Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

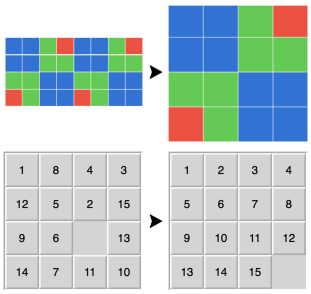
Abstraction and Reasoning Challenge

Bachelor's Thesis

Enea Peter

peteren@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich



Supervisors:
Ard Kastrati
Prof. Dr. Roger Wattenhofer

April 4, 2022

Acknowledgements

I would like to thank Ard Kastrati for supervising my work and for his continuous assistance and guidance. Furthermore, I would like to thank Prof. Roger Wattenhofer and his Distributed Computing Group at ETH Zürich for giving me the opportunity to write this interesting thesis.

Abstract

When humans encounter new puzzles and logical problems, it is often sufficient to see a few examples, rely on prior knowledge, and make a few attempts to learn how to solve them. Current, machine learning models, on the other hand, often need a lot of examples, specialize only in one task, and do not learn based on previously acquired knowledge. DreamCoder is a machine learning model that tries to get closer to the human learning process. This thesis aims to understand and evaluate the ability of DreamCoder to learn as a human and its capability to solve different logical tasks. A system that can learn from previous experiences and react to challenges having only little data available would have numerous applications in the present and the future. In this work, it was found that DreamCoder shows promising results but is still far from being able to learn as a child. DreamCoder tries to solve problems by abstracting useful primitives. However, to do so, there is a need for tasks with increasing difficulty. These, have to be designed or created in such a way, that they allow for primitives to be discovered. What becomes evident with the results of this thesis is that there is a lack of the capability to create these increasingly difficult, but meaningful tasks.

Contents

Chapter 1

Introduction

When we come across a new puzzle while reading a newspaper or magazine, we often don't even think about how we, as humans, are able to understand the puzzle and learn how to solve it. With just a few rules, we can start playing and developing strategies to solve the game. We do not need many examples and improve our ability to solve step by step by drawing on previous experience and reasoning about the given problem. This approach is very different from the one used by most current machine learning methods. They are often data hungry, require many examples, and cannot improve their ability to solve a problem by combining newly acquired techniques to solve sub-problems or other problems. In addition, they can only work on problems and tasks they have seen before and they are often specialised in solving a single task. This makes it difficult to develop systems that can solve a variety of different, unrelated problems. DreamCoder [1] is a machine learning model that takes a different approach than usual and tries to get closer to the way humans learn and solve problems. In this work, DreamCoder is being run on two different datasets. On the one hand, a collection of abstract reasoning tasks, and on the other a known puzzle.
The definition by François Chollet of the intelligence of a system is as follows: "The intelligence of a system is a measure of its skill-acquisition efficiency over a scope of tasks, with respect to priors, experience, and generalization difficulty"[2]. This thesis aims to understand and evaluate the ability of DreamCoder to learn as a human and its capability to solve different logical tasks. For this purpose, first, the model itself is introduced and followed by the description of the puzzles. In the second chapter, the experiments and technical detail are presented. In the third chapter, the results are presented and discussed. In the fourth, and last chapter, the results of the experiments are discussed as a whole and the conclusions are drawn.

1.1 DreamCoder

DreamCoder, introduced by *Ellis et al.*,2020 [2], is a machine learning system based on an approach called "wake-sleep Bayesian program induction". The au-

