Creating Solvable Instances for Unsolved Puzzles

Bernard J. Willemsen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
b.j.willemsen@student.utwente.nl

ABSTRACT

Procedural grid-world puzzle generation is a fairly well-studied field with applications both in machine learning as well as more directly in game development. Most of these efforts are focused either on Generate-and-test methodologies, requiring solvers, or rely on specific mathematical properties of their games.
Kwirk is a 1989 block-pushing puzzle game [3] that does not have a solver[1], and while while bearing a striking resemblance to Sokoban [4], a 1982 puzzle game that has seen a lot more research in the domain of level generation, Kwirk violates the properties that the generative methods researched for Sokoban rely on. In this paper we will discuss how we adapted one of these Sokoban generation algorithms to function on Kwirk regardless.

Keywords

Puzzle, Procedural Generation, Level Generator, Kwirk, Sokoban

1. INTRODUCTION

We will begin with a brief description of Kwirk, discuss Sokoban level generation practices, and end this section laying out what separates this research from the work done before.

1.1 Kwirk

Kwirk is a 1989 puzzle game originally developed for the Nintendo Game Boy. In Kwirk, the player controls a number of player characters on a discrete grid who must all make their way to the level exit. Once there are no player characters left, the puzzle is solved. Besides simple stationary wall pieces, the game employs a number of obstacles, including boxes which can be pushed around; Pits that cannot be navigated over unless filled in with boxes; and finally rotators, elements with a static core and a number of orthogonal arms that may rotate around the core in 90 degree intervals. Lastly it is important to note that only one of these objects may be interacted with at a time, so a row of boxes cannot be pushed along collectively, but must be broken up to be moved. For a detailed description of Kwirk, see section 2.1.

[1]at the time this research was started, a solver was developed in parallel

28^{th} *Twente Student Conference on IT* Febr. 2^{nd}, 2018, Enschede, The Netherlands.

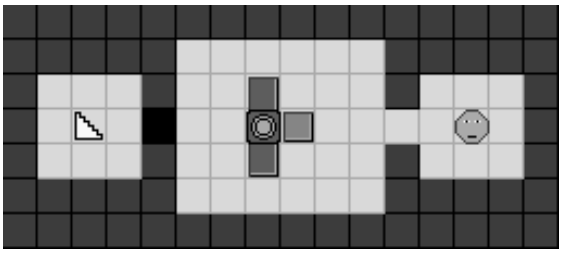

Figure 1. A Kwirk level that is essentially a Sokoban level, requiring the block to be pushed in the hole at (4,3) to be able to be solved

1.2 Established Practices

No literature on the game Kwirk was found. The closely related grid-world tile-sliding character navigation puzzler Sokoban has seen a fair amount of research however.

1.2.1 Enter Sokoban

In Sokoban, you control a character in a grid-world, and using orthogonal movement must push a collection of single-tile boxes onto marked spaces of the level.
Key to the complexity of Sokoban puzzles is the observation that a box located next to a wall cannot be separated from said wall, as that would require a player getting in between said box and said wall (by this same logic, a box placed in a corner is simply lost). While some Kwirk levels essentially are Sokoban levels (see figure 1) there are a number of important differences that will come up as we discuss applying Sokoban level generation methodologies to Kwirk.

B. Kegel and M. Haahr suggested the defining factor of a Sokoban-style puzzle to be the fact that "no items/characters are ever lost or added to the board; the solution exists as a rearrangement of the original configuration." [6], effectively categorizing Sokoban as variation of a sliding block puzzle.
Between players leaving levels and blocks sliding into holes in the floor to create new floor Kwirk violates this requirement. While it could be argued that this is analogous to boxes pushed into corners being lost, level generation for Kwirk must violate this property, since according to Kegel and Haahr, Kwirk would be classified as a Maze [6].
The importance in viewing Kwirk as a maze puzzle over a Sokoban-style one is that a sliding-block-puzzle generator may be given a solved level to be un-solved, where a maze generation algorithm is required to generate the maze itself. Or in other words: a Kwirk level generation algorithm must itself be able to place obstacles in the player's way

that are not part of the input.

