For this reason, we compute the cost of a maximal matching and divide the cost by 2. In this way, the heuristic will not overestimate the cost assuming that objects can be moved only one tile per player move.

Notice that this heuristic is not always admissible depending on the rules. This is not a problem in most cases. For example, in Sokobond, a game where one needs to connect atoms until all electrons have bonded, the win condition is *No Orbitals*. In this game, it is possible to bond multiple orbitals with a single move. The heuristic only ever overestimates the solution by a constant factor of at most 3 (since one can connect up to 4 atoms with a single move). For Sokoban, the heuristic remains admissible.

As we have mentioned in the introduction, the *perceived difficulty* of a level depends on the solver and is subjective. We approximate this difficulty by counting the number of states which any of the search algorithms needs to explore in order to find a solution. Finally, we expect that a level which is easy to solve for one of the mentioned search algorithms also tends to be easily solved by human players. Thus our difficulty metric is simply the minimum complexity of all mentioned search algorithms.

```
difficulty := min(diff(Greedy), diff(AStar), diff(BFS))
```

This approach is not the only approach to approximate difficulty. [Murase et al. 1996] gauged the difficulty of Sokoban levels through a combination of the parameters: length of solution,

number of changes in directions of pushing of minimal solution & number of detours in a solution sequence. In the suggestions, we mention a way in which designers could choose their own curation criteria for their puzzle game.

Another interesting idea comes from [Radek 2011], [Jarušek Petr 2011], who noticed that most Sokoban levels have a bottleneck. It is possible to look at the graph of all possible paths leading to a solution and notice an hourglass shape. The value of this bottleneck is the maximum flow from the start to the goal state. Both easy and hard games tend to have this bottleneck, and computing its value is not feasible except for small examples. However, this made them model the human player as a solver moving uniformly at random until it comes within a certain distance of the goal, starting from which it will more accurately find its way towards the solution. Notice that these methods will not make solving the levels any easier but might give a more accurate result on the level difficulty if the additional time can be afforded. We decided not to implement this as finding a solution is significantly easier than constructing the graph of all possible paths let alone computing the maximum flow on that graph but can be considered for future work.

3.4. Transformer

The transformer allows the user to steer the passively shown suggestions by letting the user specify rules on what valid suggestions are.

We discovered a neat way of doing this which was to extend PuzzleScript with two simple commands *choose* and *option* making it non-deterministic:

```
option 0.4 [Wall] -> []
```
will remove every wall with a probability of 0.4

```
choose 5 [Wall] -> [Crate]
```
will replace 5 walls chosen uniformly at random and replace them with crates. If there are less than 5 walls, turn all of the walls into crates.

```
choose 5 option 0.4 [Wall] -> []
```
will choose 5 walls uniformly at random and remove these with a probability of 0.4.

From these fundamental rules, the designer can create more elaborate transformations. For the user-study we provided four such transformations: *Moving Player/Crates, Add/Removing Walls, Removing/adding a target/crate pair* and finally *'do a bit of everything'* which is a combination of the other three methods.

Add/Removing Walls/Crates

Here we remove and add walls (on average we tend to add more walls instead of removing more walls as that seemed to provide better suggestions).

```
(randomly remove or add 20 walls with prob. 0.4)
choose 20 option 0.4 [Wall] -> []
or option 0.6 [No Obstacle] -> [Wall]
```

Move Player/Crates This preset moves around the crates and the player(s). Instead of using forces to move these objects, we can use place/replace rules to move them around. In this way, the objects can be moved more than one tile.

```
(move players and crates around)
choose 20 [Player | No Obstacle] -> [ | Player]
or [Crate | No Obstacle] -> [ | Crate]
```

Add/Remove a Crate/Target Pair First tries to remove a crate/target pair in the modify section and then tries to remove it.

```
(remove one crate/target pair if it exists and add one)
choose 1 [Crate][Target] -> [][]
choose 1 [No Obstacle or Target][No Obstacle or Target] -> [Crate][Target]
```

3.4.1. Suggestions

The transformer specifies valid possible level suggestions. However, the tool cannot show all these possible suggestions and needs a way to display the best levels to the user automatically. As mentioned in the introduction, we decided to use *difficulty* as a discriminator and passively show the four most difficult levels. More precisely, we use the notion of difficulty discussed in the previous section, namely, the minimum number of states any of the three solvers needs to explore before finding a solution.

Because solving a level takes time, we only ever use all three solvers on potential curated level candidates. We start by solving a level using only one solver and, if it turns out to be difficult enough to lie within the curated level candidates, we proceed by checking the difficulty with the other two solvers.

Additionally, we added a timeout to the solver, so from the point of view of the solver there are now three types of levels: Solvable levels, unsolvable levels, and levels on which the solver times-out before figuring out whether it is solvable or unsolvable. This requires a careful timeout balance in order to not skip solvable but interesting levels and to not waste computational time on unsolvable levels which are difficult to prove unsolvable. Initially, the timeout for each new level is set to a tenth of the time it already spent on trying to find a solvable level. As soon as a solvable level is found, the tool will always set the timeout threshold to 8 times the amount of time it took to solve the current curated levels. This method of increasing the threshold leads to a quick succession of freshly curated levels at the start, which slows down as it gets proportionally harder to find more difficult levels.

4

User Study and Results

We did two user-studies: One need-finding survey, which we discussed in the introduction, and one study to evaluate MixedAim based on a think-aloud session followed by a structured interview.

Both user-studies are qualitative rather than quantitative and have the purpose of informing, i.e., obtaining compelling research questions and finding interactions that provide 'genuine value' to the designer.

Quantitative studies are usually carried out on 'tried-and-tested' design approaches and serve to test a hypothesis or to compare the effectiveness of one approach towards other approaches. Due to the scale of the project and the dearth of similar tools, we decided that a qualitative study, specifically the think-aloud session, would provide more insights into the design of mixed-initiative systems for puzzle games. In hindsight, this turned out to be a good decision as our users found new ways of using the tool which we did not anticipate. While we did gather clickstream data, due to the different ways that users approached the tool, we were not able to draw meaningful conclusions from it.

This second user-study had 7 participants, 6 of which are very experienced puzzle game designers (with a median experience of 3 years). The remaining participant is an experienced game designer (5 years of experience outside of puzzle games). For more details on the participants, see A.1.

The user-study was carried out remotely via a video call and a screen capture of the participants' machine except for participant 6, who did the same user-study process but on our machine. The outline of the study looked as follows:

1. First, we asked the participants some questions related to their experience as a (puzzle) game designer (see A.1).

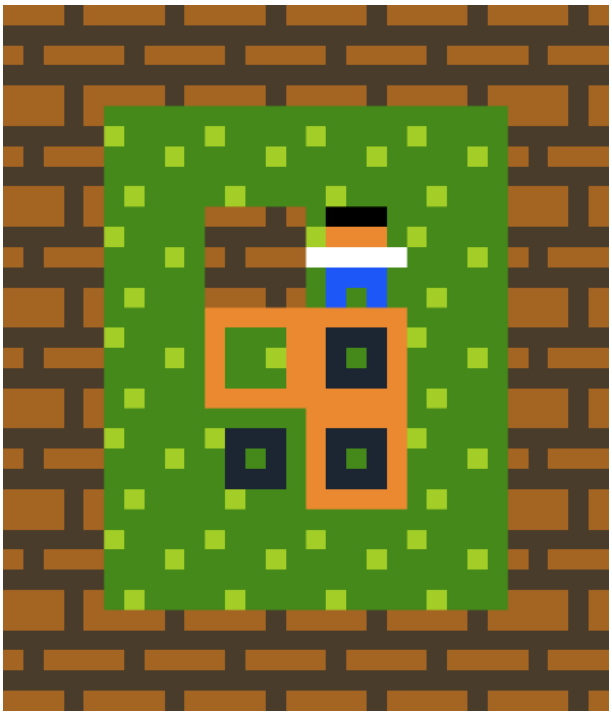


Figure 4.1.: *Sokoban level participants were prompted to iterate upon*

2. Second, we asked the participants to design one or more Sokoban-levels for 60 minutes and asked them to think-aloud their thoughts during the design process.
3. Third, we optionally gave the participant time to explore the tool further and use their PuzzleScript games to design levels. In particular, participant 1 & 4 have worked multiple days with the tool and have found interesting use-cases which we did not anticipate before proceeding with the final interview.
4. Lastly, we concluded with a structured interview.

4.1. Think-aloud study results

For the think-aloud study, we asked participants to design one or more Sokoban-levels. Since we are interested in seeing whether mixed-initiative systems can help designers iterate upon their design, we suggested to try turning the following Sokoban level 4.1 into a more challenging level. Some designers then used our mixed-initiative system the way we anticipated, while some participants found other surprising ways of utilizing the tool. For this reason, we decided to divide this section into the different styles in which the participants used the tool to design levels.

