Data-Driven Sokoban Puzzle Generation with Monte Carlo Tree Search

Bilal Kartal, Nick Sohre, and Stephen J. Guy
Department of Computer Science and Engineering
University of Minnesota
(bilal,sohre, sjguy)@cs.umn.edu
http://motion.cs.umn.edu/r/sokoban-pcg

Abstract

In this work, we propose a Monte Carlo Tree Search (MCTS) based approach to procedurally generate Sokoban puzzles. Our method generates puzzles through simulated game play, guaranteeing solvability in all generated puzzles. We perform a user study to infer features that are efficient to compute and are highly correlated with expected puzzle difficulty. We combine several of these features into a data-driven evaluation function for MCTS puzzle creation. The resulting algorithm is efficient and can be run in an anytime manner, capable of quickly generating a variety of challenging puzzles. We perform a second user study to validate the predictive capability of our approach, showing a high correlation between increasing puzzle scores and perceived difficulty.

Introduction

Puzzle games play an integral role in entertainment, intellectual exercise, and our understanding of complex systems. Generating these puzzles automatically can reduce bottlenecks in design, and help keep games new, varied, and exciting. Furthermore, generating a variety of puzzles with controlled difficulty allows us to custom tailor game experiences to serve a much wider population, including those with little previous video game or puzzle solving experience.

Here, we study the above challenges within the context of the puzzle game of Sokoban. Developed for the Japanese game company *Thinking Rabbit* in 1982, Sokoban involves organizing boxes by pushing them with a player controlled agent on a discrete grid board. The goal of this work is to produce a system that automatically generates Sokoban puzzles. In order to support the dynamic needs of a large variety of users, our system needs to address several challenges inherent in the field of puzzle generation. These include the speed of the system, supporting on-demand puzzle generation, and producing a variety of puzzles. These properties support a range of player skills, and are key factors in keeping player experiences engaging.

Current methods for Sokoban puzzle generation tend to use exponential time algorithms that require templates or other human input. Achieving the goal of a fast, varied, and predictive system requires overcoming several challenges in automating the understanding of puzzle difficulty and generating puzzles of desired difficulty levels. Our work moves towards addressing these challenges via the following contributions:

- *Assessing level difficulty.* We utilize a user study to annotate the perceived difficulty of an initial set of Sokoban puzzles.
- *Learning features predictive of difficulty.* We use statistical analysis to infer features that are predictive of puzzle difficulty and are efficient to compute.
- *Generating varied, solvable puzzles that optimize key features.* We formulate puzzle generation as an MCTS optimization problem, modeling the search tree structure such that puzzles are generated through simulated game play.

The result is an anytime algorithm that produces levels of varying difficulty that are guaranteed to be solvable. To the best of our knowledge, this is the first such system to combine simulated gameplay and level optimization into a single stochastic tree search for puzzle generation

Figure 1: A high scoring 5x5 Sokoban puzzle generated by our method. The goal is to move the agent to push boxes (brown squares) so that all goals (yellow discs) are covered by the boxes. Yellow filled boxes represent covered goals. Obstacles (gray squares) block both agent and box movement

This work has been supported in part by the NSF through grant #CHS-1526693.

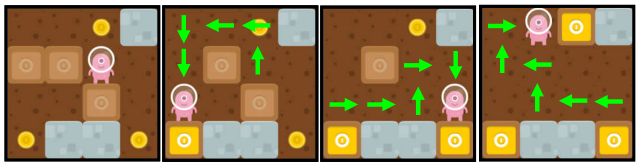

Figure 2: A generated Sokoban puzzle with solution (score = 0.31).

Background

There have been many applications of Procedural Content Generation (PCG) methods to puzzle games, such as genetic algorithms for *Spelunky* (Baghdadi et al. 2015), Bayesian Network based approaches (Summerville et al. 2015) for dungeons generation, map generation for video games (Snodgrass and Ontanon 2015), and regular expression based level generation (Maung and Crawfis 2015). Other approaches propose search as a general tool for puzzle generation (Sturtevant 2013), and generation of different start configurations for board games to tune difficulty (Ahmed, Chatterjee, and Gulwani 2015). Some even dynamically adapt to player actions (Stammer, Gunther, and Preuss 2015). Smith and Mateas (2011) propose an answer set programming based paradigm for PCGs for games and beyond. A recent approach parses game play videos to generate game levels (Guzdial and Riedl 2015). We refer the reader to the survey (Togelius et al. 2011) for a more detailed overview. Closely related to our work, Shaker et al. (2015) proposed a method for the game of *Cut the Rope* where the simulated game play is used to verify level playability.