Kegel and Haahr outlined two main approaches for procedural content generation across games: Constructive and Generate-and-test. Constructive algorithms generate content as they go along, ensuring solvability along the way; Generate-and-test-algorithms generate randomized output that is then evaluated and in case of insolvability discarded [6].

1.2.2 Generate-and-test

The first foray into automatic Sokoban problem construction by Murase, Matsubara, and Hiraga utilizes a Generate-and-test approach. They first used templates to construct an empty level and placed goal states, then placed the player and boxes randomly. Then using attributes of the solution such as length, amount of changes in direction and amount of "counterintuitive" moves to estimate how "interesting" a level is [8]. While Generate-and-test approaches are still used for learning algorithms [7], Bhaumik, Khalifa, and Green found that a search-based constructive approach may outperform an evolutionary method [2]. Furthermore Bento, Pereira and Levi successfully used a constructive approach to superhuman performance in terms of difficulty for a solver [1]. As such, along with the need for a solver in order to function we elected for a constructive approach over a Generate-and-test approach for generating Kwirk levels.

1.2.3 Constructive approaches

The first documented use of backwards generation is described by Taylor and Parberry [9]. They used a similar approach to Murase, Matsubara and Hiraga for generating the base levels, and for each possible start state (brute-forced) used an un-informed iterative deepening approach to find the most "interesting" of the farthest states from the given start state [9]. The main problem with adapting this approach to Kwirk is in the non-deterministic nature of the desired input format. The location of the player in the initial state of a Sokoban level given the position of the boxes the position of the player poses a similar non-deterministic input problem, but Taylor and Parberry did also not place the player explicitly. They instead represented the player by the currently accessible region, minimizing the amount of start states to be separately evaluated [9]. For Kwirk however we do not want to require the interactable elements to already be placed, and even the walkable area optimization while possible, becomes challenging to implement with multiple players. Other players block attempts to move objects, and one player's region could be bisected by the action of another player (pushing a block into the other player's path for instance). This would lead to a single action generating multiple predecessor states.

Bento, Pereira and Levi build on the work of Taylor and Parberry with a particular focus on the farthest state search process. Using a backward greedy best-first search with heuristics for conflicts and novelty derived from pattern databases -known solution lengths for small randomly chosen subsets of boxes present- [1]. This approach appeared to be successful as Bento, Pereira and Levi claim it is the first Sokoban level generation system with superhuman performance in terms of level difficulty for a solver. The cost and method of populating the pattern databases for each input level was not reported however, so it is presumed these were created by solver, making the approach unavailable for generating Kwirk levels.

1.3 Proposed Solutions

In this paper, we suggest using the backwards search algorithm in [1], but not defining a level as a set of values, but rather a set of constraints on the potential values of all tiles in the level geometry. In actions, by Bento, Pereira and Levi defined as a vector $a = \langle Pre, Post \rangle$, containing two collections of defined variables where one must match and be overwritten by the other to generate the predecessor/successor state [1], this means that $Post$ defines constraints that must be a subset of the constraints in the target region, and Pre must define a function changing these constraints to ensure that the intended game input is possible. A series of actions then, when composed and given some start state, ensure that any matrix of defined tiles(a tilegrid) matching the level's constraints will allow for a series of inputs such that a state matching the start state is reached. We will investigate the capabilities of various heuristics to replace the pattern database, as well as an optimization to furthest state searching based on detecting proper sub-sets of constraints besides merely an exact match for the purposes of finding the furthest state from the start state. Since puzzle generation for Kwirk specifically is a new field that techniques from Sokoban do not map onto perfectly, the main goal of this paper is the following:

GOAL: develop an algorithm for generating Kwirk levels.

To this end we will attempt to answer the following research sub-questions:

RQ1: With this adaptation, what kind of new affordances does this level generation algorithm provide?

RQ2: what level/algorithm state features can be used effectively for generating kwirk levels?