thors claim that DreamCoder is capable to discover interpretable, reusable, and generalizable knowledge over different domains. Learning, for DreamCoder, is based on the search for a functional program that solves a given problem, given by input-output examples, using a set of functions in a given library. DreamCoder apparently tackles two fundamental bottlenecks encountered traditionally when talking about the application of program induction. On the one hand, it learns to compactly represent programs, reducing the problem of very long programs. On the other hand, it learns to induce programs in a given domain reducing the problem of a huge search space. Dreamcoder solves tasks using its library of functions, it then learns new functions out of existing functions through compression, by capturing which combination of functions was used in several solved tasks. Thanks to these newly learned functions it can then try to solve other tasks it was not able to solve before. In addition, Dreamcoder also trains a neural network that predicts which functions most likely will solve a given task. Thanks to the compression of the combination of functions into a new function and a neurally guided search of programs to solve tasks, DreamCoder learns to solve increasingly complex tasks related to a domain. It is interesting to note that when DreamCoder solved, thanks to its grown library of functions and trained model, a very complex task, in theory, the same tasks would have also been solvable with the initial library. But the program to solve the task with the initial library would be huge and not be foundable in a reasonable amount of time. DreamCoder grows its knowledge of a particular domain by ”wake-sleep” cycles. The schematic view of the model, taken from the original publication, can be found in Figure-1.1. It iterates through a wake phase, where it tries to solve tasks, and through two sleep-phases, where it learns to solve new tasks. In the wake phase, the model tries to find for each task a program that solves it. It uses the functions of the library and is helped by the neural recognition model. The Abstraction sleep-phase has the role of finding common program parts in solutions of tasks found in the wake phase. It then abstracts these common program parts into new code primitives and can extend the existing library. In such a way in the next wake phase more complex tasks can hopefully be solved. In the second sleep phase, dreaming, the neural network is trained that helps to search for programs that solve the tasks in the wake phase. It is trained on solved tasks, as well as new ”fantasies” generated by sampling programs from the learned library.

1.2 The Puzzles

To understand better the capabilities of DreamCoder and explore its ability of abstract reasoning, DreamCoder was adapted to solve the Abstraction and Reasoning Corpus (ARC) introduced by François Chollet in ”On the Measure of Intelligence”[1]. In addition, it was also tried to adapt it to solve the ”Sliding puzzle” to see if it was possible to bring it to solve a logical game that a human

