Generating Sokoban Puzzle Game Levels 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

Abstract

In this work, we develop a Monte Carlo Tree Search based approach to procedurally generate Sokoban puzzles. To this end, we propose heuristic metrics based on surrounding box path congestion and level tile layout to guide the search towards interesting puzzles. Our method generates puzzles through simulated game play, guaranteeing solvability in all generated puzzles. Our algorithm is efficient, capable of generating challenging puzzles very quickly (generally in under a minute) for varying board sizes. The ability to generate puzzles quickly allows our method to be applied in a variety of applications such as procedurally generated mini-games and other puzzle-driven game elements.

1 Introduction

Understanding and exploring the inner workings of puzzles has exciting implications in both industry and academia. Many games have puzzles either at their core (e.g. *Zelda: Ocarina of Time, God of War*) or as a mini-game (e.g. lock picking and terminal hacking in *Fallout 4* and *Mass Effect 3*). Generating these puzzles automatically can reduce bottlenecks in design phase, and help keep games new, varied, and exciting.

In this paper, we focus on the puzzle game of Sokoban. Developed for the Japanese game company *Thinking Rabbit* in 1982, Sokoban involves organizing boxes by pushing them with an agent across a discrete grid board. Sokoban is well suited for consideration for several reasons. A well-known game, Sokoban exhibits many interesting challenges inherent in the general field of puzzle generation. For example, the state space of possible configurations is very large (exponential in the size of the representation), and thus intractable for search algorithms to traverse. Consequently, ensuring generated levels are solvable can be difficult to do quickly. Furthermore, it is unclear how to characterize what makes an initial puzzle state lead to an interesting or non-trivial solution. While Sokoban has relatively straightforward rules, even small sized puzzles can present a challenge for human solvers.

In this paper, we propose a method to procedurally generate Sokoban puzzles. Our method produces a wide range of puzzles of different board sizes of varying difficulty as evaluated by a novel metric we propose (Figure 1 shows an example puzzle generated by our method). Beyond simply being able to generate more content for Sokoban enthusiasts, we can envision puzzles such as these used as inspiration or direct input to game elements that correspond well to puzzle mechanics. Examples include skill-based minigames or traversing rooms with movable obstacles. Furthermore, human designed puzzles only represent a small fraction of the possible puzzle space. Procedural puzzle generation can enable the exploration of never before seen puzzles.

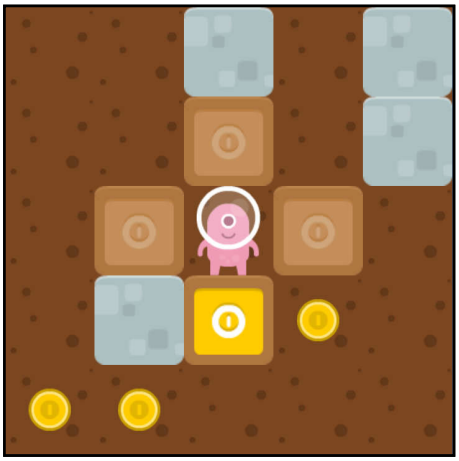

Figure 1: One of the highest scoring 5x5 puzzles generated by our method. The goal of the puzzle is to have the agent push boxes (brown squares) such that all goals (yellow discs) are covered by the boxes. Yellow filled boxes represent covered goals. Obstacles (gray squares) block both agent and box movement (Sprites from *The Open Bundle*[1])

Currently, the state of the art in procedural Sokoban puzzle generation tends to use exponential time algorithms that require templates or other human input. As a result, these methods can take hours or even days to generate solvable puzzles even on small boards. Finding ways to address this issue can significantly enhance our ability to achieve the potential benefits of procedurally generating Sokoban puzzles.

[1] `http://open.commonly.cc`

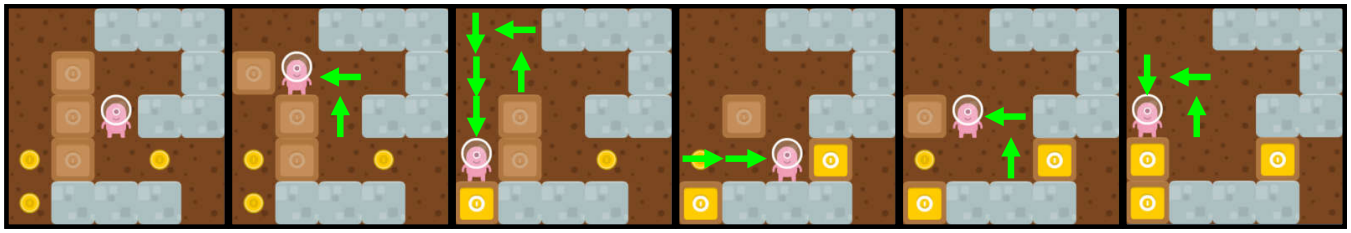

Figure 2: A solution to one of the generated Sokoban puzzles (score = 0.17). Each successive frame depicts the point at which the next box push was made, along with the movement actions (green arrows) the agent took to transition from the previous frame to the current one.