4.1.1. Iterative design

The iterative design approach was the primary way we intended users to design levels. We encountered this method in our need-finding survey and in blog posts of Sokoban designers [1]. The design process roughly looks as follows:

1. Have a set of rules for the puzzle game.
2. Create an aesthetically pleasing level / decide on a theme / find an interesting mechanic.
3. Iteratively turn this construction into an enjoyable level.

Most of our participants (2,3,4,5,6,7) have, among others, employed this method while designing their Sokoban level. A good way of illustrating how users iterated upon their design is to look at how participant 6 used the tool. The corresponding design can be seen here 4.2:

First, he started by employing the 'do a bit of everything' transformation on the presented Sokoban level to go from the initial design to a design with two crates and two targets and one additional crate at the right. He removed this crate and decided that he likes the two crates, two target configuration and planned to design a level around it. When prompted, he said he liked it aesthetically and because it had two interesting solutions.

Next, he decided to remove as much from the top of the level as possible (we think this is because he wanted to disable the solution through the hoop). He continued doing this until he found that the last wall he placed made the level unsolvable (as reported by the mixed-initiative system). As soon as this happened, he then decided to let the transformer make it 'somehow work' again (after saving a copy of the design). This was a pattern we frequently saw designers take: As soon as their design became unsolvable, they decided to let the transformer make it somehow work again through transformations.

From there, he picked a few that looked interesting and decided to build on one of them. Because our mixed-initiative system optimized for levels that have a lot of possible states (which are easily identifiable as dead ends to the player), it often generated levels with some unnecessary space. Our participant then finalized the design by removing these unnecessary dead ends. Again, this was a pattern we frequently observed: The transformer (or the designer) made a level which had redundant parts which then had to be manually removed.

In the next section, we try to list some of these re-occurring design patterns, analyze them, and compare them with the literature.

4.1.2. Design patterns

In this section, we list design patterns for the iterative design approach which we obtained from the think-aloud study and see in which way participants profit from mixed-initiative methods.

[1] `http://sokoban-jd.blogspot.com/2015/02/how-to-build-sokoban-level.html` @ How to build a Sokoban level – Serg Belyaev

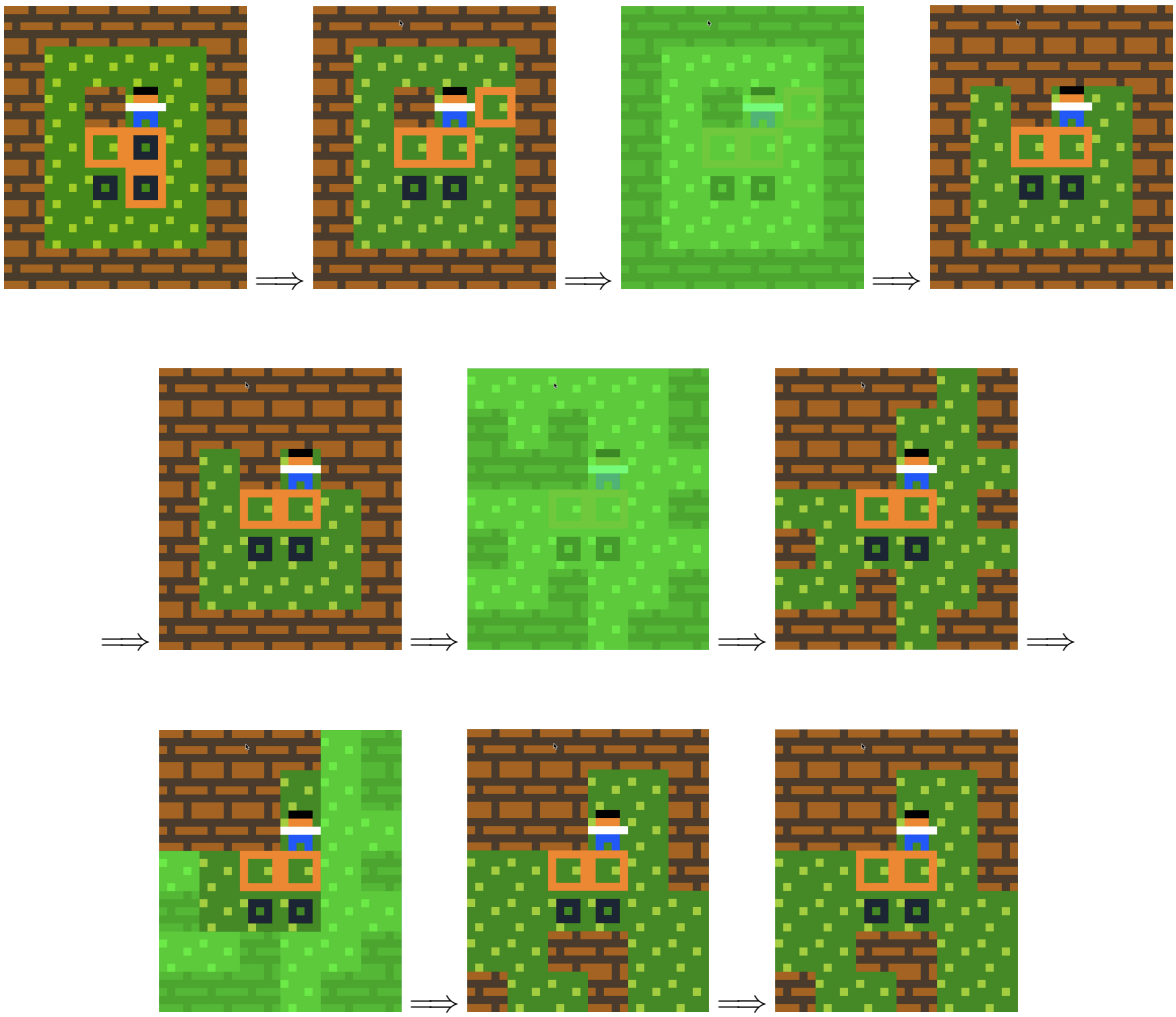


Figure 4.2.: *Participant 6: Iterative design processed*

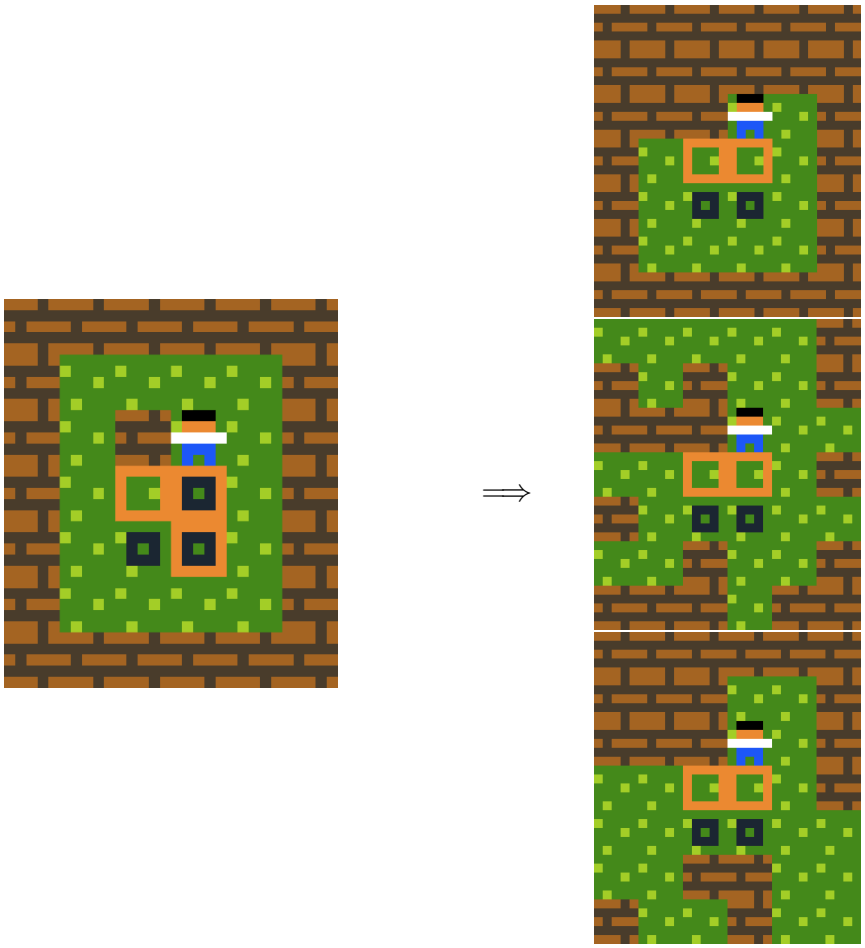


Figure 4.3.: *Participant 6: Final iterative designs*

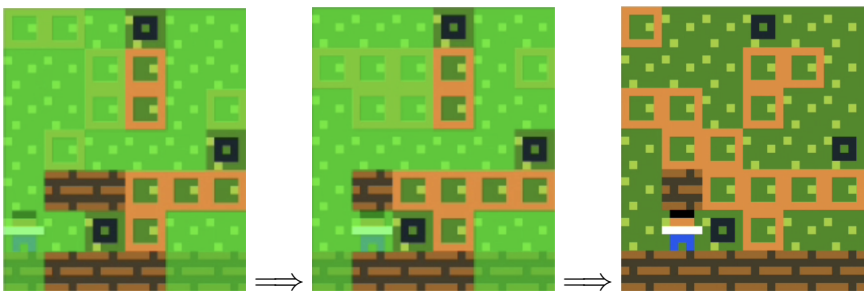


Figure 4.4.: *Participant 7: Iterative design process*

Identifying aesthetics/mechanics After using the tool for a while, many our participants have found *aesthetically* pleasing sections or *mechanically* interesting sections upon which they liked to expand on. For example, participant 6 liked the two targets and crates in the design mentioned above.

