Automated Puzzle Difficulty Estimation

Marc van Kreveld
Utrecht University
M.J.vanKreveld@uu.nl

Maarten Löffler
Utrecht University
M.Loffler@uu.nl

Paul Mutser
Utrecht University
P.Mutser@students.uu.nl

***Abstract*— We introduce a method for automatically rating the difficulty of puzzle game levels. Our method takes multiple aspects of the levels of these games, such as level size, and combines these into a difficulty function. It can simply be adapted to most puzzle games, and we test it on three different ones: Flow, Lazors and Move. We conducted a user study to discover how difficult players find the levels of a set and use this data to train the difficulty function to match the user-provided ratings. Our experiments show that the difficulty function is capable of rating levels with an average error of approximately one point in Lazors and Move, and less than half a point in Flow, on a difficulty scale of 1–10.**

I. Introduction

With the rise of smartphones over the past few years, the market for apps is growing rapidly. Especially small *pick-up-and-play* games have seen a large increase in popularity. This group of games includes puzzle games, which require a large number of sufficiently different and interesting levels that challenge the players, so that they will continue to play them.

The creation of levels is a time-consuming process that is usually performed by hand. A large part of this process consists of determining how challenging a puzzle level is. By automating the evaluation of level difficulty, the level generation pipeline can be made more efficient and cheaper. Furthermore, user-specific level generation becomes possible; for example, software may generate levels and select the very hard ones for the small group of players that are particularly good at solving a puzzle type, without the need of making such a level set available globally.

Automated difficulty estimation is far from trivial, as we need to determine what factors play a role in the difficulty of a level. Such factors can be the size of a level, the number of colors, the number of obstacles, the number of bends in a solution path, etc. Once these factors have been found, they need to be combined into a *difficulty function* that has a level as its input and a difficulty rating as its output.

Difficulty is subjective, implying that there is not one true difficulty rating for each level. The goal of the difficulty function is to be able to rate levels as close to the average rating given by players as possible. This means that it has to be trained to match human ratings, and for this we need to acquire data on the difficulty of levels in a game. This can be done by play-testing. Fortunately, play-testing only needs to be done once on a small set of levels. The trained function can then be used to estimate the difficulty of newly created levels without the need for play-testing each level.

A side effect of using a difficulty function is that the formula itself can be used to understand what makes a level difficult. This knowledge can be used to create levels with specific difficulty, and is a step towards automated user-specific content generation.

Our goal is to create a method for automated, effective difficulty estimation. The method is based on a function that should be able to accurately assign difficulty ratings to puzzle levels. We aim to create a generic method that can be applied to most puzzle games. To test how our method applies to different games, we perform our experiments on three different puzzle games. They are:

- *Flow:* the player connects pairs of equal-colored dots with non-intersecting paths.
- *Lazors:* the player moves mirrors to direct laser-beams to hit certain targets.
- *Move:* the player moves a set of colored balls to their matching positions.

As the games are quite different, we expect that using a difficulty function will work well on many other abstract puzzle games.

This paper is structured as follows. Section II gives a short overview of previous work in the area of automated

difficulty rating. Section III presents our method for difficulty estimation. Section IV introduces the games we use for this research. We also describe how we apply the difficulty function to these games in this section. We perform a user study to train the difficulty function on actual data, which is described in Section V. We evaluate the results in Section VI by correlation analysis and cross-validation. The evaluation reveals which factors correlate most with difficulty, according to the users, and how well the difficulty function can estimate the difficulty of a level. We conclude in Section VII and give directions for future research.

II. Related Work

This section gives a brief overview of research on difficulty assessment with a focus on automated methods for puzzle games.

Ashlock and Schonfeld [1] assess the difficulty of Sokoban puzzles by solving levels automatically. The average time taken until a solution is found and the probability that an attempt to find a solution fails are used as difficulty measures. These measures are subsequently used to order the levels by difficulty.

Mantere and Koljonen [2] use genetic algorithms to solve, generate and rate Sudoku puzzles. They make use of a genetic Sudoku solver to rate the difficulty. Their assumption is that puzzles that take longer to solve by the algorithm are also perceived as more difficult by humans, which is tested by comparing their solving times to the provided difficulty ratings of the puzzles. They conclude that their results support this assumption.

Jarušek and Pelánek [3] argue that human factors should be taken into account explicitly. User study data for Sokoban levels is used to determine difficulty, measured as the average time taken to solve a level. They use various metrics based on the shortest solution to find the correlation between those metrics and the difficulty of a level. They also use more abstract measures.