To that end, we propose the use of Monte Carlo Tree Search (MCTS) for this puzzle generation. We show that the generation of Sokoban puzzles can be formulated as an optimization problem, and apply MCTS guided by an evaluation metric to estimate puzzle difficulty. Furthermore, we model the MCTS search an an act of simulated gameplay. This alleviates current bottlenecks by eliminating the need to verify the solvability of candidate puzzles post-hoc. Overall, the contributions of this work are three-fold:

- We formulate the generation of Sokoban puzzles as an MCTS optimization problem.
- We propose a heuristic metric to govern the evaluation for the MCTS board generation and show that it produces puzzles of varying difficulty.
- Our method eliminates the need to check post-hoc for board solvability, while maintaining the guarantee that all of our levels are solvable.

2 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], MCTS based *Super Mario Bros* [Summerville *et al.*, 2015], and map generation for Physical TSP problem [Perez *et al.*, 2014b] and video games [Snodgrass and Ontanon, 2015]. Other approaches proposed search as a general tool for puzzle generation [Sturtevant, 2013], and generation of different start configurations for board games to tune difficulty [Ahmed *et al.*, 2015]. Recent work has looked at dynamically adapting games to player actions [Stammer *et al.*, 2015]. Smith and Mateas (2011) proposed 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]. 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. We refer readers to the survey [Togelius *et al.*, 2011] and the book [Shaker *et al.*, 2014] for a more comprehensive and thorough discussion of the PCG field, and to the survey particularly for PCG puzzles [Khalifa and Fayek, 2015].

2.1 Sokoban Puzzle

A 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 tile that contains a goal 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]. Several approaches have focused on proposing approximate solutions to reduce the effective search domain [Botea *et al.*, 2002; Junghanns and Schaeffer, 2001; Cazenave and Jouandeau, 2010].

Recently, Pereira *et al.* (2015) have proposed an approach that uses pattern databases [Edelkamp, 2014] for solving Sokoban levels optimally, finding the minimum necessary number of box pushes (regardless of agent moves). The authors in [Perez *et al.*, 2014a] applied MCTS for solving Sokoban levels, but concluded that pure MCTS performs poorly.

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 firstly creates 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 by [van Kreveld *et al.*, 2015], which combines features common to puzzle games 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.

2.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a best-first search algorithm that has been successfully applied to many games [Cazenave and Saffidine, 2010; Pepels *et al.*, 2014; Jacobsen *et al.*, 2014; Frydenberg *et al.*, 2015; Mirsoleimani *et al.*, 2015; Sturtevant, 2015; Steinmetz and Gini, 2015] and a variety planning domains such as multi-agent narrative generation [Kartal *et al.*, 2014], multi-robot patrolling [Kartal *et al.*, 2015] and task allocation [Kartal *et al.*, 2016], and others[Williams *et al.*, 2015; Sabar and Kendall, 2015; Hennes and Izzo, 2015]. Bauters *et al.* (2016) show how MCTS can be used for general MDP problems. More recently, Zook *et al.* (2015) adapted MCTS such that it simulates different skilled humans for games enabling faster gameplay data collection to automate game design process. We refer the reader to the survey on MCTS [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 actions from the root to the expanded node is unlikely to find a complete solution, i.e. a Sokoban puzzle for our purposes, MCTS uses random actions, a.k.a. *rollouts*. Then the full action sequence, which results in a candidate puzzle for our domain, obtained from both tree actions and random actions is evaluated. For each potential action, we keep track of how many times we have tried that action, and what the average evaluation score was.

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 don't represent the true reward mean of that action. This exploration/exploitation dilemma has been well studied in other areas.