MixedAim helped the designer identify some interesting designs by passively suggesting levels during the design phase (5 participants claimed they got inspired by a level without clicking on it) or by the designer explicitly looking for them with the transformer. Especially participants 1, 2 & 4 focused on using the transformer to come up with original designs upon which they then were able to improve, see also the section 5.1.3.

Creative constraints These are situations where designers constrain themselves only to the use of a previously identified aesthetic or mechanic. Every participant worked with creative constraints in one way or the other.

Mixed-initiative tools helped the designer to satisfy *aesthetic constraints* by allowing them to *freeze* certain sections that the transformer would not be able to change. For example, participant 6 decided he would keep the two crates and the two walls and constrained the transformer to only change the level around the crate target pairs. Further suggestions for *aesthetic constraints* followed from the structured interview included adding tools for *symmetry* and being able to *combine* two levels in interesting ways.

For *mechanical constraints*, the transformer rules have to be employed to create interesting mechanics. These are for example the preset buttons like only moving the crates or removing/adding some walls and keeping the rest intact. Suggestions included not only *specifying* ways of transforming but also specifying means of constraining allowed transformations.

Unsolvable into solvable This was not something we encountered in the literature, but it was something that happened very frequently with our mixed-initiative system. As soon as designers were passively shown that the level was unsolvable, they did not spend the effort to make it solvable manually but told the tool to make the current design work somehow. A lot of these designers then reported being pleased by the methods the transformer came up with to make the level solvable again.

This was also mentioned directly by participant 3 during the think-aloud study: *"A perfect way to use this [tool] is if you find something that looks interesting, but you cannot quite get it to be solvable or something like that."*

Window dressing This happens when an interesting level is found (through the transformer or the designer) that contains a lot of unnecessary states. The designer then tries to eliminate all these unnecessary states (without making the problem trivial) to find the 'canonical form', the 'kernel' of the problem. If the level becomes too easy the designer can again add redundant pathways and see in which ways the level can be made more interesting again. It was mentioned by participant 1, 6 & 7, and clearly noticeable in participant 7's design process, see Figure 4.6.

Our mixed-initiative tools only help to address this problem in so far that they help the designer to edit the level quickly. Passive information about solvability can tell the designer whether or not his simplifications suddenly made the level easier, for example.

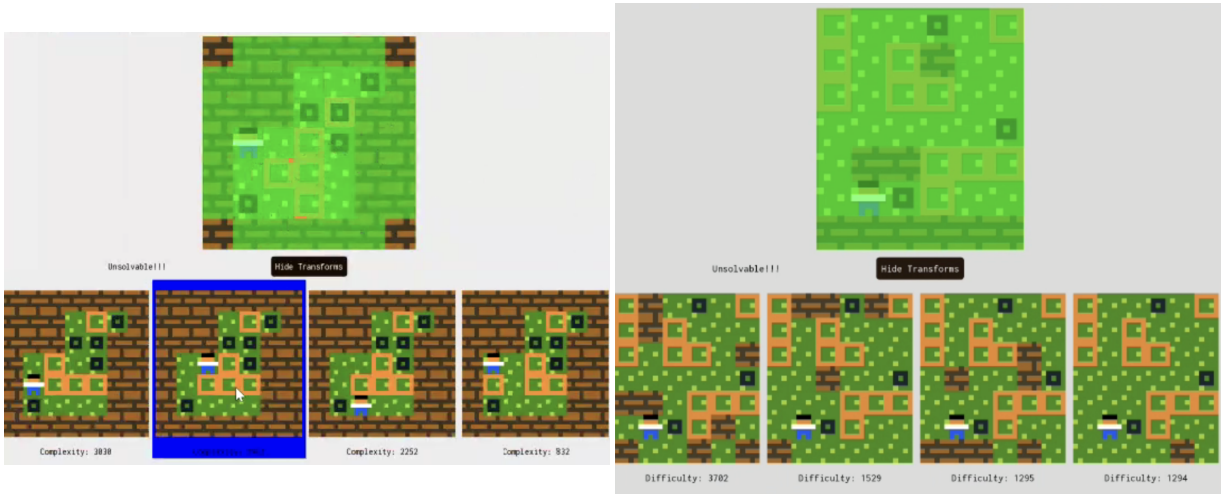


Figure 4.5.: *Participant 4 & 7: Unsolvable to solvable*

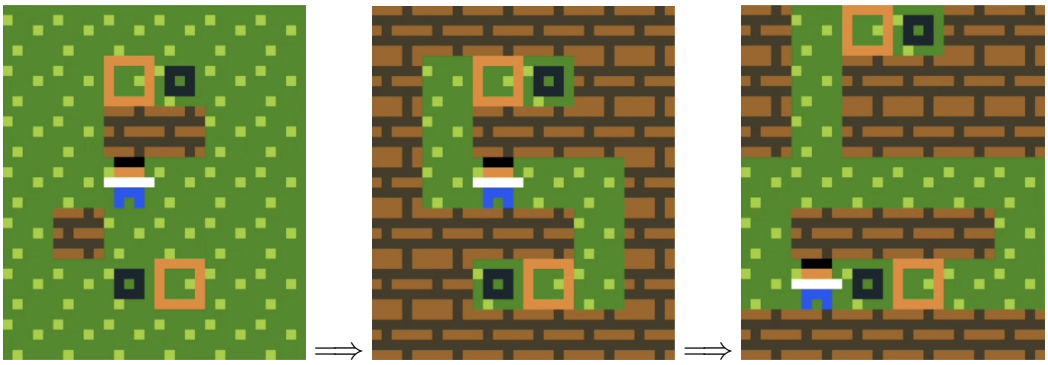


Figure 4.6.: *Participant 7: Window dressing*

Mechanic swapping Participant 1 already had an extensive library of Sokoban-like PuzzleScript games and used this to utilize the tool in an interesting manner. He loaded up a list of Sokoban levels into the tool and then started to change the rules of the Sokoban game. For example, he made it so players can push the block vertically but only teleport through the block horizontally. After doing this, the tool quickly showed him whether these levels were solvable under the new rules and allowed him to find novel solutions quickly.

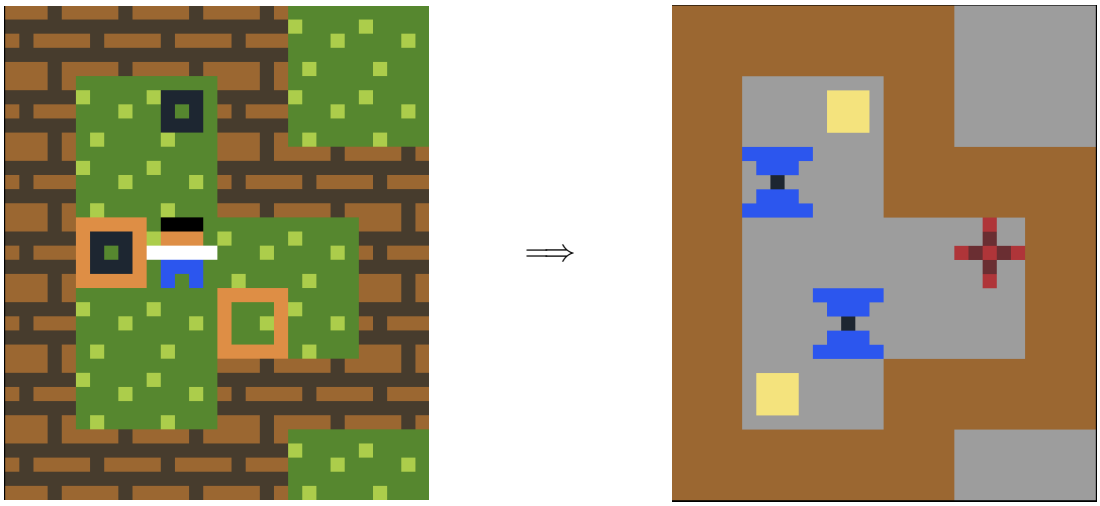


Figure 4.7.: *Participant 1: Mechanic swapping*

Gauging completeness During the iterative design process participant 6 mentioned that he found the suggestions to be a useful stopping criterion. As soon as the suggestions did not improve the level in surprising ways, he would feel confident enough to stop iterating on the level further.