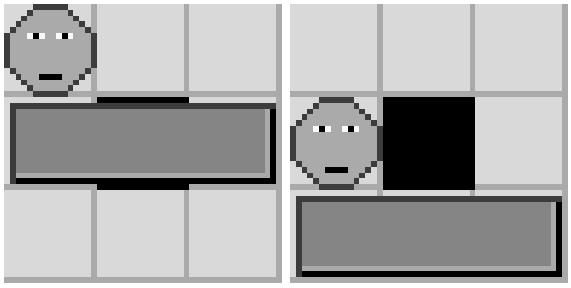

Figure 2. A 3x1 box being pushed, partially overlapping a hole and thus not falling in.

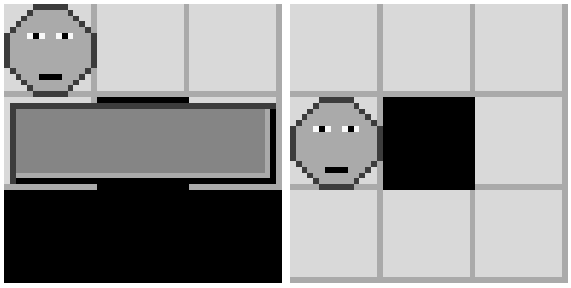

Figure 3. A 3x1 box being pushed into a hole.

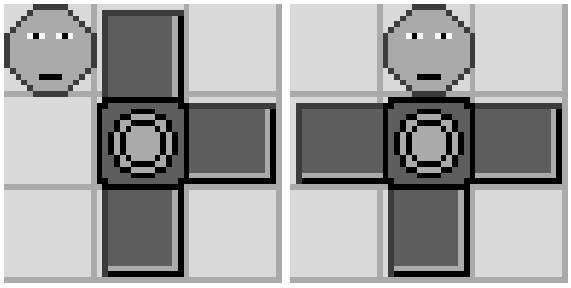

Figure 4. T-shaped Rotator, being pushed

2. IMPLEMENTATION

In this section we will provide detailed descriptions of the game, requirements and generation algorithm as implemented in order to collect the results in 3.

2.1 Kwirk, a detailed description

Kwirk is a game that takes place on a discrete grid without any dynamic elements devoid of player input, effectively making the game turn-based.

2.1.1 Players, Walls, Floor and the Exit.

As previously mentioned, A Kwirk level starts with one or more players (largest number observed in the original game is four) all of whom need to make their way to the exit tile. Once a player reaches the exit tile, they are removed from the game. As a result, a Kwirk level is considered 'solved' once there are no more players on the board. Only one player may be moved each turn, but which player character is to be moved may be chosen freely each turn. Besides changing the active player, the only game-related input is for orthogonal movement. If the tile next to the active player in the chosen direction is an empty Floor tile, the player will move over to that tile, leaving an empty Floor tile behind. If that tile is the level exit, the player is removed; if the tile is a Wall or Pit nothing happens; if the chosen tile is a floor tile with a box or rotator arm on it, the player attempts to interact with the element. With this means will be discussed below for each element. The level exit and inactive players are treated as walls while checking collision.

2.1.2 Pits

Pits similar to Walls restrict Player movement, except that in addition a pit tile may be additionally occupied by an overhanging box or rotator arm. If this is the case, this element cannot be interacted with via this tile.

2.1.3 Boxes

Boxes are defined by a rectangular region of one or more tiles. If a player attempts to move into a box, if a column or row of tiles in the opposite direction contains exclusively empty Floor or Pit tiles the player may move into the square next to them and the box region is moved to accommodate(see figure 2). If there is no space for the box to be moved into, the player's action simply fails. If after a turn every tile the box's region occupies is a Pit tile, all tiles in the region are converted to Floor tiles and the box is removed(see figure 3).

2.1.4 Rotators

A rotator exists of a core, which behaves similarly to a wall, with any layouts of orthogonal single-tile protrusions, that attempts to rotate as a singular entity around its core when interacted with (see figure 4). Just as the Boxes, rotator arms may hang over either Floor or pit tiles and may only be interacted with if it is the former. Two non-obvious rules apply to rotators. First, as mentioned, while they only rotate in 90 degree intervals, if the diagonal an arm would partially pass through contains some blocking entity, interacting with the rotator will fail (see figure 5). Second, if the tile the player would move into after rotation would contain an arm after rotation, the player is moved an extra square to the diagonal(see figure 6), if this diagonal is a pit, the interaction fails.