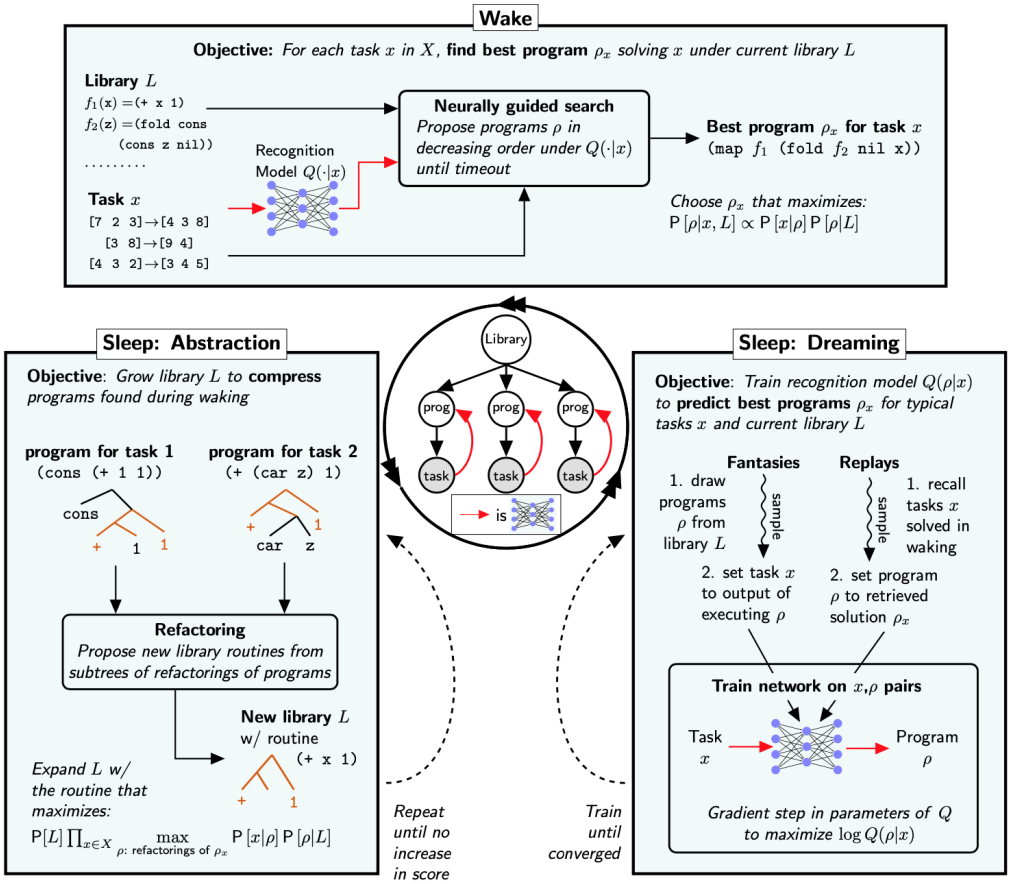


Figure 1.1: Schematic view of DreamCoder taken from the original publication (Figure 2 of [2]). The basic algorithm is shown in the middle. On top, the wake phase is shown whereas on the left and the right the two sleep phases can be found.

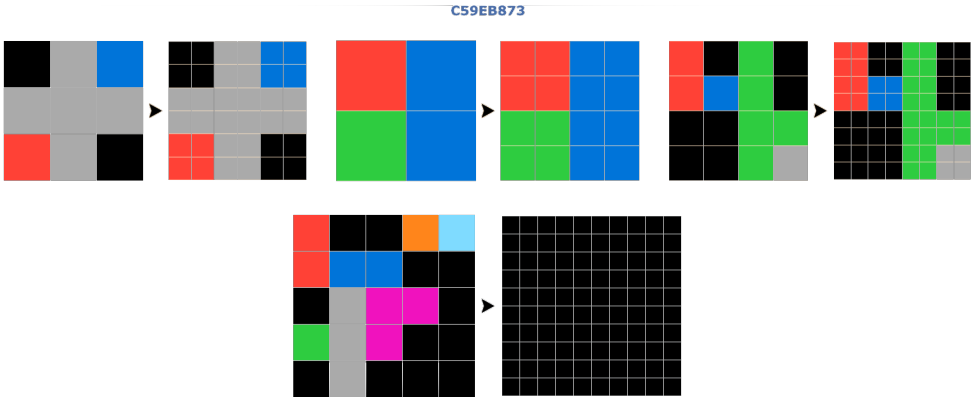


Figure 1.2: ARC task C59EB873, difficulty level entry. [3]

being can learn after some tries.

1.2.1 ARC

ARC is a dataset introduced by François Chollet. Its general purpose is to serve as an artificial intelligence benchmark, a program synthesis benchmark, or a psychometric intelligence test [1]. ARC has various top-level goals. Most important for this thesis are the following:

- Tasks generally provide only a few examples, this requires solving a task with little experience
- there is a need for generalization as in the evaluation set only unseen tasks are present
- It is solvable by humans without prior knowledge and training and it permits a broad intelligence comparison with machines.

The dataset is composed of 800 different tasks, 400 training tasks, and 400 evaluation tasks. These tasks are furthermore divided into different categories based on difficulty. The different categories are entry, easy, medium, difficult, tedious, multiple solutions, and unfixed. The dataset can be found at `github.com/fchollet/ARC`. Each task is composed of a few examples and one test (only in the category multiple solutions there are more tests than one). All examples are composed of one input/output pair where both are colored grids of dimensions between 1×1 and 30×30. The grids are colored with 10 possible colors. In Figure-1.2 and Figure-1.3 two tasks are shown of difficulty level entry.

