Generic Puzzle Level Generation for Deterministic Transportation Puzzles

Jelle Postma
Master Student at Utrecht University
Student number: 3840859

December 2016

Abstract

The focus of this thesis lies on the generic evaluation of levels. We believe this is the most crucial component of a generic level generator. We describe three 'generic' evaluation methods, of which the breaking-rule approach was the most promising. A heuristic is used that predicts the problems the player has to overcome, to solve the level. It uses a restricted input space and breaks certain rules to estimate the distance from a solution. To test the evaluation method, we generated levels for 5 different puzzles and tested their difficulty label in a small experiment. The levels are deemed hard by the experiment, even after generating an easier level set, when most participants could not solve any levels in an hour of play. The solving time and difficulty rating the participants gave to the levels has a high correlation with the evaluation scores of the breaking-rule approach.

Contents

1 Introduction **4**
- 1.1 What is a Puzzle Level . . . 4
- 1.2 Automatic Level Generation . . . 4
- 1.3 Transportation Puzzles . . . 4
- 1.4 Focus on the Evaluation of Levels . . . 4
- 1.5 Thesis Overview . . . 4

2 Literature **5**
- 2.1 Puzzle Level Solving . . . 5
- 2.2 Puzzle Level Evaluation . . . 5
- 2.3 Puzzle Level Search . . . 6
- 2.4 General Evaluation Properties . . . 6

3 What is Necessary for an Evaluation Function and what Characteristics are Desirable? **7**
- 3.1 Is a Level Solvable? . . . 7
- 3.2 Simple Iterative Local Search Generator . . . 7
  - 3.2.1 How the Iterative Local Search Generator works . . . 7
  - 3.2.2 Analyzing the Generated Levels . . . 7
  - 3.2.3 Small Conclusion . . . 8
- 3.3 Performance, Desired Correlation of Level Difficulty and Processing Time . . . 8
- 3.4 Relative Difficulty . . . 8
- 3.5 Iterating on the Evaluation Function using the Generation-Validation Method . . . 9

4 Sub-Problem Driven Approach **10**
- 4.1 Sub-Problem Driven Approach Overview . . . 10
- 4.2 Goal-Step-Plan . . . 10
- 4.3 Goal-Step Graph . . . 10
- 4.4 Creating and Resolving Sub-Problems . . . 11
- 4.5 Expanding Nodes . . . 11
- 4.6 Ways to Optimize the Sub-Problem Driven Approach . . . 12
- 4.7 The Sub-Problem Estimating Heuristic . . . 12
- 4.8 Filtering Redundant and Unwanted Nodes . . . 13
- 4.9 Defining an Order in the Node Generation Process . . . 13
- 4.10 Shortcomings of the Sub-Problem Driven Approach . . . 14

5 Constraint Clash Approach **14**
- 5.1 Constraint Clash Approach Overview . . . 15
- 5.2 Goal-Step-Plan Revised to: Must-Step-Plan . . . 15
- 5.3 Setting up the Constraint Graph . . . 15
- 5.4 Identifying Constraint Clashes . . . 16
- 5.5 Resolving Constraint Clashes . . . 16
- 5.6 Why the Constraint Clash Approach is Promising . . . 16
  - 5.6.1 Solving Performance gains . . . 16
  - 5.6.2 Isolating Problems with just the Relevant Mutables . . . 17
  - 5.6.3 Informed Search is Desirable . . . 17
- 5.7 Shortcomings of the Constraint Clash Approach . . . 17

6 Breaking-Rule Approach **18**
- 6.1 Breaking-Rule Approach Overview . . . 18
- 6.2 Expanding Nodes . . . 19
- 6.3 Breaking-Rule Metric . . . 19
- 6.4 New-Nodes Metric . . . 19

6.5 Movement-Freedom Metric . . . 20
6.6 Metric Influence Comparison . . . 20
6.7 Pre-Processing: Naive Solving Filter . . . 20
6.8 Post-Processing: the Expanded-Node Graph . . . 21
6.9 Why the Breaking-Rule Approach Performed Better . . . 22

7 Generic Rule API **22**
7.1 Implemented Puzzle Rules . . . 22
7.2 Transforming the Sokoban Generator to a 'Generic' Generator . . . 23
7.3 A Generic Must-Step-Plan . . . 23
7.4 What is Needed to Start Generating Levels for a new Puzzle? . . . 24
7.5 Rule Analyzer to Avoid Writing Code when a new Puzzle is Introduced . . . 26

8 Experiment and Results **27**
8.1 Setup and Execution . . . 27
8.2 Hypothesis . . . 27
8.3 Results . . . 27
8.3.1 First Solved Levels . . . 28
8.3.2 Time Spent on Levels . . . 28
8.3.3 Player Difficulty Rating . . . 28
8.4 Comparing the Results of other Work . . . 28

9 Future Work **30**

10 Conclusion **31**

Appendices **32**

1 Introduction

1.1 What is a Puzzle Level

People have different definitions for 'puzzle' and 'level'. By a *puzzle* we mean the set of rules that describe input possibilities, their consequences and goal conditions: all necessary information needed in order to solve each level created for the puzzle. While a *level* is a certain configuration of puzzle elements and provides the actual challenge to the player.

We want to generate fit levels, where a fit level is perceived as enjoyable and interesting by the player.

1.2 Automatic Level Generation

Automatically generating puzzle levels has been done for years on numerous puzzles and can provide several benefits over hand-crafting levels, for example, automatic generation is often faster than the level design process done by humans. The puzzles can be evaluated on difficulty, thereby presenting players with an appropriate challenge. Furthermore, the generated levels can aid as a base for human designers to create better levels.

1.3 Transportation Puzzles

A deterministic puzzle game is a one player game that involves no chance. If a sequence of moves results in a solution, that sequence will always result in a solution. In this thesis we focus on a sub-set of deterministic puzzle games: *transportation puzzles*.

What is a transportation puzzle? Transportation puzzles have multiple elements that are transformed by input from the player. This transformation is often a change in position, but can also be a change in element type, size, rotation, etc. Transportation puzzles always have a goal-state which is defined on an element level. In the case of Sokoban, the goal-state is reached when the position of each box element is equal to the position of one of the depot elements. Transportation puzzles have mutable elements, and immutable elements. In the case of Sokoban, the mutables are the boxes, while the immutable elements are the walls and depots.

Note that we did not include the worker (which interacts with the boxes) as a mutable here. Throughout the course of this thesis we discuss Sokoban and variations of Sokoban, we distinguish between the worker and mutables themselves (the boxes). So when we refer to mutables it does not include the worker.

Examples of transportation puzzles are: Rush Hour, Sokoban, Jelly no Puzzle, Snake Bird, and Zen Garden.

1.4 Focus on the Evaluation of Levels

In our literature research [1] we broke the generation of levels down into three parts:

- **Puzzle level solving:**
  Is the level solvable?
- **Puzzle level evaluation:**
  How fit/difficult is a certain level?
- **Puzzle level search:**
  How to find fit levels?

The focus of this thesis is on the level evaluation part. We think the quality of the generated levels is most dependent on the evaluator. The solver is only influencing how quick we will get results (when the solver is not intertwined with the evaluator). The search method has influence on which levels are found, but most search methods are guided by the evaluation method, thus even mediocre search methods are likely to find good levels when they are guided by a perfect evaluation method. In the case of the evaluation method being mediocre, the search method is guided to worse quality levels. Furthermore, the program is more likely to output levels with a wrong difficulty label.

The level evaluator is also the most specific part of puzzle level generation. Well known search algorithms can be used to make a generic solver [2]. Genetic Algorithm (GA) and Local Search can be used as a generic search method. No such algorithm can be used for generic puzzle level evaluation, as the evaluation method needs to identify what makes a problem hard for a human.

To test the evaluator's performance we wanted to generate new levels, so we had to implement a search method. However, minimal effort is spent on the search method.

The way we evaluate levels is closely related to the solver and in order to keep our approach feasible we do concern ourselves with the time and mostly memory complexity of the level solver. Generating levels quickly is deemed less important than the quality of the levels.

1.5 Thesis Overview

We have narrowed generic down to: generic for transportation puzzles. We started with a specific generator for one of the most popular transportation puzzles (Sokoban), while ensuring that the program is already generic for transportation puzzles (so we would not have to remove Sokoban specific exploits later on).

When the program worked satisfactorily well, we made it generic without touching the core evaluation method. Instead we just transformed the core system's API to work on more different puzzle rules incrementally.

We will first look at the literature on generic puzzle level generation. Then a extensive introduction is given on the problem we want to tackle in this thesis. From there the core part of the thesis begins: the three approaches we have implemented and an explanation on how generic the final method is. Each approach led to the next. The third approach gave the best results by far. We present all three to inform the reader of our decisions along the way. Afterwards we present a pilot experiment to test our results. We end the thesis with our view on future work and a conclusion.

2 Literature

This section will provide a compact literature overview, for the full literature research see [1].

As mentioned before the generation of levels can be broken down into three parts. We will look at the literature on each of the three parts separately (solve, evaluate and search).

The focus of this thesis is on the evaluation part. So at the end of this section we will list some general properties on level difficulty that can be used in a generic evaluation function. We have put this list together from the literature research.

2.1 Puzzle Level Solving

The puzzle level solving methods used throughout previous research can be split into the following categories:

- **Exhaustive Search Solvers:**
  By far the most general and easy to implement is an exhaustive search based solver. Drawbacks are often computing resources. Depending on the algorithm large amounts of time, memory, or both can be needed in order to find a solution.

  Some examples of exhaustive search algorithms used in puzzle level solving are: breadth first search (BFS) [3] [2], depth first search (DFS) [4], A*/BestFS [2] [3], iterative deepening (ID/IDA*) [5], and dynamic programming (DP) [6].

- **Techniques Partially Considering the Search Space:**
  Researchers have tried to omit exhaustively searching a level's search space by partially traversing it [7] and with plausible move generation [8]. Plausible move generation cuts off branches that are never to be considered again. A drawback of this is the possibility to cut off essential moves.

- **CSP and ASP:**
  Constraint Satisfaction Programming (CSP) is a popular and general technique for modeling combinatorial optimization problems. Pelánek [9] and Jefferson et al. [10] make use of CSP to solve their generated levels, and use the solving time directly as evaluation score.

  Answer Set Programming (ASP) [11] is similar to CSP. Smith et al. [12] [13] and Neufeld et al. [7] use ASP to ensure their generated levels are solvable.

- **Deductive Solvers:**
  In deductive reasoning methods guarantee that any advances within the reasoning, are correct. No guesses are made at any time.

  Deductive solvers use deduction techniques to gather more information on a level. With this information, more deductions can be made until a solution is found. Researchers often claim that their deduction solvers are based on general techniques that can be applied to many puzzles [14] [15] [16].

2.2 Puzzle Level Evaluation

- **Trained Model as Evaluation Method:**
  The work of Mutser [17], and Vendrig [18] both involve fitting user data on a linear model with several hand-crafted metrics.

  Wang et al. [19] present a Genetic Algorithm (GA) approach that evolves an evaluation method to fit user data. This approach is tested on Sudoku only, although Wang claims it to be generic.

  These methods are generic as long as there is good quality user data available. In our thesis we cannot use these methods, as we aim for a method that works when only a definition of the puzzle rules is given.

- **Deduction Based Evaluation Method:**
  Deductive reasoning is based on information that is known to be true.

  Meng and Lu [15] presents a method to generate *Sudoku* levels, the levels are evaluated by the necessary application of five deduction methods.

The most difficult required assignment is used as rating, so a level gets a rating between 1 and 5. A similar approach is presented by Oranchak [20] for *Shinro*[1] levels.

Browne [14] presents a solving and evaluation technique for logic puzzles called Deductive Search (DS). The authors measure the required recursive depth of the deductive reasoning, and use this to evaluate the levels. The technique has been tested on 5 different logic puzzles and the difficulty rating seems accurate.

Other work that uses deduction based evaluation are [9] [10].

- **Search Space Related Evaluation Method:** A puzzle's search space is the directed graph (may contain cycles) that has a node for each state the level can be in. It has a directed edge from node N_1 to N_2 when the corresponding state N_2 is reachable from N_1.

  Jarušed and Pelánek [21] [22] [23] label each state in the search space of Sokoban, with the distance from the closest solution and analyze how a human solving model solves the level. The method shows promising results, although it will require a lot of computing resources.

  Guid and Bratko [24] argue that a problem is more difficult when new and better solutions are found at higher search depths in Chess.

  Kotovsky and Simon [25] present experiments aimed to explain the differences in difficulty of isomorphic puzzles. As the search spaces are identical in isomorphic puzzles, something about the presentation of puzzles has influence on their difficulty.

  Other relevant research related to the search space of puzzles are [12] [26] [27].

2.3 Puzzle Level Search

- **Local Search:**
  Local Search is a very general search method.

  Oranchak [20], Jefferson et al. [10] and Hauptman et al. [28] all use local search in their search method.

- **Generation with Genetic Algorithms:**
  Genetic Algorithms are another common search method in puzzle level generation.

  The drawback is that in practice it is hard to define a useful crossover method for many puzzles, which reduces the GA on those puzzles to a simultaneous local search algorithm, that manages the time spent per local search instance (which is still more effective than plain local search).

  Williams-King et al. [26], Ashlock [6], Khalifa and Fayek [2] and Lim and Harrell [3] all use GA in their search method.

- **Constructive Search Approaches:**
  Constructive search approaches define parameters for a random generator, which then generates levels. The evaluation method is only consulted after the level is created. This approach often ensures the solvability of levels with the generation parameters.

  Browne [16], Vendrig [18], Taylor and Parberry [5] and Murase et al. [29] all use constructive algorithms in their search method.

2.4 General Evaluation Properties

We have encountered several general properties that influence puzzle level difficulty. The list of general properties we deem most important follows:

- **Required changes of interaction / Mutable switches:** In *Sokoban* for instance, interacting with a different box.
  The need to change the interaction a number of times, in order to find a solution, seems to indicate difficulty and interesting level design [5] [30].

- **Shortcut solutions:** Unintended solutions that can be found without grasping the concept the level designer intended the player to understand.
  Shortcut solutions function like cheats and are very likely to reduce a level's difficulty [12].

- **Potential sub-problems that lead to checkpoints:** A solved sub-problem gives a partial solution, when this partial solution is included in a complete solution it is a checkpoint. A level is generally more difficult if a lot of potential sub-problems exist and only a few of these form checkpoints. A similar observation was made by [27].

- **Linear search space:** The player has no choices to make in a linear search space, there is basically one option. Once the player realizes the search space is linear the level becomes trivial. A similar observation was made by [25].

[1] en.wikipedia.org/wiki/Shinro

- **Tricky search space:** When a search space contains a lot of seeming shortcut solutions that fail once the search is continued to a deeper level, this is a tricky search space. The player will probably get tricked in thinking he or she found a shortcut solution.
  Once the player is tricked, he must then think outside 'the box' to find the real solution. A similar observation was made by [24].

- **Elegance:** This is more related to player interest than level difficulty. Elegance is the ratio between perceived difficulty and the complexity in terms of elements in the level.
  For instance, a *Sokoban* level with 3 boxes, that is equally difficult as a level with 10 boxes; is considered more elegant than the latter, and thus the former is preferred.

Each of the properties listed above is utilized in at least one of our approaches.

3 What is Necessary for an Evaluation Function and what Characteristics are Desirable?

The purpose of this section is to bring the reader up to speed with some of the problems we want to overcome in this thesis. We will explain what we are looking for in terms of characteristics and what we want to avoid, in an evaluation function.