Browne [4] uses linear functions and features of board games to generate enjoyable games. While his goals are different, his approach is also applicable to determining difficulty. Using a user study, Browne finds the aspects of board games that makes those games enjoyable. This data is then used to create a linear function that determines the enjoyability of other board games. An evolutionary algorithm is then used to create a very enjoyable game, Yavalath. The same process can be used to rate the difficulty of levels, and generate levels with a certain difficulty.

We also mention the work of Aponte, Levieux and Natkin [5], who calculate the difficulty of a game by seeing it as a series of challenges whose difficulty is calculated separately by a probability of succeeding. András, Sipos and Sóos [6] analyse the difficulty of Happy Cube puzzles, and performed a user study to determine the time taken to solve different puzzles. Furthermore, they develop a solver and try to characterize difficulty by analyzing the number and types of backtracks, etc. Guid and Bratko [7] estimate problem difficulty for humans based on searching between alternatives, using chess. Finally, we note that procedural level generation also needs evaluation of the levels (e.g., Taylor and Parberry [8]).

It has been suggested that the difficulty of a level can be found by measuring the time players take to solve that level. This is not necessarily true, as a level consisting of one hard challenge may require less time to solve than a level consisting of multiple simple challenges. Similarly, the number of moves required to solve a level is often not related to the difficulty of that level. There may be a high correlation between the difficulty of a level and the number of moves, but sometimes a level requires many moves in its solution that are all simple. In other puzzles, like Flow, the number of moves may be fixed, since every empty grid cell must be filled with exactly one connection, rendering number of moves useless to evaluate difficulty.

We use a similar method to Browne [4] in our research by also combining several aspects of a game into a linear function. Our function is also trained with real data that we obtain from a user study. We use this function to define the difficulty of levels.

III. Difficulty Estimation

To automatically determine the difficulty of a level, we design a function that takes a level as its input and outputs a value that indicates the difficulty. As every game is different and has other elements defining the difficulty, each game requires its own function. We use the function in a general methodology to estimate the difficulty of a level of a puzzle game. The methodology consists of three steps. First, we determine measurable, simple aspects of levels in a game that are likely to influence their difficulty.

Second, we combine these aspects by weighted linear combination into a single difficulty function. Third, we determine the weights by optimizing the function over a set of levels with a known "ground truth" difficulty, yielding a function that can estimate the difficulty of any level of the game.

A. Choosing Variables

For our difficulty function to be effective, each game requires a set of variables that together are enough to obtain a good estimate of the difficulty. These variables are usually chosen by playing the game, or observing others playing the game, and looking for properties that are common in difficult levels and uncommon in easy levels, or vice versa. This is mostly a trial and error process, as occasionally the chosen variables are in reality a poor representation of level difficulty. As the variables are meant to give a partial measure of the difficulty of a level, we also refer to them as *difficulty measures*.

When choosing the variables, we take into account the number of variables and the ease of measuring the variables. As we want to use the difficulty function for fast classification of levels, we prefer variables for which the value is easily determined. These are usually simple counts of the number of objects of a certain kind. To improve the process of determining the weights in the difficulty function, we try to limit the number of variables each game uses. We found that four to seven variables is enough to make a good estimate of the difficulty. A smaller number does not contain enough data to accurately estimate the difficulty of a level, while a larger number would require a lot more reference data to properly set the weights of the variables.

B. Models

There are two ways to combine values of different factors: either treat these variables independently and add them together after weighing, or combine multiple variables into single terms by more general methods. The second option is more versatile, but also more complicated to use and fine-tune than independent variables. Since simplicity of a method is preferable (cf. Occam's Razor), we choose to treat the variables independently.

There are still multiple options when using separate variables. It is simplest to use a weighted linear combination, although it can be that for some game and some factor, the difficulty grows quadratically or even exponentially in that factor. Again, for the sake of simplicity we use a weighted linear combination:

$$\text{Difficulty}(L) = \sum_{i=1}^{n} (w_i * V_i(L)) + w_0 \qquad (1)$$

Each game is assigned a number of variables V_i which influence the difficulty of a level L in the game. This number, n in the function, may vary between games. As our model is linear, it allows us to use linear regression to set the values for the weights.