"Yes, especially as a level of confidence, so I was sure that the current design was pretty solid when the suggestions seemed worse." – Participant 6

4.1.3. Adapting to the transformer

Every participant did, to some degree, modify their approach towards designing puzzle levels to incorporate the transformer. While our main focus was on how we can support the user in their design process of designing puzzle levels, some participants changed their approach towards designing puzzle levels significantly in order to exploit the strengths of the transformer.

We saw a clear difference in the perceived usefulness of the tool depending on which role the tool took in the design process.

Take as an example participant 5's approach when designing puzzle levels:

Participant 5 first decided that his Sokoban design should be created from an empty but large field. From there, he placed a few crates and walls but quickly noticed that the tool did not find interesting transformations. This was partially because our initial settings of the transformer only added a few walls on average making almost all levels trivial but requiring much pushing around (making the solver believe that the level is difficult).
He then added more crates and targets believing that it would relinquish this error, but now the complexity of solving a single transformed level was upwards of 20 seconds.
Thinking that it takes too long to solve a single instance of a level he then added walls around the level, which should have helped to reduce the time it took to solve a single transformed level but this allowed for more unsolvable levels which were difficult to prove unsolvable.
Lastly, he manually added a wall on the top of the level, making the level significantly more difficult to his surprise.

Participant 5 tried to fit the tool into his method of designing puzzle games. In the beginning, he decided he would make a large Sokoban level with many crates and reluctantly changed his plans after realizing that the tool did not do so well with these kinds of levels.

Participant 1 meanwhile had a few ideas at what MixedAim might be good at and tried to find the most interesting interactions with the tool and thus did not mind changing his approach towards designing puzzle levels.

The other participants fell somewhere in between, participants 2,3,6 & 7 used MixedAim mostly in the way we intended it to be used and did not have to significantly adapt their design approach while using the tool. Participant 4 also tried to fit the tool in his method of designing puzzle games but with success, see the backward design section for more.

These different ways of adapting to mixed-initiative tools have also been mentioned by [Guzdial et al. 2019], who identified a few types of participants themselves. One group of their participants attempted to fit every suggestion their AI made into a level, something we did not encounter, and other groups of participants who adapted their behavior to discover how best to interact with the tool, similarly to participant 1.

We identified the following adaptions in the way users designed levels and provide a measure for judging the usefulness of an adaption based on one of these findings.

Threshold-based searching Participant 1 mentioned that there is a *threshold* in the difficulty scores starting from which the levels become *interesting*. This made him quickly change the level or the transformation rules until interesting suggestions appeared. As soon as they did, he then waited for more suggestions to appear.

"[...] there seems to be a threshold starting from which the levels get interesting, at least that's how it feels like qualitatively."

The other participants, though not stating this explicitly, acted similarly, swiftly changing the transformation rules until interesting levels occurred. When they occurred, they waited for more. For example, participant 5 frequently mentioned that the levels do not look difficult and quickly changed the transformation rules or the level until good suggestions came along.

This threshold is entirely dependent on the level as the difficulty does not always corre-

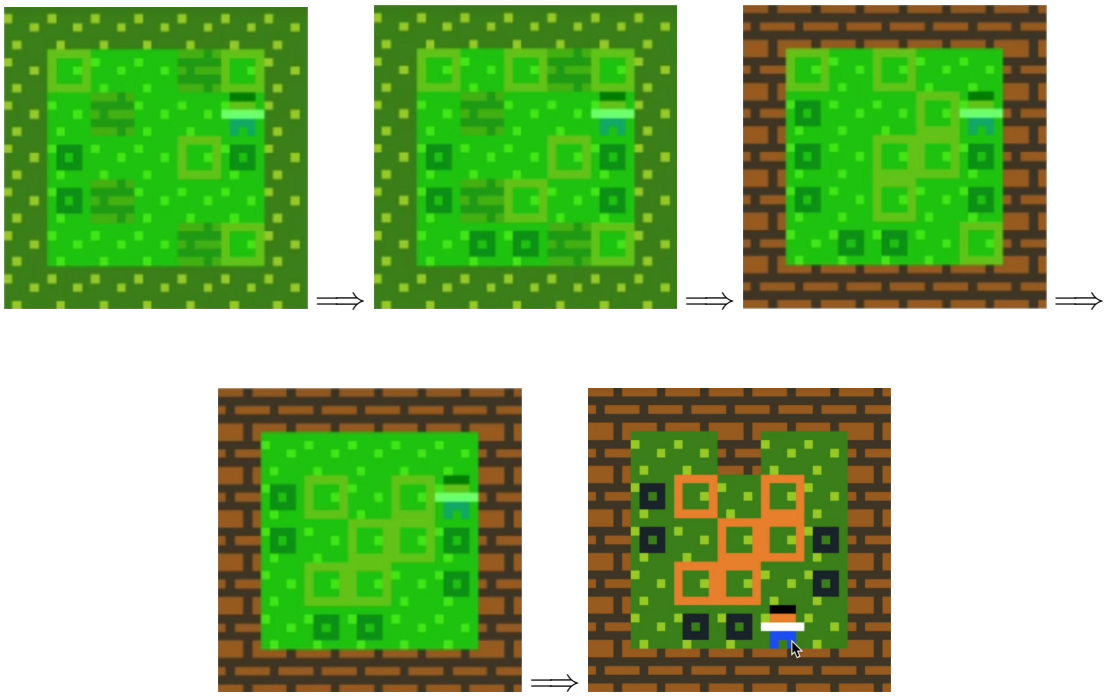


Figure 4.8.: *Participant 5: Adaptive design process*

late well with the perceived difficulty of the designer, something which was frequently mentioned (see negative feedback section). Sometimes this threshold is not reached due to it being difficult to solve the levels in time or because no interesting levels exist in the transformations but as soon as the threshold is reached practically all the levels are interesting.

With this insight we can model the usefulness of a transformation as the throughput of levels above this difficulty threshold and might give more insight as to why the other adaptions worked / did not work.

$$\text{usefulness of transformation} = \frac{1}{\text{avg. time for interesting level}} \tag{4.1}$$

$$\text{avg. time for interesting level} = \left(1 - \frac{\text{\# interesting/difficult transformed levels}}{\text{\# total transformed levels}}\right) \cdot \text{avg. solve time} \tag{4.2}$$

Freezing sections Participant 3 and 5 (among others) have chosen to freeze a few sections in the level, not because they necessarily thought they found these sections interesting, but to skew the search space to contain more interesting levels. For example participant 3 said quote *"let me help it a little"* when he froze parts of the transformer section, see Figure 4.9.

This adaption increases the throughput by increasing $\frac{\text{\# interesting difficult transformed levels}}{\text{\# total transformed levels}}$.

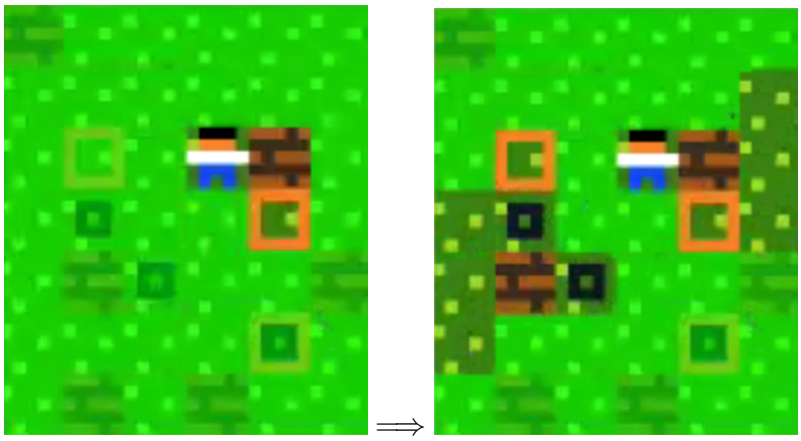


Figure 4.9.: *Participant 3 freezing sections to aid the transformer*

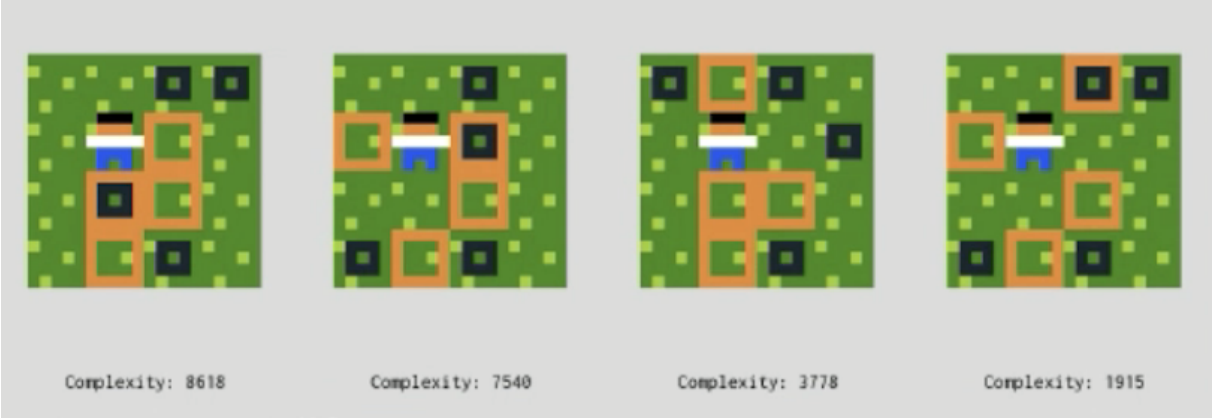


Figure 4.10.: *Participant 1's small Sokoban transformations*

Smaller levels Participant 1 concluded that the transformer is well-suited to create small Sokoban levels, see figure 4.10. He also experimented with variations of Sokoban rules and created a few puzzle games with small levels using this method ('All Green To Blue'[2] and 'All Green To Blue On Yellow'[3] which are playable online), see Figure 4.11.