Sokoban Puzzle

The Sokoban game board is composed of a two-dimensional array of contiguous tiles, each of which can be an obstacle, an empty space, or a goal. Each goal or space tile may contain at most one box or the agent. The agent may move horizontally or vertically, one space at a time. Boxes may be pushed by the agent, at most one at a time, and neither boxes nor the agent may enter any obstacle tile. The puzzle is solved once the agent has arranged the board such that every goal tile also contains a box. We present an example solution to a Sokoban puzzle level in Figure 2.

Previous work has investigated various aspects of computational Sokoban including automated level solving, level generation, and assessment of level quality.

Sokoban Solvers Previously proposed frameworks for Sokoban PCG involve creating many random levels and analyzing the characteristics of feasible solutions. However, solving Sokoban puzzles has been shown to be PSPACE-complete (Culberson 1999). Some approaches have focused on reducing the effective search domain (Junghanns and Schaeffer 2001). Recently, Pereira et al. (2015) have proposed an approach for solving Sokoban levels optimally, finding the minimum necessary number of box pushes. Pure MCTS has been shown to perform poorly for solving Sokoban puzzles (Perez, Samothrakis, and Lucas 2014).

Level Generation While there have been many attempts for solving Sokoban puzzles, the methods for their procedural generation are less explored. To the best of our knowledge, Murase et al. (1996) proposed the first Sokoban puzzle generation method which initializes a level by using templates, and proceeds with an exponential time solvability check. More recently, Taylor and Parberry (2011) proposed a similar approach, using templates for empty rooms and enumerating box locations in a brute-force manner. Their method can generate compelling levels that are guaranteed to be solvable. However, the run-time is exponential, and the method does not scale to puzzles with more than a few boxes.

Level Assessment There have been several efforts to assess the difficulty of puzzle games. One example is the very recent work of (Van Kreveld, Loffler, and Mutser 2015), where features common to puzzle games are combined into a difficulty function, which is then tuned using user study data. Others consider Sokoban levels specifically, comparing heuristic based problem decomposition metrics with user study data (Jarušek and Pelánek 2010), and using genetic algorithm solvers to estimate difficulty (Ashlock and Schonfeld 2010). More qualitatively, Taylor et al. (2015) have conducted a user-study and concluded that computer generated Sokoban levels can be as engaging as those designed by human experts.

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a best-first search algorithm that has been successfully applied to many games (Frydenberg et al. 2015; Steinmetz and Gini 2015; Sturtevant 2015; Silver et al. 2016), and a variety planning domains such as multi-agent narrative generation (Kartal, Koenig, and Guy 2014). More recently, MCTS has been employed to simulate different player models to improve game design process (Zook, Harrison, and Riedl 2015; Holmgård et al. 2015). For a more comprehensive discussion on MCTS, we refer the reader to the MCTS survey in (Browne et al. 2012).

MCTS proceeds in four phases of selection, expansion, rollout and backpropagation. Each node in the tree represents a complete state of the domain. Each link in the tree represents one possible action from the set of valid actions in the current state, leading to a child node representing the resulting state after applying that action. The root of the tree is the initial state, which is the initial configuration of the Sokoban puzzle board including the agent location. The MCTS algorithm proceeds by repeatedly adding one node at a time to the current tree. Given that a single action from any one node is unlikely to find a complete solution, i.e. a Sokoban puzzle for our purposes, MCTS adds several random actions referred to as *rollouts*. The full action sequence, which corresponds to the candidate puzzle, is evaluated. For each action, we keep track of the number of times it is tried, and its average evaluation score.