C. Determining the Weights

To find the weights w_i in the difficulty function, we attempt to fit the function to a dataset of levels with known difficulty, obtained from a user study. The way these levels were obtained is detailed in Section IV-C. Details on the user study are given in Section V. We assume for now that the difficulty of each level is known, for example as the average of the user ratings.

Once we obtain this dataset, we attempt to set the weights in the difficulty function in such a way, that when we apply the function to the levels from the dataset, a difficulty score is calculated that is as close as possible to the score assigned to that level by the participants in our user study. This is a well-researched problem in mathematics, and there are many methods available to solve it, each with its pros and cons.

Among the exact solutions, *linear programming* [9] seems a logical choice to determine weights. The dimension of the linear program is equal to the number of variables, so it is typically a small constant less than ten. Hence, linear programming can be done in time linear in the number of constraints [10]. While our constraints are exact if we consider the difficulty ratings to be exact, the linear program will generally not be solvable since we have (many) more constraints than unknowns. Instead, we may choose a margin. Instead of letting $\text{Difficulty}(L_j) = D_j$ for a level j with difficulty D_j, we can use constraints $\text{Difficulty}(L_j) \leq D_j + \delta$ and $\text{Difficulty}(L_j) \geq D_j - \delta$ for some chosen margin $\delta > 0$. However, a small number of levels with outlier ratings may force us to choose the margin large, leading to poor performance. There are other ways to apply linear programming, but it is beyond the scope of this paper to discuss these.

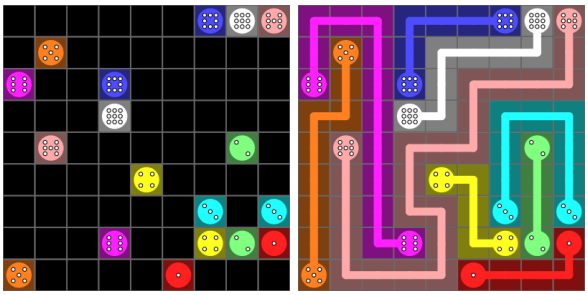
Since outlying values may be present we choose to use a standard *linear regression* [11] to set the weights in

such a way that the average squared difference between the result of the difficulty function and the user-assigned score is minimal.

IV. Games for our Experiments

A. Games

For our experiments we used three different games. We decided to use multiple games to be able to test how well our method can be adapted to different applications. As our aim was to create a general method, we want it to perform well in more than one case. The rules of these games will be explained next.



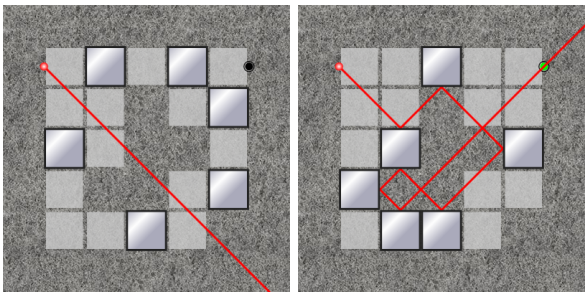
(a) initial configuration **(b)** solution

Fig. 1: Example of a level in Flow and its solution

1) Flow: A level of Flow[1] consists of a grid that contains multiple pairs of balls, where each pair of balls has a different color. Figure 1a shows a level of the game in its initial configuration. The goal of the game is to draw paths on the grid to connect both balls of each pair. Paths are not allowed to intersect, and the paths should cover the entire grid. The level is solved once all pairs are connected, and the entire grid is covered, see Figure 1b.

Although the rules of Flow are fairly straightforward, levels can become difficult when paths need to twist around each other to reach a solution. In some levels it is possible to connect all pairs without using the entire grid. This results in holes that need to be filled to reach a solution. Creating a path that fills these holes is also not always a simple task.

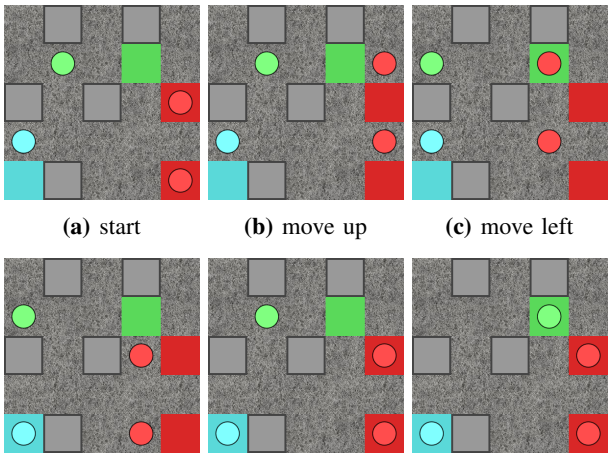
[1]Big Duck Games, http://blog.bigduckgames.com/. Also known as Number Link or as Arukone. The origins are unclear, but date back to 1932 or earlier.