This adaption increases the throughput by increasing the *average time for solving a transformed level*.

Larger levels Participant 5 decided that larger levels would be better for the transformer because it would allow for a larger quantity of interesting levels. However, this approach equally allows for more uninteresting levels and significantly decreased the *average time for solving a transformed level*, which made choosing larger levels a bad adaption to the system.

More variation Lastly, we observed that some games are better suited for identifying interesting mechanics than others. Both we and participant 2 have had such success with puzzle games that feature polyominoes.

[2] `https://www.increpare.com/game/all-green-to-blue.html`
[3] `https://www.increpare.com/game/all-green-and-blue-on-yellow.html`

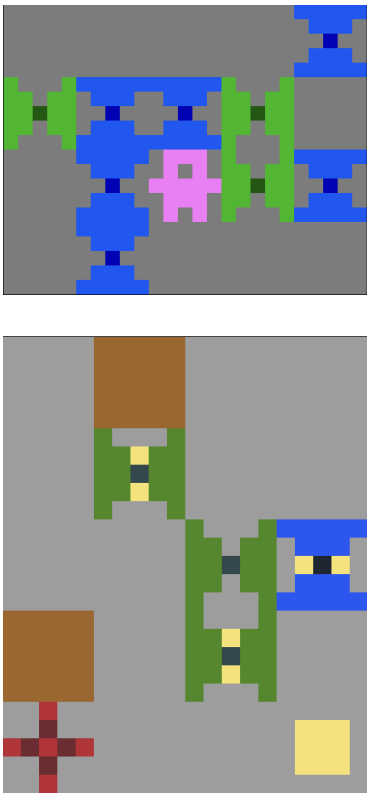


Figure 4.11.: *Participant 1's custom games*

This leads us to believe that Sokoban-like puzzle games with a lot of possible variations in a small space are especially well-suited for identifying interesting mechanics. When focusing on small levels it does not take a lot of time to find a solution or prove that no solution exists and would take at most in the order of $\mathcal{O}\left(\binom{n^2}{a}\right)$ computational steps, where a are the number of blocks in the generated level and n^2 are the possible places the blocks can be placed. Notice that using a variety of different objects (for example differently shaped polyominoes) does not change the value of a. Thus one can build more interesting small levels with puzzle games that have a lot of different types and shapes of objects, as opposed to games with only three types of 1×1 objects.

4.1.4. Backward designing

This method of designing levels with the mixed-initiative system was discovered and coined by participant 4 and was not an approach of designing puzzles with MixedAim we had foreseen. In backward design, one starts to design a level from a solved state and slowly turns it into a more complicated/interesting, but always solvable, level. Participant 4 used it specifically for their PuzzleScript game, see Figure 4.12, but this backward approach can also be used for Sokoban.

Both [Taylor and Parberry 2011] and [Kartal et al. 2016] have employed such a backward design approach for generating Sokoban levels. We will now illustrate how their approaches can be formulated as transformer rules in MixedAim with which designers could interactively backward

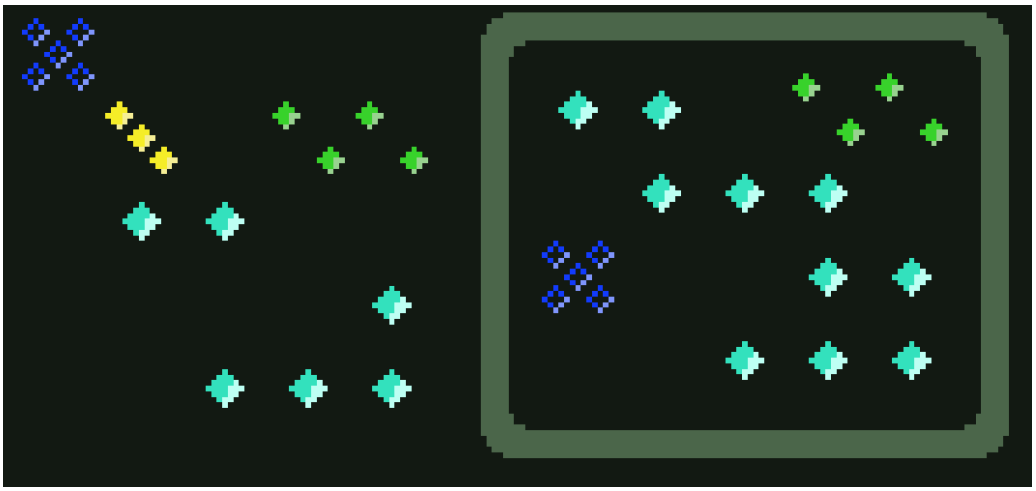


Figure 4.12.: *Backward design of participant 4's puzzle game*

design.

[Taylor and Parberry 2011] This assumes that the target positions are fixed and no crates are in the initial design. The player then pulls the crates away from the goal.

```
(place crates on all targets)
[Target No Crate] -> [Target Crate]
(move the player around and let him pull crates)
choose 50 [Player | No Obstacle] -> [| Player]
or [No Obstacle | Player | Crate] -> [Player | Crate | ]
```

[Kartal et al. 2016] This assumes that the crate positions are fixed and no targets are in the initial design. The player then pushes the crates around and at the end turns them into targets.

```
(add a placeholder to all crates)
[Crate] -> [Crate Placeholder]
(move the player around and let him push crates)
choose 50 [Player | No Obstacle] -> [ | Player]
or [Player | Crate | No Obstacle] -> [ | Player | Crate]
(replace crates with targets and placeholders with crates)
[Crate] -> [Target]
[Placeholder] -> [Crate]
```

4.2. Structured Interview Responses

For most participants, we took the structured interview immediately after the think-aloud study, depending on the preference in writing or orally. Participants 1 and 4 did the structured interview after using the tool for a few days testing it not only on Sokoban but also on their games. Since the questions are quite open-ended, we would skip a few of them if the participant had already answered them in a previous question.

We compiled a list of the most interesting answers to the questions in the appendix section A.4 but believe that the most important insights from the interviews have either been mentioned or are elaborated in the next sections.

4.3. Negative feedback

Unfortunately, not all participants had a very smooth experience with the tool. Especially participant 5 immediately decided that interesting Sokoban levels were large and would contain many crates which unfortunately the transformer did not handle very well. The designer then had to limit himself to designs which the transformer handled better.

He also admitted that while making levels with the tool can be more efficient, passively showing information can take the fun out of solving the level and one might be easily tempted to look at the solution instead of working it out in the head 4.13.

Participant 6 addressed the same concern mentioning that there is a 'Google effect' that makes him stop thinking about solvability (Google effect meaning that people do not attempt to remember a fact but instead remember how to look it up instead). He did not see this in a negative light though, and enjoyed that it made the process faster.

"Part of designing a puzzle is trying to make the best puzzle you can make, but part of it is also the fun of solving intermediate states yourself. So that's one thing that I would miss with the software (you might be able to work around it) but one thing I like about designing puzzles, is to constantly be thinking about solutions in my head, so this is kind of giving this up a little bit, and I'm not actively thinking about solutions when I'm designing."

"I can create better puzzles with this tool but might have less fun while doing it."
"There's a kind of Google effect where I stop thinking about solvability."

Figure 4.13.: *Participant 5 & 6's quotes regarding solvability*

Participant 1, who is very experienced with puzzle design, has a different mindset towards the transformer and thinks that it adds another tool for inventing levels.

Participant 1 fears that designers who have not had much manual puzzle-developing experience would end up making uninteresting levels, which are just 'interesting enough' to put into a game. [Guzdial et al. 2019] mentioned similar concerns that their tool could replicate an overused design. This gives an additional responsibility to the designer, who now has to resist relying too much on the system.

