An Efficient Approach of Sokoban Level Generation

An Efficient Approach of Sokoban Level Generation

WU YUEYANG
Graduate School of Computer and Information Science
Hosei University
yueyang.wu.6f@stu.hosei.ac.jp

***Abstract*—This article describes an algorithm for the procedural generation of the Sokoban puzzle. This algorithm can generate Sokoban levels according to the given parameters. The algorithm is meant to generate Sokoban levels efficiently but maintains acceptable quality. This article provides evidence that this algorithm is efficient and produces levels with a quality comparable with other existing levels which can be found online.**

The approach contains two parts. They are forward process and backward process. The forward process creates the goal position and empty room for the result. And the backward process makes initial status further away from its goal status. In each iteration of the forward or backward process, a box and a direction will be selected based on the strategies being set in the generator parameters. The number iterations are able to be configured by changing the parameters. With certain configuration, the generated levels can be with acceptable average quality. The detailed explanations are also included in this article.

I. Introduction

Sokoban is a classic tile-based puzzle game, where the player plays a warehouse manager, pushing boxes to their goals. The player, boxes, and walls collide with each other. Boxes can be pushed by a player to move around. The player can only push one box at a time.

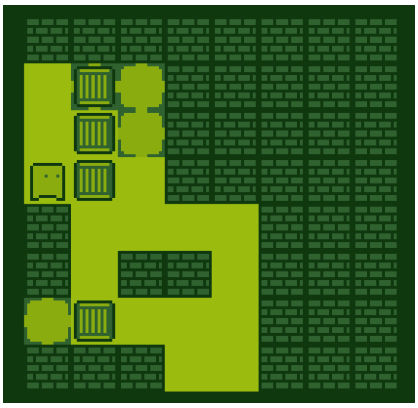

Fig. 1. A Sokoban level

The design of a puzzle can largely affect the difficulty of a level. It's a NP-Complete problem [1]. A puzzle can be extremely hard if the author of the puzzle is an experienced Sokoban designer [2]. I present an algorithm to generate levels with reasonable difficulty yet with very little time spent. A completely randomly generated puzzle can be either too easy or impossible to solve. A properly generated Sokoban puzzle should at least be possible to solve. And it should be of fair difficulty. The difficulty of a Sokoban level is a little hard to distinguish because it's about human beings' feelings.

The easiest way to evaluate the difficulty of a Sokoban puzzle is by looking at the steps of the optimized solution. A solution means a series of directions the character move and pushes boxes to complete a level. A better solution is which has fewer steps. The number of pushes can be used for evaluating a level's difficulty. A better way to evaluate a Sokoban puzzle is to calculate the box lines of it's optimized solution. In a solution of a Sokoban puzzle, each continuous push without changing direction is called one box line. A level with more box-lines tends to be more difficult. But it's easy to make a very easy to solve the level with high box lines, such as a bunch of boxes in a line right next to a bunch of goals. The box-lines can go as much as you want, but it just makes the level more tedious rather than interesting and difficult.

The main objective is to build a generation method which produces Sokoban levels with an interesting-to-solve difficulty, but also within a limited amount of time. This article describes an algorithm for the procedural generation of Sokoban puzzles. In this work, we focus on the efficiency of the algorithm. The algorithm can produce a level with a large size and more boxes in a relatively short time. The quality may be lower than the algorithms which search for every possible result. but still reasonable.

II. Related Works

Procedural content generation is always an important topic in the game industry. There are works about generating every aspect of a game. There are works about procedurally generate terrian [3], dungeons [4], maze-like levels [5], etc. When a game needs a large amount of content, the procedural generation can help a lot. It will make an approximately infinite number of contents.