(a) initial **(b)** solution

Fig. 2: Example of a level of Lazors and its solution

2) Lazors: In Lazors[2] there are three types of objects. First, there are lasers. These emit a beam of light across the level. Second, there are light targets. These are activated when there is light shining through it. Once all targets in a level are activated, that level is solved. Third, there are blocks, objects that interact with the light. The player can move these blocks to certain open spaces in the level to solve it. Our implementation is a simplified version of the game with only mirror blocks. This is done to reduce the number of variables possibly influencing the difficulty of the level. Figure 2a shows a level in the game, together with its solution in Figure 2b.



(a) start **(b)** move up **(c)** move left

(d) move down **(e)** move right **(f)** move right (2x)

Fig. 3: Example of a level in Move and its solution

3) Move: The game Move[3] is is played on a small grid onto which three types of elements are placed: balls,

[2]Pyrosphere, http://pyrosphere.net/lazors/. A type of light-and-mirror puzzle, in physical form several centuries old. A related computer game is Deflektor (1987).

[3]Nitako Brain Puzzles, http://www.nitako.com/.

goals and rocks. Balls and goals are each assigned a color, with each ball having a corresponding goal of the same color. Multiple balls can have the same color, in which case they can be matched to any goal of the same color. Finally, a number of rocks are placed on the grid. The initial configuration of a level can be seen in Figure 3a.

The goal of the game is to move each ball to a goal of the same color. Balls can be moved one square left, right, up or down, but the player can only move all balls at the same time. Rocks block the movement of the balls, as do the edges of the grid and other balls. When a ball is blocked, that ball will not move, but the other balls can still be moved in that direction. Figure 3 shows how a level is solved from its initial configuration to the solution shown in Figure 3f.

This game becomes hard as often moving one ball towards its goal results in another ball moving away from its goal. Players must make clever use of the rocks to get each ball in the right position to solve the level.

B. Implementation

All three games were reimplemented using javascript on an html5 page. The choice for a web-based approach was made to ensure that as many people as possible would be able to participate in our user study, and to keep the effort required to participate to a minimum. Javascript also allowed us to have support for both mouse/keyboard control as well as touchscreen devices.

C. Levels

The three games came with sufficiently many levels to be able to select a good subset for the user study. For Flow, we selected 40 levels for the user study and another 10 for training. For Lazors, we selected 65 levels for the user study and another 10 for training. However, the supplied levels for Move were all rather easy, so we implemented a level generator to make our own set to have a larger variety in difficulty among the levels. We picked 80 levels in such a way that for $3 \leq x \leq 10$ there were 10 levels solvable in x moves.

D. Difficulty Measures

In this section we detail the difficulty measures we use in the three games. Recall that our goal is to find a small set of features for each game, which together are enough to give a good estimate of the difficulty of a level in that game using a weighted linear combination. We want this set to be small, as larger sets of features can lead to overfitting on our dataset, causing our difficulty function to work well on our dataset but not on others.

We identify three different types of difficulty measure. The first type is *initial features*, and includes features of the level in its initial configuration. Examples of initial features are the (grid) size of the level and the number of game elements the player can interact with. These values are usually easy to determine for a given level.

The second type is *solution features*. This type includes features of the level in its solved configuration. An example of a solution feature is the location and state of game elements in the solution. They can in some cases be more useful than initial features, as for certain games, the solved state tells more about the difficulty than the initial configuration does. Values for difficulty measures of this type are harder to determine, as they require a level to be solved first.

The last type is *dynamic features*, including all features related to the process of solving a level. Examples of dynamic features are the minimum number of moves required to solve a level, and the type of moves used. The values for these difficulty measures are usually the hardest to determine, as they often require a solver that can find a solution with a minimal number of moves. Albeit different in nature, the time to solve a level for a player or automated solver is also a dynamic feature.

We note that levels that have multiple solutions require extra attention to define solution features and dynamic features appropriately. We also note that some games do not have solution features other than the initial features. An example is Move, where the final state of the game is immediately obvious from the initial configuration.