Participant 1's quote sums it up nicely: *"They're cool to work with [referencing the tools], but it's so easy to make bad and difficult levels (a fatal combo)."*

4.4. Suggestions

Use more sophisticated curators / more accurate difficulty measure A very common complaint (every participant except for 1, 5 and 6 mentioned it) was that we curated levels based only on the difficulty of the solver and that the difficulty of the solver also does not always match the perceived difficulty.

"The limitations are [the systems] judging abilities." – Participant 3

Due to time constraints, we decided from the beginning to keep the curator very simple. However, after interviewing participant 1 & 4 we have a suggestion for future work to improve upon the curator. We suggest adding an optional 'cost' modifier to operational rules.

For example: `[ > Player | Crate ] -> [ > Player | > Crate ] COST 10`

This way, levels that require more costly rules in order to be solved are given a higher difficulty score. The rule above would discourage levels that need a lot of moving around without pushing blocks.

Feedback for adapting to the transformer Participant 4 suggested showing how many levels have already been transformed to get an estimate of the effectiveness of the transformer.

We suggest to take this a step further and also show the throughput of curated levels and also show how many levels are solvable/unsolvable/timed-out, respectively. These changes would allow the user to better adapt their actions towards the transformer.

Unresponsiveness Many participants felt like the transformer was unresponsive at times, due to there being no visual feedback when MixedAim found no suitable suggestions. Participant 4 & 7 both suggested adding a button to explicitly generate the levels instead of automatically transforming them, while Participant 1 suggested labeling more clearly that the transformer was doing its work.

Improve UI A suggestion by participant 5 was that the tool should have an option to enable/disable showing passive information about a level since knowing that a solution exists can tempt you to click on it instead of solving it yourself, which is an aspect he would otherwise miss when designing puzzle games.

Level progression aid Participant 4 suggested that the tool should also help with creating an engaging level progression sequence. This is far beyond the scope of our work. However, [Butler et al. 2013] has promising work in this direction.

4.5. Conclusion

We developed the mixed-initiative system MixedAim for creating PuzzleScript games interactively with the computer. For this purpose we generalized the Sokoban heuristic of [**?**] to solve levels more efficiently, we crafted a user interface which supported various forms of direct ma-

nipulation, like using a level editor and play-testing the level and added support for interactive suggestions. We hypothesized and later confirmed that giving more difficult suggestions is a feasible way to assist the designer. Specifically, we have identified that there is a threshold in difficulty starting from which levels become interesting.

We identified various key interactions with those suggestions that designers and illustrated in which way designers could benefit from them. Interactions included *Identifying and constraining mechanics and aesthetics in designs*, *making unsolvable levels work again through transforming*, *window dressing*, *mechanic swapping*, *gauging completeness* and *backward designing*.

Furthermore, we analyzed how participants adapted to the MixedAim suggestions, identified that the perceived usefulness is dependent on the role of MixedAim in the design process and noticed that it was particularly good at identifying interesting mechanics in small levels with a lot of possible variations.

Through the user interviews, we identified that some designers experienced a 'Google effect' that made users delegate the responsibility of solving the level to the MixedAim system. Some view this as taking out the fun, while others think this is efficient. Lastly, we also received concerns that the tool can make it easy to create difficult and bad levels which give additional responsibility to the designer planning to incorporate such tools in their workflow.

Overall, we believe that mixed-initiative methods add value to the design process if they are approached as another tool in the toolset for creating puzzle games and not as the de facto replacement of one's usual design process.

A

Information For The Few (Appendix)

A.1. Needfinding survey

Why do people play puzzle games?
"to feel smart", "experience aha-moments", "increased spatial reasoning skills", "to get into the state of flow", "because it is fun"

What's the hardest aspect about making puzzle games?
"Deciding whether the mechanics are fun.", "Completeness. Having confidence that you fully explored the possibility space of a mechanic is daunting.", "Finding a balance between completeness and drawing a strong boundary around the concept.", "Creating a set of puzzle mechanics that's elegant, uses as little elements as possible in as many as possible interesting combinations, and generating a puzzle set that's both sufficent in quality and quantity."

In which way do you hope automated tools will help design puzzle games? *"Finding novel configurations and solutions.", "For creative inspiration. Finding ideas that a person would have trouble discovering.", "Mostly to help speed up finding puzzles that might have certain properties, or to allow me to experience types of puzzles I might not normally have designed on my own.", "Reduce hand work on the designer, so that the designer can focus more on improving the mechanics and increasing the set of puzzles."*

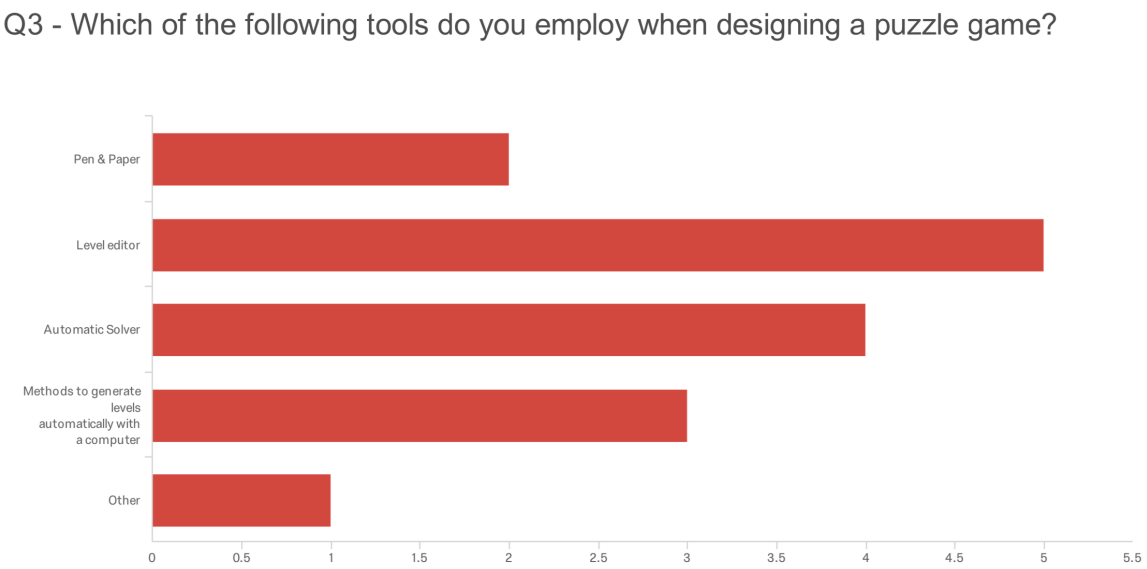


Figure A.1.: *Needfinding survey Q3*

A.2. PuzzleScript

1

[1] Note that matched_Property has to be a property that either appears only once on the left-hand side or a property that is exactly at the same location where the property appears on the left-hand side. Whatever property it matches will be replaced on the right-hand side.

```
========
OBJECTS
========
Background
LIGHTGREEN

Target
DarkBlue

Wall
BROWN

Player
White

Crate
Orange

=======
LEGEND
=======
. = Background
# = Wall
P = Player
* = Crate
@ = Crate and Target
O = Target

=======
SOUNDS
=======
Crate MOVE 36772507

================
COLLISIONLAYERS
================
Background
Target
Player,Wall,Crate

======
RULES
======
[ > Player | Crate ] -> [ > Player | > Crate ]

==============
WINCONDITIONS
==============
All Crate on Target

=======
LEVELS
=======
####..
#.O#..
#..###
#@P..#
#..*.#
#..###
####..
```