Sokoban generation is a topic with a fairly long history. A generation method was introduced by Murase Yoshio and Matsubara Hitoshi and Hiraga Yuzuru in 1996. They use templates to generate room and remove unsolvable levels using a solver [6] . Joshua Taylor and Ian Parberry have introduced a generation approach in 2011 [7]. This approach uses templates to generate the empty room, and then place the goals by brute force searching, and then find the farthest possible state. This approach uses a load of searching which costs a large amount of time to finish a relatively larger level. The empty rooms are generated by putting together several 3 by 3 templates, and a lot of possibilities are eliminated because other restrictions

like no 4 by 4 or larger empty space are allowed. There is also a method by Bilal Kartal, Nick Sohre and Stephen Guy which uses Monte Carlo Tree Search to generate Sokoban puzzles [8]. Most of the other researches about Sokoban is about solving the puzzles. Junghanns and Schaeffer have introduced a method for solving Sokoban problems [9]. Adi Botea introduced a method that uses abstraction to solve Sokoban puzzles [10]. There are also some work have been done to estimate the difficulty of a given Sokoban puzzle [11].

There are also researches about some more generous puzzle generation. Such as Automatic Puzzle Level Generation using a Description Language [12] by Khalifa and etc.,and An approach to general videogame evaluation and automatic generation using a description language [13] by Lim and etc.

III. Method Overview

The basic idea of this method is to generate the level from its initial status. Each time choose a box to push. Every tile that the player walked on, or a box been pushed on, will be marked as an empty space. After this pushing stage, some pulling operations should be applied to increase the level's difficulty. So generally there are two stages, forward and backward. The forward procedure should try to open up as less empty space as possible to make a difficult level. There is configuration

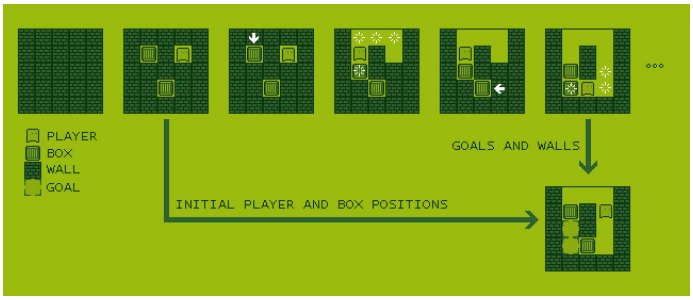

Fig. 2. A basic flow chart of the method

functionality for the generator. It's possible to choose what kind of methods are to be used in the process of generation. Each different strategy of doing box selection, push direction selection, route generation and so on will significantly affect the outcome. Here is a list of the basic steps of the generation method.

- Forward Process
  - Initialize
  - Iteration
- Backward Process
  - Initialize
  - Iteration
- Finalize

IV. Implementation Details

A. Data Structure

Several maps of the data used in the generation will be stored as one-dimensional arrays. The width of the map will be recorded separately to make the "maps" two dimensional. Using this method to represent a level will allow the representation of the positions using just one integer instead of two integers representing X and Y value.

B. Initialize

The random seed, width, height, and the box count should be set at this step. Map size is calculated by multiplying width and height. Box positions, initial box positions, push of the boxes, and "walked" map information will be stored in arrays. The box positions are randomly generated and then copied to the box positions and initial box positions arrays. A player position is represented with an integer which will be randomly picked within the map size and without colliding with box positions. The initial player position will also be recorded at the same time. Then all the positions with boxes or player are marked in the walked area array as true.

C. Forward Process

Firstly, a reachable map will be generated based on the player's position and the box positions. Then a box should be selected based on the reachable map. Then a direction will be selected based on the surroundings of the box. The direction should make sure it's possible for the player to push it in the opposite position, and also the new position of the box doesn't collide with another box. If every check is done and the push is valid, then a route should be generated from the player's original position to the pushing position. Then the box position, the player position are changed. And the walked area map will record the walking route and the new box position as true.

D. Backward Process

The Backward Process changes the initial positions of the boxes. Firstly a reachable map should be generated basing on the initial player position, boxes and walked-map. Then a box and the pulling direction is selected based on this reachable map and the pulling strategy set up in the beginning. The new position of the box and the player shouldn't collide with other boxes and the un-walked tiles (which is considered as walls. After checking the situation, and everything is good to go, the initial player position and the initial box positions are changed.

E. Result Output

The result will be generated based on the data after doing several iterations of the forward and backward process. First, the walls will be defined by the un-walked map. And the result goals are positioned based on the box positions. The result box positions will be based on the initial box positions. The player position will be the player position as it is. All the content will be converted to a string with an integer number representing the width.