1) Flow: For the game Flow, we use the following features as variables in our difficulty function:

- **Level size (initial)**: The size of the level, defined by the number of squares on the board. We expect larger levels to be more difficult, as the player has more space to fill. The creators of the original game use the size of the level as their only rating of level difficulty.
- **Colors (initial)**: The number of different colors to connect to each other. With smaller numbers of colors, paths will need to fill more space, and become longer, and possibly more difficult. With larger numbers of colors, we expect the paths to

block each other more, which may also lead to higher difficulty. We therefore do not know beforehand how this measure will affect the difficulty of a level.
- **Average distance (initial)**: The average distance between the start- and endpoints of the same color. We measure this using city block distance (L_1 distance). When this value is low, it means that the paths of the solution are more indirect, as the complete level still needs to be filled. We therefore expect lower values to cause higher difficulty.
- **Turns (solution)**: The number of turns used in the solution. More turns implies that the connection between start- and endpoints is more convoluted, which may imply a more difficult puzzle.

2) Lazors: For the game Lazors, we use the following features for our difficulty function:[4]

- **Level size (initial)**: The size of the level, defined as the area of the smallest bounding box around the usable tiles.
- **Usable tiles (initial)**: The number of usable tiles.
- **Emitters (initial)**: The number of laser emitters.
- **Receivers (initial)**: The number of targets that need to be activated to solve the level.
- **Mirrors (initial)**: The number of mirrors.
- **Reflections (solution)**: The number of times a laser reflects from a mirror in the solution. This measure indicates how indirect the path of the lasers is.
- **Intersections (solution)**: The number of times two laserbeams cross path in the solution. We use this as a measure of how cluttered a level is.

3) Move: For the game Move, we use the following features for our difficulty function:

- **Level size (initial)**: The size of the level, defined by the number of squares on the board.
- **Balls (initial)**: The number of balls controlled by the player.
- **Colors (initial)**: The number of unique ball colors.
- **Rocks (initial)**: The number of rocks in the level.
- **Moves (dynamic)**: The minimum number of moves required to solve a level.
- **Counterintuitive moves (dynamic)**: The number of moves in the shortest solution that are counterintuitive. A move is counterintuitive when the average distance between the balls and their closest goal of the same color increases in that move.

[4]We no longer discuss how a feature may influence the difficulty, given the space limitations.

V. User Study

To be able to set the weights in our difficulty function in such a way that it accurately predicts the difficulty of a level, we require "ground truth" data. This data comes in the form of a set of levels and the difficulty rating they should have, obtained from a user study. In total, 86 participants played Flow, 105 participants played Lazors, and 57 participants played Move. Their ages differed, but the majority was between 18 and 28. The level of education was relatively high.

Participants started the experiment with a tutorial of a game with 10 levels of increasing difficulty. Then the other levels were offered in random order to avoid learning effects. After solving a level, the participant was asked to rate the difficulty on a scale of 1–10. The levels can still be played at `http://www.staff.science.uu.nl/~loffl001/puzzles/experiments/games/` in the same interface as used in the study.

VI. Results

In this section we give an overview of the results of the user study: in particular, a correlation analysis between difficulty and the chosen measures, and the quality of fit of the difficulty function after setting the weights.

A. Flow

We start by comparing the results of the user study to the values for the difficulty measures. Besides the difficulty rating provided by the users, we also consider the scaled difficulty rating (where each user has their scores scaled to cover the full range 1–10), the time taken to solve a level, and the number of moves performed by the users. We define a move in Flow as the drawing of one (partial) path from the moment the mouse button is pressed to the moment the mouse button is released. The correlation coefficients are shown in Figure 4.

We notice that the correlation coefficients of solving time with the difficulty measures are much lower. This can be explained by a high variation between users for the time taken to solve a level. Users tend to agree on difficulty ratings and use a similar number of moves, but some players are significantly faster than others.

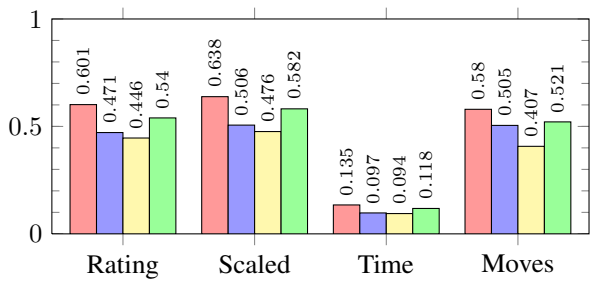


Fig. 4: Correlation coefficients between difficulty Rating, Scaled version of this rating, solving Time, and number of Moves, and difficulty measures for Flow. From left to right these are level size (red), colors (blue), average distance (yellow), and turns (green).