Exploration vs. Exploitation Dilemma Choosing which child node to expand (i.e., choosing which action to take) becomes an exploration/exploitation problem. We want to primarily choose actions that had good scores, but we also

need to explore other possible actions in case the observed empirical average scores do not represent the true reward mean of that action. In this work, we employ Upper Confidence Bounds (UCB) (Auer, Cesa-Bianchi, and Fischer 2002), a selection algorithm that seeks to balance this exploration/exploitation dilemma. By using UCB, the tree can grow in an uneven manner, biased towards better solutions.

Anytime Formulation with MCTS

One of the challenges for generating Sokoban puzzles is ensuring solvability of the generated levels. Since solving Sokoban has been shown to be PSPACE-complete, directly checking whether a solution exists for a candidate puzzle becomes intractable with increasing puzzle size. To overcome this challenge, we exploit the fact that a puzzle can be generated through simulated gameplay. To do so, we decompose the puzzle generation problem into two phases: puzzle initialization and simulated gameplay. Puzzle initialization refers to assigning the box start locations, empty tiles, and obstacle tiles. Simulated gameplay consists of a simulated player performing sequences of box pushes to determine goal locations. As the agent moves around during the simulation, it pushes boxes to different locations. A final snapshot of the resulting board configuration defines goal locations for boxes.

We apply MCTS by formulating the puzzle creation problem as an optimization problem. The main reasons for using MCTS to generate Sokoban puzzles include its success in problems with large branching factors, the anytime property, and the search structure that guarantees solvability. As discussed above, the search tree is structured such that the game can be generated by simulated gameplay. The search is conducted over both puzzle initializations and gameplay actions. Because the simulated gameplay is conducted using Sokoban game rules, invalid paths are never generated. In this way, our method is guaranteed to generate only solvable levels.

Anytime algorithms return a valid solution (if a solution exists) even if it is interrupted at any time. Given that our problem formulation is completely deterministic, MCTS can store the best found puzzle after rollouts during the search and optionally halt the search at some quality threshold. This behavior also enables us to create many puzzle levels from a single MCTS run with monotonically increasing scores.

Action set

Our search tree starts with a board fully tiled with obstacles, except for the agent start position. Initially, the following actions are possible at any node in the search tree:

1. *Delete obstacle*: An obstacle that is adjacent to an empty space is replaced with an empty space. This progressive obstacle deletion prevents boards from containing unreachable regions.
2. *Place box*: A box may be placed in any empty tile.
3. *Freeze level*: This action takes a snapshot of the board and saves it as the start configuration of the board.

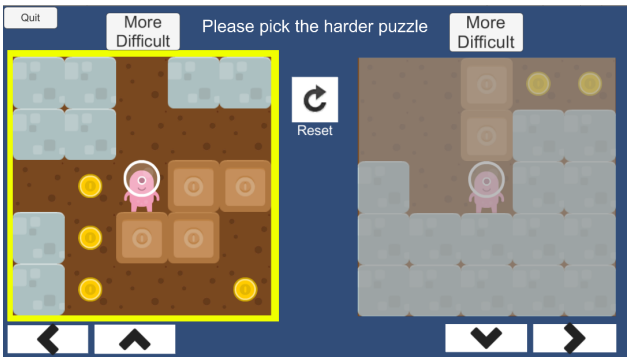

Figure 3: Our user study application, which presents pairs of puzzles to subjects and asks them to identify the one that is more challenging. Subjects were able to play each level presented as much or as little as desired before making a decision.

After the *Freeze level* action is chosen, the action set for descendant nodes of the frozen puzzle node is replaced by two new actions:

1. *Move agent*: This action moves the agent on the game board. The agent cannot move diagonally. This action provides the *simulated gameplay* mechanism, where the boxes are pushed around to determine goal positions.
2. *Evaluate level*: This action is the terminal action for any action chain; it saves the rearranged board as the solved configuration of the puzzle (i.e. current box locations are saved as goal locations).

These two action sets separate the creation of initial puzzle configurations (actions taken until the level is frozen) from simulated gameplay (agent movements to create goal positions). A key property of this two-phase approach is that it maintains the uniqueness of states throughout the tree; no two nodes represent the same board layout and agent path. This helps improve efficiency by reducing redundant search paths.