A common method of representing a Sokoban level is to represent it with a string. Different characters represent different tiles. After generation, this string format is used for storage and evaluation. Because this is a common format of

a Sokoban Level, the information can be used in many other Sokoban related software like YASC and JSoko.

F. Reachable Map Generation

The reachable map is generated using a naive method, flooding the whole map from the player's position. A queue is used for the calculation. First, push the starting position into the queue, and mark it as a position that is visited. Then, take an element out of the queue. Visit the neighbors and check if they are accessible. Push the accessible and not yet being visited neighbor into the queue, and so on. Thus a reachable map from the starting position is generated. To determine whether a tile is accessible, the obstacles should be defined before this process. Using this "reachable map" can largely improve the overall generation time because it provides the information used so many times in each step.

V. Parameters of the Generator

There are several parameters can be configured before doing a generation. These parameters can largely influence the result. They are the size of the level, box count, forward iterations, backward iterations, the strategies of box selection, push direction selection, pull direction selection, route generation. Also, all of the randomnesses are based on a random seed. The random function is provided by the Unity engine. So by changing the random seed, different results can be generated with other parameters not being changed.

A. Box Selection Strategies

There are three available strategies implemented by the time this article is written: Random, Least Pushed, and In Order.

"Random" is a strategy that is totally random.

"Least Pushed" is a strategy that selects the boxes with the least pushes applied before. Every boxes' number of pushes are recorded during the iteration. So it's easy to find a "least pushed" box. This strategy is found useless because in many cases, there are several boxes can never be reached, which makes their pushing history being zero. In this situation, the generator is always trying to move that particular box which will end up doing nothing.

"In Order" is a strategy that selects the boxes one by one in order. The box entities are stored in an array, so simply increase an index is enough to keep track of which box to push. This is the most "fare" strategy because every box will be moved with a very similar amount of times after iterations.

B. Push Strategies (Forward Direction Selection)

There are three available strategies implemented by the time this article is written: Random, Farthest, Most Obstacles.

"Farthest" is a strategy that tries to find the direction farthest from the original position of the selected box. Calculate the total Manhattan distance to all the goals from each direction. The highest-distanced direction will be selected. When using this strategy, the boxes tend to be farther away from their initial positions.

"Most Obstacles" is a strategy which tries to find the direction which makes the boxes end up in a position where there are the most un-walked area or boxes surrounding. This method simply checks each new position's neighbors and see if they are boxes or un-walked area. The direction with the highest total number will be selected.

C. Pull Strategies (Backward Direction Selection)

There are four available strategies implemented by the time this article is written: Random, Farthest, Most Obstacles, Most Access.

"Most Obstacles" and "Farthest" are the same thing inverted with the ones with the same name in the Push Strategies.

"Most Access" is a strategy where leaves out most accessible tiles for the player after the operation. In this method, every new status will generate a different "reachable map" from the position of the new player position. The direction with the largest number of reachable tiles will be selected. This strategy is introduced because, during the backward process, the player tends to stuck in a pit of boxes or walls which makes it's impossible to further modify the level.

D. Route Generation Strategies

There are two available strategies implemented by the time this article is written: Direct, Closest Active Tile.

"Direct" is a strategy that generates a direct route to the target position starting from the current player position. To find the route from the player position to the target position, a flooding algorithm is used. First, push the player position into a queue and mark it as "visited". Then each time the system pops an element from the queue and push the un-visited neighbors into the queue. Also when pushing an element into the queue, the recently popped element will be recorded to that new position. Iterate until the target position is visited. Then using the recorded information to backtrack to the original position. Thus a route is generated.

"Closest Active Tile" is a strategy that generates a route from the closest reachable walked tile to the target position. Using this strategy will mark fewer tiles as "walked", so it preserves more walls. The closest tile is found simply by going through all the acceptable empty areas and calculate their Manhattan distance to the target. And then using the same algorithm used in the "Direct" strategy to generate the route.

The order of visiting neighbors of different directions should be randomized to avoid different weights of different directions. Without randomize the order, the route will always be like, more likely to go one direction rather than the opposite one. (Because some direction is always visited before others, thus the other ones only have the chance to be in the result when the opposite one fails.)