The size of the level (red) has the highest correlation with all four aspects, followed by the number of turns (green), the number of colors (blue), and last the average distance between endpoints (yellow). Even so, the average distance measure still shows a fairly high correlation with all aspects (except the solving time).

We split our data into 5 sets, each containing the data for 8 different levels in order, such that set 1 contains the data for levels 1–8, set 2 contains the data for levels 9–16 and so on. We train the difficulty function for each set using the data from the other four sets, so that the difficulty function for each set is not influenced by the data in this set. This way we can perform a proper analysis on how the difficulty function would perform on new levels outside of our dataset (*cross-validation*).

The difficulty function, averaged over the sets, is $0.088\cdot$ Level size $- 0.30 \cdot$ Colors $- 0.40 \cdot$ Average distance $+ 0.040 \cdot$ Turns. Note that the weights do not have a clear meaning, nor the fact that some weights are negative. The measures were not normalized and they are not independent variables. We use the difficulty function to calculate the difficulty for all levels, so that we can compare this to the original ratings by the users. We also do this for scaled difficulty, time to solve, and number of moves. An overview of the average error over the sets is presented in Table I.

The difficulty function gives us an average error of 0.4103 points, or 0.4040 points when we weigh the error for each level by the number of times that level has been played. For the scaled difficulty rating this error is 0.6502 points, or 0.6423 when the error is weighted. This is noteworthy, since we introduced the scaled rating to decrease the error, but it actually increases. This can partly be explained by the fact that the scaled difficulty makes use of a larger range of scores. The range is, however, only ~ 1.3 times as large on the average, and this accounts for only half of the difference.

	Rating	Scaled	Time	Moves
Unweighted	0.4103	0.6502	42.84%	12.19%
Weighted	0.4040	0.6423	43.21%	11.86%

TABLE I: Average errors of Flow produced by the difficulty function when compared to user study data. Shown for difficulty rating and scaled difficulty rating (in absolute points), and time and number of moves (as relative error). All scores are given both unweighted and weighted by number of level ratings.

B. Lazors

For Lazors we give the correlation coefficients in Figure 5 and the errors obtained after fitting, using cross-correlation, in Table II. The scaled rating again makes the error larger, and more than can be expected from the increase in range. Therefore, the scaled difficulty rating is omitted in Figure 5. The number of emitters and receivers correlates little with the difficulty, which may be due to too little variation in the levels. We observe that the remaining error after fitting is considerably higher than for Flow: slightly over 1 point on a scale 1–10.

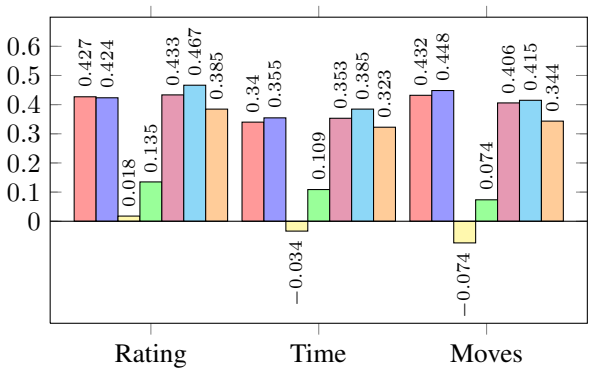


Fig. 5: Correlation coefficients between user difficulty Ratings, solving Time and number of Moves, and difficulty measures for Lazors. From left to right, the difficulty measures are level size, usable tiles, emitters, receivers, mirrors, reflections, and intersections.

	Rating	Scaled	Time	Moves
Unweighted	1.0957	1.5410	96.79%	72.84%
Weighted	1.0138	1.4337	108.39%	82.54%

TABLE II: Average errors of Lazors produced by the difficulty function when compared to user study data. Shown for difficulty rating, scaled difficulty, time, and number of moves, both unweighted and weighted by number of level ratings.

C. Move

For Move we again get similar observations. The variable counterintuitive moves has the highest correlation with the difficulty, and the number of rocks has a negative correlation. The size of the level has only a small positive correlation. Table III shows that an error of 0.93 on a scale of 1–10 is obtained in the weighted case.