Once the *Evaluate level* action is chosen, we apply a simple post-processing to the board in order to remove elements that are known to be uninteresting. In particular, we turn all boxes that are never pushed by the agent into obstacles as this does not violate any agent movement actions. We also replace boxes that are pushed only once with an empty space (and delete the associated goal). This post-processing is performed before evaluating the level.

A critical component of our MCTS formulation that has yet to be addressed is the evaluation function. As MCTS is an optimization algorithm, we must provide it with an objective function that describes the desired properties of candidate puzzles. To accomplish this, the function maps from candidate puzzles to a score dependent upon how difficult or interesting the puzzle is. This involves finding features of Sokoban puzzles that can be computed quickly and are predictive of puzzle difficulty. We propose a data driven way to produce such a function in the following section.

Data-Driven Evaluation Function

Our goal is to generate levels which are not only solvable, but also engaging or difficult. We address this with a data-driven approach. First, we perform a user study analyzing the perceived difficulty of Sokoban puzzles. We then use this analysis to propose and validate new features estimating the level difficulty. Finally, we utilize these inferred features in our MCTS framework to efficiently generate Sokoban puzzles.

Estimating Perceived Difficulty

One challenge in taking a data-driven approach for difficulty estimation is the lack of large datasets of Sokoban puzzles that have known difficulty. The purpose of our user study was to create such a dataset. To facilitate this, we developed a custom Sokoban player application where users are shown two levels and asked to select the one that is more difficult (Figure 3). The users can switch between shown levels anytime and decide on the harder level without needing to complete the games. This application was placed on an Android tablet and users were allowed to rate as many puzzle pairs as they liked.

We collected user ratings for 120 preexisting puzzles including both human-designed puzzles obtained from (Skinner 2000) and computer generated ones obtained from (Kartal, Sohre, and Guy 2016). Over the course of two weeks, we had approximately 30 participants provide 945 pairwise comparisons.

In order to estimate the perceived difficulty of each puzzle, we employed the TrueSkill Bayesian skill estimation system (Herbrich, Minka, and Graepel 2006). Briefly, each puzzle's estimated difficulty is represented by a Gaussian, with a mean at the estimated difficulty score, and a standard deviation representing the uncertainty in the estimation. Each time a puzzle is decided to be more difficult than another, its mean (estimated difficulty) increases and the other puzzle's mean decreases. The estimated uncertainty decreases as more ratings are gathered for each puzzle. These TrueSkill means typically range from 0 (least difficult) to 50 (most difficult), and are referred to in this paper as *Perceived Difficulty*.

Feature Analysis

The comparison results from the user study were compiled and used to annotate the puzzles with their perceived difficulty. Because the evaluation function is invoked for every *Evaluate level* action of MCTS, we restricted our search for features to only those that were efficient to compute. In particular, we do not include features based on an optimal solution, as finding an optimal solution is a PSPACE-complete task.

We tested several features for correlation with perceived difficulty, including:
Metrics analyzing the layout of obstacles and free space

- *Tile Mixing.* The number of free space tiles next to obstacle tiles, and obstacle tiles next to free space.
- *3x3 Block Count.* The number of tiles not in a 3x3 block of solid obstacles or open space.

Metrics which measured the placement of boxes and goals:

- *Box Count.* The number of boxes on the board.
- *Goal Distance.* The average distance between all possible pairings of boxes and goals

And metrics which measured how congested the paths from boxes to their goals were:

- *Congestion v1.* A weighted sum of the number of boxes, goals, and obstacles in the bounding rectangle between a box and its goal.
- *Congestion v2.* A refinement on the above congestion measure designed to maximize correlation with perceived difficulty (see below).

Importantly, each of these metrics can be computed in just a few microseconds, even for larger boards, allowing them to be efficiently used during MCTS rollout evaluation.

To test the efficacy of candidate features, the signed Pearson correlation coefficient r was computed for each feature with respect to perceived difficulty of the puzzles. For features which contained tuning parameters, we ran a grid search to find which parameters yielded the highest correlation. Table 1 shows the correlation between each metric and the perceived difficulties of the puzzles. We also show the correlation with only the procedural generated puzzles (PCG) tested, as the human crafted puzzles tended to have a significant effect on the analysis.