VI. Experiment and Result

Here are some examples of the generated levels.

The generated result has been evaluated in several ways. Here are the comparisons with Joshua Taylor's method. Here I used the YASC Sokoban Solver to evaluate the levels generated with my method and Joshua Taylors method. First,

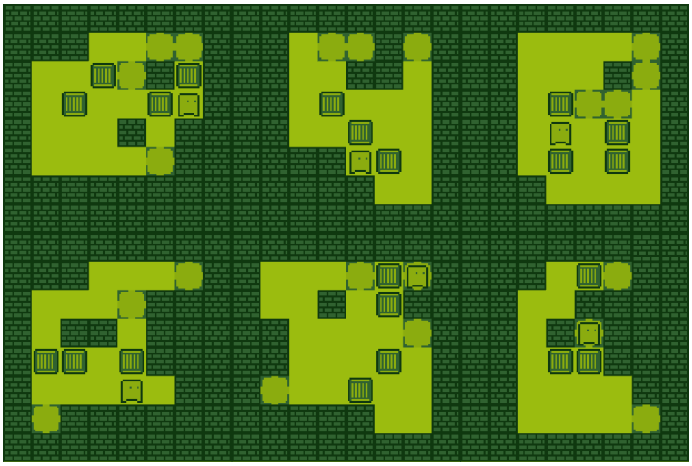

Fig. 3. The levels generated with my method

the level sets of both mine and Joshua Taylor's are fed into YASC Sokoban Solver. And the resolution will be stored in the files with the levels. Then I used regular expressions to extract the essential data from the files. The generation time, the optimized solution, the moves. And then the box-lines are calculated based on the optimized solution generated previously with the YASC Sokoban Solver.

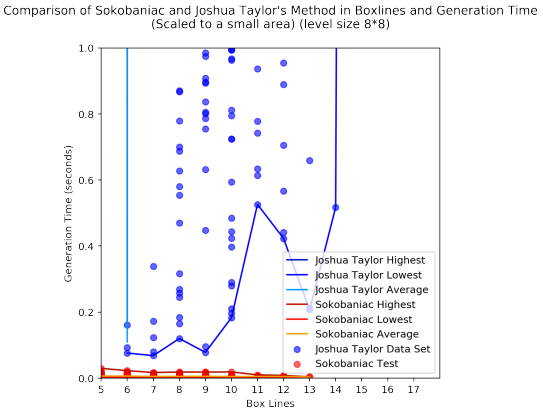

Fig. 4. Generation Time Comparison (with less than 4 boxes)

As is seen in figure 4, it takes much less time than Joshua Taylor's method to generate an equivalent difficult level when the level has 6-14. It can also be seen that my method is capable of generating 6-14 box-lined levels with a reasonable ratio. Due to the much lower generation time, it's a valuable method if the target is to generate levels with box-lines is 6-14. In my method, the generation time doesn't change much when the result's complexity goes up. But the average complexity is not guaranteed. This means that if a level with high complexity is required, more levels have to be generated so that a level that meets the requirement can be found.

A. *Comparisons For Different Parameters*

The result can also vary due to different paramters being applied.

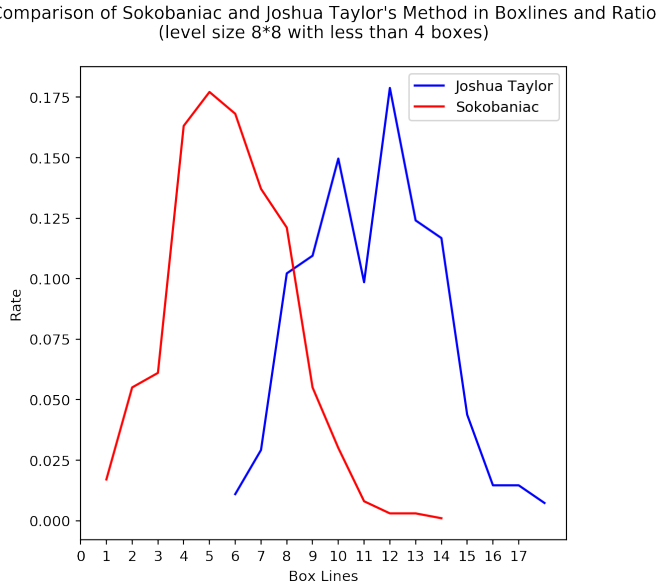

Fig. 5. Boxlines ratio (with less than 4 boxes)