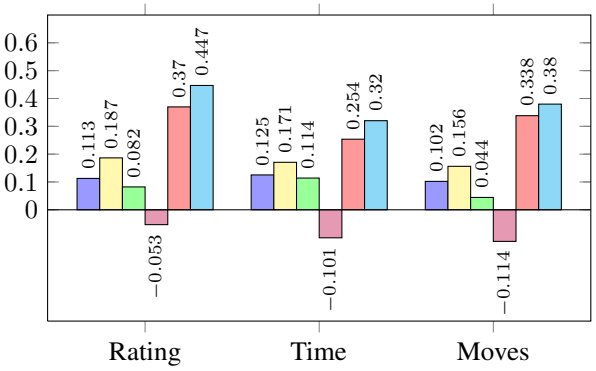


Fig. 6: Linear correlation coefficients between user difficulty ratings, solving times and number of moves, and difficulty measures for Move. From left to right, the difficulty measures are level size, balls, colors, rocks, moves, and counterintuitive moves

	Rating	Scaled	Time	Moves
Unweighted	0.9456	1.3399	84.05%	63.45%
Weighted	0.9276	1.2930	86.34%	64.35%

TABLE III: Average errors of Move.

VII. Conclusion and Future Work

We presented a methodology to determine the difficulty of a level in a puzzle game automatically after a user study on a number of levels. We tested the method on three simple puzzle games, Flow, Lazors, and Move, and showed that the difficulty can be estimated with an error of roughly 1 or less on a scale of 1–10. This means that our difficulty function can give a quite good estimation. For solving time or number of moves, a similar approach gives less good estimates. Along the way we studied various features in these puzzle games, classified them as initial, solution or dynamic, and determined how they correlated with different ways of measuring difficulty: by user ratings, solution time, and number of moves. We expect that our methodology extends to other puzzle games that involve connecting, moving, or coloring.

Whether our methodology can also be used for puzzle-like games that involve some dexterity (Cut the Rope, Angry Birds) is topic of future research. It is also interesting to analyze whether more complex, non-linear combinations of features gives error reductions that are substantiate the increased complexity of the model.

Acknowledgements

M.L. was partially supported by the Netherlands Organisation for Scientific Research (NWO) under grant number 639.021.123.

References

[1] D. Ashlock and J. Schonfeld, "Evolution for automatic assessment of the difficulty of Sokoban boards," in *IEEE Congress on Evolutionary Computation (CEC)*, July 2010, pp. 1–8.

[2] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," in *IEEE Congress on Evolutionary Computation (CEC)*, 2007, pp. 1382–1389.

[3] P. Jarušek and R. Pelánek, "Difficulty rating of Sokoban puzzle," in *Proc. Fifth Starting AI Researchers' Symposium (STAIRS)*. IOS Press, 2010, pp. 140–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=1940526.1940539

[4] C. Browne, *Evolutionary Game Design*. Springer, 2011.

[5] M.-V. Aponte, G. Levieux, and S. Natkin, "Measuring the level of difficulty in single player video games," *Entertainment Computing*, vol. 2, no. 4, pp. 205–213, 2011.

[6] S. András, K. Sipos, and A. Sóos, "Which is harder?-classification of happy cube puzzles," 2013.

[7] M. Guid and I. Bratko, "Search-based estimation of problem difficulty for humans," in *Artificial Intelligence in Education, 16th International Conference AIED*, ser. LNAI, no. 7926, 2013, pp. 860–863.

[8] J. Taylor and I. Parberry, "Procedural generation of sokoban levels," in *Proc. 6th Annual North American Conference on AI and Simulation in Games (GAMEON-NA)*, 2011, pp. 5–12.

[9] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*. Athena Scientific, 1997.

[10] M. Dyer, N. Megiddo, and E. Welzl, "Linear programming," in *Handbook of Discrete and Computational Geometry*, 2nd ed., J. E. Goodman and J. O'Rourke, Eds. Chapman and Hall/CRC, 2004, pp. 999–1014. [Online]. Available: http://dx.doi.org/10.1201/9781420035315.pt6

[11] N. Draper and H. Smith, *Applied Regression Analysis*, ser. Applied Regression Analysis. Wiley, 1981, no. dl. 766. [Online]. Available: http://books.google.nl/books?id=7mtHAAAAMAAJ