First we will look at level solvability, then introduce a fully functional Sokoban specific generator as a prototype to build upon in the rest of the thesis. Afterwards we cover 3 subsections on the characteristics we desire our evaluation method to possess and why.

3.1 Is a Level Solvable?

We want our generated levels to be solvable. There might be puzzle rules for which the existence of unsolvable levels does have meaning, but we will assume we are generating for puzzles that always need solvable levels.

To ensure all final levels are solvable, we simply made a puzzle solver that verifies whether a level is solvable. The solver uses a 'ruleAPI' so it will solve levels for each puzzle that has implemented the 'ruleAPI'.

We cannot use a naive breadth first search (BFS) implementation for our solver, as transport puzzles are often PSPACE Complete and even reasonably sized instances can consume several gigabytes of memory in the solving process. We did end up using an A* algorithm with our own invented heuristics to be able to solve a bigger range of levels.

3.2 Simple Iterative Local Search Generator

We set up a Sokoban level generator prototype. The main purpose of the prototype was to get a feeling of the different parts needed for a generic generator, to give a better judgment on how the evaluator / generator performed later on.

3.2.1 How the Iterative Local Search Generator works

The level evaluator counts the smallest number of box switches necessary to solve a level. A box switch is counted when a different box is pushed from the last box that has been pushed. The level search / generation part uses a simple iterative local search (ILS) like algorithm. The algorithm works as follows: a local search is performed on Y neighbors. Neighbors are generated by applying X changes to a level. Once no improvement is found the process is repeated with $X + 1$ changes. The next iteration, local search will be performed on the best improvement. If an improvement is found, X is set to its minimum value: two. This process is repeated until G generations/iterations of the local search phase have been performed. After G generations/iterations the best level is returned as the result.

3.2.2 Analyzing the Generated Levels

This simple setup produced decent quality levels in a reliable time frame, but the level evaluator was of course not perfect. We would often see that the first 3 box switches were trivial in the sense that only a small number of wrong moves can be made during this phase of the level. Furthermore, the possible wrong moves immediately result in a deadlock or a box unable to reach any goal independent of other boxes. In other words; the possible wrong moves (in the start of the level) were easy to identify as such. This pattern is destructive as the actual level difficulty has not increased much by these 3 necessary early switches. A neighbor level with more non-trivial box switches is now harder to find, since we require a level with 4 extra non-trivial box switches, or a similarly locally optimized level with one extra box switch. The latter is easy to achieve with a local search on this particular neighbor, but not likely to occur directly in the neighborhood.

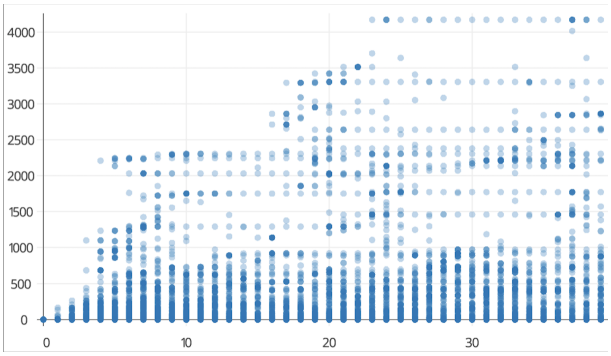


Figure 1: ILS performance, generations versus evaluation score of tried levels. A more opaque color means more levels scored this evaluation.

3.2.3 Small Conclusion

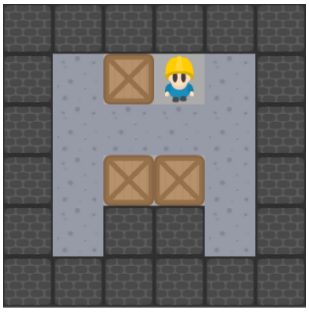
As mentioned before, the purpose of this prototype was not focused on the generation results yet. We gathered data on the results from the generation process. In Figure 1 a scatter diagram of the evaluation score of tried levels per elapsed generation is shown. The data indicates that further on in generation, the levels get a higher evaluation score, as well as that most tried levels are either unsolvable or get a low evaluation score throughout the whole generation process. The increase of evaluation score is due to the nature of ILS which iterates on the previously best level found. It would be interesting to have an adapting algorithm that interprets the metrics while it is solving and adapts its strategies likewise, however this thesis is more focused on the evaluation part. The graph is thus only used to indicate that the ILS generator does decent work and is viable for testing evaluation functions (more on this is explained in subsection 3.5).

Throughout the rest of the thesis we have used the ILS generator to generate levels in combination with several different level solvers and evaluation functions.

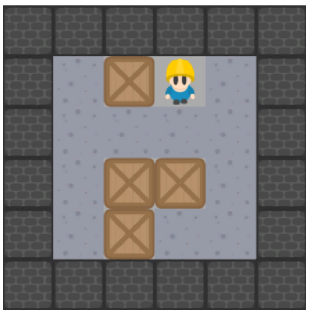
3.3 Performance, Desired Correlation of Level Difficulty and Processing Time

Since we strive for a generic generator it is difficult to keep the solving process optimized. Optimization is often about finding specific cases and exploiting them with a clever data-structure, which is the opposite of a generic solution.

Generating hard/interesting levels for transport puzzles will in general take longer than the generation of a simple and easy level, not the least because for most transport puzzles, there are much fewer interesting levels than easy or unsolvable ones.



(a) A deadlock example, the two boxes at the bottom cannot ever move.



(b) Not a deadlock example, each box can move if we disregard the boxes that can move initially.

Figure 2: Deadlock example.

If program A is going to take X time to evaluate any level, and program B takes, for instance, X/Y time to evaluate an easy/uninteresting level, while taking $X * Y$ time to evaluate a difficult/interesting one; we prefer program B over A, since the vast majority of levels our generator tries in the process, are either unsolvable or uninteresting (see Figure 1). This leads to two conclusions. Firstly: if easy levels are processed quickly, very long evaluation processes of hard levels are feasible. Secondly: optimizing the filtering of unsolvable levels is relevant.

There are two easy and generic ways to optimize the filtering of unsolvable levels (both of these often also speedup the process for solvable levels by stopping the solver from expanding an unsolvable state). One way is 'deadlock detection': are mutables that have not yet reached their goal unable to move independent of other mutables that can still be moved? (See Figure 2). The other way is 'unreachable goals': mutables that are no longer able to reach a goal state, independent of any other mutable.

3.4 Relative Difficulty

In a perfect world, where a heuristic models the way humans solve puzzles perfectly, the number of nodes the solving algorithm has to expand in order to find

a solution, will have a linear correlation with the humanly perceived difficulty. Because the heuristic guides the solver to the same clever shortcuts (or misleading pitfalls for that matter) as a typical human would (we will come back to this observation in the next subsection).

Note that this 'perfect' heuristic does not exist by fact, since humans perceive relative difficulty different from one another.

Player A might have solved a thousand Sudokus that are very similar to Sudoku instance X but different from Sudoku instance Y, while player B might have solved a thousand Sudokus that all have very different characteristics. Level X is probably easier for player A compared to level Y, while player B might think of both levels to be equally difficult.

In this work we want to create an objective difficulty evaluator with the possibility to distinguish between the required solving strategies during the solving procedure, so that the relative difficulty within a level set, can be accounted for later on. We will not do anything with this relative difficulty in this thesis, apart form keeping it in mind while designing the difficulty evaluation function.

3.5 Iterating on the Evaluation Function using the Generation-Validation Method

From the observation about the perfect human like heuristic, from the previous subsection, follows that once we have a sub-optimal model of a typical human solving behavior, we can measure the heuristic's effectiveness with the correlation between expanded nodes and the human perceived difficulty. We can use the observation of subsection 3.3: the evaluator must be quick on easy levels, and may be slow on hard levels, in our algorithm design combined with the ILS generator, to find levels that took long to evaluate. So when an easy level comes up, we can analyze the flaws of the evaluator.

The generator searches for the level with the highest evaluation score. Thus once a level is evaluated too high, because of a certain flaw in the evaluator, this flaw will have been exploited as much as the generator possibly could, which makes it easier to understand and address the problem. We call this the generation-validation method

The same holds for hard levels that receive a relatively low score. If the heuristic works too well on these levels, there is probably some computer solving advantage in the level that the heuristic exploits. For instance, a harder level often has less free space to traverse, as movement through the level needs to be restricted to one or two clever and concealed solution paths. This results in less feasible nodes in the search space making a brute-force search quicker on a tight hard level than on an open easy level, such a flaw is easy to see when multiple easy levels with a lot of open traversing space come up in the generation process.

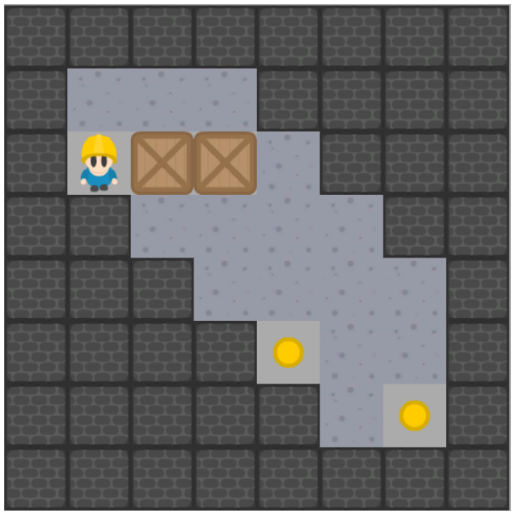


Figure 3: Sokoban example level that requires 14 box switches, we call it the 'staircase example'.

An example of a flaw of the prototype generator, identified through the generation-validation method, can be seen in Figure 3 of the staircase example. It has only two boxes and requires 14 switches, this is the highest number of box switches encountered (and maybe possible) for this level size and box count. Interestingly, the level turned up on multiple occasions which was never the case for any other level. The search space of this level is linear (when the player has deduced that a box cannot reach its goal when pushed in one of the corners). This level exploits the fact that the mutables-switched metric does not consider a level's search space, to such an extreme, that it is easy to identify the problem and a direction on how to fix it: make the evaluation dependent on the search space.

Overrating easy levels is much worse than underrating hard levels. We want hard levels, and eventually a set of interesting levels. When the easy levels (which are not interesting after one or two introduction levels) are overrated, these levels are more likely to end up in the final level set, which results in wrongly labeled levels in the final level set. On the other hand, when hard levels are underrated, these levels are less likely to end up in the final level set: other levels get a better score and overshadow the wrongly labeled hard levels. However, as long as the easy levels receive a correct label and not all hard

levels are underrated, the final level set will not contain wrongly labeled levels. The biggest impact of underrating a percentage of hard levels would be a longer generation time for the same quality of a level set.

The generation-validation method has less benefit on 'hard but underrated' levels as these are often kept under the radar (others score higher and are pushed to the top), but for puzzles like Sokoban we can use the widely available level-sets with hard levels, to help analyze when the evaluator is underrating levels.

Note that any algorithm characteristic can be evaluated using the generation-validation method. For instance, we tested memory use by making the evaluating score dependent on the memory consumed while solving a level, as well as time and many other characteristics.

4 Sub-Problem Driven Approach

Humans often break problems down into multiple sub-problems. As we explained in subsection 3.3 we believe that mimicking human solving strategies has a lot of potential in the evaluation of human perceived difficulty of a level. With the sub-problem driven approach we wanted to break the problem of a whole level up into smaller pieces and estimate how hard it would be for humans to split the problem up into meaningful sub-problems.

In this section we will take an in depth look at the sub-problem driven approach. We will first provide an overview of how the approach works. Then we introduce an abstracted view of a level's search space with the goal-step-plan and the goal-step graph, look at how the problem is split up into sub-problems using this abstraction, and explain how the search for a solution is driven by the expansion of nodes. Afterwards we explain how the heuristic works that greatly improved the performance of the approach, how the filtering of less potential nodes was unsuccessful, and how an order in newly generated nodes could potentially improve the computational performance. The section is concluded with the reasoning behind the choice to continue with a different approach.

4.1 Sub-Problem Driven Approach Overview

The main method of the sub-problem driven approach receives a problem $P0$, we check if this problem is trivially solved, and when it is not, we split the problem up into two smaller sub-problems. We split the problem in different places, each time creating a new sub-problem pair and a new node for the search algorithm. A node contains a sub-problem pair and a set of valid moves to reach its current state. We decide which sub-problem pair is most promising and repeat the process on each sub-problem separately.

The first problem that is handed to the algorithm is the whole level we want to solve. We calculate a higher level abstraction of the level's search space and traverse it until we get stuck. We use the information on where we got stuck at, to create the new sub-problem pairs.

4.2 Goal-Step-Plan

The goal-step-plan is an abstraction over the raw input possibilities of a puzzle, to a level and mutable specific input scheme. It resembles a naive linear plan for one mutable towards its goal. The plan is broken into steps of equal subsequent player inputs. See Figure 4 for the individual steps each box (mutable) will take in the displayed Sokoban level. The goal-step-plan is the path with the smallest number of steps towards the mutable's goal.

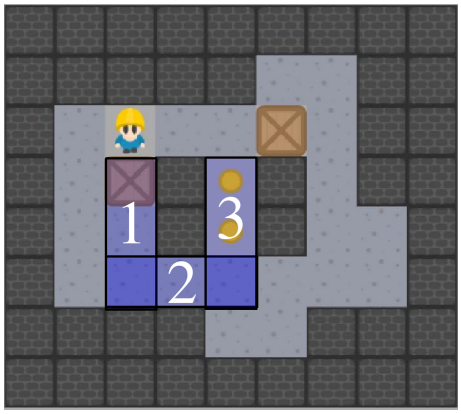
In the next section we will explain how the goal-step-plan is used in the sub-problem driven approach.

4.3 Goal-Step Graph

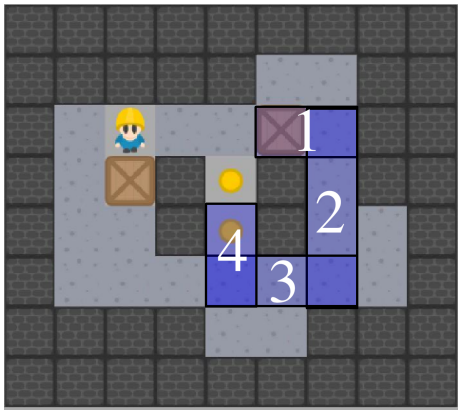
For the abstraction of the level we make use of the goal-step-plan. It is set up as follows. Each mutable generates a goal-step-plan, and we use these together to create a goal-step graph. The goal-step graph is a directed graph, and it resembles the search space in which mutables can only move forward in whole steps from their goal-step-plan. So a node represents at which step each mutable is. Nodes have at most M outgoing edges, where M is the number of mutables in the level. Outgoing edges only exist when the changed step can be accomplished without interacting with any other goal related mutable.

The goal-step graph is exhaustively searched for the goal-state, in which all mutables have finished their goal-step-plans. When the goal-state is found, a valid sequence of moves that results in the goal-state is determined. Note that the goal-state might be the solution to the level, but could also be a solution to a sub-problem, depending on the original problem ($P0$) that is focused on at the time.

If the goal-state cannot be reached with the goal-step graph, the traversal of the graph will reach dead ends: none of the mutables can take the next step towards their goal. We use the dead ends as points to



(a)



(b)

Figure 4: Goal-step-plan example, subsequent input is seen as one step. The numbers indicate the order in which the steps are taken.

split the problem up into sub-problems, this process is explained in the next subsection.

One benefit of using this goal-step graph, is that a search to the goal-state in this abstracted view can only move towards a potential solution and not away from it. Any possible cycles have been eliminated and mutables only move from their current state to their next goal-step. This makes an exhaustive search approach with the goal-step graph input space feasible in terms of memory and computing complexity, while it was not with the raw player input for most puzzles. A level is however, often not solvable using this naive input scheme.