2.2 Requirement adaptations

The one requirement for this generator is tied to the fact that until near completion of this research no solver for Kwirk existed, as such the generator may not require one to function. Furthermore, the algorithm will be given an input level along with an hour of runtime. In order for the best output after that runtime to be considered "successful" it has to pass the following requirements:

- **De-tours required**: The minimum solution length must exceed the minimum path distance from the player to the goal ignoring Boxes, Pits and Rotators. That is to say, if the player just walked over to the goal, the must not solve the level "by accident".

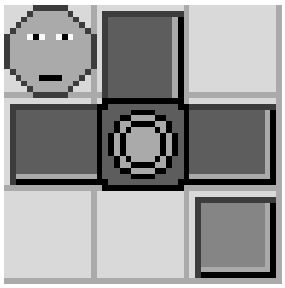

Figure 5. T-shaped Rotator, Failing to be pushed

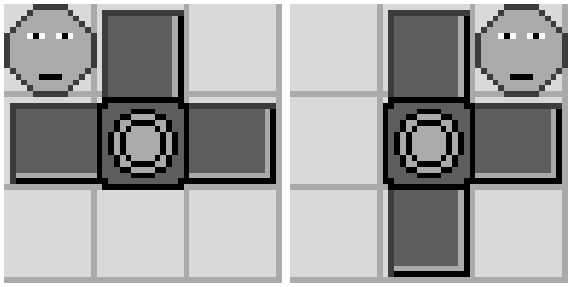

Figure 6. T-shaped Rotator, skipping a tile

- **No mazes**: There must not be a path between the player and the goal that does not cross any Pits, Boxes or rotators. This is to say, the player must interact with some dynamic elements in order to solve the level.

2.3 The Search algorithm

First, we will discuss the base version of the search algorithm, akin to that suggested in [1], then we will discuss the various adaptations as they become relevant when discussing the heuristics.

2.3.1 Base

Aside from the move over to stochastic variables, under most conditions the search algorithm is no different from that suggested in [1]. Consisting of a heap called *Open*, and a set *Closed*. Each cycle the top of *Open* is taken, all previous states are calculated, all states not already in closed are evaluated, and those are then added back into *Open* and *Closed*. If one is evaluated better than the current best, it is printed and saved, giving this algorithm the "anytime" property, allowing the algorithm to be run for an arbitrary amount of time, be terminated and have some usable output.

2.3.2 Heuristics

The following Generation heuristics were tested.

- **Element Count (Ec):** Simply the number of discrete Boxes and Rotators in the Level
- **Just A (As)*:** Ignoring Boxes,Rotators and Pits, the sum of the distances of the shortest paths between each player and their closest exit.
- **A*: Crossed Elements (Ce):** Using the path generated by A*, the number of tiles crossed that contain Pits, Boxes or Rotators.
- **A*: Minimum Crossed Elements (Mce):** A*: Crossed Elements, but before calculation each tile containing a Box, Rotator or Pit is given an additional weight exceeding the maximum path length in the level. The heuristic counts the amount of Box, Rotator or Pit tiles that were passed anyway.
- **Random (Rn):** Simply a random number, n indicates the seed used.
- **Tree Depth (TdFF):** In line with the reasoning by Taylor and Parberry, attempt to find the state that is the most steps away from the goal state [9]. This heuristics requires a number of changes to the algorithm, as a collision (finding a state that has already been found) may mean a short-cut to that state has been found (also implying a short-cut from this state to the goal state when playing), at which point the version in the *Closed* set may need to be adjusted, and then also needs to be re-explored in order to propagate the adjustment to its predecessor states. To aid in this process two optimizations where implemented.
  1. propagating adjustments (TdFT): The first optimization involves propagating tree-depth heuristic adjustments. Every state in the 'closed' set also links to it's successor, allowing to recursively search the *Closed* set until the tree-depth values for the entire predecessor tree has been adjusted for a new-found short-cut.
  2. encapsulation filtering, or subset testing (TdTF): The second optimization makes use of the change in the representation to stochastic tile values. Each level is constructed with a "canonical" solution in mind (the temporally ordered, composed list of taken backward actions), ensuring that for each tilegrid that satisfies the constraints said "canonical" solution is possible. As a result, if a certain level $L1$ is a proper subset of another level $L2$ (i.e. all tilegrids matching $L1$ also match $L2$, if they are equal it is a collision), we know for a fact that the canonical solution to $L2$ is also applicable to $L1$. This optimization in this case would recursively prune $L1$ and all of it's predecessors from the search tree. Given enough time this optimization should ensure that the canonical path through a given level is the minimal solution path, allowing for search tree depth to be an accurate description of solution length.