Upper Confidence Bounds (UCB) [Auer *et al.*, 2002] is a selection algorithm that seeks to balance the exploration/exploitation dilemma. Using UCB with MCTS is also referred to as Upper Confidence bounds applied to Trees (UCT). Applied to our framework, each parent node p chooses its child s with the largest $UCB(s)$ value according to Eqn. 1. Here, $w(.)$ denotes the average evaluation score obtained by Eqn. 2, $\hat{\pi}_s$ is the parent's updated policy that includes child node s, p_v is visit count of parent node p, and s_v is visit count of child node s respectively. The value of C determines the rate of exploration, where smaller C implies less exploration. $C = \sqrt{2}$ is necessary for asymptotic convergence of MCTS [Kocsis and Szepesvári, 2006].

$$UCB(s) = w(\hat{\pi}_s) + C \times \sqrt{\frac{\ln p_v}{s_v}} \tag{1}$$

If a node with at least one unexplored child is reached ($s_v = 0$), a new node is created for one of the unexplored actions. After the rollout and back-propagation steps, the selection step is restarted from the root again. This way, the tree can grow in an uneven manner, biased towards better solutions.

Variations in Selection Methods. There are numerous other selection algorithms that can be integrated to MCTS. In this work, as a baseline, we employed UCB selection algorithm. However, considering a possible relationship between the variance of tree nodes and agent movement on the board, we experimented with UCB-Tuned [Auer *et al.*, 2002], and UCB-V [Audibert *et al.*, 2007] for Sokoban puzzle generation using our evaluation function shown in Eqn. 2.

UCB-Tuned and UCB-V both employ the empirical variance of nodes based on the rewards obtained from rollouts with the intuition that nodes with high variance need more exploration to better approximate their true reward mean. UCB-Tuned purely replaces the exploration constant of the UCB algorithm with an upper bound on the variance of nodes, and hence requires no tuning, whereas UCB-V has two additional parameters to control the rate of exploration.

3 Approach Overview

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 itself.

To do so, we decompose the puzzle generation problem into two phases: puzzle initialization and puzzle shuffling. Puzzle initialization refers to assigning the box start locations, empty tiles, and obstacle tiles. Puzzle shuffling consists of performing sequences of *Move agent* actions (listed in section 3.1) to determine goal locations. In a forward fashion, as the agent moves around during the shuffling phase, it pushes boxes to different locations. The final snapshot of the board after shuffling defines goal locations for boxes.

We apply MCTS by formulating the puzzle creation problem as an optimization problem. As discussed above, the search tree is structured so that the game can be generated by simulated gameplay. The search is conducted over puzzle initializations and valid puzzle shuffles. Because puzzle shuffles are conducted via a simulation of the Sokoban game rules, invalid paths are never generated. In this way our method is guaranteed to generate only solvable levels.

The main reasons for using MCTS for Sokoban puzzle generation is its success in problems with large branching factors and its anytime property. MCTS has been applied to many

problems with large search spaces [Browne *et al.*, 2012]. This is also the case for Sokoban puzzles as the branching factor is in the order of $\mathcal{O}(mn)$ for an $m \times n$ puzzle.

Anytime algorithms return a valid solution (given a solution is found) 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 with some quality threshold. This behavior also enables us to create many puzzle levels from a single MCTS run with varying increasing scores.

3.1 Action set

Our search tree starts with a board fully tiled with obstacles, except for an agent which is assumed to start at the center of the board. At any node, the following actions are possible in the search tree:

1. *Delete obstacles*: Initially, only the obstacle tiles surrounding the agent are available for deletion. Once there is an empty tile, its surrounding obstacle tiles can also be turned into an empty tile (this progressive obstacle deletion prevents boards from containing unreachable hole shaped regions).
2. *Place boxes*: 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.
4. *Move agent*: This action moves the agent on game board. The agent cannot move diagonally. This action provides the *shuffling* mechanism of the initialized puzzle where the boxes are pushed around to determine box goal positions.
5. *Evaluate level*: This action is the terminal action for any action chain; it saves the shuffled board as the solved configuration of the puzzle (i.e. current box locations are saved as goal locations).

This action set separates the creation of initial puzzle configurations (actions taken until the level is frozen) from puzzle shuffling (agent movements to create goal positions). Before move actions are created as tree nodes during MCTS simulations, the agent moves randomly during rollouts. As search is deepened, agent moves become more non-random.

Once *Evaluate level* action is chosen, before we apply our evaluation function to the puzzle, we apply a simple post-processing to the board. We turn all boxes that are placed to the board but never pushed by the agent into obstacles as this doesn't violate any agent movement actions. By applying evaluation function after post-processing, we make sure our heuristic metrics' values are correctly computed.

The proposed action set has several key properties which make it well suited for MCTS-based puzzle generation. In contrast to our approach, interleaving puzzle creation and agent movement requires the MCTS method to re-simulate all actions from the root to ensure valid action sequences, which would render the problem much more computationally expensive and MCTS less efficient.