Figure A.2.: *Example Sokoban implementation in PuzzleScript*

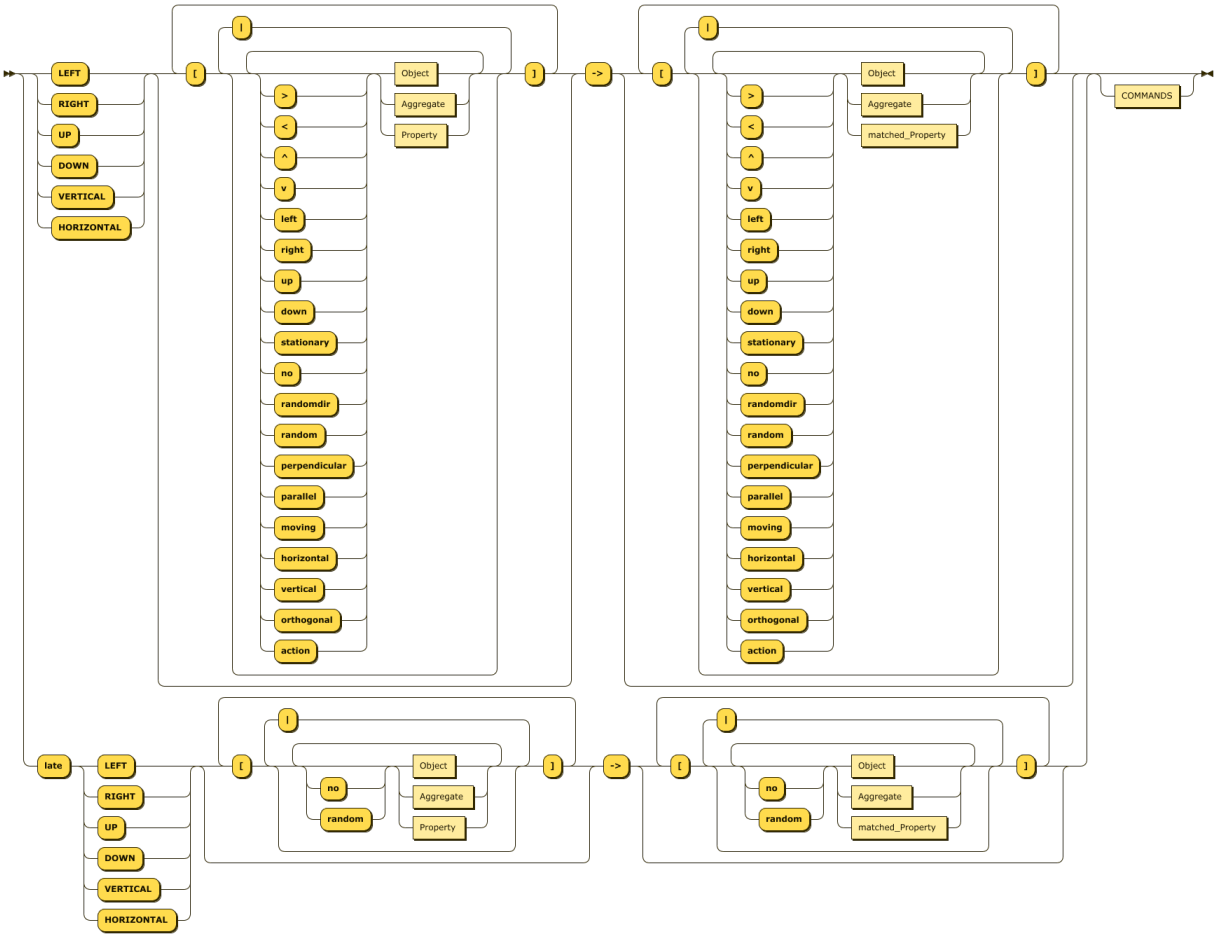


Figure A.3.: *BNF Diagram for a PuzzleScript rule* [a)]

[a)] Note that matched_Property has to be a property that either appears only once on the left-hand side or a property that is exactly at the same location on the left-hand side. Whatever property it matches will be replaced on the right-hand side.

A.3. User study

A.4. Structured interview questions

This is a compiled list of the most interesting answers to the interview questions:

Tell me about your experience with the editor.

"In broad terms it does some cool things. Bunch of usability suggestions. I have to figure out more use cases." – Participant 1

"Overall it's a very interesting project. Some quirks of the interface are to be solved but I hope you can expand it and make it more powerful." – Participant 2

"The editor was fun to use, but oftentimes I found it was lacking visual feedback of the ongoing generation process." – Participant 4

How was this design process different from your usual practice?

"Designing with a level generator gives the advantage of encountering new object 'constellations' that are sometimes unintuitive and difficult to solve." – Participant 4

"Faster, felt like communication, not trivial." – Participant 6

"The loop of mutating/trying/repeat is very appealing, reminiscent to genetic art generators. Usually I make a loop of generating/curating/repeat for my games, perhaps this type of tool can insert a very useful step with the mutator." – Participant 2

How satisfied are you with the design(s)?

"Some of the levels I found are very surprising and pleasing." – Participant 2

"I wasn't happy with the designs (of the Sokoban levels) but attribute that mainly to my error, could have used a more interesting generation algorithm and level base. Sokoban levels have been generated for ages, so my levels were probably not original. Maybe going with a different base game (block pulling maybe?) would have resulted in more originality." – Participant 4

"The levels are already quite neat! I'd say about 20% were really interesting." – Participant 6

"Not particularly." – Participant 5

How original do you think are the levels?

"Not really. There should be more curated suggestion levels." – Participant 6

Did the functionalities hinder you to do anything and if yes what was it?

"I did more than I imagined I could with the tool." – Participant 3

"Some elements of the interface are difficult to understand at first, and some others are difficult to use, like the text editor." – Participant 1

"The editor had most of the functionality that it needs, some 'quality of level' improvements would be required to tie it all together though. Not being able to alter level size from within the

Participant	1	2	3	4	5	6	7
Age	34	46	20	25	30	27	28
Gender	male	male	male	trans	male	male	male
Profession	game designer	programmer & game designer	junior programmer	design student	doctor	game designer	game designer
Puzzle Design Experience	15 years	8 years	3 years on and off	1 year, 3 games	1 year, 200 hours	2 days 5 years (gd)	4 years

Table A.1.: *Demography of user study*

editor was annoying, also the Transform text box clearing on tab switch was a hindrance. Not being able to alter level size from within the editor was annoying, also the Transform text box clearing on tab switch was a hindrance." – Participant 4

"I was able to do more in a short amount of time and check more things than I would usually." – Participant 6

Can you tell me how you experienced the interaction with the editor?

"I used mainly the mouse, and some of the suggested shortcuts. I did not try to use the text editor very much." – Participant 2

"Without the verbal confirmation that the generator is working I'd have believed it was not generating and kept restarting the process. Otherwise it was mostly fine." – Participant 4

"It felt like a communication with the exception of the things I controlled." – Participant 6 (Note: This participant only used the suggested buttons and did not implement his own transformation rules).

"Watching numbers go up and gradually in the transformer and if it's too slow i'll change the transforms and fiddle with the level size. The polishing at the end I do by hand." – Participant 1

How useful were the suggestions by the system? How did they help you?

"The mutations were generally chaotic, but some of them were surprising, apparently impossible to solve, hence very fun to solve." – Participant 2

"The level suggestions were as useful as the generation rules I fed into the system. The suggestions can be helpful for coming up with own design ideas." – Participant 4

Did you think that the system pointed you in different directions than you intended?

"If the purpose of the system is generating new ideas, then new ideas would be an intended direction, so no." – Participant 4

"I was just trying to figure out what the tool can and cannot do." – Participant 1

"Not much, only worked well on smaller sized levels." – Participant 5

In your view, how did the system make the suggestions?

"I suspect the system uses digital mutation operations that modify the current element of the levels, and tries to solve the new levels, discarding unsolvable/trivial ones." – Participant 2

"It made the suggestions based on the results of my transformation rules, a solver and a complexity rating algorithm." – Participant 4

Can you reflect on how your behaviour impacted the suggestions of the system?

"I feel that the selection of levels from me reinforces the kind of variations the system makes later." – Participant 2

"The transformation rules directly impact what levels are generated, the solvable subset of those gets rated by complexity, the four highest-razed levels get presented to me, so the impact of the rules is indirect." – Participant 4

"Well I kind of knew what it did after you explained it so it just did what I wanted it to do." – Participant 6

"The modify and leave buttons impacted it." – Participant 3

How much did you feel the system understood your aim?

"I feel like in its current state the system does not use all the feedback loops it could use." – Participant 2

"The system has the aim of creating complex levels from a set of rules. Formulating intentions would happen by setting rules and base level but the system would not 'understand', just process." – Participant 4

"Pretty well." – Participant 1

"It didn't really 'understand' it since it's just a tool." – Participant 3

"It is not intelligent, it just did what I told it to do and it was a useful tool." – Participant 6

How would you characterise the tool within the design process?

"As an integral part, like a canvas where experiment within the level space." – Participant 2

"Perhaps useful for coming up with variations and unintuitive level designs." – Participant 4

"The role of saving levels, playing them, checking if they are solvable and finding their solutions if so. Additionally, it brought me ideas and did its job quite well." – Participant 6

"I don't have a loop of iterative design, I'm just looking for a good starting point. The suggestions are nice and interesting." – Participant 1

How well did the scores for the difficulty of the level match your estimation?

"Good enough to be useful, but somewhat uneven. I believe the system could integrate more evaluation criteria to make the estimation more exact." – Participant 2

"Difficulty rating did not align with my perception of difficulty, but I've not yet explored enough to be sure." – Participant 4

"I did not look at the difficulty scores, just at the curated levels." – Participant 6

"Pretty accurate, there seems to be a threshold starting from which the levels get interesting, at least that's how it feels like qualitatively." – Participant 1

Did a suggestion inspire you without you clicking on it?

"Yea, exactly what happened in the end." – Participant 3

"Yes, especially as a level of confidence so I was sure that the current design was pretty solid when the suggestions seemed worse." – Participant 6

"No, but could happen. Looking at them and trying to solve them in the head is slower than just clicking on them to try it out." – Participant 1

Note: The people who said no were simply adapt at quickly trying out a suggestion and popping back to the old design.