2.4 previous state generation

In the algorithm, a state is defined by two parts, a tilegrid and an elementset. Since boxes and rotators can hang over pits, they are defined separately in the elementset. The tilegrid is a matrix matching the level dimensions, where each unit exists of either an explicit Tile, or an abstract list of tiles. This list contains the values the tile may have, as well as whether or not the tile may have the property "blocking", for the purposes of rotator and box fitting (saving if a tile needs to currently be free for an action we have already decided we will take later). For the purposes of requirement fitting, a 'compiled' version of the level is used, where all tiles containing parts of elements (except for, if applicable, an element we are trying to re-use) are replaced with walls. Upon printing, all abstract tiles are collapsed, preferring Floor tiles.

For boxes, a list of applicants is created, consisting of existing boxes next to the player; boxes that could have existed already (detected by the "blocking" property) and boxes that could have just fallen into a region of pits to form floor. All these applicants are then tested whether a valid push action may have applied to them, in which case they are un-pushed, and added as previous states.

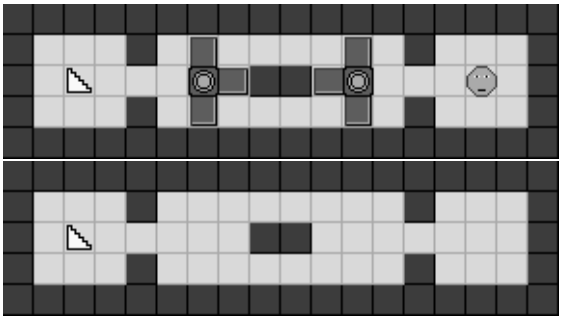

Figure 7. The first level, original and as input for the algorithm

For other actions, there exists a list of pattern-delta pairs, the first a matrix of functions returning Boolean values, the second a matrix of functions returning tiles, each taking tiles (either explicit or abstract) as input. These pairs are then flipped and rotated for every applicable symmetry(symmetry groups D_4 or C_4, depending on the action, as described in chapter 16 of [5]) , adjusted for the positions or their origin tiles(in most cases, one of these functions will only match an explicit player tile), and if all pattern functions return True the delta functions are applied and the resulting level is added as a predecessor.

3. RESULTS

A complete systematic exploration of all state combinations was not possible within the timescale of this project, so the following describes an exploratory look into the performance of the various heuristics.

We started with running all heuristics individually on an empty version of the first level in the game(see figure 7). Table 1 shows for each heuristic setting for the algorithm (random was run using the seeds zero until four, mean and standard deviation are reported), set to generate a one-player level, the size of the *Closed* set at the time the level was generated(does not reflect size of *Closed* set upon termination, and the treedepth algorithm with subset filtering prunes the *Closed* set.), the amount of steps it takes to solve the generated puzzle, and the puzzle coefficient, defined as $\frac{\text{steps to solve puzzle}}{\text{length shortest path from player to exit}}$, the latter ignoring puzzle elements. As such, unless mentioned otherwise, a puzzle with a puzzle coefficient of 1 fails the "de-tours required" clause. If a level fails the "no mazes" clause it is prepended with an '!'. Finally, here was an issue in test automation, likely leading to an amount of progress by the algorithm being lost. In these instances the last recoverable result is used, and the affected rows are marked with an '*' at the end of the setting name.