3.2 Evaluation Function

MCTS requires an evaluation function to optimize over during the search. For game playing AI, evaluation functions typically map to 0 for loss, 0.5 for tie, and 1 for winning sequence of actions. For Sokoban puzzle generation, this is not directly applicable as we do not have winning and losing states. Instead, we propose to use a combination of two metrics, i.e. terrain and congestion, to generate interesting Sokoban levels. Our approach seeks to maximize the geometric mean of these two metrics as shown in Eqn. 2. This provides a good balance between these two metrics without allowing either term dominate. Parameter k is employed to normalize scores to the range of 0 to 1.

$$f(P) = \frac{\sqrt{Terrain \times Congestion}}{k} \tag{2}$$

These two components of our evaluation function are intended to guide the search towards boards that have congested landscapes with more complex box interactions.

Congestion Metric

The motivation for the congestion metric is to encourage puzzles that have some sort of congestion with respect to the paths from boxes to their goals. The intuition is that overlaps between box paths may encourage precedence constraints of box pushes. We compute this by counting the number of boxes, goals, and obstacles in between each box and its corresponding goal. Formally, given an $m \times n$ Sokoban puzzle P, and given b boxes on the board, let b^s denote the initial box locations, and let b^f denote the final box locations. For each box b_i, we create a minimum area rectangle r_i with corners of b_i^s and b_i^f. Within each rectangle r_i, let s_i denote the number of box locations, let g_i denote the number of goal locations, and let o_i denote the number of obstacles. Then

$$Congestion = \sum_{i=1}^{b} (\alpha s_i + \beta g_i + \gamma o_i). \tag{3}$$

where α, β, and γ are scaling weights.

Terrain Metric

The value of $Terrain$ term is computed by summing the number of obstacle neighbors of all empty tiles. This is intended to reward boards that have heterogeneous landscapes, as boards having large clearings or obstacles only on the borders make for mostly uninteresting puzzles.

4 Results

In this section, we firstly present a comparison between different selection algorithms. Then, we report the anytime performance of our algorithm. Lastly, we present a set of levels generated by our approach.

4.1 Experimental Design

All experiments are performed on a laptop using a single core of an Intel i7 2.2 GHz processor with 8GB memory. Search time is set to 200 seconds for all results and levels reported in