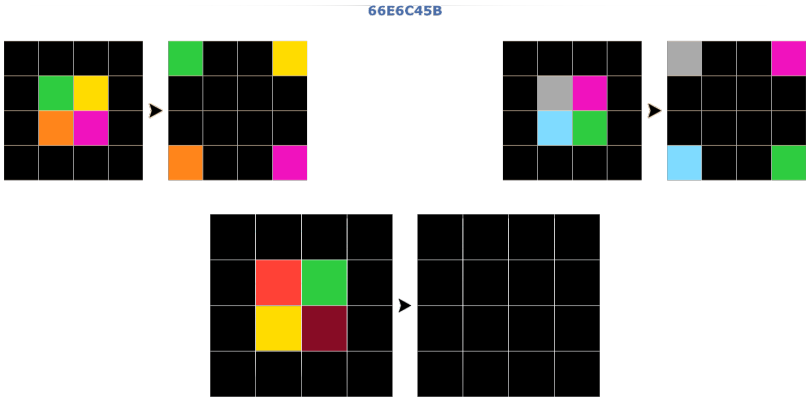


Figure 1.3: ARC task 66E6C45B, difficulty level entry. [3]

1.2.2 Sliding puzzle

The sliding puzzle exists in many variations and under many names. The focus was on the following puzzle: there is a $n \times n$ sized grid with $n^2 - 1$ tiles in it. The tiles are numbered from 1 to $n^2 - 1$ and one space is empty. The objective of the game is to rearrange the tiles into order. Problem instances were restricted to square grids. To move the tiles, the only allowed moves are sliding a neighbor (a tile that is above, below, on the right, or the left of the space) of the space onto the space. This results in exchanging the space with the moving tile. In Figure-1.4 an example is shown: it is shown how the tile number 13 is moved down by one.

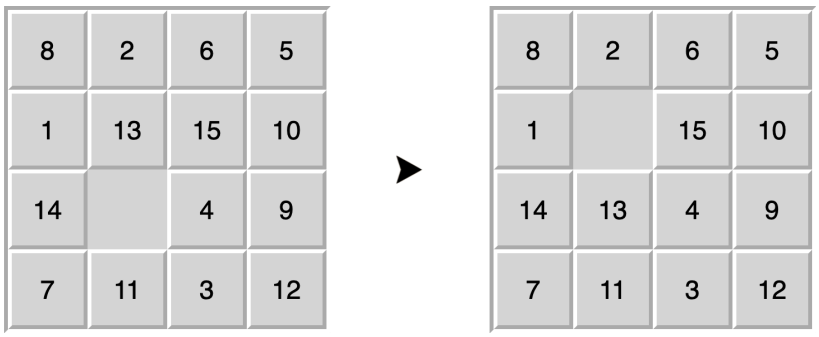


Figure 1.4: Example of a 4×4 instance of the sliding puzzle, illustrating one aloud movement. For instance, in the right image tile 13 was moved down by one. [4]

Not knowing the game a human being would probably simply try to solve it by brute force. Beginning moving the smallest number to the top left corner and then continuing with the bigger number one number at a time. After some attempts, a strategy would probably become clear. There are two main strategies

how to solve the puzzle. Either you try to solve row by row until you reach the last two rows and then proceed to solve column by column. Another possibility is to solve first a row, then a column, then again a row until you only have one more 2x2 square in the bottom right-hand corner to solve. The most difficult part of the process is how to place the last two elements of a row or column respectively. The easiest way to do so is to bring the second-last element of the row to the rightmost position of the row. Then bring the last element just below it. Then move the empty tile into the second-last position of the row and move the last two elements in position by simply exchanging the empty tile first with the second-last element and then with the last element of the row. The process is illustrated in Figure-1.5. To solve the column the process is analogous except that you work with columns and bring the last element to the right of the second-last element instead of bringing it below it.

Figure 1.5: Example of a 3×3 instance of the sliding puzzle illustrating how to correctly position the last two elements of a row. First, bring the number 2 to the last position of the row. Then position the 3 below it and move the empty tile next to the 2. Then exchange first the empty tile with the 2 and then with the 3. This results in a solved first row. [4]

Chapter 2

Experiments

2.1 Datasets

This section describes how the different problems were presented and how the tasks and the new domains for DreamCoder were created.