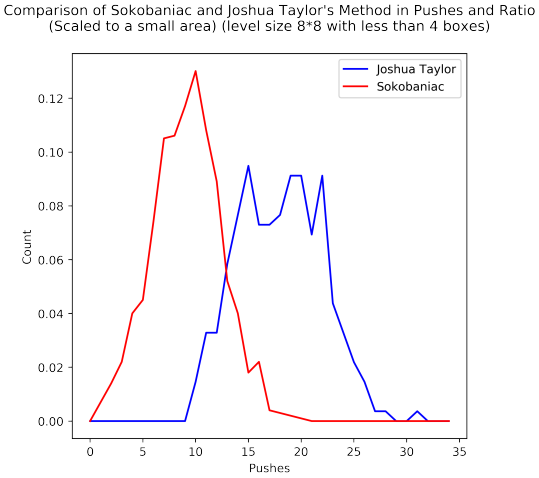

Fig. 6. Pushes ratio (with less than 4 boxes)

The different settings can affect the result. There is 8 example set being tested and the result is shown in figure 7.The settings are:

- S1 - Random Box Selection Strategy, Random Backward Strategy, Random Forward Strategy, Direct Route Generation Strategy
- S2 - Random Box Selection Strategy, Random Backward Strategy, Random Forward Strategy, Closest Active Route Generation Strategy
- S3 - In Order Box Selection Strategy, Random Backward Strategy, Random Forward Strategy, Direct Route Strategy
- S4 - Random Box Selection Strategy, Reserve Most Acess Backward Strategy, Random Forward Strategy, Direct Route Strategy

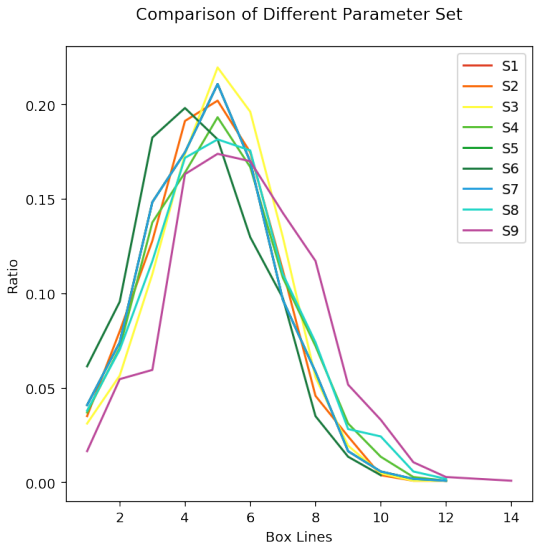

Fig. 7. The comparison of Different Parameter Set

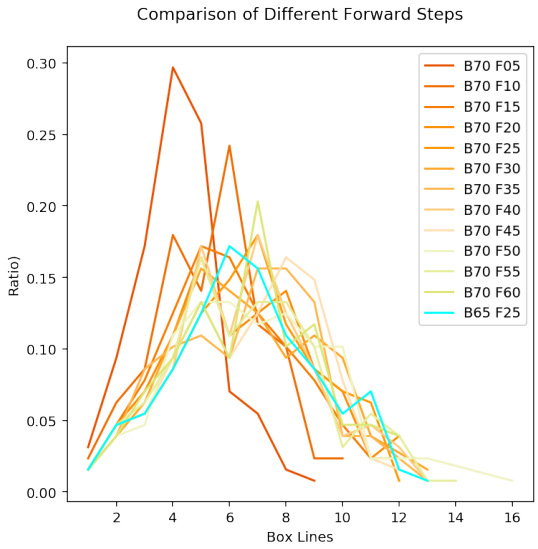

Fig. 8. The comparison of Different Forward Steps

- S5 - Random Box Selection Strategy, Most Obstacles Backward Strategy, Random Forward Strategy, Direct Route Strategy
- S6 - Random Box Selection Strategy, Farthest Backward Strategy, Random Forward Strategy, Direct Route Strategy
- S7 - Random Box Selection Strategy, Random Backward Strategy, Most Obstacles Forward Strategy, Direct Route Strategy
- S8 - Random Box Selection Strategy, Random Backward Strategy, Farthest Forward Strategy, Direct Route Strategy
- S9 - In Order Box Selection Strategy, Reserve Most Acess Backward Strategy, Farthest Forward Strategy, Closest Active Route Generation Strategy