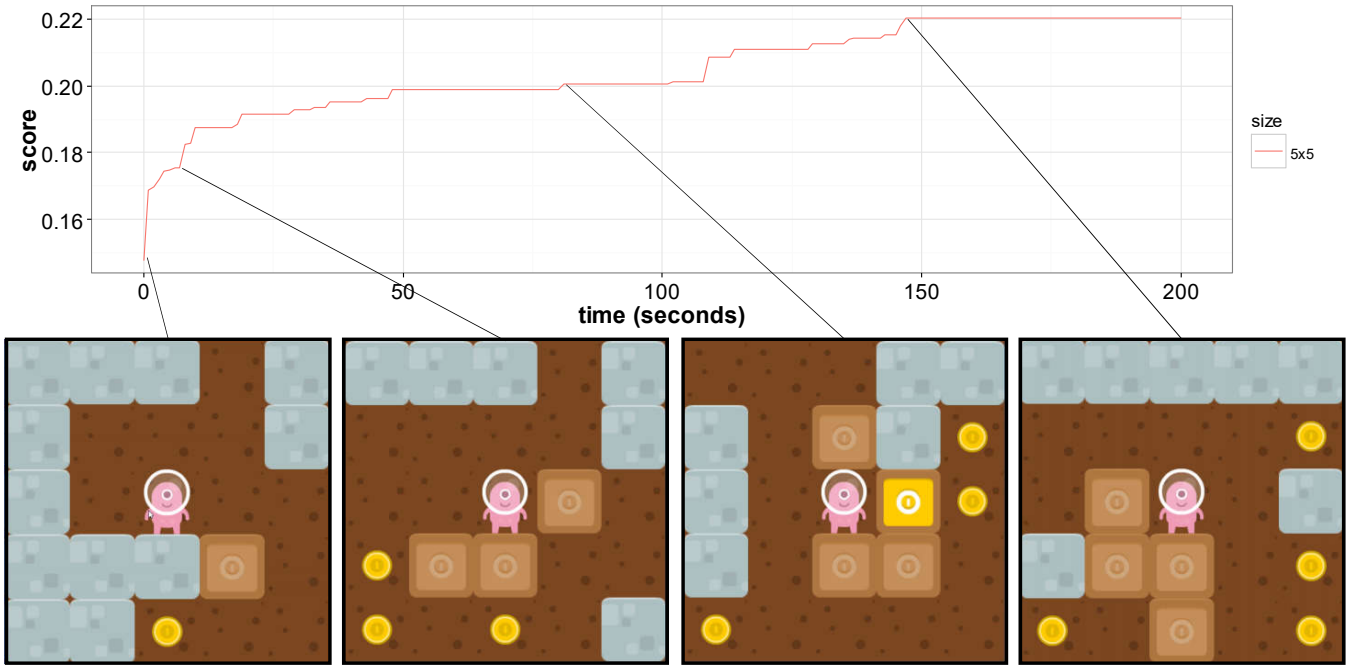

Figure 3: Our method works in an anytime fashion, improving the quality of generated puzzles over time. The red line corresponds to the performance of UCB-V for 5x5 board generation over time. The first two levels, i.e. generated instantaneously by MCTS, are trivial to solve. As time passes and the puzzle scores increase, more interesting puzzles (such as the last two shown above) are found. The anytime property of MCTS proves to be useful as a single MCTS run can generate a variety of levels.

this work. Given the simple game mechanics of Sokoban puzzles, our algorithm on average can perform about 80K MCTS simulations per second for 5x5 tile puzzles.

For our experiments, we set $C = \sqrt{2}$ for UCB which is presented in Eqn. 1. For the two parameters involved with UCB-V, we use those suggested in [Audibert *et al.*, 2007]. For the congestion metric (Eqn. 3), we used the following weights for all boards generated: $\alpha = 4$, $\beta = 4$, and $\gamma = 1$. The normalization constant of $k = 200$ is employed for all experiments. We ran our experiments for 5 runs of varying random seeds across each board size generated (5x5 to 10x10). Boards were saved each time the algorithm encountered an improvement in the evaluation function.

4.2 Computational Performance

Our method was able to produce approximately 700 candidate puzzles (about 100 per board size) on a single processor over a period of 6 hours. Because of the anytime nature of our approach, non-trivial puzzles are generated within a few seconds, and more interesting puzzles over the following minutes (see Figure 3). The anytime behavior of our algorithm for different puzzle board sizes is depicted in Figure 4. A puzzle set is shown in Figure 5.

We compare the results of different MCTS selection methods in Figure 6. While UCB-Tuned has been shown to outperform these selection algorithms for other applications [Perick *et al.*, 2012], our experiments reveal a slight advantage for UCB-V. Although all selection methods perform statistically similarly, UCB-V slightly outperforms other techniques consistently across different tested board sizes.

Compared to other methods that are exponential in the number of boxes, our method is a significant improvement on the run time. Our algorithm is capable of quickly generating levels with a relatively large number of boxes. As can be seen in Figure 5, puzzles with 8 boxes or more are found within a few minute time span. Previous search-based methods may not be able to generate these puzzles in a reasonable amount of time given the number of boxes and board size.

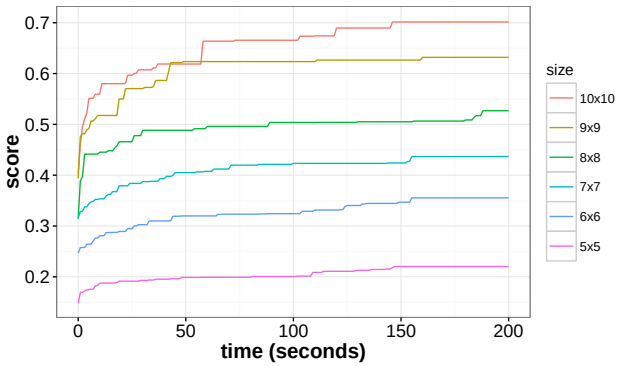

Figure 4: Anytime behavior of MCTS algorithm for varying board sizes is depicted; here each line corresponds to the average of the best score found so far by UCB-V across 5 runs with different seeds, grouped by the size of boards being generated. We have chosen UCB-V for display as it outperformed other methods for our domain. MCTS quickly finds trivially solvable puzzles within a second. Given more time, more interesting puzzles are discovered.