2.1.1 ARC

Representation of the problem

In the ARC dataset, the tasks were already given. Therefore, the representation used in these tasks could simply be used. To represent one input/output pair, two lists of lists of integers were used. The integers allowed are the numbers from 0 to 9. All grids present in the dataset were squares. Each of these lists of lists can be represented as a colored grid by mapping each number to a color in the way shown in Figure-2.1. The grid can be seen as a matrix of numbers from 0 to 9 represented as a list of lists in row-major manner. One task of the dataset is considered solved when the right output grid, given an input grid and some examples, can be obtained. Meaning that the outputted list of lists is compared with the given solution and if they match the task is solved.

Creation of tasks

In this case, tasks were already given by the ARC dataset. The tasks are subdivided in different difficulty levels: entry, easy, medium, difficult, tedious, multiple

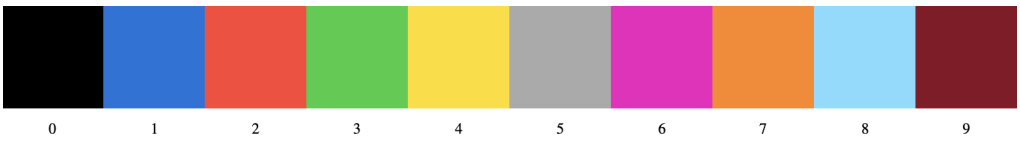


Figure 2.1: ARC color mapping. [3]

solutions, and unfixed. For the purpose of this thesis, the ARC tasks were simply kept as they were and used as tasks for DreamCoder. Also, the original differentiation in levels and training/evaluation split were kept the same. The focus was laid on the task sets of the first three difficulty levels (entry, easy and medium). This resulted in 340 training tasks, where 27 were entry-level, 210 were easy level and 103 were medium level. In addition, there were 255 evaluation tasks with 6 entry, 92 easy, and 145 medium tasks. Each task has from 2 to 4 input/output train examples and 1 input/output test. Task "C59EB873" is shown in Figure-1.3 and has 3 train examples and 1 test example. The first input/output example equals the following list of lists pair:

$\{'input' : [[0,5,1],[5,5,5],[2,5,0]],$
$'output' : [[0,0,5,5,1,1],[0,0,5,5,1,1],[5,5,5,5,5,5],$
$[5,5,5,5,5,5],[2,2,5,5,0,0],[2,2,5,5,0,0]]\}.$

2.1.2 Sliding puzzle

Representation of the problem

This problem, consisted of $n \times n$ grids filled with numbers and one empty space. The problem can be seen as a $n \times n$ matrix filled with integers from 1 to $(n*n-1)$ and one blank space. The number 0 was simply assigned to the empty space and therefore a square matrix filled with numbers from 0 to $(n*n-1)$ was obtained. This matrix can be easily represented in form of a list of lists in row-major order. The problem is considered solved when the flattened list of lists is in the following form: $[1,2,...,(n*n-1),0]$. In the sliding puzzle, only the empty tile can be moved. This is modeled by the fact that the only allowed movements are exchanging the number zero with one of its orthogonal neighbours in the matrix.

Creation of tasks

In this case, the creation of the tasks was not as straightforward as it was with the ARC as there was no existing dataset that could be used. The simplest idea that comes to mind is to simply take a $n \times n$ matrix filled in random order with the numbers from 0 to $(n*n-1)$ as input and the solved puzzle in matrix form as output. The first thing to note here is that not all such input matrices representing an instance of the sliding puzzle are solvable. In fact only half of all possible initial configurations are solvable. If an instance of the puzzle is solvable or not can be easily checked using the function "is_solvable" in the appendix in Section-B.1.1. The next question is whether the task should contain only a single instance of the problem or several instances of the problem, and in the latter case whether the instances should be of the same size or different sizes. As DreamCoder seemed to have more difficulty when presented with lots and lots of tasks, grouping the instances of the same size seemed also a valid idea. In the

end, both ideas were used by creating tasks with single solvable instances of the puzzle and tasks with 5 solvable instances of the puzzle of the same size. The tasks were created as follows:

- 60 tasks containing each 5 input/output pairs with grid size 3×3
- 50 tasks containing each 5 input/output pairs with grid size 4×4
- 40 tasks containing each 5 input/output pairs with grid size 5×5
- 30 tasks containing each 5 input/output pairs with grid size 6×6
- 20 tasks containing each 5 input/output pairs with grid size 7×7
- 1 task containing all 12 solvable 2×2 size instances
- 100 tasks containing only 1 solvable instance size 3×3
- 50 tasks containing only 1 solvable instance size 4×4

All instances, except the size 2×2, were created by taking a random permutation of a list of numbers from 0 to $(n * n - 1)$. From the permutation, a matrix of size $n \times n$ was created by considering the permutation as the result of a row-major traversing of the matrix. The instance was only kept if solvable. In total this resulted in 351 different tasks.

Because of the difficulties, DreamCoder encountered in solving this puzzle, also a second group of tasks was created. These tasks were more carefully designed following a possible path to solving the sliding puzzle. This was done with the objective in mind that in this way, DreamCoder would possibly be able to learn how to solve the puzzle. It was tried to design the tasks so that they each constituted a step on the way to solving the puzzle. The objective of this task set was to constitute the intermediate steps in the process of bringing the number 1 to the top left corner. The thinking behind this was that this is often the first step taken when solving this puzzle. The tasks created can be found in the list below. In each task all possible instances of size 2×2 were present. In addition, 40 instances in each of the sizes 3×3, 4×4, and 5×5 were present. The tasks are listed in different categories depending on the purpose:

- Move the 0 related tasks: these tasks aimed to learn to bring the number 0 (the empty tile) to the right border, to the bottom, and to the bottom right corner. Having the 0 in that position represents a good starting point to proceed with solving the puzzle at various stages of the process. In this category there were 6 tasks:
  p_move_0_to_right, p_move_0_to_right2,
  p_move_0_to_bottom, p_move_0_to_bottom2,
  p_move_0_to_bottom_right, p_move_0_to_bottom_right2,

- Tasks aiming to remove the 1 from the last line, if that was the case. This opens up the possibility of moving the zero below the 1. The tasks were the following:
  p_move_1_away_from_last_row_starting_with_0_bottom_right,
  p_move_1_away_from_last_row,
  p_move_1_away_from_last_row_and_0_to_last_row.
  The tasks are slightly different. The first one is the easiest and the others become progressively more difficult.

- Tasks aiming to move the 0 below the 1: these tasks aimed to bring the zero below the 1. This represents an intermediate step in the process to move the 1 to the upper left corner. From the situation with the 0 below the 1, it is then easier to bring the 1 to the upper left corner. This is because it is clear where the 0 is located with respect to the 1. Therefore the 1 can be moved more easily.
  In this category, there were 3 objectives to learn. This resulted in 12 tasks, 4 times the same 3 objectives but always with slightly different starting points. The objective were: zero_is_in_same_col_as_1, zero_is_one _row_below_1, zero_is_below_1. The 4 different starting points differed in the conditions that had to hold for the input images: no condition, 0 had to be in the bottom right corner, 1 was not allowed to be in the last row, 1 was not allowed to be in the last row and 0 had to be in the bottom right corner. For all of these 12 tasks another equal task, but with other input instances, was created to improve the learning of new primitives because of the repetitiveness.

- Helper tasks to bring 1 away from the last row: As DreamCoder could not solve the tasks about bringing away the 1 from the last row right away this additional tasks were added. The tasks were:
  p_move_zero_above_1_if_1_is_in_last_col ,
  p_move_zero_above_1_if_1_is_in_last_col_zero_everywhere.
  Both aim to bring the 0 above the number 1, but only if the number 1 was located in the last row. Also, these two tasks were duplicated.

- A single task aiming to move the number 0 below the 1. All the above tasks can be considered preparatory tasks for solving this task.

- Tasks regarding the moving of the 1: In this category, there were 3 tasks all with duplicates making a total of 6. The tasks were the following: p_move_1_up_and_0_below,
  p_move_1_left_and_0_below, p_move_1_corner_up_left.
  To solve those tasks the 0 had first to be brought below the 1 and then the 1 could be moved. In the first two cases, it is simply one step up or left in the last task the aim is to bring the 1 in the top left corner. In all 3 tasks, it holds that in the input instances the 1 was not in the last row.