Looking at the correlations we can see several interesting trends. For example, the tile mixing metric is well correlated with difficulty for the entire dataset, but when only computer generated levels are considered the metric is not very predictive. In contrast, the simpler metric penalizing 3x3 blocks is more consistent. Likewise, the total distance the user must push all the boxes is slightly less correlated than the simpler approach of just counting the number of boxes to push (recall that boxes that are not pushed at least two spaces will be removed).

Simpler methods were not always the most predictive. In particular, the first version of the congestion metric was a weighted sum of the number of initial box positions b_i, number of goals g_i, and number of obstacles o_i within the bounding rectangle between the start and the goal for each box i. That is

$$\sum_{i=1}^{n} \alpha b_i + \beta g_i + \gamma o_i. \tag{1}$$

where, n is the number of boxes to be pushed, and α, β, and γ are scaling weights. While intuitive and relatively well correlated with difficulty for computer generated puzzles, this simple metric was almost completely uncorrelated with level difficulty when including human designed puzzles, even after tuning the values of α, β, and γ. Investigating the puzzles suggests this lack of correlation arises in part because the metric rewards pushing a box past obstacles even if there are no other boxes directly in the way. To address this issue, we refined the metric to be

$$\sum_{i=1}^{n} \frac{\alpha b_i + \beta g_i}{\gamma(A_i - o_i)}, \tag{2}$$

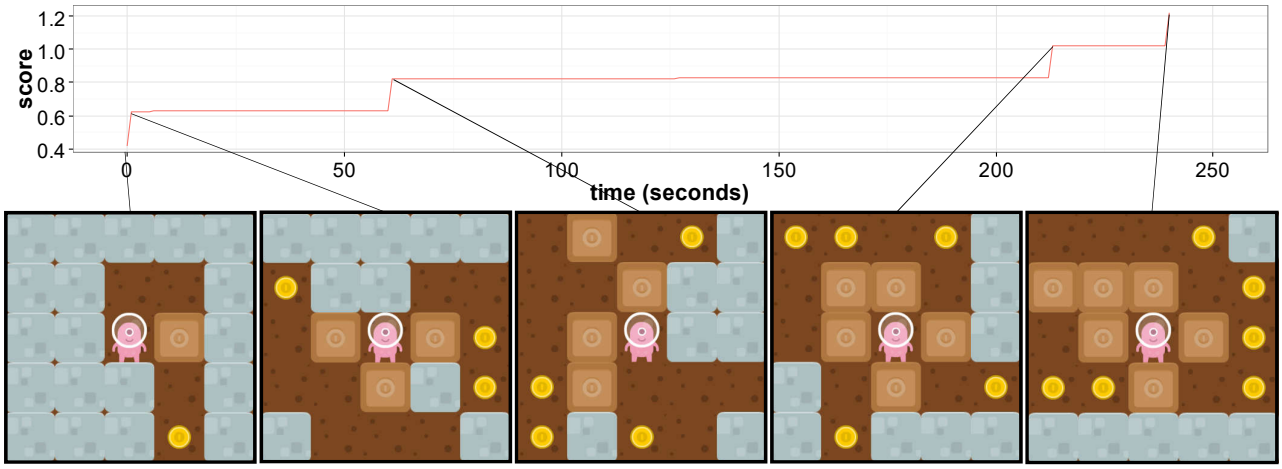

Figure 4: Generating Level Sets. (Top) The evolution of the best score from a single run of MCTS. (Bottom) Several levels generated from the same run. Later levels have higher score, and are therefore predicted to be more difficult.

Feature	r value (PCG levels)	r value (all levels)
Tile Mixing	-0.05	0.17
3x3 Block	0.09	0.24
Box Count	0.48	0.23
Goal distance	-0.16	0.20
Congestion v1	0.41	0.05
Congestion v2	0.43	0.32

Table 1: Correlation (Pearson r correlation coefficients) for six features from the user study. The most correlated features were used for level evaluation function.

where A_i is the total area enclosed in the rectangle from the box i to its goal. The intent was to make the metric reward box paths that actually encounter boxes and obstacles, instead of just having them nearby an otherwise unconstrained path. While this was a small change to the measure of congestion, this new metric now correlates well with difficulty in both procedurally-generated and human-generated levels.