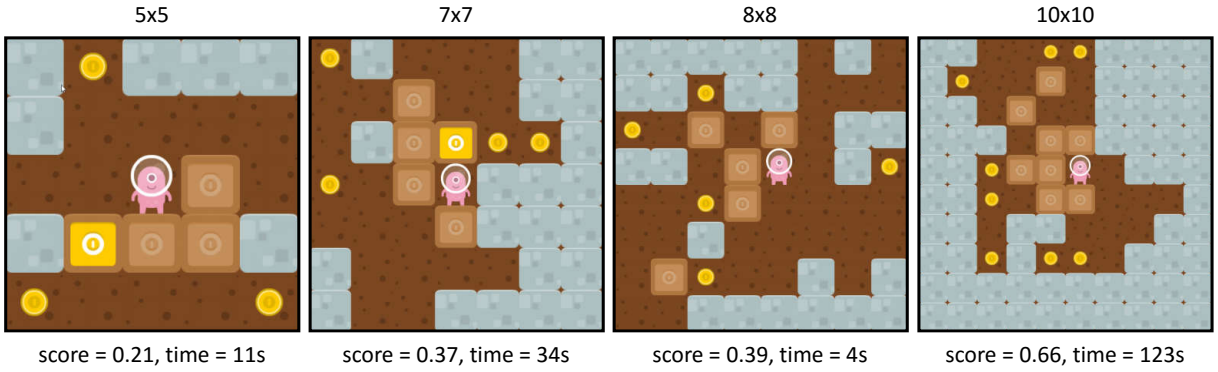

Figure 5: A variety of puzzles with different board sizes. Our algorithm makes use of larger amounts of board space in optimizing the evaluation function, leading to interesting box interactions in all the various sizes. Score value is computed by our evaluation function, and time refers to how many seconds it takes for MCTS to generate the corresponding puzzle.

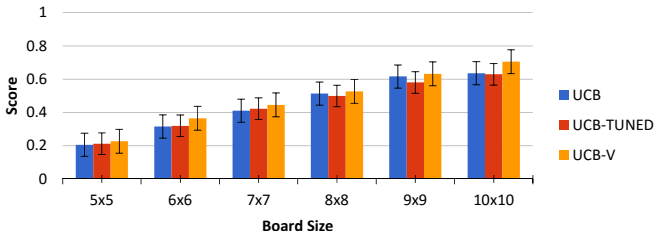

Figure 6: Different MCTS selection algorithms are compared. UCB-V slightly outperforms UCB and UCB-Tuned algorithms. All algorithms were provided the same search budget. The results are obtained by averaging 5 runs with different seeds.

4.3 Generated Levels

Our algorithm is capable of generating interesting levels of varying board sizes. Figure 5 showcases some examples. Our algorithm does not require any templates, generating the levels from an initially obstacle-filled board. Figure 3 shows how board quality evolves over time. Lower scoring boards are quickly ignored as the search progresses towards more interesting candidates. The leftmost board depicted in Figure 3, while only 5x5 tiles, presented a challenge for the authors, taking over a minute to solve. As MCTS is a stochastic search algorithm, we obtain different interesting boards when we initialize with different random seeds.

5 Discussion

The difficulty of a Sokoban puzzle is not easily quantified (even knowing the optimal solution would not make determining relative difficulty easy). Some properties of the scores produced by our evaluation function are worth noting. In general, higher scores led to higher congestion (more box path interactions) and a larger number of required moves. This does lead to more challenging and engaging puzzles as the score values increase. However, there are other aspects of difficulty that are not explicitly captured by our metrics. A level that requires more moves than another doesn't necessarily mean it is more difficult to recognize the solution (e.g, having to move 3 boxes 1 space each vs 10 boxes 1 space each). Examples exist in which a very challenging level has a much lower score than a "busier" but otherwise menial one. Additionally, the box path interactions in our challenging puzzles usually correspond to determining a small number key moves, after which the puzzle is easily solved. In contrast, human designed puzzles can require the player to move boxes carefully together throughout the entire solution.

A key motivation of our work is the ability to use our generated levels for games that have puzzle mechanics as an integral part of gameplay. Here, the ability to generate the puzzles behind these elements could make the entire game experience different on every play through. Additionally, levels of varying difficulties could be generated to represent mini-games corresponding to various skill levels. For example, one could imagine the reskinning of Sokoban puzzles as a "hacking" mini-game where the difficulty can be tuned to the desired challenge based on contextual information.

The properties of our algorithm in particular make it well suited to the aforementioned applications. Since the speed of our method could allow new puzzle to be generated for every instance, players could be prevented from searching for solutions on game forums (if that property is desired). Additionally, an evaluation metric that corresponds well to puzzle difficulty could allow the game to dynamically present puzzles of various difficulty to the player based on their character's profile (e.g."hacking skill") or other contextual information. Finally, the anytime nature of the algorithm allows for the generation of multiple levels of varying difficulty in a single run, which could be stored for later use in the game.