It seemed likely that the bad performance of the subset optimization might be due to the order of exploration. To test this it was ran alongside A* and Element Count, which had shown remarkable performance in previous rounds of testing (see table 2). The ones using EC as a negative have the heuristic weighted at -1.1, reasoning that we only want elements to be added when they would increase the necessary steps to solve. In all others EC is weighted to 0.1, just like A* against the 1 of tree depth. Note that the solution to 1-player Tree-depth fully optimised with A* costed and Element Count discounted may not be minimal (see figure 9).

In addition, the more promising heuristics were tested for a maximum of four players (see table 3).

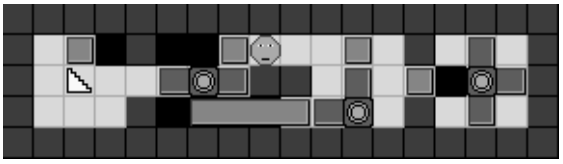

Figure 8. The best performing puzzle of the initial first level batch

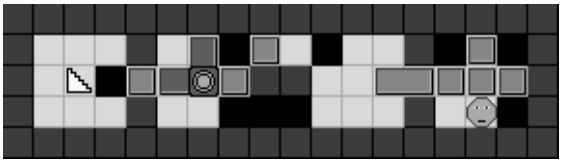

Figure 9. The best performing puzzle of the second first level batch

To provide some variation in context, a number of Generation settings were also run on the fourth level of the game, a similar, but far more spacious level (see figure 10, table 4).

Notable here is the under-one performance of 4-player Element Count, where a direct path to the exit skips over several tiles due to rotators under the right circumstances moving the player an extra space.

Furthermore the performance of 4-player Tree-depth with Element Count weighted negatively, the resulting levels contained no intractable elements whatsoever, since the search algorithm prioritized merely moving the players to all available squares first, and with the increase in level area of the fourth level over the first one this means that the algorithm is terminated before the area was fully explored.

Lastly, the steps-to-solve for 4-player tree-depth unoptimized, while we know for sure the level is solvable. The level can be solved with a number of different strategies all involving different action dependencies between several player-characters. As a result we were not able to come up with a definitive solution path, let alone one approaching the minimum solution path (see figure 11). Lastly, to further test the capabilities of the algorithm two custom levels were entered: 3x3, a 3x3 grid of 3x3 rooms with single-tile doorways between them and the exit in the central room; and an 9x9 empty room with the level exit in

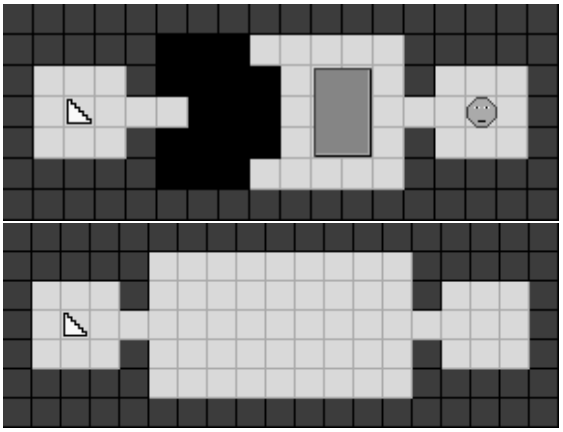

Figure 10. Level 4, original and as given as input (note miscount in the size of the level, though it should not affect the algorithm significantly)

Table 1. 1-Player Results for Level 1. (Names indicate number of players followed by heuristics used. Random was run five-fold, as such the mean and standard deviation are reported. TdTT refers to Tree depth with both proposed optimizations)

name	Closed set size	Steps to solve	Puzzle coefficient
originial	None	36	2.25
1pAs	1360	17	1
1pCe	34842	17	1
1pEc	19680	14	1
1pMce	39920	17	1
1pR(0-4)	20326.2(9489.9)	≈12.2(≈3.66)	1
1pTdFF	1100	20	≈2.86
1pTdFT	1100	20	≈2.86
1pTdTF	17966	11	1
1pTdTT	17949	11	1