The level is simplified so much that when a level is solvable from in this abstracted state, the level is often trivially easy. Also, when the traversal comes to a dead end and no step can be taken, there is a good chance this is a planning challenge the player has to overcome in order to solve the level. More on this will be explained further in this section.

4.4 Creating and Resolving Sub-Problems

When we split up a problem we generate multiple sub-problem pairs. Each sub-problem pair contains a starting problem $P1$ that starts the same as $P0$, but has a different goal-state, and an ending problem $P2$ that starts at the goal-state of $P1$, $P2$ has the same goal-state as $P0$. The starting state of $P2$ is chosen such that the goal-step graph from that state has at least one mutable which is guaranteed to make progress on its goal-step-plan further than the place it got stuck at before. The exact generation of the starting state of $P2$ is explained in the next subsection.

While the algorithm progresses, we break the problem in more and more sub-problem pairs, one pair describes a potential path to the solution but not necessarily from the start of the level. A node in the solving algorithm is one pair of sub-problems and the valid moves which have led to the start of the first sub-problem. When we expand a node, we always try to resolve the sub-problem closest to the start first. As only afterwards, we are sure that the starting position of the next sub-problem is actually reachable. When a sub-problem is solved, the necessary moves are added to the node.

The problem the level imposes is broken down step by step until each individual sub-problem is trivially solved and we have found a solution.

4.5 Expanding Nodes

To calculate the minimal number of sub-problems needed to find a solution, we used an A* algorithm.

A node in the A* algorithm consists of the sub-problems that are not solved yet, and a list of moves that brings us from the start of the level to the first unsolved sub-problem. The heuristic score of nodes is calculated as follows: number of sub-problems (including the solved ones), times the sum of the distance each mutable has to travel on their goal-step-plans before they reach their goal. Which results in a BFS like behavior, with DFS jumps when the mutables made decent progression in their goal-step-plans. This heuristic gave the best result when only distance functions and sub-problem count is considered in the heuristic.

Once a node is expanded, we generate a set of new nodes. For each mutable we calculate the list of places it can reach without breaking a puzzle rule. We then take these lists and calculate all the combinations wherein one place is picked from each list, such that the final configuration is consistent with the puzzle rules (for instance no overlap of boxes in the case of Sokoban). The new place combinations are then filtered to guarantee for at least one of the mutables that it can proceed further in its goal-step-plan, than where it got stuck at before.

4.6 Ways to Optimize the Sub-Problem Driven Approach

The explored sub-problems depth throughout the search was very low. Even for hard levels this number would not reach 10 (at least not before running out of memory). However, the number of sub-problems that were possible and thus generated in large levels, was enormous (see Table 2). This resulted in infeasible memory use for those levels.

There are three evident ways to improve on this matter:

- **1. Heuristic:** We can optimize which sub-problems are tried first with a heuristic that picks out the most promising node.
- **2. Filter:** We can reduce the number of sub-problems generated at each dead end with the use of a filtering process.
- **3. Ordering:** We can create an ordering in how the sub-problems are generated, so a single choice of expansion becomes less critical. The progression in the ordering would be seen as a new expansion and is only performed when this node is still deemed promising.

We choose to go with the heuristic first, as the filter has the potential of filtering out tricky solutions when it gets confused by good level design (resulting in not finding a solution), and the ordering option raised hard questions of when to continue expanding an already expanded node, over a new one. A heuristic seemed more promising as the sub-problems that were mistakenly scoring low would eventually still be expanded, so there is no danger of not solving solvable levels. We can even use the misguidance of the heuristic as an additional measure of difficulty: expanding more nodes before finding a solution indicates a harder level. The solving time of hard levels would be much higher than that of easy ones, but as mentioned in subsection 3.3 this is acceptable for hard/high quality levels.

4.7 The Sub-Problem Estimating Heuristic

What kind of heuristic can be used to pick better nodes to expand? We will first define a 'better node'. When we created the sub-problem algorithm, we had a hypothesis in mind: the number of (trivially solved) sub-problems needed to form a solution, correlates with human perceived difficulty. So we want to find the shortest (or at least a relatively short) sequence of sub-problems that leads to a solution. With this observation the best node to expand is the node for which the sub-problem distance away from a solution, plus the sub-problems tackled already, is minimal. So our heuristic must estimate how many sub-problems will be needed to reach a solution.

Sub-problems are generated when the traversal of the goal-step graph reaches a dead end. So naturally the heuristic traverses the goal-step graph and inspects the dead ends. We could base a metric on the number of steps left at a dead end. But a better way to estimate the number of problems ahead comes from estimating the minimal number of dead ends the traversal will get stuck on, before reaching a solution. So instead of finding the dead end that gets us closest, in terms of steps away from a solution, we compute the least number of illegal steps we need to take, in order to reach a solution. This is done by computing the cheapest path through the goal-step-cost graph. This graph has the same nodes as the goal-step graph but the edges that represent mutables taking an illegal step, are present in this graph. The edges are weighted with either 1 or 0. Edges representing an illegal step get a weight of 1, the others get a weight of 0.

This heuristic performed very well as can be seen in Table 1. The problem estimating heuristic decreased the number of expanded nodes by a factor of 2.5 to 4. However, calculating the heuristic took a long time and in order to select the best node, all the en-queued nodes must receive a heuristic score. The number of potential nodes generated at each expansion was too high (see Table 2) which meant the al-

Table 1: Expanded nodes comparison of the heuristics, n/a means more than 8 GB RAM was needed.

Level name	BFS like	Problem estimating	Factor
Hand-crafted 1	79	22	3.6
Hand-crafted 2	132	51	2.6
Hand-crafted 3	445	73	6.1
Hand-crafted 4	991	393	2.5
Hand-crafted 5	n/a	n/a	n/a
Hand-crafted 6	n/a	n/a	n/a
[21] 43 minutes	n/a	n/a	n/a

Table 2: Average generated nodes per expansion.

Level name	Generated nodes per expansion
Hand-crafted 1	9
Hand-crafted 2	8
Hand-crafted 3	42
Hand-crafted 4	32
Hand-crafted 5	18093
Hand-crafted 6	2306
[21] 43 minutes	41433

gorithm was still infeasible on medium to large sized levels, especially in combination with a high number of mutables. The average nodes generated at an expansion were over 40000, for a level of size 8x6 and 4 boxes that players solved in a median time of 43 minutes [21]. We did not manage to solve this level with the sub-problem driven approach.

4.8 Filtering Redundant and Unwanted Nodes

The natural next approach is filtering the unlikely nodes. This improvement complements the heuristic, and should solve the problem of having too many potential sub-problems for which the heuristic must be calculated. However, the anticipated drawback of filtering came up of almost every non-trivial level: removing necessary nodes and eventually deeming a level unsolvable when it is actually solvable.

The filtering process checks for every new node, if they are trivially accessible from any other new node. They are deemed trivially accessible when they are one mutable-switch apart from each other, those nodes are discarded one after the other.

We tried this way of filtering because it seemed unlikely that the filter would interfere with the solver's ability to find a solution. When it turned out most levels were deemed unsolvable after this filtering attempt, we thought it was due to a bug we just created. However, the levels became unsolvable (for our solver) because of how the goal-step-plans were set up. Even though the nodes were trivially reachable from other queued nodes, their goal-step-plans could take a completely different path the queued nodes might not consider. Thus the filter process could filter out solution-nodes even when none of the other nodes were leading to a solution.

The goal-step-plan is the complete path towards a mutable's goal, in the most naive way possible. This path could be drastically different from another position, even if the starting positions are trivially accessible from each other, as can be seen in Figure 4. Thus the filter process could filter out solution-nodes even when none of the accepted nodes were leading to a solution.

4.9 Defining an Order in the Node Generation Process

Because so many nodes are generated each expansion (see Table 2), we want to look at a subset of the possible new nodes. Also, when we look at more of them, we want to do so in a meaningful order. We could achieve this with an ordering function that discriminates between generated nodes: first generate all new nodes and then order them so we can look at them in a meaningful way. Or, we could define an ordering inherent in the node generation process, in that way the nodes of a higher order do not have to be generated. We can then generate new nodes in meaningful chunks, without having to generate the other possible nodes at all. The number of generated nodes each expansion was so huge, that this last approach was desired.

There are two reasons why it is a good idea to define an ordering in the node generation process: Firstly, we can speed up the solving process by stopping to look for more nodes when we already have one or more good candidates (and save valuable time by not calculating the expensive heuristic score for these nodes). Secondly, if we can correlate the order in which nodes are generated by how likely they are to be thought of by humans, we can estimate how hard a certain node is to be found and improve our difficulty evaluation with this information.

How would such an ordering be defined? We want the ordering to correlate with how likely humans are to see the potential of this new node when they first stumble on the current problem. The most naive way to do this, is generating new nodes in the order of distance away from the problem state. For instance, the combined number of grid-points all mutables moved. Another ordering may reuse the *mutables switched* metric: generate new nodes with only one mutable

changed, before generating nodes with two, three, etc, changed mutables. This ordering can be combined with the problem-distance ordering.

We could use the reachability of new nodes as well. For instance, first generate all nodes that can be reached from the problem state. Then take one (goal-)step back from the problem state for each mutable and generate nodes that can be reached from there. Or, instead of the mutables that have been changed, define an order in the number of mutable switches that must be made to reach this new node. Note however, that if we were to expand in this manner indefinitely, we would just brute force the level's search space, thus a restriction must be added.

Lastly we can define an ordering by estimating how hypothetical a new node is. This ordering is best used in combination with another ordering, as it is computationally expensive. A node that is reachable like the way we described above is not hypothetical: we have just calculated that this node can be reached. The degree of how hypothetical a node is, is best estimated with the heuristic we use to estimate the number of problems we are likely to face. Even though we are looking to optimize the number of nodes for which the heuristic must be calculated in the first place, the heuristic can be useful here. The complexity of the heuristic is highly dependent on the goal-step-plans and the number of mutables involved. So by restricting these factors we can generate nodes in the order of how hypothetical they are, in a feasible way and thus balance between node quality and the number of new nodes generated each expansion.

4.10 Shortcomings of the Sub-Problem Driven Approach

The computation time of a brute force level solver is highly dependent on the possible states of the search space. Thus levels with a large grid and many mutables take the longest to solve, even though they might require only a few trivial moves. These properties are undesirable and we tried to avoid them in the sub-problem driven approach. However, moderately difficult levels with a lot of open space did not fulfill this desire. A small number of expansions were needed, but the openness of the level allowed for many new nodes each expansion. Furthermore, the solving algorithm often needed more then 8 GBs of memory on hard levels with 4 or more mutables (again magnified by the amount of open space in the level), which led to not finding a solution.

Problems were encountered from multiple places (different traversal dead ends) without the algorithm being aware of this, so the guarantee of one mutable progressing in its goal-step-plan does not always influence the actual problem that we want to solve, which made this guarantee not as useful as we had intended it to be. In other words: the search was not informed enough.

The last three subsections are full of promising improvements to the sub-problem driven approach in its current state. We believe that all these improvements have potential and would make the sub-problem driven approach perform much better. However, we decided to try a new approach.

Time spent with the sub-problem driven approach is not wasted, as we have gained valuable knowledge and experience with level abstractions, and the problem estimating heuristic, both are promising tools for a good evaluation function. However, the difficulty of optimizing the computational complexity in the current implementation made us consider a different approach.

The next section is about this new approach, we set up the approach with a different but similar level abstraction as the goal-step graph to allow for a faster computation of the heuristic, as well as a more informed solver with better evaluation opportunities, and a computation complexity more dependent on level difficulty.

5 Constraint Clash Approach

After the sub-problem driven approach, we decided to use our newly gained knowledge and try a new approach for the evaluator.

The goal of this approach is to support a broader range of levels. We wanted to improve the calculation time of the problem estimating heuristic by redesigning the abstraction of the level. As in the goal-step graph the same problems were encountered from multiple places, without providing more information to guide the search towards more promising places in the future. The calculation of identifying the problems ahead, is much less dependent on grid-size and the number of mutables in the constraint clash approach.

This section is about the constraint clash approach, we will first look at an overview of how the approach works. Then explain how we updated the abstraction of the search space from the goal-step graph to the *constraint graph*. We then look at how the constraint graph can be used to identify problems the level imposes to the player and how these can be resolved. Afterwards explain in depth what makes this approach more promising than the sub-problem driven approach, and conclude with an argumentation on why we ultimately decided to continue with

the breaking-rule approach.

5.1 Constraint Clash Approach Overview

The constraint clash approach calculates an abstracted view of the level in the form of the constraint graph. This is a directed graph. A node in the graph represents the state of one mutable in terms of which step it is at in its *must-step-plan* (see the next subsection). Each edge in the constraint graph represents a constraint between two mutables. When we identify a cycle in this graph, we have identified a problem. We then zoom in on a constraint cycle/problem and fix it for the relevant mutables.

Resolving a problem is similar to the way we resolved problems in the sub-problem driven approach: we find configurations of the mutables such that the problem is improved upon. However, this time we do not try hypothetical configurations, and we have more reliable information on whether a problem has actually been improved.

5.2 Goal-Step-Plan Revised to: Must-Step-Plan

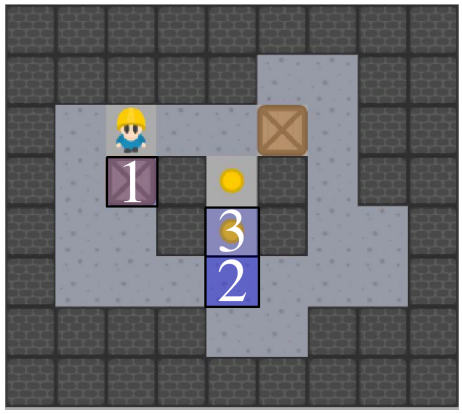
The goal-step-plan represents a completely thought out plan. However, there might be more ways to progress towards our goal. For the constraint clash approach we revised the goal-step-plan to: the must-step-plan.

The must-step-plan contains only the absolute necessary states the relevant mutable should be in, in order to reach its goal. See Figure 5 for the individual steps each box (mutable) should take in the displayed Sokoban level. A step is the path from one state the mutable must reach, to the next. If the next state can be reached by repeating an input, these states are grouped together to one step.

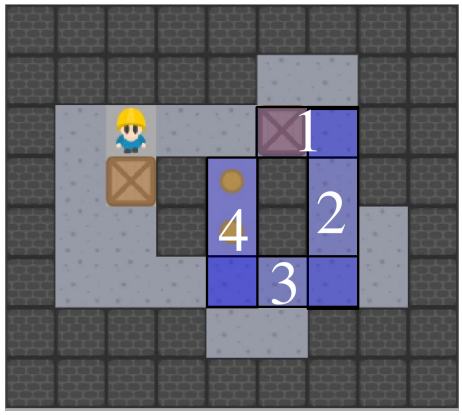
5.3 Setting up the Constraint Graph

The constraint clash approach creates a set of constraints for each mutable. Constraints come in the form: mutable A should move to X before mutable B has been at Y. Or: mutable B should move to Y after mutable A has been at X. Constraints are calculated for the must-step-plan abstraction. So for each step S of mutable M we calculate the constraints between all the steps of the other mutables and S. The upper bound on constraints is thus $O(n^2)$ where n is the combined number of steps of all mutables.

Constraints are also differentiated by the elements that impose the constraint. In the case of Sokoban, constraints exist for the worker: can he find his path when box B has been at place Y? And for boxes:



(a) A must-step-plan does not necessarily plan out every state the mutable will be in, along the way to its goal.