Level Evaluation

Using the results from our feature analysis, we developed an evaluation function for use in MCTS. For game playing AI, evaluation functions generally map to 0 for loss, 0.5 for a tie, and 1 for a win, with MCTS implementations typically calibrated to optimize on this scale. For Sokoban puzzle generation, this is not directly applicable (as the measure of success is not analogous to loss/tie/win). Instead, we propose to use a weighted combination of puzzle features to estimate the difficulty on a scale close to this 0 to 1 range.

By optimizing several metrics which are each independently correlated with puzzle difficulty, MCTS can be used to find more difficult puzzles than optimizing any one feature alone. Here, we used the *Box Count*, *3x3 Blocks*, and *Congestion v2*, as they were the most positively correlated with difficulty. Additionally, each of these metrics captures an intuitive aspect of what makes an interesting Sokoban level: the 3x3 Blocks metric (P_b) rewards heterogeneous landscapes, and discourages large clearings which are easy to navigate; the Congestion metric (P_c) rewards box paths which overlap with each other and are thereby likely to develop precedence constraints between box pushes; and the Box count (n) rewards levels with more boxes which makes complex interactions between boxes more likely. The resulting function is as follows:

$$f(P) = \frac{w_b P_b + w_c P_c + w_n n}{k} \quad (3)$$

The parameter k is employed to help normalize scores to the range of 0 to 1, though some of our top scoring puzzles can fall outside of this range.

While generating puzzles, $f(P)$ was used to evaluate MCTS rollouts. The weights w_c, w_b, and w_n were set empirically to be 10, 5, and 1 respectively and k was set to 50. Other weights can be used, and will lead to different puzzles being generated.

Generating Level Sets

A given run of the MCTS tree search will generate several levels of increasing (predicted) difficulty. We exploit this feature to reach our goal of creating a *level set*, that is, a series of levels that are of increasing difficulty. Because MCTS is an anytime algorithm that explores a wide, randomized section of the search space, it is well suited for this task; each run of MCTS creates several levels as it explores deeper in the tree, each with increasing difficulty.

While each run of MCTS can generate a large number of levels, many are slight variations of each other. To help create variation in the level sets, we chose a subset of these levels with different estimated difficulty scores. Figure 4 shows the results from one of these level sets. The entire run of MCTS for this set took 240s, and generated 20 of levels of varying difficulty.

Score	Size (Empty Tiles)	Num. Boxes	Computation Time (s)
0.4	10.8	2.0	0.01
0.8	16.8	4.2	0.54
1.2	21.1	6.0	39.7
1.4	27.3	7.0	120

Table 2: Puzzle Scaling. Average computation time to find puzzles of various scores and size. While smaller puzzles with few boxes can be found in a under a second, scaling to larger sizes and box counts requires several minutes of computation. Results are averaged over 5 runs with with different random seeds.

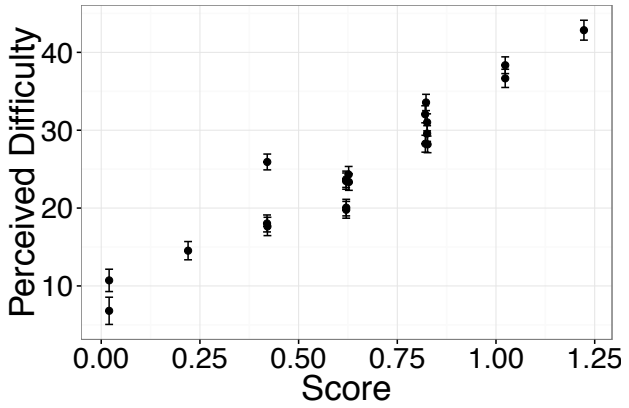

Figure 5: The evaluation score of the puzzles in the generated level set were well correlated with perceived difficulty (r^2 = .91).

Analysis and Discussion

Our approach efficiently generates dozens of levels with monotonically increasing scores within 5 minutes on on a laptop using a single core of an Intel i7 2.2 GHz processor with 8GB memory. We observe that it generates more levels with low and medium scores than high scores. An instance of this behavior can be seen in Figure 4.