- The last task brings this series of tasks to a conclusion and aims to bring the number one to the top left corner starting from an arbitrary instance of the problem.

This makes a total of 45 tasks for this second set. A full listing of these tasks can be found in the appendix in Table-B.3.

2.2 Building new domains

When building a new domain in DreamCoder different decisions have to be taken. Tasks have to be defined including the input and output type. These can then be split into training and testing sets. In addition, primitives, declaring input and output types have to be defined. In the following, the ideas behind the defined primitives for both domains, ARC and sliding puzzles, are presented.

2.2.1 ARC

As explained in Section-2.1.1 the input and output types of the tasks were lists of lists of integers. The corresponding type in DreamCoder is (tlist (tlist (tint))). The general approach that was taken to define the primitives to solve ARC tasks was to simply try to solve different tasks by hand and get a feeling of what primitives could be useful to solve general ARC tasks. The idea behind that was to test whether and how well it is possible to use DreamCoder to solve abstract reasoning tasks without having in-depth knowledge about the tasks and by only providing general primitives.
In the following, four different primitive sets are presented. These were used in the experiments. The difference between the sets lies in the number of primitives given and not in the implementation of the single primitives. The full list of the names of primitives contained in the sets can be found in the appendix in Section-A.1.1.

- Elementary: this set of primitives contains only the most low-level kind of primitives. Primitives that add and remove rows and columns and a primitive that changes entry at row i and column j to color c. In addition, it contained the numbers from 0 to 9 representing the 10 colors.
- Basis ARC: This set of primitives contained all primitives of ”elementary” and additional primitives which seemed intuitive on a colored grid. The primitives can be separated into different groups depending on what type of operations they perform or on what properties of the image they work on. The following primitives were added:

  - Color specific: returning the color at a specific position, coloring a row or a column with a specific color, color a specific position with color c1 if it previously had color c2, color in black all colored pixels and in color c all black pixels, mapping some color c1 to some color c2, count all colored pixels, return the first encountered color.
  - Division of the image: identity function giving back the original image, taking only part of the image by giving the coordinates of the upper left and lower right corners, take the top half or the left half of the image, take only the first row/column.
  - Transposing and reflecting: transposing the image, reflecting the image vertically/horizontally.
  - Amplification: stacking two different images one above the other or one next to the other, double/triple every pixel horizontally or vertically
  - Logic: if some Boolean input is true apply a function to some input value, union of two images by taking the color of the second image for pixels that are black in the first, intersection of two images by blacking out all colored pixels of the first image where the second image was black, if first row/column is all equal return true and otherwise false, return whether two images are equal or not, return whether an image is horizontally/vertically symmetric.
  - Move and remove black: move an image up, down, left, or right by inserting a black row or column and eliminating the row or column on the opposite end, remove all black row/columns on top, bottom, left, or right.
  - Others: flatten the image to a list row by row, access a list at some index, return an image composed only of one row of size n.

- All: here some primitives were added that could have been discovered using the primitives in the set "Basis ARC". Adding them can help to solve more tasks and maybe discover more easily other new primitives. For example, primitives taking the bottom half and taking the right half have been added. The same effect can be achieved by composing mirroring primitives with take-upper-half or take-left-half primitives. In addition rotation by 90 degrees clockwise and counterclockwise were added (can be achieved with transposing and mirroring). Also, the primitives flip down (stack vertically the image and the horizontally mirrored image) and duplicate horizontally/vertically were added (all 3 can be emulated with the stack primitives and the mirror primitives).

- Selected: the primitives in this set are selected primitives from the set "All". The primitives that showed the most promising results in solving tasks were added here. The aim was to reduce the number of primitives in order to

reduce the search space and potentially discover more new primitives and solve more tasks. It has been seen that especially for primitives taking various arguments DreamCoder had more difficulty using them for solving problems.