(b) A must-step-plan can be the same as a goal-step-plan.

Figure 5: Must-step-plan example, subsequent must-visit gird points are seen as one must-step, when the path to its goal is restricted. The numbers indicate the order in which the steps are taken.

can box A reach place X when box B has been at place Y? This distinction is important as a clash of boxes is harder to workaround than a clash of the worker and a box. The worker might, for instance, push this particular box along while he proceeds.

While calculating the constraints, we set up a constraint graph. Each individual step represents a node in the graph, and a directed edge is added for each constraint. So if mutable A should move to X before mutable B has been at Y, an edge is added from respective node A to node B. In other words, all incoming edges to arbitrary node N_x are constraints that must be resolved before the corresponding step S_x can be performed.

When the constraint graph is set up, edges are added from node N_1 to node N_2 when the corresponding step S_2 comes after step S_1. This is necessary to calculate where the constraints clash and a problem occurs.

5.4 Identifying Constraint Clashes

The constraints are based on the must-steps each mutable has to take in order to reach its goal. This abstracted view on mutable movability will identify problems, just like traversing the goal-step graph did in the sub-problem driven approach. However, instead of 'traversing' the constraint graph to find situations where the mutables can no longer move forward, we search for cycles in the constraint graph.

When there is a cycle in the constraint graph, the traversal of the goal-step graph would have gotten stuck around those steps as well. This is true because a constraint from step S_x to S_y is added when step S_y cannot be taken in the goal-step graph, before step S_x is performed, so when the constraints form a cycle, there are two or more steps that cannot be taken before one another. Therefore, none of these steps can be performed and the goal-step graph will inevitably get stuck.

5.5 Resolving Constraint Clashes

Once the problems are identified, we want to get rid of them. This phase is similar to that of the sub-problem approach. Just as with the sub-problems we find places for which the problem P does not exist anymore. This time however, we consider only the mutables on the cycle and take them one step back in their must-step-plan (continue taking steps back for individual mutables, if the state is not feasible) let us call this state S. From S we consider the places each mutable can travel to and search the space for configurations of the mutables such that they do not create any constraint cycle, or the first cycle these mutables create, is further towards the goal than the cycle we are trying to resolve. When we reach that state we have made progress on problem P, but not necessarily on the level. We have to merge the new must-step-plans and update the constraint graph. We do this for each improvement found for P and pick the best emerging constraint graph. A new heuristic is used to evaluate which constraint graph is the best.

The state S is preserved as new solutions to P might emerge when we take the mutables back another step.

5.6 Why the Constraint Clash Approach is Promising

The constraint clash approach has many aspects in common with the sub-problem driven approach. Problems are (indirectly) still based on the traversal of the goal-step graph getting stuck at some point, and the generation of new sub-problems is very similar to the way we look for configurations which omit a constraint cycle. There are however, a number of significant changes that result in big differences on how the algorithm progresses.

5.6.1 Solving Performance gains

The heuristic used in the sub-problem approach traverses through every reachable edge in the goal-step graph. The constraint graph is searched for cycles to identify all constraint clashes. The number of constraints can only be less than the edges in the goal-step graph (and they are usually much less). More importantly though, the constraint graph can be updated by deleting and inserting constraints. New cycles can then be found very efficiently.

We will now discuss the performance gains on unsolvable levels. In section 3 we discussed certain properties we wanted our solver to have. We argued that the solver can be slow on hard levels as long as it is fast on the easier ones. Fast detection of unsolvable levels was very important too. Because the goal-step graph was used by the sub-problem driven approach, some unsolvable levels are detected very fast: if no complete goal-step-plan can be formulated, the level is unsolvable. But in most other cases the unsolvable levels could take up a lot of time.

The constraint clash approach has a few promising characteristics when it comes to detecting unsolvable levels. For one, the identified problems can be tackled in any order and it is obvious whether a problem has been solved, or when it is just shifted; either to another sub-set of mutables, or further along their must-step-plans. This means that a problem can be

tracked down, until it is evident that the problem is resolved. When a single problem cannot be resolved, we do not have to look for possibilities to advance on other problems. Something that is inevitable in the sub-problem approach, as we would then not know if we are trying to solve a different problem. The sub-problem driven approach would first try a lot of useless angles, but because the constraint clash approach uses facts to progress instead of hypothetical situations, it is less likely to get lost in the search space, when a human would not, before it deems a level unsolvable.

5.6.2 Isolating Problems with just the Relevant Mutables

Once the clashes are identified, they must be resolved. When a clash is resolved only the relevant mutables are taken into account, we isolate them from the rest. This was not possible for the sub-problems; there was no information on which mutables actually posed the problem encountered on a stuck branch. Furthermore, in the sub-problem driven approach, one problem can be encountered from many different angles resulting in many different branches getting stuck on the same problem. To gather the same knowledge in the sub-problem driven approach, another analysis should be done to identify individual problems from the set of dead ends that have occurred.

The isolation of the relevant mutables functions as a reliable filter on the potential configurations that can resolve the problem (which was hard to do for the sub-problems), this will speed up the solving process.

Another advantage is that we can distinguish between the problems in size by counting the number of mutables involved: is a problem big or small? We can use that information (as well as the number of constraint clashes) when we predict which constraint graph is closer to a solution, this will speed up the solving process and enhance the evaluation score (another metric that can evaluate if a counter intuitive move has to be made, when the right move seems less obvious than other wrong moves).

5.6.3 Informed Search is Desirable

The biggest advantage of the constraint clash approach, is the extra information we can access about the problems in a level. We can gather reliable information by analyzing what happened to the constraint graph after a problem is resolved: are other problems resolved as well? Are the other problems exchanged for different problems? Did the problem slightly shift to a further state? Did we just wriggle a mutable out of the problem? If so did we also introduce a new mutable into the problem? Etc. This information is very useful for making the decision of with which constraint graph we want to continue, as well as when we estimate how difficult that same choice was for a human being.

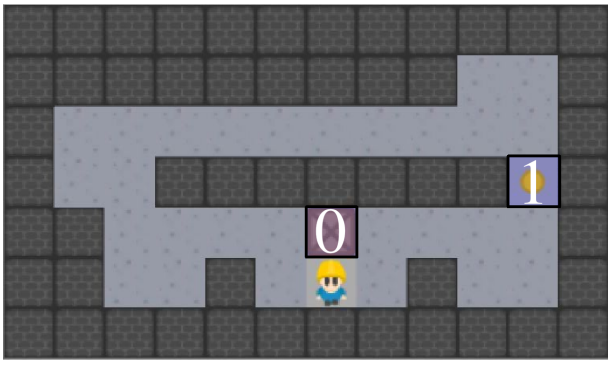
5.7 Shortcomings of the Constraint Clash Approach

Just as with the sub-problem approach, the open spaces form a problem, but for different reasons this time. See Figure 6a displaying a Sokoban level. The must-step-plan of box M has one step: from its current state A to its goal. M has just one step because there are two completely different paths from A towards its goal, so the only state he must be in, is being at the position of its goal. An undetailed must-step-plan like this, leads to an undetailed set of constraints.

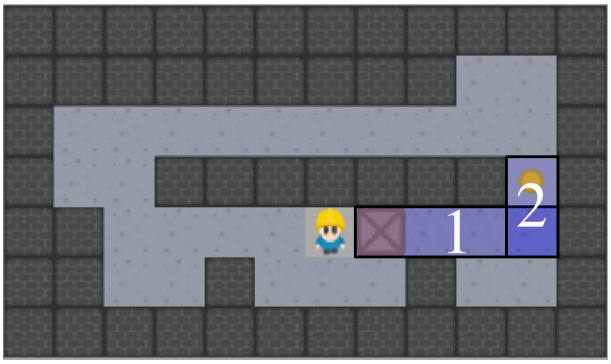
The places the box is in at Figure 6b and Figure 6c have a detailed must-step-plan and thus constraint set, even though they are one distance away from A. The reason for this is of course the choice available at state A, in this particular case the problem will resolve itself as the box at A has to move to a more restricted area right away in order to get to its goal. However, consider a big hallway, mutables could traverse this in many ways on their own. A challenge will not be recognized when the hallway is crowded with mutables, because each individual can formulate a flexible plan resulting in undetectable clashes.

The flexibility of these plans makes constraint clashes rare in levels that give mutables multiple options to reach their goal. So unless all mutables are considered at once, clashes are hard to detect in levels with many mutables, because these often provide multiple options for a single mutable to reach its goal state, while in reality, most of those options are blocked by other mutables. This is a big problem as our expanding heuristic relies highly on information about the problems ahead, if the problems are not identified, the solving process is slowed down until the mutables get to a more restricted area (more towards the goal-state mutables often have less options) in the meanwhile the heuristic is blind and the algorithm will perform more like a brute-force approach.

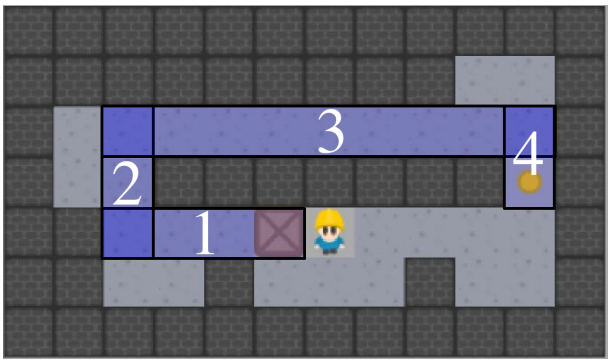
If we take another look at Figure 6 where state A has no meaningful constraint set. We see that upon branching the state of A out into B and C we do get meaningful constraints. As soon as we branch out to a place from which a meaningful set of constraints can be formed, the flexible plan issue is resolved. This leads to the idea of branching in



(a) Flexible path, only one must-step is defined.



(b) Strict path 1.



(c) Strict path 2.

Figure 6: Flexible path example.

constraint sets when a certain part of a mutable's path has multiple options. However, the same levels still impose a problem: when a level has a lot of open space, each mutable will have a big constraint tree/graph and in order to define clashes we must prove that the branches alone are not enough to get all mutables to their goal. This requires going through all combinations of relevant branches for all mutables. The bigger these trees/graphs the more time the computation will take, while often the more open space a level has, the easier they are; this clashes with our desired characteristics explained in subsection 3.3.

When all mutables are considered at once the constraints become less useful: we could also try valid input and see where we end up. However, the flaws of the constraint clash approach point towards considering all mutables and creating a path on the go, instead of analyzing the whole level at once. In the next section we will look at the evaluation approach used in the final product and how the insights gained from the previous two approaches have led to a cleaner solution that works on a broad set of levels and can be built upon in future research.

6 Breaking-Rule Approach

In this section we will take an in depth look at the breaking-rule approach. First an overview is given on how the approach works. We then explain the sophisticated heuristic that is build out of three separate metrics, the heuristic is the core of this approach. Afterwards we explain how each individual metric works and what it adds to the final heuristic. We then explain the pre-processing and post-processing procedures that enhance the evaluation score for a more accurate difficulty label.

6.1 Breaking-Rule Approach Overview

The A* algorithm's search is guided by a sophisticated heuristic. Nodes in the search are states of the mutables' must-step-plan. Each node expansion resembles a mutable switch.

The final evaluation score is calculated with the following formula: $score = E * S$ where E is the number of nodes that are expanded before a solution is found, and S is an estimation of the number of mutable-switches required to find the solution.

We used three different metrics that together form one heuristic that determines which node is expanded. There is a hard ordering in the metrics. To discriminate between two nodes, a less important

metric is only consulted when the more important one scored equally.

The three metrics in order of importance follow:

- **Breaking-Rule Metric**
  The first and most important is the breaking-rule metric (BRM), it is similar to the heuristic introduced in subsection 4.7 which guided the expansion of nodes in the sub-problem driven approach by estimating the number of problems that lay ahead.
- **New-Nodes Metric**
  The new-nodes metric is not trying to predict how distant a node is from a potential solution. Instead it tries to point towards directions that have not been explored yet.
- **Movement-Freedom Metric**
  The movement-freedom metric is trying to predict how much movement is possible from a given node. It is similar to the new-nodes metric, but instead of guiding towards new places, the movement-freedom metric guides the search towards less intertwined states.

After a level is verified to be solvable, the expanded nodes are analyzed in a graph representation (expanded-node graph) to make an estimation of the required mutable-switches to reach a solution. We tried to get more metrics from the expanded-node graph, but these were not useful.

6.2 Expanding Nodes

Throughout the solving phase we explore the level's search space by expanding from one node to several others. Instead of applying the raw player input on the level state, we expand towards all states that are reachable when we interact with one mutable only. So an expanded state might be 50 moves away from the expanding node, but only one mutable will have been interacted with, in those 50 moves.

Each expansion resembles a mutable switch, we continue the expansion from the node we estimate to be the most potential. This estimation is the most important part of this approach and determines the level evaluation, as the number of node expansions greatly impacts the final evaluation score.

6.3 Breaking-Rule Metric

The most important metric is the breaking-rule metric (BRM). The other metrics are only consulted when the BRM score is scored equally. When we calculate the BRM for node X, we will first calculate all the must-step-plans and then set up the must-step-cost graph. This is similar as the goal-step-cost graph, except the weights are not always 1 when a step cannot be taken. Instead the weights are determined by the cost of the puzzle rules that are broken. The cost of rules follow a hard ordering. A rule that is broken for elements that are closer related to the puzzle goal are of a higher order. In the case of Sokoban, the boxes are closer related to the puzzle goal, compared to the worker. Moving the worker X times through a box is thus always cheaper than moving a box through another box.

With an exhaustive search through the must-step-cost graph we find the cheapest path towards the goal-state. The cost of this path is the final BRM score.

A level is deemed solved once the BRM score of a node is 0, as no rules have to be broken to reach a goal-state, even though this node might be several moves (and mutable switches) away from the actual goal-state.

We have experimented with fusing the mutables-switched metric into the BRM score: keep track of the mutable that has been interacted with last; when this is not the same as the one that is interacted with right now, add additional cost to this move. The mutables-switched cost is of a lower order than the cost of breaking any single rule, so we only discriminate between paths that have the same rule breaking cost. The expanded nodes were reduced on many levels and often to 50% of the original expansions or lower. However, we did cut this addition as the expanded nodes on the harder levels is often reduced too much in comparison with other levels, thus labeling these hard levels much worse than without the addition, since the final evaluation score is highly dependent on the number of expanded nodes.

6.4 New-Nodes Metric

The new-nodes metric's main purpose is to enhance the diversity of the search for a solution. It is less important than the BRM. The new-nodes metric itself does not direct the solver towards a solution.

The new-nodes metric identifies which nodes can reach more unseen nodes than others. A node N_r is considered reachable by node N_e, if N_r is generated when N_e gets expanded. A node N_u is considered unseen when it has never been expanded towards.

In cases where the BRM does well and nodes receive a big diversity in BRM scores, the new-nodes metric is not as important. However, the BRM often scores equal on nodes, as the number of rules that must be broken to reach the goal-state in the must-step graph, is often below 5 throughout the whole

solving process. This is where the new-nodes metric comes in. It prevents the solver from behaving like a BFS when the BRM scores are equal. Instead the solver's attention is pointed to places that have not been expanded much, for a better diversity of the expanded nodes, until a leap forwards in BRM score takes over again.

The new-nodes score is dependent on the nodes that have been expanded so far. When a node A gets elected as new best node, others have probably been expanded while A was queued. This means that the new-nodes score A received when it got enqueued, could be outdated. We thus recalculate the new-nodes score and A is only expanded when the final score is equal to the old one. We cannot underestimate the number of new nodes, so there is no need to update all the queued nodes every expansion to ensure we find the best one.