It can be inferred that using different strategies can make the result with more box-lines which potentially increases the difficulty of the level. The S9 setting is shifted further right than other settings being tested.

The effects of the forward and backward steps are also evaluated. All of the levels being tested in this session are generated in the size of 8 by 8, using the S9 setting of strategies. There are 128 levels in each configuration. The result of different forward steps is shown in figure 8. It's very obvious that more forward steps can increase the result box-lines. Average box-lines of different forward steps are shown in the figure 9. The effect of backward steps is also tested. The result is shown in figure 10. The box-lines of the result tend to increase as the backward steps go up. The box-lines statistics is shown in the figure 11. There are some unexpected spikes in the graphs. These phenomenons are probably caused by the number of samples is too low. But even though, we can still tell that larger forward and backward steps can make the result tend to be of more box-lines, which potentially increases the difficulty of the result.

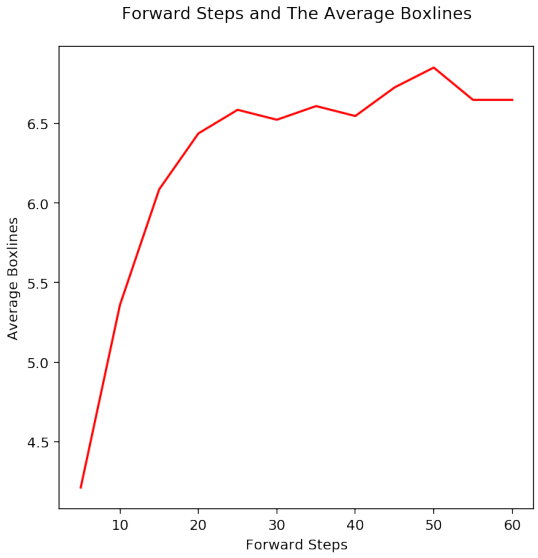

Fig. 9. Average Boxlines of Different Forward Steps

B. Manual Evaluate

A manual evaluation application SOKOBANIAC is also implemented to evaluate the levels manually. Levels form different levels of sets are presented to the players (participants) with no difference. After they complete solving a level, the solution will be sent to the GameJolt sever. Also if they rated a level's difficulty, the information will be sent to the server too. This application is designed to gain the playing data of both levels generated by my method and the levels from Joshua Taylors' data set. So that more human-related properties can be compared.As is seen in figure 12, the overall rating of Joshua's level set is higher than the levels generated with my method. But there is some overlapping area of scoring.