Table 2. Additional 1-Player Results for Level 1

name	Closed set size	Steps to solve	Puzzle coefficient
1pTdTT-Ec-As	26006	19	≈1.27
1pTdTT-Ec+As	23557	15	1
1pTdTT+Ec-As	16078	47*	≈2.94*

Table 3. 4-Player Results for Level 1, the summations show steps-to-solve separated by player.

name	Closed set size	Steps to solve	Puzzle coefficient
4pCe	1403	2+2+17=21	1
4pEc	28856	2+2+14=18	≈1.56
4pTdFF*	6386	1+2+2+16=21	1.4
4pTdFT	6548	1+2+2+24=29	≈1.81
4pTdTF	1668	1+2+2+18=23	≈1.10
4pTdTT*	1329	2+1+1+16=20	1
4pTdTT+Ec-As*	6092	1+2+15=18	1
4pTdTT-Ec+As	15849	16+17+17+16=66	≈1.27
4pTdTT-Ec-as*	8588	1+2+3+4=10	≈1.11

Table 4. Results for Level 4, steps to solve for 4pTdFF unknown but estimated (see figure 11)

name	Closed set size	Steps to solve	Puzzle coefficient
original	-	24	≈1.85
1pTdTT*	16246	26	≈2.17
1pTdTT+R(0-4)*	5674.4 (4274.5)	16.0(2.45)	≈1.08(≈0.11)
4pEc	7654	2+2+13=17	≈0.81*
4pTdFF*	7012	∼60?*	∼2.4?
4pTdFT*	19499	2+19+18+20=59	≈1.74
4pTdTF*	848	2+2+2+9=15	1
4pTdTT*	1125	2+2+3+9=16	1
!4pTdTT-Ec+As	17058	2+9+16+16=43	1
!4pTdTT-Ec-As*	11465	1+1+2+2=6	1
4pTdTT+R0*	5034	1+5+11+18=35	≈1.09

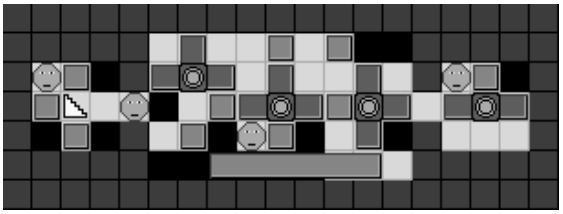

Figure 11. 1pTdFF for Level 4, where we were not able to find the steps-to-solve

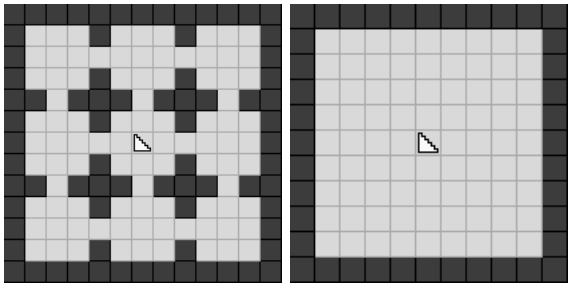

Figure 12. The 3x3 and open levels, as given as input

the center(see figure 12). Both designed specifically to test the algorithm's ability to box the player character in, as both allowed for numerous potential routes to attempt to circumvent parts of the puzzle (see tables 5 & 6).

As expected, the open level performed noticeably worse than the 3x3 grid level, and punishing adding elements in the hopes of aiding exploration made it so that no elements were ever added to the levels. In the open level, 1-Player Element Count fails the no mazes requirement not by failing to add elements, but by being entirely agnostic to the player's position, and ultimately placing them the tile next to the exit.