6.5 Movement-Freedom Metric

The least important metric is the movement-freedom metric. This metric pre-expands the given node and counts how many places are reachable with one mutable-switch. A higher score is better. The relative movement-freedom between states gives an indication of how free the mutables are to move around the level. We want to move towards states with a lot of movement freedom to make sure we do not miss out on promising paths. The movement-freedom metric mostly serves the new-nodes metric. It tries to find states that provide a lot of movement possibilities.

The new-nodes metric evaluates the relative number of new nodes a node will generate upon expansion. It pushes the search towards new places, while the movement-freedom metric evaluates the absolute number of paths possible from a certain node and pushes the search towards states in which the mutables are less intertwined.

Many transport puzzles cannot be reversed so the freedom of mutables can be very low in a solved state. To compensate for this, we assign the maximum movability score (dependent on the level's grid-size) to a mutable that is already at its goal state. This will guide the solver to bring single mutables to their goal state which likely prevents other mutables from ever reaching their goal state (at least in most interesting puzzle levels). However, this actually improves the difficulty evaluation, since the cases in which this naive solving strategy does work, the solution is found earlier on and will result in a lower difficulty score which is desirable.

6.6 Metric Influence Comparison

We started developing the new-nodes and movement-freedom metrics after we noticed that generating levels with only the BRM as heuristic resulted in some easy levels with too high evaluation scores. We selected a few easy levels that received high evaluation scores to test our new metrics on while developing them. For this comparison we searched through old levels generated with only the BRM and selected four levels we felt were easy, without looking at the scores they received and these levels were not used during the development of the new-nodes and movement-freedom metrics.

The main purpose of the new-nodes and movement-freedom metrics is to reduce the expanded nodes of easy levels, occasionally reducing the expanded nodes on hard levels is less important, as we have stated before in section 3.3. The comparison results are shown in Table 3. The levels are ordered on difficulty by our two experts. Except for the last two levels that have our experts have not (tried to) solve, instead we based the order on the findings of [21]. Furthermore, for the first 4 levels no meaningful order is agreed upon, they are all deemed very easy. These are the levels we selected from the generated levels with only the BRM as heuristic. The levels of [21] are all hand-crafted, the accompanying times are median human solving times [21] gathered from online playing data. The numbers shown are all final evaluation scores.

From the results we can see that adding the BRM is clearly better, mostly in the easy levels. When only the BRM is used we see that the 'hardest' level scores even lower, and that two of the four easy levels score much too high. Not using the new-nodes metric results in an equal or higher number of expanded nodes for all levels except the two 'hardest' levels. Overall we see that the levels "hand-crafted 5" and " [21] 53 minutes" score too low for their order of difficulty with every tried metric combination.

6.7 Pre-Processing: Naive Solving Filter

In the section on the sub-problem driven approach, we discussed optimizing the solving process by filtering unpromising nodes. We did not put this optimization into work because it made the solver label many levels as unsolvable when they were not. However, for the breaking-rule approach we did end up using a similar filtering process to identify easy levels early on, with a cheap operation. Levels that where still solvable, even after we would filter out many nodes and only try the most naive options, were of-

Table 3: This table shows a performance comparison of the metric combinations. The levels are from various sources, see the appendix, and [21].

Level name	All metrics	Only BRM	No new-nodes	No movement-freedom	No BRM
Only BRM gen 1	6	6	6	6	528
Only BRM gen 2	25	85	25	35	150
Only BRM gen 3	6	423	6	9	132
Only BRM gen 4	126	725	126	156	540
Hand-crafted 1	104	42	136	88	144
Hand-crafted 2	28	108	32	28	104
[21] 3 minutes	128	852	1296	66	1734
Hand-crafted 3	200	192	240	184	240
Hand-crafted 4	370	905	830	425	1120
Hand-crafted 5	175	585	175	180	750
Hand-crafted 6	1155	1520	1254	1287	2508
[21] 43 minutes	7680	12256	13070	8570	22140
[21] 49 minutes	16226	12796	13776	15988	32424
[21] 53 minutes	2562	432	2112	2160	6846

ten lacking any interesting characteristics and thus deemed easy. When a new level is tried we first try to solve it in this naive way, before applying the full solving algorithm.

This is most useful at the start of the generation process as most levels further in the process were either unsolvable or decently hard. In both cases the naive solver would not find a solution and the full solving algorithm had to be applied anyway.

6.8 Post-Processing: the Expanded-Node Graph

After the solving process, we analyze the level structure with the expanded-node graph. The expanded-node graph is an undirected graph that is set up during the solving process. Each node that gets expanded, is also a node in the expanded-node graph. When we expand a node we add an edge from this node to each node it expands towards. So we build up a graph representation of the level's search space, specific for the nodes that have been expanded during the solving process.

In our prototype Sokoban level generator, we evaluated levels by the minimal number of box switches required to solve it. This worked decently well and the idea is backed by a number of previous papers [5] [30]. Therefore, we wanted to have the final difficulty evaluation to be dependent on the mutable-switches as well. Especially since the heuristic already deals with the biggest pitfalls of the mutables-switched metric: the staircase example shown in Figure 3 requires 14 mutable switches (the highest found for two boxes), but the solution is found in 14 node expansions (which is very low), so the final score would be $14 * 14 = 196$ which indicates easy for Sokoban levels, since the staircase example has a linear search space a low score is desired. Furthermore the preprocessing described in the previous subsection will filter this level out, because the solution is too naive.

To actually calculate the mutables-switched metric, we do a quick BFS on the expanded-node graph (remember a node expansion is equal to a mutable switch), the BFS is quick as the number of expanded nodes has not yet been over 3000. This calculation gives a good estimation of the minimal mutable-switches necessary to solve a level. Although it almost always estimates too low, because a node with a BRM score of 0, can still be some trivial mutable-switches away from the goal-state. This is fortunate however, as we do not want these switches to increase the evaluation score.

The search space of a level is often too big to exhaustively search it, so naturally we often do not find every possible solution. The shortest path through the expanded-node graph, therefore, does not always represent the minimal number of mutable switches required to solve the level (even with the trivial mutable-switches at the end). To minimize this problem, we expand all the nodes that are queued when we find a solution. We do not calculate the heuristic scores of these nodes as we just want a more accurate expanded-node graph, this keeps it a cheap operation.

Apart from the mutables-switched estimation, none of the analysis from the expanded-node graph was directly useful. We looked at the ratio of nodes that lead to a solution. We calculated the must-visit

nodes to identify the crux of the level, however the crux was often still hidden because of the way we expand nodes (one mutable can traverse the whole level in one expansion if there is room), this meant that many must-visit nodes were skipped, as jumps can be made between nodes even when the nodes in between have to be visited to reach a solution.

We also analyzed how hard it was for the solver to solve the level from one must-visit node to the other, however this was almost always trivially solved in one expansion, even though not all must-visit nodes were present. Only the extremely difficult levels had no trivially reachable must-visit nodes. Lastly we analyzed how often counter intuitive moves were needed in terms of the BRM score of nodes. Again this almost never occurred. There was almost always a path from start to goal that had nodes with equal or a lower BRM values than the previous nodes, thus traversing the level this way was not considered counterintuitive in terms of BRM score.

6.9 Why the Breaking-Rule Approach Performed Better

The breaking-rule approach has a similar problem as the constraint clash approach: when a mutable has multiple options towards its goal, its must-step-plan will be less useful. In the constraint clash approach this meant a problem was not identified. In the must-step graph however, it is explicitly checked whether the step can be taken or not (all mutables are taken into account), which means a problem will always be identified. Furthermore, there is a much higher chance that an undetailed must-step-plan causes problems in the traversal of the must-step graph, since the steps the mutable has to take are bigger. This means that an undetailed must-step-plan can only overestimate the problems ahead and is likely to do so. Thus the solving algorithm will progress towards places where its mutables have more detailed must-step-plans. This is desirable as the BRM is more reliable in these situations.

A node with a lower BRM score is almost always better, but during the search we also see a lot of nodes with the same BRM score. The new-nodes metric is good at enhancing the search and provides a balance in depth and breadth first search which was a big problem in the sub-problem driven approach. Also, expanding a node as if one mutable-switch happened enhances the balance in depth and breadth first search, whilst better mimicking human solving characteristics, as the solving algorithm now progresses in terms of mutable-switches.

This results in the number of expanded nodes giving a good impression of the levels search space. That combined with the number of mutable-switches that measures how complex the actual solution is, gives a good indication of level difficulty. Not counting the last few trivial mutable switches improves upon this indication.

The breaking-rule approach can evaluate a much larger range of levels making the method feasible for generating hard levels on a standard computer of 2016.

7 Generic Rule API

7.1 Implemented Puzzle Rules

We started generating test levels with our Sokoban box-switch evaluator. For every improvement of the evaluator we kept in mind what restriction it forced on the supported puzzle rules. The actual transformation towards the generic evaluator was done after the BRM gave decent results for Sokoban.

The puzzles implemented consist of Sokoban, and four variations on Sokoban:

- **Pull Mechanics:**
  In the original Sokoban the worker can only push boxes. In the pull mechanics variation, the worker cannot push any box, but will always pull a box along. When there is a box on the grid-point opposite of the direction where the worker is moving toward, this box will be moved to the grid-point the worker was moving from. This is different from reversed Sokoban rules as the worker may not choose whether to pull a box or not, it is as if the box is glued to the worker and the worker can only stop pulling a box by moving perpendicular to it.
  Behavior: we found the generated levels for this puzzle were the hardest to solve (for humans). This is probably due too the extra planning layer introduced by the mandatory interaction with boxes, as a worker cannot necessarily traverse through empty spaces without consequences, which is hard to see for humans. This is also an additional rule that may be broken, the evaluator is thus likely to work better as the heuristic will be more nuanced during the search, which in turn means that easy levels are solved with fewer node expansions.

- **Strong Worker:**
  In the original Sokoban the worker can only push one box at a time. A box cannot be pushed towards a grid-point that is already occupied by another box. The strong worker can move as many boxes as it likes at once, as long as

no box ends up in a wall. Essentially this is a relaxed version of Sokoban, since each feasible Sokoban level is feasible with the strong worker rules as well. We tried evaluating a few very hard Sokoban levels, and all of them became easy with the strong worker rules.
Behavior: these rules allow for a valid Sokoban level to be generated, but this never happened. For every generated level multiple boxes have to be pushed at once in order to reach a solution.

- **Big Boxes:**
  In the original Sokoban the boxes occupy only one grid-point. In the big box variation each box occupies four grid-points.
  Behavior: these rules naturally needed bigger levels for the same number of boxes, compared to the other variations. The levels feel quite different from conventional Sokoban levels as the boxes can partially block each other and seem to get stuck easier.

- **Swap Mechanics:**
  In the original Sokoban the worker can only interact with a box if this box is touching the worker. In the swap mechanics variation, the worker swaps positions with the nearest box in its moving direction. If the nearest box is behind a wall or when there is no box in the direction the worker moved towards, the worker will move one grid-point just as in Sokoban. Similar to the pull mechanics, the interaction with a box cannot be omitted by the player, when the worker moves towards a box in sight he will swap positions, always.
  Behavior: these rules had the most altering effect on the generation and evaluation score. The ILS generator could barely find non-trivial levels with the conventional parameter settings. After increasing iterations/generations, neighbors tried per iteration, and the number of mutables, good quality levels did emerge. What is interesting though, is the extremely low evaluation scores each swap mechanic level would receive. A level that scored 40 would be considered medium to hard for the swap mechanics while similar difficulty would receive a score of ± 1000 for the other puzzle rules. The generation process is also much quicker with the same parameter settings compared to the other rulesets.

7.2 Transforming the Sokoban Generator to a 'Generic' Generator

As mentioned before, the final product is not generic. It is not even generic for all transportation puzzles. However, the previous subsection provides us with evidence that the methods are generic to a certain degree. We want to add to this in this subsection by explaining the transformation process of the Sokoban generator to the 'generic' generator we have now.

While making the program generic, and thus work for each Sokoban variation, it was never a question of: how do we remain the functionality for the other puzzles. Rather the question of: how to make sure the evaluator uses the new puzzle rules in a meaningful way. If we wanted to do new things in our puzzles, we often had to change the ruleAPI. For instance, before the big boxes variation we had only mutables that fit on one grid-point, however, allowing mutables to occupy multiple grid-points in the ruleAPI did not change anything for the other puzzle rules. Another example is the swap mechanics. For the swap mechanics to make sense in the evaluator, the ruleAPI needed to allow an interaction with mutables at an arbitrary distance. This was not possible yet, but the change in the ruleAPI did not affect the implementations of the other puzzle rules at all.

We think this progression throughout the transformation is very promising. Never did we have to change the definition of a puzzle once it was implemented, even though the ruleAPI kept on changing as we added new possibilities for puzzle rules. The generated levels from each variation feel very different from those of the other puzzles. Still the evaluation function seems to work well on all four variations without applying any tweaks to it. This is achieved even though, during its development, it was tested on the original Sokoban puzzle only.

All these points are promising signs for the evaluation function to be generic in its core.

7.3 A Generic Must-Step-Plan

We think that the core systems are generic for transportation puzzles, although adjusting the program will require a decent amount of work. At the core of the program is the BRM. For the BRM to work, we need the must-step-cost graph and for that we need a must-step-plan for each goal-related mutable.

A big chunk of generality is lost because the must-step-plan in its current state assumes that mutables can reach their goal states without the help of other mutables. This cuts off a big variety of puzzles, for instance, in 'Jelly no puzzle' (see Figure 7) the goal state is reached once all elements of the same color

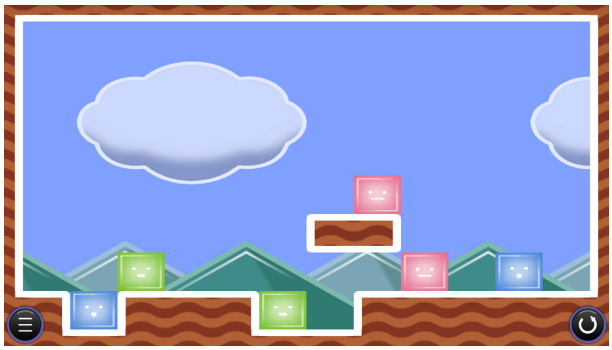
touch each other. So if we take one piece, it will always be at its goal state. For this particular example the goal state's dependence on all similar colored mutables, can easily be derived from the puzzle rules. So the program could calculate a must-step-plan for each independent color. This is indeed a valid method for this puzzle. However, when we look at another puzzle 'Snake Bird' (see Figure 8 for the rules) we have a similar case of needing other mutables to reach our goal (snakes can be lifted by other snakes). This time the single elements cannot be grouped in a meaningful way (all snakes in the level, which leaves the abstraction to be equal to the original level).

There are other options, these have however, not been tested at all. Consider a single Snake in 'Snake Bird', in order for this snake to reach its goal it often needs another snake to climb on. If we implement 'Snake Bird' as a puzzle in the current program, it will say these levels are not solvable, because the must-step-plans cannot be formulated when the goal-state cannot be reached by the Snake. However, when we allow for individual mutables to break rules for their must-step-plans as well, the must-step-plans can be formulated and the levels can be solved (as long as the right time and memory is at hand).