Our method has the potential to generalize to other puzzle games or games of other genres containing analogous mechanics. First we must assert that the game to be generated has some finite environment whose elements can be represented discretely. Then the initial phase of our method may be applied simply by changing the action set to include actions that manipulate the elements of the environment. In many puzzle games, the game is won by manipulating the environment through player actions such that the elements match some goal state conditions. The efficiency of our method comes from exploiting this property. To apply the second phase of our method, one need simply change the action set of the Sokoban agent to the action set available to the player

during game play. Given these and a simulation framework, our method will generate solvable puzzles. Finally, an evaluation function must be carefully designed to produce the desired puzzle qualities. This is typically the most difficult part of applying our method to new puzzle types. The evaluation function needs to be specific to the game being generated, and must balance between being efficient to compute but still predictive of the desired difficulty of the puzzle.

6 Conclusion

In this work, we have proposed and implemented an MCTS approach for the fast generation of Sokoban puzzles. Our algorithm requires no human designed input, and is guaranteed to produce solvable levels. We developed and utilized heuristic evaluation function that separates trivial and uninteresting levels from non-trivial ones, enabling the generation of challenging puzzles. We have shown, with examples, that our method can be used to generate puzzles of varying sizes.

Limitations: While our method can be used as is to generate interesting levels, there are areas in which the method could potentially be improved. No structured evaluation of our proposed metrics' ability to select interesting levels has been performed. While we have presented levels that were challenging to us, we have not tested the validity of these metrics statistically.

Future Work: We plan to pursue the extension and improvement of this work in two main directions. One is to deepen our knowledge of puzzle features that can be efficiently computed and reused within the puzzle generation phase while the other is improving the performance of the algorithm. To validate our evaluation function, we plan to conduct a user study to collect empirical annotations of level difficulty, as well as other characteristics. Having humans assess our puzzles would allow the testing of how well our evaluation function corresponds to the puzzle characteristics as they are perceived. To begin overcoming the challenge of better understanding puzzle difficulty, we must expand our notion of what makes a puzzle engaging or challenging. This could potentially be achieved by using machine learning methods to identify useful features on both human designed and generated puzzles with annotated difficulty. To increase the scalability and performance of our approach, we plan to parallelize MCTS potentially with the use of GPUs.

References

[Ahmed *et al.*, 2015] Umair Z Ahmed, Krishnendu Chatterjee, and Sumit Gulwani. Automatic generation of alternative starting positions for simple traditional board games. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[Ashlock and Schonfeld, 2010] Daniel Ashlock and Justin Schonfeld. Evolution for automatic assessment of the difficulty of sokoban boards. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, 2010.

[Audibert *et al.*, 2007] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Tuning bandit algorithms in stochastic environments. In *Algorithmic Learning Theory*, pages 150–165, 2007.

[Auer *et al.*, 2002] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[Baghdadi *et al.*, 2015] Walaa Baghdadi, Fawzya Shams Eddin, Rawan Al-Omari, Zeina Alhalawani, Mohammad Shaker, and Noor Shaker. A procedural method for automatic generation of spelunky levels. In *Applications of Evolutionary Computation*, pages 305–317. 2015.

[Bauters *et al.*, 2016] Kim Bauters, Weiru Liu, and Lluís Godo. Anytime algorithms for solving possibilistic mdps and hybrid mdps. In *Foundations of Information and Knowledge Systems*, pages 24–41. 2016.

[Botea *et al.*, 2002] Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In *Computers and Games*, pages 360–375. 2002.

[Browne *et al.*, 2012] Cameron B Browne, Edward Powley, Daniel Whitehouse, et al. A survey of Monte Carlo Tree Search methods. *IEEE Trans. on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[Cazenave and Jouandeau, 2010] Tristan Cazenave and Nicolas Jouandeau. Towards deadlock free sokoban. In *Proc. 13th Board Game Studies Colloquium*, pages 1–12, 2010.

[Cazenave and Saffidine, 2010] Tristan Cazenave and Abdallah Saffidine. Score bounded monte-carlo tree search. In *Computers and Games*, pages 93–104. 2010.

[Culberson, 1999] Joseph Culberson. Sokoban is PSPACE-complete. In *Proceedings in Informatics*, volume 4, pages 65–76, 1999.

[Edelkamp, 2014] Stefan Edelkamp. Planning with pattern databases. In *Sixth European Conference on Planning*, 2014.

[Frydenberg *et al.*, 2015] Frederik Frydenberg, Kasper R Andersen, Sebastian Risi, and Julian Togelius. Investigating MCTS modifications in general video game playing. In *Proc. IEEE Conf. on Computational Intelligence and Games (CIG)*, pages 107–113, 2015.

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

[Hennes and Izzo, 2015] Daniel Hennes and Dario Izzo. Interplanetary trajectory planning with monte carlo tree search. In *Proceedings of the 24th International Conference on Artificial Intelligence, AAAI Press*, pages 769–775, 2015.