Would you use such a system in your work practice and when? If Yes: What for in specific; If No: What needs to be changed?

"I would use it to find new levels for my games, but also to inspire my level generators with some kind of variations I usually don't explore because lack of real time visualisation." – Participant 2

"Yes, for creating level states that I hadn't encountered before, then using those as inspiration." – Participant 4

"Not for the game I'm currently working on but if I design a puzzle game in the future again sure." – Participant 6

"Yea, for certain particular games. Also for super compact levels on any kind of puzzle game." – Participant 1

Where do you see the potential of such systems and where the limitation?

"There is great potential to explore rule variation in addition to level variation. Limitations are, like always, memory and cpu usage when using mutators and solvers not tuned to specific rules." – Participant 2

"Potential for creating unintuitive and thus hard to solve levels. Potential for improving handmade levels. Limitation in complexity heuristics and level size (since computation effort increases exponentially)" – Participant 3

"Potential is to create low effort levels and it allows you to constrain your creativity. There's a kind of Google effect where I stop thinking about solvability. Probably if you've been designing puzzles for a specific game for a long period of time the tool probably won't help much anymore, so it's more useful in the prototype phase. The limits are the language of PuzzleScript and the language of the transformer." – Participant 4

"Performance limitations. Level size, # of rules, people making bad levels with it, people making levels that aren't fun. I'm always going to try to make small levels with it." – Participant 1

"The limitations are [the systems] judging abilities." – Participant 3

Do you have any thoughts you would like to add?

"I hope more kind of games are added later to the system (3D and/or continuous-like rules). This project is very promising, I cannot wait to try the next version, and possibly contribute to it." – Participant 2

Bibliography

ALOUPIS, G., DEMAINE, E. D., GUO, A., AND VIGLIETTA, G. 2015. Classic nintendo games are (computationally) hard. *Theor. Comput. Sci. 586*, C (June), 135–160.

BALDWIN, A., DAHLSKOG, S., FONT, J. M., AND HOLMBERG, J. 2017. Mixed-initiative procedural generation of dungeons using game design patterns. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 25–32.

BUTLER, E., SMITH, A. M., LIU, Y.-E., AND POPOVIC, Z. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '13, 377–386.

CHONG-U LIM, AND HARRELL, D. F. 2014. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games*, 1–8.

CSIKSZENTMIHALYI, M. 1990. *Flow: The Psychology of Optimal Experience*. Harper & Row.

CULBERSON, J. C., AND CULBERSON, J. C. 1997. Sokoban is PSPACE-complete. *Proceedings of the International Conference on Fun with Algorithms*, April, 65–76.

DEMAINE, E. D., DEMAINE, M. L., HOFFMANN, M., AND O'ROURKE, J. 2003. Pushing blocks is hard. *Computational Geometry 26*, 1, 21 – 36. The Thirteenth Canadian Conference on Computational Geometry - CCCG'01.

FROLEYKS, N. 2016. Using an Algorithm Portfolio to Solve Sokoban. AAAI Publications, Tenth Annual Symposium on Combinatorial Search.

GIACCARDI, E., AND FISCHER, G. 2008. Creativity and evolution: a metadesign perspective. *Digital Creativity 19*, 1, 19–32.

GUZDIAL, M., LIAO, N., CHEN, J., CHEN, S.-Y., SHAH, S., SHAH, V., RENO, J., SMITH, G., AND RIEDL, M. O. 2019. Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '19, 624:1–624:13.

HEARN, R. A., AND DEMAINE, E. D. 2005. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science 343*, 1, 72 – 96. Game Theory Meets Theoretical Computer Science.

HOLTE, R. C. 2010. Common Misconceptions Concerning Heuristic Search. *Proceedings of the Third Annual Symposium on Combinatorial Search (SOCS-10)*, 46–51.

HORVITZ, E. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of CHI '99, ACM SIGCHI Conference on Human Factors in Computing Systems, Pittsburgh, PA, ACM Press.*, 159–166.

JARUŠEK PETR, P. R. 2011. What Determines Difficulty of Transport Puzzles? Experiments with Human Problem Solving. *Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24*, 428–433.

JUNGHANNS, A., AND SCHAEFFER, J. 1999. Pushing the limits: New developments in single-agent search. AAINQ46861 PhD thesis.

KANGAS, P. 2017. The pleasures of puzzle-solving in adventure games. MSc thesis.

KARTAL, B., SOHRE, N., AND GUY, S. 2016. Data driven sokoban puzzle generation with monte carlo tree search. *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

KHALIFA, A., AND FAYEK, M. 2015. Automatic Puzzle Level Generation: A General Approach using a Description Language.

KOCH, J., LUCERO, A., HEGEMANN, L., AND OULASVIRTA, A. 2019. May ai?: Design ideation with cooperative contextual bandits. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '19, 633:1–633:12.

LAWSON, B., AND LOKE, S. M. 1997. Computers, words and pictures. *Design Studies 18*, 2, 171 – 183.

LEE, M. J., BAHMANI, F., KWAN, I., LAFERTE, J., CHARTERS, P., HORVATH, A., LUOR, F., CAO, J., LAW, C., BESWETHERICK, M., LONG, S., BURNETT, M., AND KO, A. J. 2014. Principles of a debugging-first puzzle game for computing education. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 57–64.

LIAPIS, A., YANNAKAKIS, G. N., AND TOGELIUS, J. 2013. Sentient sketchbook: Computer-aided game level authoring. *Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG 2013)*, 213–220.

LIAPIS, A., SMITH, G., AND SHAKER, N. 2016. *Mixed-initiative content creation*. Springer International Publishing, Cham, 195–214.

MACHADO, T., GOPSTEIN, D., NEALEN, A., NOV, O., AND TOGELIUS, J. 2018. AI-

Assisted Game Debugging with Cicero. *2018 IEEE Congress on Evolutionary Computation, CEC 2018 - Proceedings*.

MOHAMMADI, D., 2014. The Guardian : Online gamers solving sciences biggest problems. `https://www.theguardian.com/technology/2014/jan/25/online-gamers-solving-sciences-biggest-problems`.

MURASE, Y., MATSUBARA, H., AND HIRAGA, Y. 1996. Automatic making of sokoban problems. In *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence: Topics in Artificial Intelligence*, Springer-Verlag, London, UK, UK, PRICAI '96, 592–600.

NELSON, M. J., AND MATEAS, M. 2009. A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, ACM, New York, NY, USA, FDG '09, 137–144.

OSBORN, J. C., GROW, A., AND MATEAS, M. 2013. Modular computational critics for games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AAAI Press, AIIDE'13, 163–169.

PEREZ-LIEBANA, D., SAMOTHRAKIS, S., TOGELIUS, J., SCHAUL, T., LUCAS, S. M., COUETOUX, A., LEE, J., LIM, C., AND THOMPSON, T. 2016. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games 8*, 3 (Sep.), 229–243.

RACANIÈRE, S., WEBER, T., REICHERT, D. P., BUESING, L., GUEZ, A., REZENDE, D., BADIA, A. P., VINYALS, O., HEESS, N., LI, Y., PASCANU, R., BATTAGLIA, P., HASSABIS, D., SILVER, D., AND WIERSTRA, D. 2017. Imagination-augmented agents for deep reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Curran Associates Inc., USA, NIPS'17, 5694–5705.

RADEK, P. 2011. Human Problem Solving: Sudoku Case Study. *Proc. of the Fifth Starting AI Researchers Symposium (STAIRS 2010)*, January.

SALEN, K., AND ZIMMERMAN, E. 2004. Rules of play. *MIT Press*.

SCHMIDHUBER, J. 2010. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). *IEEE Transactions on Autonomous Mental Development 2*, 3 (Sep.), 230–247.

SHAKER, M., SHAKER, N., AND TOGELIUS, J. 2013. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2013*, AAAI press, 215–216.

SHAKER, M., SHAKER, N., AND TOGELIUS, J. 2014. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AAAI Press, AIIDE'13, 72–78.

SMITH, A. M., NELSON, M. J., AND MATEAS, M. 2008. Prototyping Games with BIPED. 193–194.

SMITH, G., WHITEHEAD, J., AND MATEAS, M. 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational*

Intelligence and AI in Games 3, 3 (Sep.), 201–215.

TAYLOR, J., AND PARBERRY, I. 2011. Procedural Generation of Sokoban Levels. *Proceedings of the 6th Annual North American Conference on AI and Simulation in Games*, 5–12.

VIGLIETTA, G. 2012. Gaming is a hard job, but someone has to do it! In *Fun with Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, E. Kranakis, D. Krizanc, and F. Luccio, Eds., 357–367.

WILLIAMS, A. 2017. MazezaM Levels with Exponentially Long Solutions. *20th Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCG$\hat{3}$ 2017)*, August.