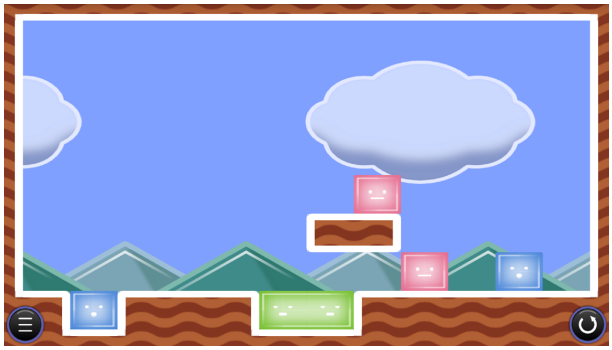
What kind of rules can an individual mutable break to find its goal, in a way that the must-step-plan still makes sense? We could restrict when rules may be broken such that the cheating path a mutable takes is theoretically possible. In the case of 'Snake Bird' this would mean that a snake A might magically move upwards, but only the number of grid-points that would be feasible when another snake B (that is present in the level) is supporting snake A. This might sound like a very specific solution, but it comes down to considering the other mutables when the must-step-plans are defined, and deducing additional properties from the puzzle rules on how they interact with each other. These additions, would not affect the way Sokoban boxes define their must-step-plan, as the interaction-rules for boxes only limit their movement and they will thus not be able to help each other. Furthermore, in the case of 'Jelly no puzzle' only mutables with the same color will be considered in their must-step-plans as here too, the others will just limit the movement.

7.4 What is Needed to Start Generating Levels for a new Puzzle?

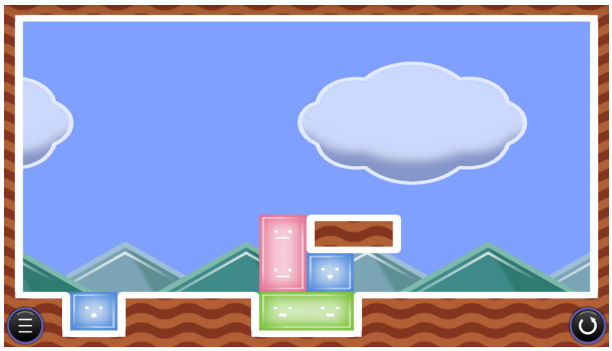
An ideal implementation of a new puzzle rule-set would be with the use of a simple scripting lan-



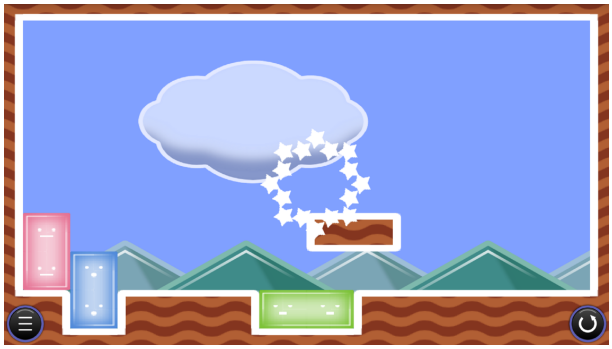
(a) You can move jellies to the left and right.



(b) When jellies touch, they are fused together.



(c) Jellies are affected by gravity.



(d) The goal state is reached when all jellies of the same color are fused together (jellies of a different color cannot be fused).

Figure 7: Jelly no Puzzle, overview of the rules.

(a) Snakes can sit on-top of each other.

(b) The up key is pressed once. Snakes move like in the original snake action game: the head goes to the grid point the snake is moving towards, the last tail part is removed.

(c) The player can switch snakes at any time.

(d) The goal state is reached when all snakes went through the rainbow/snake-hole.

Figure 8: Snake Bird, overview of the rules.

guage like, for instance, PuzzleScript[2], which is a video-game description language created by Stephen Lavelle designed to easily create transportation puzzle games. However, we knew that the time frame did not allow for all the extra components to be finished and polished by the end of the project. So we decided that making the ideal ruleAPI component was not important. There is no question of 'can it be implemented' as [2] deduces similar information as we would need from PuzzleScript scripts for their generic puzzle level generator for PuzzleScript games. Furthermore, this should be done after the solving, evaluation, and generation components are finished, since these determine what information needs to be analyzed from the rule-set. Finishing these components is already out of this thesis's scope, as the focus is on level evaluation.

To start generating levels for a new puzzle, we need a starting level for the search method to work, and implement a list of functions for the evaluator to work. As starting level we use an almost empty level as displayed in Figure 9. For the pull mechanics this level is unsolvable. This slows down the generation, but mostly just for one iteration of the ILS, because as soon as a solvable level is found the search continues with this level.

A complete list of the functions that must be implemented in order to start generating levels for a new puzzle follows:

- **PuzzleName:**
  Input: nothing.
  Functionality: this function is just a String, representing the name of the puzzle. The generator uses this to name new levels and puts it in the level's meta data.
- **WorkerMoveStaticConditions:**
  Input: current level state and player-input.
  Functionality: this function checks if the player-input provides a valid move considering the worker and all static elements in the level. For our puzzles this comes down to all the elements except the boxes.
- **WorkerMoveConditions:**
  Input: current level state and player-input.
  Functionality: this function checks if the player-input provides a valid move considering all the elements in the level.
- **MutableMoveStaticConditions:**
  Input: current level state, player-input and one mutable M.

[2] `www.puzzlescript.net`

Functionality: this function checks if the player-input provides a valid move wherein mutable M is modified if we consider the worker, the mutable M, and all static elements in the level. For our puzzles this comes down to all the elements, without the other boxes, and including the supplied box (M).

- **MutableMoveConditions:**
  Input: current level state, player-input and one mutable M.
  Functionality: this function checks if the player-input provides a valid move wherein mutable M is modified considering all the elements in the level.
- **MutableTriggered:**
  Input: current level state, player-input and one mutable M.
  Functionality: this function checks if the player-input would result in a move wherein mutable M is modified.
- **WorkerPreMutableRule:**
  Input: player-input and one mutable M.
  Functionality: this function returns a position for the worker in which the MutableTriggerd function would return 'True' for mutable M.
- **DoMutableMove:**
  Input: current level state, player-input and one mutable M.
  Functionality: this function processes the player-input and mutable M and returns the modified level along with the modified mutable. The modified level contains the modified mutable as well, but it is returned separately also. The function does not check whether the player-input is valid, nor if it would result in the mutable being modified. These conditions are assumed to be true.
- **DoWorkerMove:**
  Input: current level state and player-input.
  Functionality: This function processes the player-input and returns the modified level. The function does not check whether the player-input results in a valid move.

7.5 Rule Analyzer to Avoid Writing Code when a new Puzzle is Introduced

The programming of the functions from the previous subsection is trivial and does not cost a lot of time. However, if we would deduce the implementation of these functions from a valid script of a PuzzleScript puzzle, anybody can come up with new puzzles and generate levels for their own ideas.

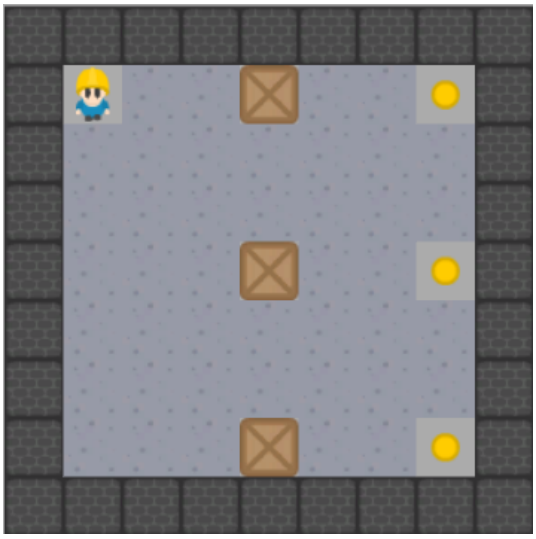


Figure 9: A basic level used to start generation. This basic level can be used for all the implemented puzzles. The number of mutables stays the same throughout the generation as does the level size.

The deductions needed at this point are very minor. We have to know which elements are mutables/immutables. How and when the player-input changes the mutables, which mutables are goal related, and when they are at their goal state.

All this information would provided in a valid PuzzleScript script, so the implementation of the ruleAPI could be deduced from such a script. This can be achieved with simple pattern matching: elements that never change after player-input are immutables, the others are mutable. Furthermore, queries like these: 'in the current state, is mutable A changed after player input I?', and: 'is mutable A currently at its goal state?' are easy to calculate and will provide the information we need for the rest of the breaking-rule approach to work.

When many different types of mutable elements are present in the puzzle rules, we also have to define an order of importance related with the goal-state, this is needed to properly define the cost order for breaking rules.

This functionality is not implemented in the final program as we have explained in the previous subsection: given the size of our scope we did not deem making the ideal ruleAPI as important. More important are things like algorithm characteristics that provide us with the potential for creating this ideal ruleAPI.

8 Experiment and Results

We did a small experiment to test if our generated levels were of high quality and whether their difficulty label was accurate.

We will explain how the experiment is set up, state the hypothesis we had for the results, interpret the results, and compare results of previous research with ours.

8.1 Setup and Execution

The focus of this research lies on the evaluation component of a generic puzzle level generator. We must thus test the evaluator for multiple puzzles, in order to get meaningful data. The evaluation scores represent an ordering of levels from the same puzzle. So we must also test multiple levels per puzzle to get meaningful data. However, we could not require our participants to solve puzzles for over 3 hours and have to be careful with the (random) selection of our test levels to get a high ratio of quality data versus required time per participant. We decided to test 3 levels per puzzle, for 3 different puzzles.

During the experiment the participant is faced with the 3 levels of one puzzle type at a time. The levels are presented next to each other horizontally, in a random order (different for each participant). The participants have to solve the levels and put them in an ordering on perceived difficulty, before they can proceed to the next puzzle type. We log every move they make so we can split the data afterwards and see the amount of time that was spent on each level separately, as well as the down time during the solving sessions. A down time of more than a minute is not counted as solving time, nor is the time spent replaying a level after the solution has been reached.

While the levels are presented the following text is displayed above the levels: "We encourage you to try and solve the level that seems easiest first. Furthermore, it is allowed to switch between levels at any time, so feel free to mess around".

To enhance the random selection of test levels, we picked a few levels from the level pool and played them (afterwards removing them so they could not be selected as test levels). We did this to define an appropriate distance in evaluation for which the levels should have a noticeable difference in difficulty (as the evaluation numbers are not automatically normalized to perceived difficulty).

We ended up iterating once on the evaluation rating of the levels, as during the first try, people were having a hard time solving levels at all. Of 5 participants 4 did not manage to solve any level other than the tutorial (which is solved in less than a minute), while all played over half an hour and most a full hour.

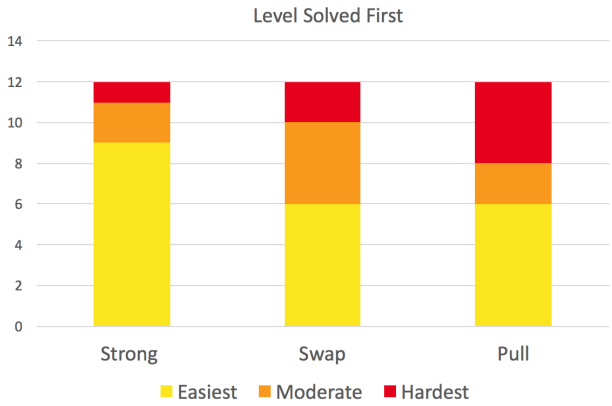


Figure 10: This column diagram shows which level is solved first per puzzle type.

We decided that easier levels should be picked in order to get meaningful data from a small group of participants. We generated 30 new levels for each puzzle type, this took 8 hours on a Macbook Pro from 2015 with the standard i5 CPU, running three processes. Over 30% of the levels received a higher score (multiple even double) than the highest level in the experiment for that puzzle type (except for the swap mechanics, with which the generator has the most trouble), this indicates that during those 7 hours many high quality levels were found.

8.2 Hypothesis

Our three hypothesis follow:

- The easy levels are likely to be solved first, even though all levels are presented together, as the big difference in evaluation score will become apparent by trying out the levels.
- Participants will spend more time on levels with a higher evaluation score.
- Participants will label a level which receives a higher evaluation score, to be harder.

8.3 Results

We have results from 12 participants.

In this subsection we will interpret the data. Since the experiment was performed by only 12 people, the significance of the results is questionable. However, good correlations are scored with the Spearman correlation coefficient.

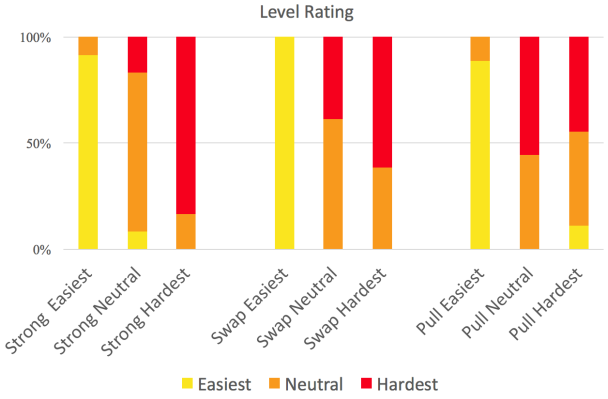


Figure 11: This column diagram shows the given difficulty ratings per level.

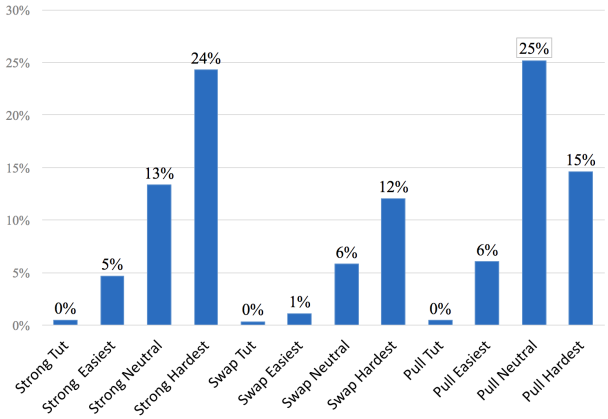


Figure 12: This column diagram shows the average time spent on solving each level in percentages per player.

8.3.1 First Solved Levels

Our first hypothesis stated that the easier levels are more likely to be solved first. When we look at the data in Figure 10 we see that for every puzzle type 50% or more of the participants solved the level with the lowest evaluation score first. Although for the swap and pull mechanics 50% also solved one of the other two levels first, this still indicates that the lowest scoring levels were in fact the easiest.

8.3.2 Time Spent on Levels

The second hypothesis stated that the time spent on each level would correlate with the evaluation score of the levels. This is the only hypothesis for which we can include the tutorial levels, each of the tutorial levels received an evaluation score of zero.

The strong worker and swap mechanics both show a high correlation by the Spearman correlation coefficient 0.88 and 0.76 respectively. The pull mechanics however, shows a much lower correlation of 0.62. This is due to the medium scoring level which received a score of 600 compared to 125 and 1458 of the lowest and highest scoring levels respectively, while people on average spent a factor of 1.45 more time on the medium scoring level versus the highest scoring level. When we switch the evaluation order of these two levels, we get a correlation of 0.77 for the time spent and evaluation order of the pull levels. This indicates the evaluation of 600 is too low.

8.3.3 Player Difficulty Rating

The third hypothesis stated that the order of evaluation score would correlate with the difficulty label the participants would rate the levels. For the strong worker mechanics the Spearman correlation coefficient is 0.85 which is a strong positive correlation. The ratings of the swap mechanics has a less strong positive correlation of 0.81. The pull mechanics received a slight positive correlation of 0.61, and when we change the results from the highest and medium evaluated levels we get a correlation of 0.72. This indicates that the order of the highest and medium labeled levels for the pull mechanics was not correct.