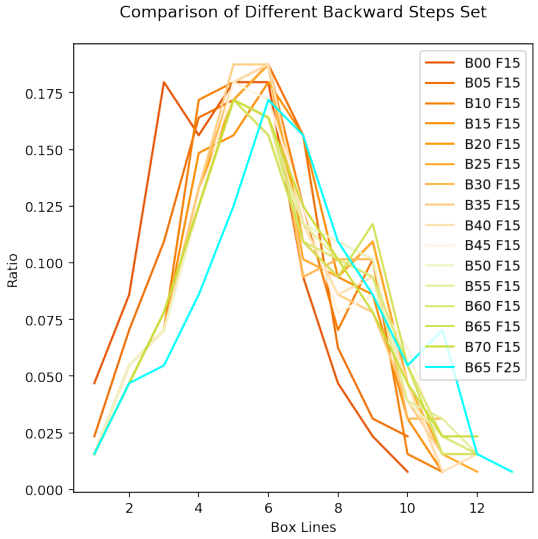

Fig. 10. The comparison of Different Backward Steps

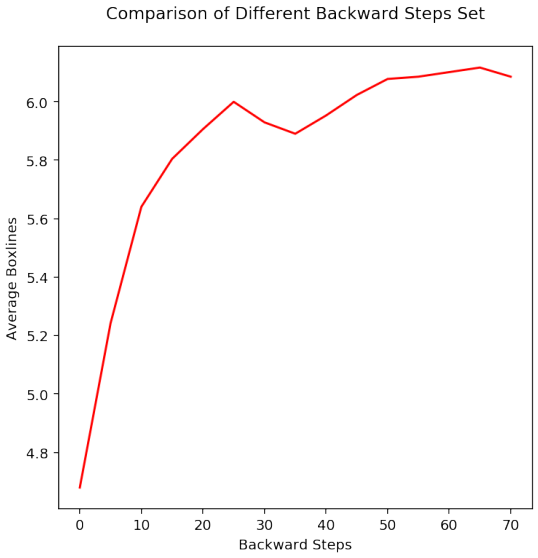

Fig. 11. Average Boxlines of Different Backward Steps

VII. Conclusion and Future Work

This approach is indeed useful when trying to generate levels with fewer box-lines. It's faster than Joshua Taylor's algorithm but with lower average quality.

The currently existing strategies are strategies that happened to come to my mind when I was trying to implement this generator. I believe there could be other strategies that I haven't found yet that could produce a better result. Also, a mixed strategy like randomly choose other strategies each time may somehow make a better result. My implementation of the generator is using only one thread only to do the generation. I believe that a parallelled implementation will largely improve performance. The evolutional method could make more sense if running on a paralleled version generator. If an efficient solver can be used within this generator, the quality of the generated level can be much better. Just generate more levels and eliminate the bad ones simultaneously. A solver may also help to do evolutional generation because, with a solver, the difficulty of a level can be more accurately evaluated.

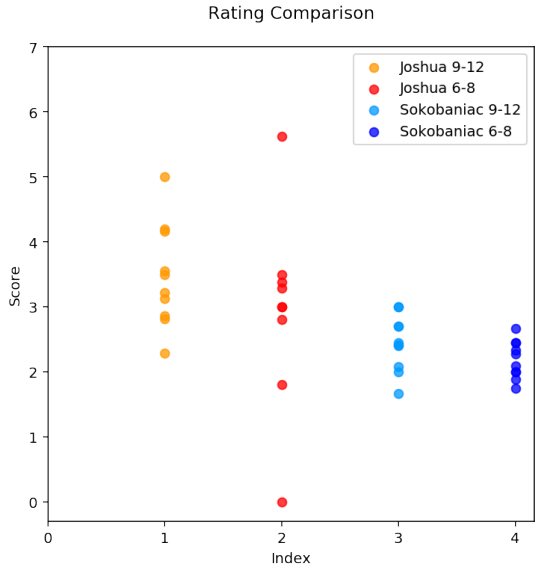

Fig. 12. The Rating of 40 Levels by 20 individuals

References

[1] Joseph Culberson. Sokoban is pspace-complete. 1997.

[2] Sokoban Wiki. http://sokobano.de/wiki/index.php?title=Main_Page, 2018.

[3] J. Doran and I. Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, June 2010.

[4] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, March 2014.

[5] D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, Sep. 2011.

[6] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. Automatic making of sokoban problems. In *Pacific Rim International Conference on Artificial Intelligence*, pages 592–600. Springer, 1996.

[7] Joshua Taylor and Ian Parberry. Procedural generation of Sokoban levels. Technical Report LARC–2011–01, Laboratory for Recreational Computing, Dept. of Computer Science & Engineering, Univ. of North Texas, February 2011.

[8] Bilal Kartal, Nick Sohre, and Stephen J Guy. Generating sokoban puzzle game levels with monte carlo tree search. 07 2016.

[9] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *IJCAI 1997*, 1997.

[10] Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. pages 360–375, 07 2002.

[11] Petr Jarušek and Radek Pelánek. Difficulty rating of sokoban puzzle. In *Proc. of the Fifth Starting AI Researchers' Symposium (STAIRS 2010)*, pages 140–150, 2010.

[12] Ahmed Khalifa and Magda Fayek. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*, 2015.

[13] Chong-U Lim and D Fox Harrell. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.