[Jacobsen *et al.*, 2014] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 293–300, 2014.

[Jarušek and Pelánek, 2010] Petr Jarušek and Radek Pelánek. Difficulty rating of sokoban puzzle1. In *Stairs 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, volume 222, page 140, 2010.

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

[Kartal *et al.*, 2014] Bilal Kartal, John Koenig, and Stephen J Guy. User-driven narrative variation in large story domains using monte carlo tree search. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 69–76, 2014.

[Kartal *et al.*, 2015] Bilal Kartal, Julio Godoy, Ioannis Karamouzas, and Stephen J Guy. Stochastic tree search with useful cycles for patrolling problems. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 1289–1294, 2015.

[Kartal *et al.*, 2016] Bilal Kartal, Ernesto Nunes, Julio Godoy, and Maria Gini. Monte carlo tree search for multi-robot task allocation. In *Thirtieth AAAI Conference on Artificial Intelligence*, pages 4222–4223, 2016.

[Khalifa and Fayek, 2015] Ahmed Khalifa and Magda Fayek. Literature review of procedural content generation in puzzle games. 2015.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. 2006.

[Mirsoleimani *et al.*, 2015] S Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos Vermaseren. Parallel monte carlo tree search from multi-core to many-core processors. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3, pages 77–83. IEEE, 2015.

[Murase *et al.*, 1996] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. Automatic making of sokoban problems. In *PRICAI'96: Topics in Artificial Intelligence*, pages 592–600. 1996.

[Pepels *et al.*, 2014] Tom Pepels, Mark HM Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Trans. on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[Pereira *et al.*, 2015] André G Pereira, Marcus Ritt, and Luciana S Buriol. Optimal sokoban solving using pattern databases with specific domain knowledge. *Artificial Intelligence*, 227:52–70, 2015.

[Perez *et al.*, 2014a] Diego Perez, Spyridon Samothrakis, and Simon Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8, 2014.

[Perez *et al.*, 2014b] Diego Perez, Julian Togelius, Spyridon Samothrakis, et al. Automated map generation for the physical traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 18(5):708–720, 2014.

[Perick *et al.*, 2012] Pierre Perick, David L St-Pierre, Francis Maes, and Damien Ernst. Comparison of different selection strategies in monte-carlo tree search for the game of tron. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 242–249, 2012.

[Sabar and Kendall, 2015] Nasser R Sabar and Graham Kendall. Population based monte carlo tree search hyper-heuristic for combinatorial optimization problems. *Information Sciences*, 314:225–239, 2015.

[Shaker *et al.*, 2014] Noor Shaker, Julian Togelius, and Mark J Nelson. Procedural content generation in games: A textbook and an overview of current research. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, 2014.

[Shaker *et al.*, 2015] Mohammad Shaker, Noor Shaker, Julian Togelius, and Mohamed Abou-Zleikha. A progressive approach to content generation. In *Applications of Evolutionary Computation*, pages 381–393. Springer, 2015.

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

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

[Stammer *et al.*, 2015] David Stammer, Tobias Gunther, and Mike Preuss. Player-adaptive spelunky level generation. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 130–137, 2015.

[Steinmetz and Gini, 2015] Erik Steinmetz and Maria Gini. Mining expert play to guide monte carlo search in the opening moves of go. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 801–807, 2015.

[Sturtevant, 2013] N Sturtevant. An argument for large-scale breadthfirst search for game design and content generation via a case study of fling. In *AI in the Game Design Process (AIIDE workshop)*, 2013.

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

[Summerville *et al.*, 2015] Adam James Summerville, Shweta Philip, and Michael Mateas. Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[Taylor and Parberry, 2011] Joshua Taylor and Ian Parberry. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pages 5–12, 2011.

[Taylor *et al.*, 2015] Joshua Taylor, Thomas D Parsons, and Ian Parberry. Comparing player attention on procedurally generated vs. hand crafted sokoban levels with an auditory stroop test. In *Proceedings of the 2015 Conference on the Foundations of Digital Games*, 2015.

[Togelius *et al.*, 2011] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, et al. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[van Kreveld *et al.*, 2015] Marc van Kreveld, Maarten Loffler, and Paul Mutser. Automated puzzle difficulty estimation. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 415–422, 2015.

[Williams *et al.*, 2015] Piers R Williams, Joseph Walton-Rivers, Diego Perez-Liebana, et al. Monte carlo tree search applied to co-operative problems. In *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pages 219–224, 2015.

[Zook *et al.*, 2015] Alexander Zook, Brent Harrison, and Mark O Riedl. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th Conference on the Foundations of Digital Games*, 2015.