8.4 Comparing the Results of other Work

In this section we will compare the levels of other research with our own. We only look at Sokoban levels of other researchers, as the only well known transportation puzzle we implemented is Sokoban.

The work of Murase et al. [29] is from 1996, thus their level results are likely of less quality due to the limitation of computation power compared to today,

Table 4: Evaluation of the four 'good' problems according to the generator of [29] and their human experts (in our perceived difficulty order lower is easier).

Level name	Evaluation score	Mutable-switches	Node expansions	Our perceived difficulty order
Problem 1	42	6	7	4
Problem 2	16	4	4	2
Problem 3	322	6	46	3
Problem 4	12	3	4	1

twenty years later. However, we can still evaluate their levels.

They generated 44 levels the generator considered 'good' (levels are only evaluated by this 'good' threshold, no further ordering is attempted) and of those, 14 were also considered good by Sokoban experts. 4 of those 14 are given at the end of the paper, they all contain 3 boxes. We evaluated them with our method, the highest received a score of 322 and the rest a score of less than 100 (see Table 4). The levels are all solved in under two minutes by two experts of our own, both deeming one level intriguing as counterintuitive moves are needed, however the search space of the level is too small and thus the level is solved quickly. This is the first level of Table 4, it received a very low rating even though it is perceived the most interesting of the four, it would be better if the evaluator labeled this highest of these four levels. However, all these levels are solved too quickly to our liking which means the low evaluation score of these levels in general is a good result.

Their generation method is not easily made generic, as they make use of hand-crafted templates specific for Sokoban.

Taylor and Parberry [5] also use hand-crafted templates in their Sokoban generator. We think these level are of higher quality than those of [29]. The levels are evaluated by how far their starting position is away from any solution. This distance is calculated by a metric they call box-lines metric. It counts the times the worker changes the direction in which a box is pushed. The authors themselves state that the box-switches would probably be a better metric, however, implementation difficulty is deemed as an obstacle. We also think that box-switches is better than box-lines, because it favors a level with more pushing in circles instead of a more intertwined solution, as changing a box count the same as one direction switch. The box-lines metric has the same problems as the box-switches metric (see Figure 3), and a few additional problems of its own.

Table 5: Evaluation of the first (top-left to right-bottom) 8 problems presented by [5]. The levels are in order of difficulty labeled by one of experts.

Level name	Evaluation score	Node expansions	Mutable-switches	Expert time
level 5	2	1	2	0:11
level 6	20	5	4	1:53
level 8	112	28	4	0:18
level 7	130	26	5	0:55
level 1	165	33	5	0:51
level 3	1260	126	10	2:16
level 4	600	100	6	3:54
level 2	749	107	7	11:29

16 levels are presented in the appendix with the caption: "Some levels of varying difficulty created by our generator.", one of our experts has played and ordered the levels on difficulty. In Table 5 the evaluation of the levels is compared with the order of our expert, along with the time our expert spent on the levels.

The time spent on level 6 is very high compared to its difficulty order, this is because the level requires one specific move in the beginning and is trivial afterwards. However, the level requires a lot of moves that our expert had to reverse and do over.

Level 3 receives a too high score to our liking, it is not very difficult. The solution involves a lot of mutable-switches, but they were perceived as intuitive, we are thus disappointed with the relatively high number of node expansions needed to find the solution.

Furthermore, we deem level 2 to be underrated, it is a tricky level that requires expertise. It is by far the best level of this set, a score of ± 1200 would have been more appropriate for this level. Overall we think these results are good, as the order seems to correlate with that of our expert and most easy levels get a low rating. Also, the levels of [5] are the best generated levels of those we have tested, however, apart from level 2 they are still fairly easy, and not of the extreme difficulty that we want from our levels.

Jarušed and Pelánek [21] [22] [23] present an evaluation method for Sokoban using a human solving model. For the model to work, the full search space must first be labeled with the number of moves each state is away from a solution. This makes the evaluation method infeasible for generic level generation of transportation puzzles as of today, since several levels must be evaluated if not hundreds, labeling the whole search space for these PSPACE Complete

Table 6: Evaluation of the four problems presented in [21].

Level size	Median solving time in minutes	Evaluation score	Mutable-switches	Node expansions
9x6	3	128	8	16
6x6	54	2562	14	183
8x6	43	7680	10	768
7x7	49	16226	14	1159

problems will not be feasible.

The data used in this study is gathered from 785 hours of play, in the paper 4 levels are presented along with their median solving time; only the times of players that have solved the level are counted. We evaluated these levels with our method. The level with a median solving time of 3 minutes received an score of 128, the others have a median solving time of over 40 minutes and are evaluated in the thousands (see Table T4).

Many papers on puzzle level generation or difficulty evaluation claim their methods are generic. Three papers [2], [3] and [14] actually test their methods on multiple puzzles. [3] evolves puzzle rules within given levels and does not generate any new levels, their results[3] show only easy levels. [14] does not work on transportation puzzles, they focus on Sudoku like puzzles.

Khalifa and Fayek [2] present a generic puzzle level generator for transportation puzzles and has the same goal as we have, except we choose to focus on the evaluation function as we believed this was the most important part for good results. Their levels are playable here: `http://amidos-games.com/puzzlescript-pcg/`. Khalifa and Fayek managed to create a generator that 'works' on all transportation puzzles were ours does not (but might with the enhancements of the must-step-plan described in the previous section). However, they test their method only on puzzles our generator can generate for in its current state as well.

The levels that are generated are of much worse quality than the levels generated when we only used the mutable-switched metric for evaluation. The mutable-switched metric is generic for all transportation puzzles and easily deducible from the puzzle rules, which makes it a generic evaluation method.

Khalifa and Fayek state that the: "Heuristic measures ensure that the level's solution is challenging". However, our two experts have tested over 40 levels, *all* of them are solved in less then half a minute. Furthermore, our evaluator would give a score of zero to more than 40% of those levels, as the starting must-step graph can reach the goal-state. We did not evaluate the others, as it was apparent these would need less than 10 node expansions by our solver, and are thus deemed trivial. Lastly, many levels are present twice in a level set.

Although their work is interesting and probably the first generic level generator for transportation puzzles, the actual generation results are less than mediocre and not likely to engage people, even though they "ensure that the level's solution is challenging".

[3] `http://imgur.com/a/AkiMv`

9 Future Work

Throughout the thesis we have introduced many ideas for improvement on existing methods, as well as why these ideas are worth looking into. In this section we group the improvement ideas into categories and give a summary on what is most urgent in each category.

- **Broader Experiment:**
  Designing an experiment for a much higher number of participants is desired. We cannot require multiple hours of our participants (at least not from all). So we need more in order to test the evaluation score on a broader range of levels. We also need to test a broader range of levels for a more accurate experiment, as well as more data per level to test our hypothesis with proper statistics.

- **More Generic:**
  The evaluation method does not work when mutables need to help each other in order to reach their goals, because the must-step-plans cannot be formulated. With an extension of the implementation, such that a sensible must-step-plan can be formulated for this case, the method will be applicable for a very broad range of transportation puzzles.

  We can achieve this when we reason about the other mutables in a level, whilst formulating the must-step-plan.

- **More Thorough Analysis/Post-Processing:**
  The expanded-node graph is now only used to estimate the number of mutable switches required to solve a level. If the analysis is done more thoroughly, many other insights on the search space of a level can be estimated. The evaluation method can be improved further with this information.

- **Feasible Informed Search with the working Heuristic:**
  The sub-problem driven approach and the constraint clash approach both show a lot of potential for an accurate evaluation, because of their informed solving characteristics.

  In the final product we have stripped away most of these characteristics to optimize the breaking-rule heuristic in an isolated environment. To utilize the potential of this heuristic to its fullest, we will have to create an informed search method, without requiring infeasible amounts of memory or time.

- **Improved Level Search over the ILS:**
  This thesis is focused on the evaluation method. Generating puzzle levels also requires algorithms that find fit levels. The simple ILS works well enough for demonstration purposes, using more sophisticated meta-heuristics in the search method has the potential to find more quality levels in less time. Also, to generate the most interesting level set, an adapting search method which looks at the already generated levels is desired.

10 Conclusion

We presented three new methods for evaluating puzzle levels. The methods are made with the intention to work well on all transportation puzzles. The breaking-rule approach worked best. We have generated levels with the breaking-rule approach as our evaluator for 5 transportation puzzles to verify the method is indeed not specific to, for instance, Sokoban. However, the must-step-plan in its current state does not work on transportation puzzles with mutables that can support each other to their goal. We do have a plan thought out to address this problem, whether this plan works well in practice still has to be seen.

We can reliably generate several very hard levels using a simple ILS generator with the breaking-rule approach as evaluator, in only a few hours on modern computers. Thereby generating faster than our experts can solve the levels.

We tested the generated Sokoban levels of other researchers and evaluated them with the breaking-rule approach. Most of the previous work generates very poor quality levels except for [5]. They however, make use of a worse metric than the mutables-switched metric. They state so themselves, reasoning the box/mutable-switched metric is harder to implement, while we have shown our method tackles the biggest problems of the mutables-switched metric.

We thus believe our generator generates levels of higher quality than any other Sokoban like generator.

References

[1] J. Postma, "Literature study on generic puzzle level generation for deterministic puzzles," Master's thesis, Utrecht University, 2016.

[2] A. Khalifa and M. Fayek, "Automatic puzzle level generation: A general approach using a description language," in *Computational Creativity and Games Workshop*, 2015.

[3] C.-U. Lim and D. F. Harrell, "An approach to general videogame evaluation and automatic generation using a description language," in *2014 IEEE Conference on Computational Intelligence and Games*, pp. 1–8, Aug. 2014.

[4] M.-Q. Jing, C.-H. Yu, H.-L. Lee, and L.-H. Chen, "Solving Japanese puzzles with logical rules and depth first search algorithm," in *2009 International Conference on Machine Learning and Cybernetics*, vol. 5, pp. 2962–2967, July 2009.

[5] J. Taylor and I. Parberry, "Procedural generation of sokoban levels," in *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pp. 5–12, 2011.

[6] D. Ashlock, "Automatic generation of game elements via evolution," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pp. 289–296, Aug. 2010.

[7] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "Procedural level generation with answer set programming for general Video Game playing," in *Computer Science and Electronic Engineering Conference (CEEC), 2015*, pp. 207–212, Sept. 2015.

[8] R. Grimbergen and H. Matsubara, "Plausible Move Generation Using Move Merit Analysis in Shogi," in *Thresholds in Shogi. In Computers and Games: Proceedings CG2000*, pp. 9–16, Springer, 2000.

[9] R. Pelánek, "Difficulty Rating of Sudoku Puzzles: An Overview and Evaluation," *arXiv:1403.7373 [cs]*, Mar. 2014.

[10] C. Jefferson, W. Moncur, and K. E. Petrie, "Combination: automated generation of puzzles with constraints," in *Proceedings of the*

2011 ACM Symposium on Applied Computing, pp. 907–912, ACM, 2011.

[11] V. Lifschitz, "What is answer set programming?.," in *AAAI*, vol. 8, pp. 1594–1597, 2008.

[12] A. M. Smith, E. Butler, and Z. Popovic, "Quantifying over play: Constraining undesirable solutions in puzzle design.," in *Foundations of Digital Games*, pp. 221–228, 2013.

[13] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović, "A case study of expressively constrainable level design automation tools for a puzzle game," in *Proceedings of the International Conference on the Foundations of Digital Games*, pp. 156–163, ACM, 2012.

[14] C. Browne, "Deductive search for logic puzzles," in *2013 IEEE Conference on Computational Intelligence in Games*, pp. 1–8, Aug. 2013.

[15] J. Meng and X. Lu, "The Design of the Algorithm of Creating Sudoku Puzzle," in *Advances in Swarm Intelligence*, no. 6729 in Lecture Notes in Computer Science, pp. 427–433, Springer Berlin Heidelberg, June 2011. DOI: 10.1007/978-3-642-21524-7_52.

[16] C. Browne, "Metrics for Better Puzzles," in *Game Analytics*, pp. 769–800, Springer London, 2013. DOI: 10.1007/978-1-4471-4769-5_34.

[17] P. Mutser, "Automated rating of level difficulty for puzzle games," Master's thesis, Utrecht University, 2014.

[18] N. Vendrig, "Automated level generation and difficulty rating for trainyard," Master's thesis, Utrecht University, 2013.

[19] H. Wong *et al.*, "Rating logic puzzle difficulty automatically in a human perspective," in *Proceedings of DiGRA Nordic 2012 Conference: Local and Global–Games in Culture and Society*, 2012.

[20] D. Oranchak, "Evolutionary Algorithm for Generation of Entertaining Shinro Logic Puzzles," in *Applications of Evolutionary Computation*, no. 6024 in Lecture Notes in Computer Science, pp. 181–190, Springer Berlin Heidelberg, Apr. 2010. DOI: 10.1007/978-3-642-12239-2_19.

[21] P. Jarušek and R. Pelánek, "Difficulty rating of sokoban puzzle," in *Stairs 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, vol. 222, p. 140, IOS Press, 2010.

[22] P. Jarušek and R. Pelánek, "What determines difficulty of transport puzzles," in *Proc. of Florida Artificial Intelligence Research Society Conference (FLAIRS 2011)*, pp. 428–433, 2011.

[23] P. Jarušek and R. Pelánek, "Human problem solving: Sokoban case study," *Technická zpráva, Fakulta informatiky, Masarykova univerzita, Brno*, 2010.

[24] M. Guid and I. Bratko, "Search-Based Estimation of Problem Difficulty for Humans," in *Artificial Intelligence in Education*, no. 7926 in Lecture Notes in Computer Science, pp. 860–863, Springer Berlin Heidelberg, July 2013. DOI: 10.1007/978-3-642-39112-5_131.

[25] K. Kotovsky and H. A. Simon, "What makes some problems really hard: Explorations in the problem space of difficulty," *Cognitive Psychology*, vol. 22, pp. 143–183, Apr. 1990.

[26] D. Williams-King, J. Denzinger, J. Aycock, and B. Stephenson, "The Gold Standard: Automatically Generating Puzzle Game Levels," in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, Oct. 2012.

[27] M. van Kreveld, "The search for a cube puzzle," *Pythagoras*, pp. 10–13, 2004. In Dutch.

[28] A. Hauptman, A. Elyasaf, M. Sipper, and A. Karmon, "Gp-rush: using genetic programming to evolve solvers for the rush hour puzzle," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 955–962, ACM, 2009.

[29] Y. Murase, H. Matsubara, and Y. Hiraga, "Automatic making of Sokoban problems," in *PRICAI'96: Topics in Artificial Intelligence*, no. 1114 in Lecture Notes in Computer Science, pp. 592–600, Springer Berlin Heidelberg, Aug. 1996. DOI: 10.1007/3-540-61532-6_50.

[30] J. Donkervliet, A. Iosup, M. van Kreveld, M. Löffler, and N. Vendrig, "A definition and classification of puzzle games for automated level generation." Unpublished.