To validate our updated evaluation function, we performed a second user-study on 20 levels generated in a single run (a subset of which is shown in Figure 4). This user study included 6 participants who provided 210 level comparisons. The perceived difficulty scores were then compared to the scores assigned by our evaluation function. The results can be seen in Figure 5. We observe a very high correlation ($r^2 = 0.91, p < 0.001$) between the perceived difficulty of a level and the score assigned by MCTS. This confirms that MCTS will produce levels of increasing difficulty by optimizing this function.

Our method is capable of producing a wide variety of levels. Because MCTS is a stochastic algorithm, each run naturally generates different levels from previous runs, and even within a single run (see Figure 4). Additional variation can be achieved by changing the maximum size of the board, randomizing the start position of the agent, or limiting the number of boxes in a level (see Figure 6). Figures 1, 2, 3, and 6 showcase the variety in the puzzles that are generated.

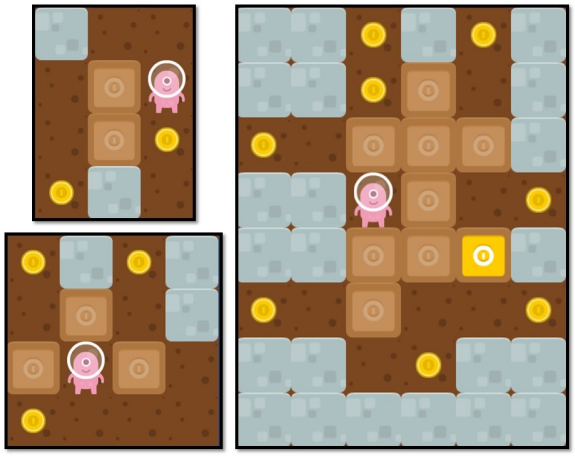

Figure 6: Procedurally generated puzzles of varying sizes

Limitations Generation of large puzzles remains a bottleneck as the time grows exponentially as the explored space and number of boxes grow linearly (see Table 2). This is due to a quickly growing branching factor in the puzzle initialization phase; every *Delete obstacle* action that is taken adds up to three more available *Delete obstacle* actions. Additionally, there are some generated puzzles that were perceived to be more difficult than those with a higher score (Figure 5); this suggests there are some aspects of difficulty our score does not capture well.

Conclusions

In this work we have proposed and implemented a method for Sokoban puzzle generation. We formulated the problem as MCTS optimization, generating puzzles through simulated gameplay to ensure solvability. We developed an evaluation function with a data-driven approach, utilizing a user study to find puzzle features well correlated with perceived difficulty. Our method is efficient, producing a variety of puzzles with monotonically increasing scores within minutes. We validated our evaluation function through an additional user study and show that it correlates very well with perceived difficulty.

Going forward, we plan to investigate ways of efficiently creating larger puzzles of increasing difficulty. Some ways to overcome the current challenges in scaling up puzzles may include composing larger puzzles from smaller puzzle elements, reducing the size of the search space via data-driven heuristics, and exploring if some properties of optimal solutions may be computed quickly. Additionally, we plan to perform a user study focused on larger levels and those of very high difficulty. We also plan to parallelize MCTS to increase computation performance. There are also other extensions of our method for future research. We intend to explore the abstraction of our method and its application to other puzzle games and game genres. Additionally, we will study the extent to which the difficulty and fun aspects of puzzles differ. Lastly, we plan to study human designed levels and better incorporate these into our data driven approach.

References

Ahmed, U. Z.; Chatterjee, K.; and Gulwani, S. 2015. Automatic generation of alternative starting positions for simple traditional board games. In *Twenty-Ninth AAAI Conf. on Artificial Intelligence*.

Ashlock, D., and Schonfeld, J. 2010. Evolution for automatic assessment of the difficulty of sokoban boards. In *Evolutionary Computation*, 1–8.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Baghdadi, W.; Eddin, F. S.; Al-Omari, R.; Alhalawani, Z.; Shaker, M.; and Shaker, N. 2015. A procedural method for automatic generation of spelunky levels. In *Applications of Evolutionary Computation*. 305–317.