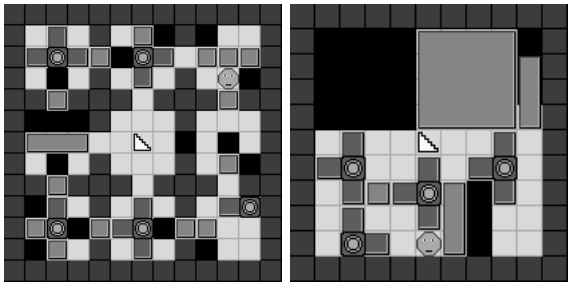

Figure 13. The best performing puzzles of the 3x3 & open batches

Table 5. Results for 3x3 Level

name	C.s.s.	Steps to solve	P.c.
1pEc	5363	8	1
1pTdTT+R0*	7379	10	1.25
1pTdFF	585	6	1
1pTdTT+Ec-As	14671	17	≈2.43
4pTdTT+R0*	2486	23	≈1.44
4pTdTT+Ec-As	4558	13	1
!4pTdTT-Ec-As*	9024	12	1

Table 6. Results for Open Level

name	C.s.s.	S.ts.	P.c.
!1pEc	3535	1	1
1pTdFF	16630	6	≈1.5
1pTdTT+R0*	6608	7	1.4
1pTdTT+Ec-As	14482	6	1
4pEc	600	14	≈1.08
4pTdTT+R0*	5039	16	≈0.89
4pTdTT+Ec-As	4292	14	≈1.08
!4pTdTT-Ec-As*	30306	8	1

3.1 Observations

A few interpretive notes on the data which will inform the conclusions drawn.

First, it should be noted that every setting tested has at least one result with a puzzle coefficient equal to, or less than one, failing the "De-tours required" requirement.
We also believe that the excellent, though inconsistent performance of the Element Count heuristic to be significant as this heuristic is entirely agnostic to the solution path. It shows that with this generation method merely blindly adding obstacles may provide positive results. This also provides a reason for the lacking performance of the subset-testing optimization compared to its unoptimized version. We speculate this is because in the runtime required it fails to gain control over the minimum solution. Having the optimizations effect be a decrease in the amount of obstacles that are "unneeded" from the algorithms point of view, with the success of Element Count showing these "unneeded" obstacles may still have made for more difficult puzzles.

4. CONCLUSION & DISCUSSIONS

RQ1: We suggested a pruning algorithm based on the adaptation in level representation which is supposed to take control of the generation path in order to have the canonical solution also be the minimal one. In practice however it was outclassed by its unoptimized counterpart in most comparative tests. We believe the order of state exploration to be a major factor in this. As a result, we would suggest investigating this approach in combination with an iterative deepening search algorithm rather than a depth-first search.

RQ2: In the initial round of tests we found that tree-depth and Element Count both gave rise to interesting levels. Both of these have proven inconsistent however be they on their own, summed or subtracted. Ultimately we were not able to find a setting that reliably generated levels with solutions more complicated than the shortest route straight to the exit.

To conclude, we believe that the larger goal for this project ultimately was successful. Using this algorithm we are able to reliably generate solvable, not somehow obviously abhorrent looking Kwirk levels. We were not able to have the levels reliably be interesting or difficult however. But we believe that with further research into various heuristics, pruning policies and search algorithms generation of reliably difficult levels should be possible.

5. REFERENCES

[1] D. S. Bento, A. G. Pereira, and H. L. Levi. Procedural generation of initial states of sokoban. *IJCAI International Joint Conference on Artificial Intelligence*, 2019-August:4651–4657, 2019.

[2] D. Bhaumik, A. Khalifa, M. Green, and J. Togelius. Tree search versus optimization approaches for map generation. *Proceedings of the 16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2020*, pages 24–30, 2020.

[3] S. Englhart. Puzzle boy. *Power Play*, 25, 1990.

[4] S. Englhart. Sokoban. *Power Play*, 25, 1990.

[5] R. P. Grimaldi. *Discrete and Combinatorial Mathematics*. Pearson Education, 2003.

[6] B. Kegel and M. Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, PP:1–1, 05 2019.

[7] A. Khalifa and J. Togelius. Multi-objective level generator generation with marahel. *ACM International Conference Proceeding Series*, 104, 2020.

[8] Y. Murase, H. Matsubara, and Y. Hiraga. Automatic making of sokoban problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1114:592–600, 1996.

[9] J. Taylor and I. Parberry. Procedural generation of sokoban levels. *6th International North-American Conference on Intelligent Games and Simulation 2011, Game-On 'NA 2009, 3rd International North American Simulation Technology Conference, NASTEC*, pages 5–12, 2011.