Appendices

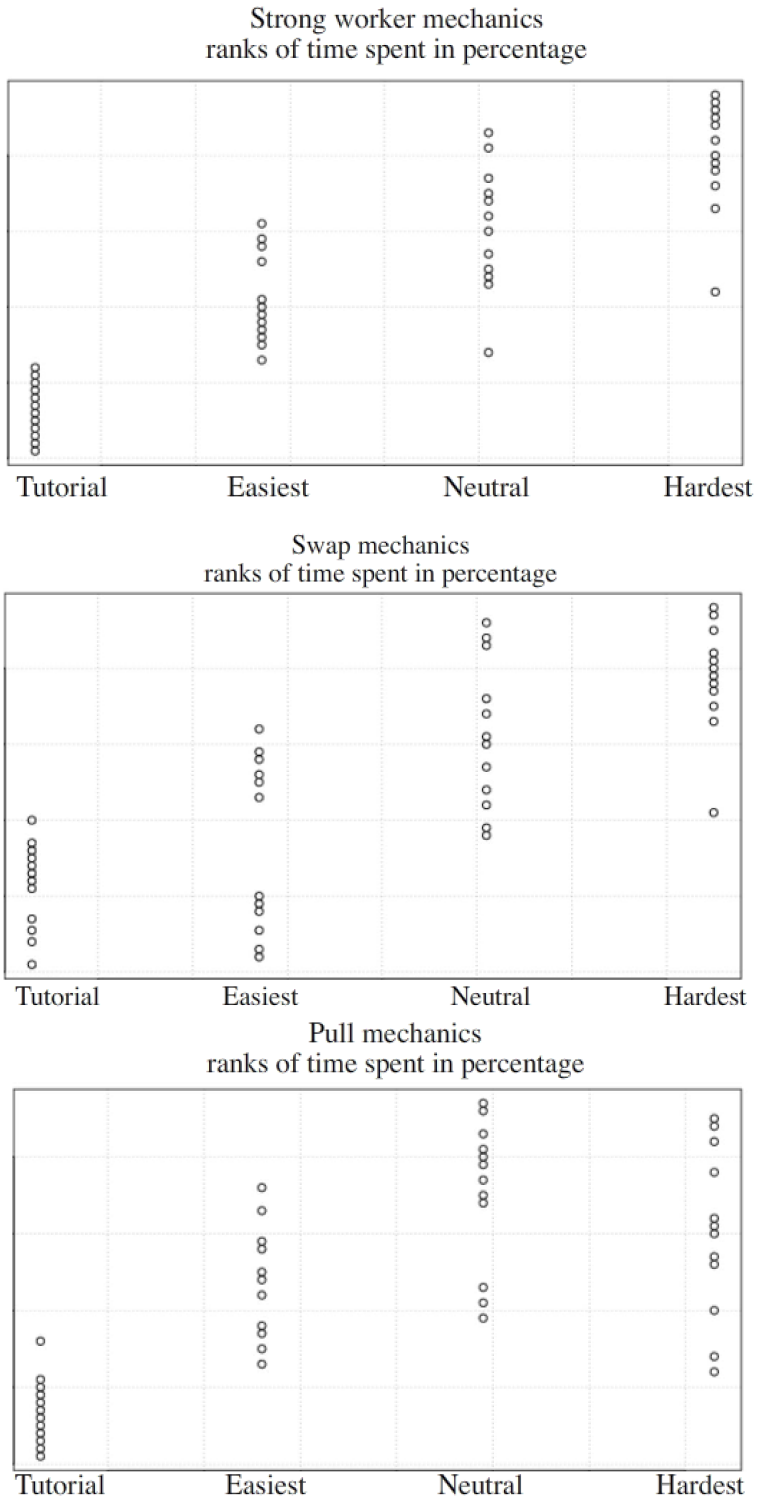
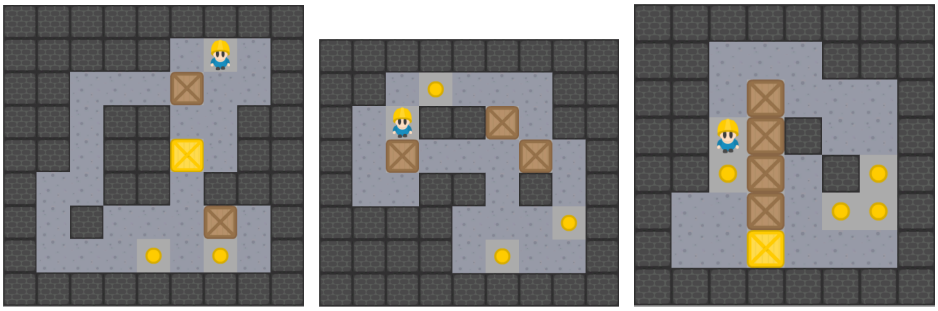


Figure 13:
The ordered data of the time spent on levels. The order is on percentages of time spent per level per participant.

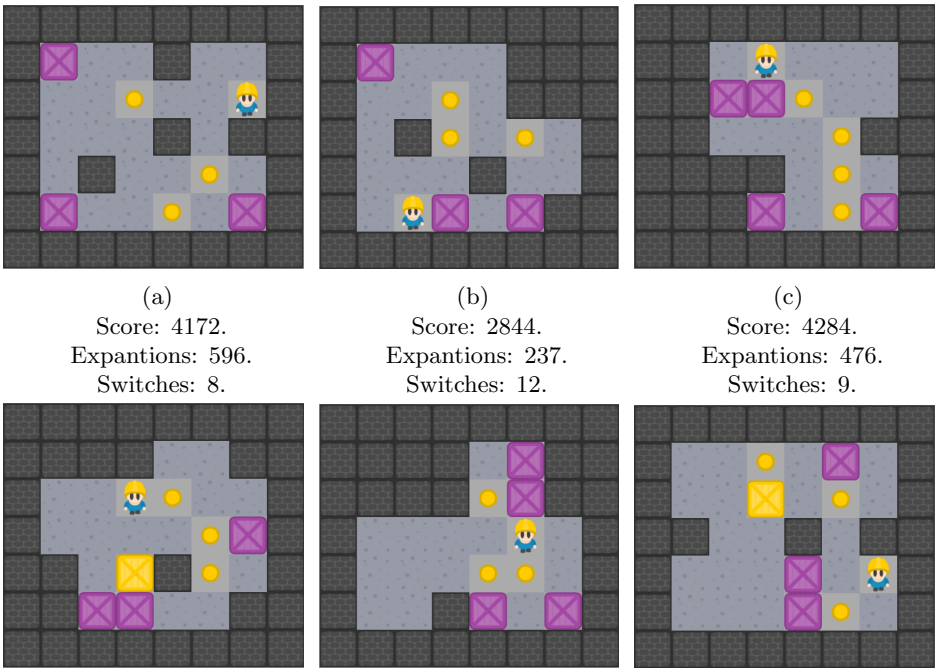


(a)
Score: 1504.
Expantions: 188.
Switches: 8.

(b)
Score: 2233.
Expantions: 319.
Switches: 7.

(c)
Score: 35074.
Expantions: 2698.
Switches: 13.

Figure 14: Generated Sokoban levels.



(a)
Score: 4172.
Expantions: 596.
Switches: 8.

(b)
Score: 2844.
Expantions: 237.
Switches: 12.

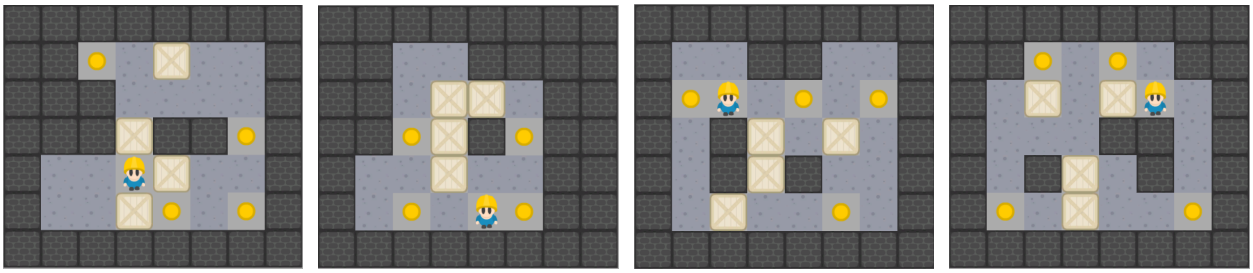
(c)
Score: 4284.
Expantions: 476.
Switches: 9.

(d)
Score: 16030.
Expantions: 1145.
Switches: 14.

(e)
Score: 9010.
Expantions: 901.
Switches: 10.

(f)
Score: 12811.
Expantions: 557.
Switches: 23.

Figure 15: Generated pull mechanic levels.



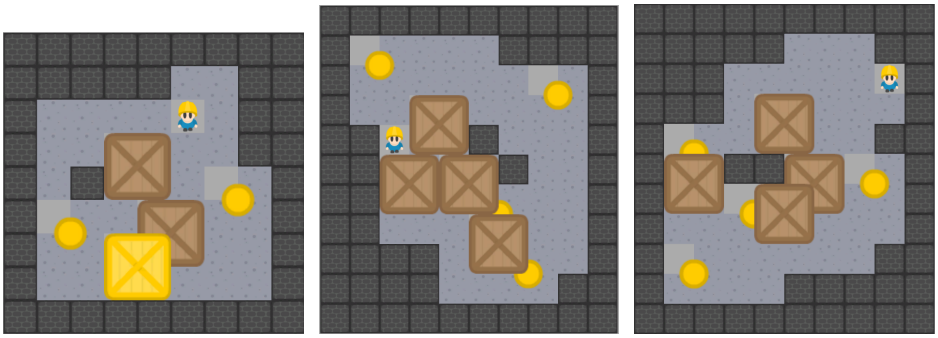
(a)
Score: 6062.
Expantions: 433.
Switches: 14.

(b)
Score: 6648.
Expantions: 554.
Switches: 12.

(c)
Score: 7588.
Expantions: 542.
Switches: 14.

(d)
Score: 17292.
Expantions: 1572.
Switches: 11.

Figure 16: Generated strong worker mechanic levels.

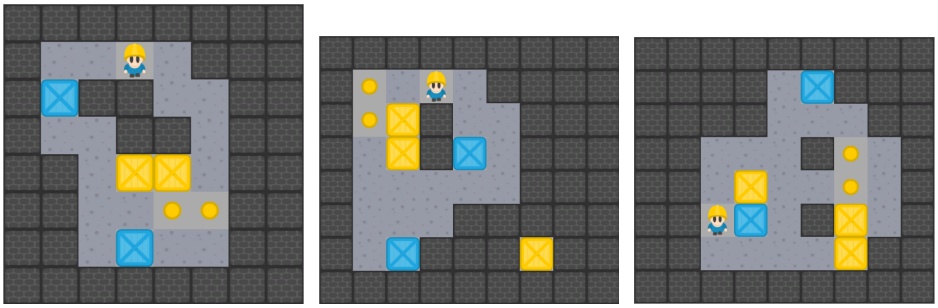


(a)
Score: 530.
Expantions: 106.
Switches: 5.

(b)
Score: 5310.
Expantions: 590.
Switches: 9.

(c)
Score: 11590.
Expantions: 1159.
Switches: 10.

Figure 17: Generated big box levels.
(The overlapping boxes were a bug, but the levels became better of it, you can only push them out of each other)

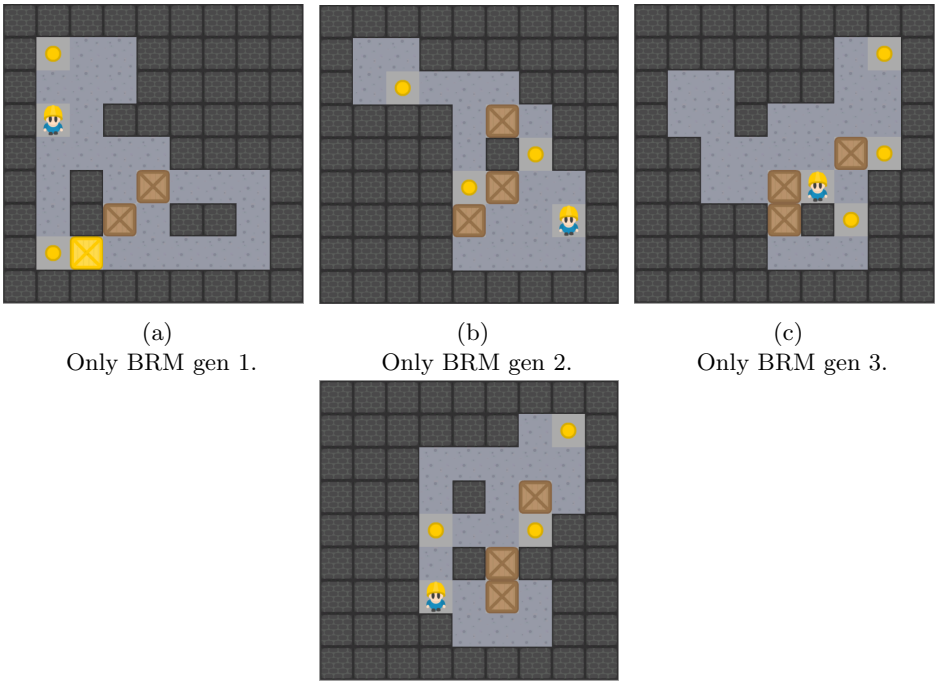


(a)
Score: 28.
Expantions: 7.
Switches: 4.

(b)
Score: 32.
Expantions: 8.
Switches: 4.

(c)
Score: 32.
Expantions: 8.
Switches: 4.

Figure 18: Generated swap mechanic levels.



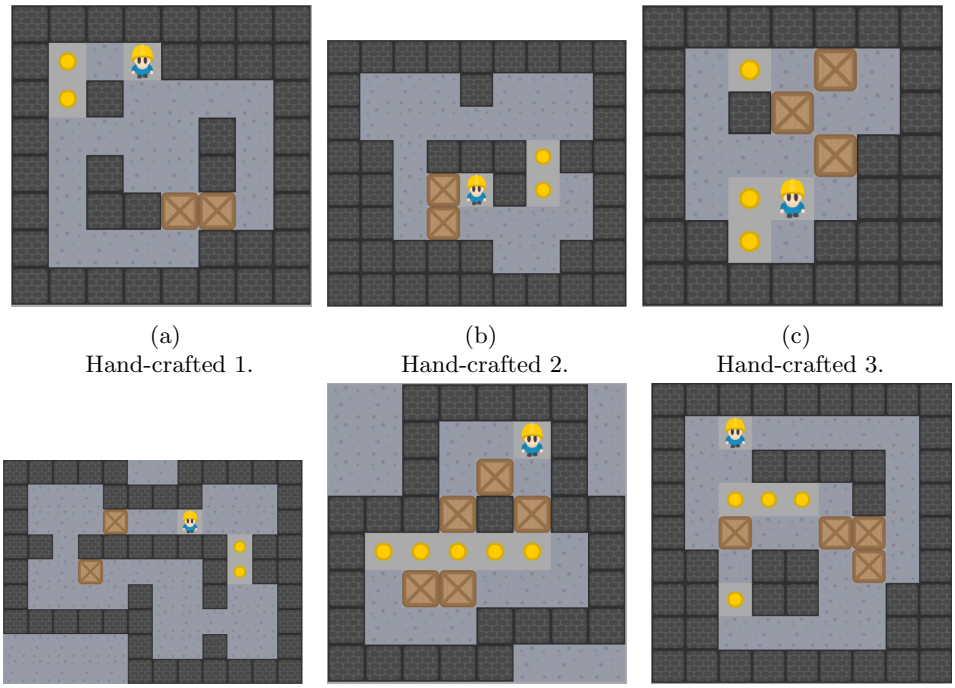
(a)
Only BRM gen 1.

(b)
Only BRM gen 2.

(c)
Only BRM gen 3.

(d)
Only BRM gen 4.

Figure 19: Easy Sokoban levels generated with only the BRM as heuristic.



(a)
Hand-crafted 1.

(b)
Hand-crafted 2.

(c)
Hand-crafted 3.

(d)
Hand-crafted 4.

(e)
Hand-crafted 5.

(f)
Hand-crafted 6.

Figure 20: Hand-crafted levels used to test the methods during development.