Browne, C. B.; Powley, E.; Whitehouse, D.; et al. 2012. A survey of Monte Carlo Tree Search methods. *IEEE Trans. on Computational Intelligence and AI in Games* 4(1):1–43.

Culberson, J. 1999. Sokoban is PSPACE-complete. In *Proceedings in Informatics*, volume 4, 65–76.

Frydenberg, F.; Andersen, K. R.; Risi, S.; and Togelius, J. 2015. Investigating MCTS modifications in general video game playing. In *IEEE Computational Intelligence and Games*, 107–113.

Guzdial, M., and Riedl, M. O. 2015. Toward game level generation from gameplay videos. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

Herbrich, R.; Minka, T.; and Graepel, T. 2006. Trueskill: A bayesian skill rating system. In *Advances in Neural Information Processing Systems*, 569–576.

Holmgård, C.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2015. Monte-carlo tree search for persona based player modeling. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Jarušek, P., and Pelánek, R. 2010. Difficulty rating of sokoban puzzle. In *Stairs*, volume 222, 140.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1):219–251.

Kartal, B.; Koenig, J.; and Guy, S. J. 2014. User-driven narrative variation in large story domains using monte carlo tree search. In *Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, 69–76.

Kartal, B.; Sohre, N.; and Guy, S. 2016. Generating sokoban puzzle game levels with monte carlo tree search. In *The IJCAI-16 Workshop on General Game Playing*.

Maung, D., and Crawfis, R. 2015. Applying formal picture languages to procedural content generation. In *Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), 2015*, 58–64.

Murase, Y.; Matsubara, H.; and Hiraga, Y. 1996. Automatic making of sokoban problems. In *PRICAI'96*. 592–600.

Pereira, A. G.; Ritt, M.; and Buriol, L. S. 2015. Optimal sokoban solving using pattern databases with specific domain knowledge. *Artificial Intelligence* 227:52–70.

Perez, D.; Samothrakis, S.; and Lucas, S. 2014. Knowledge-based fast evolutionary mcts for general video game playing. In *IEEE Computational Intelligence and Games*, 1–8.

Shaker, M.; Shaker, N.; Togelius, J.; and Abou-Zleikha, M. 2015. A progressive approach to content generation. In *Applications of Evolutionary Computation*. Springer. 381–393.

Silver, D.; Huang, A.; Maddison, C. J.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Skinner, D. W. 2000. Sokoban puzzle dataset microban. http://www.abelmartin.com/rj/sokobanJS/Skinner/David W. Skinner - Sokoban.htm.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):187–200.

Snodgrass, S., and Ontanon, S. 2015. A hierarchical mdmc approach to 2d video game map generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Stammer, D.; Gunther, T.; and Preuss, M. 2015. Player-adaptive spelunky level generation. In *Computational Intelligence and Games, 2015 IEEE Conference on*, 130–137.

Steinmetz, E., and Gini, M. 2015. Mining expert play to guide monte carlo search in the opening moves of go. In *Int'l Joint Conf. on Artificial intelligence*, 801–807.

Sturtevant, N. 2013. An argument for large-scale breadth-first search for game design and content generation via a case study of fling. In *AI in the Game Design Process*.

Sturtevant, N. R. 2015. Monte carlo tree search and related algorithms for games. *Game AI Pro 2: Collected Wisdom of Game AI Professionals* 265.

Summerville, A. J.; Behrooz, M.; Mateas, M.; and Jhala, A. 2015. The learning of zelda: Data-driven learning of level topology. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

Taylor, J., and Parberry, I. 2011. Procedural generation of sokoban levels. In *North American Conf. on Intelligent Games and Simulation*, 5–12.

Taylor, J.; Parsons, T. D.; and Parberry, I. 2015. Comparing player attention on procedurally generated vs. hand crafted sokoban levels with an auditory stroop test. In *Conf. on the Foundations of Digital Games*.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):172–186.

Van Kreveld, M.; Loffler, M.; and Mutser, P. 2015. Automated puzzle difficulty estimation. In *IEEE Computational Intelligence and Games*, 415–422.

Zook, A.; Harrison, B.; and Riedl, M. O. 2015. Monte-carlo tree search for simulation-based strategy analysis. In *the 10th Conf. on the Foundations of Digital Games*.