2.2.2 Sliding puzzle

Also for this puzzle, to represent the input and output types, lists of lists of integers were used. Using the inbuilt types of DreamCoder tint and tlist the resulting type was: tlist (tlist (tint). In addition, the inbuilt type tbool was used to represent Booleans. Booleans were needed as input and output for some primitives. The first created primitives were the 4 most basic primitives, which would perform the only aloud movements in the puzzle. The aloud movements are moving the empty tile, in this case the 0, right, left, up, and down. This means that the only aloud modification of the matrix is the swapping of the 0 with one of its 4 orthogonal (horizontal and vertical) neighbors. In the first trials, it was observed that with these and other rather generic and low-level primitives and general tasks there was no way DreamCoder could learn to solve the puzzle.

Therefore a different approach was tried. First, the problem was studied more in-depth by trying to solve it. This resulted in a program capable of solving any instance of the problem up to 9×9. This program can be found in the appendix in Subsection-B.1.3. Afterwards, the most basic possible primitives were extracted from this program. In order to be sure that these primitives would be enough to solve the problem, a program was written that solved the puzzle in a way as potentially DreamCoder would also be able to learn. This second program can also be found in the appendix in Subsection-B.1.4. The primitives extracted were the starting set of primitives to run DreamCoder with. This set is called "Basis sliding puzzle" in the following. The objective was to see if DreamCoder would be able to learn to solve the puzzle. The list of primitives in the set "Basis sliding puzzle" can be found in the appendix in Section-B.2.1. To help DreamCoder in learning to solve the puzzle 3 other sets were created. Each of these sets contains the primitives of "Basis sliding puzzle" and two other additional primitives. These additional primitives are all higher level than the primitives in the basis set. They gather a bigger part of the solution program (which can be found in Subsection-B.1.3) into one primitive. In the following, the sets and the additional primitives, as well as their actions are listed. From set to set, going downwards, the primitives solve a bigger part of the problem.

- Basis + Help_1: In this set, the following primitives are added:
  - move_to_right_position_row
  - move_to_right_position_col

Both these primitives take as input one slider instance and the number n that has to be moved to the right position. If 0 is situated below n, then they both output the slider instance with n moved into the right position. All movements are only performed using the 4 basic movements defined above. The first primitive moves n to the right position when a row is trying to be solved whereas the second one works in the case of a column.

- Basis + Help_2
  - solve_one_number_of_a_row
  - solve_one_number_of_a_col

  These primitives, instead of only moving a number to the right position if some preconditions are met, move the number directly to the right position. There is only one small precondition, namely the 0 has to be in the bottom right corner before beginning. Also in this case one primitive is for the case solving a row and one for the case solving a column.

- Basis + Help_3
  - solve_row
  - solve_col

  These primitives are very high level, they directly solve a row or a column of a problem instance. Nevertheless, they are only able to solve rows and columns up to the last 2. This means that when using them correctly in the end only a 2×2 square in the bottom right corner remains unsolved. This can then be easily solved using a primitive found in the set "Basis sliding puzzle" which solves this last small square.

In addition, as explained in Section-2.1.2, a second way was tried that focused more on the creation of specific tasks. In this case, the idea was to create tasks that represented intermediate steps in solving the problem. Specific primitives were created that would allow these tasks to be solved. The set of primitives in question is called "Tasks and Primitives engineered". Some primitives are shared between this set and the "Basis sliding puzzle" set, but not all. However, in theory, these tasks could also be solved with the primitives in the "Basis sliding puzzle" set, but the primitives in the "Tasks and Primitives engineered" were less general and therefore easier to use. A complete list of the primitives contained in this set can be found in the appendix in Section-B.2.2.

2.3 Performed experiments

The experiments where performed on the Slurm cluster of the D-ITET at ETH Zürich.

2.3.1 ARC

As explained in Subsection-2.1.1 there were a total of 340 training tasks subdivided by difficulty. In addition, there were 255 evaluation tasks. Moreover, there was also a collection of different primitives as explained in Subsection-2.2.1. To test the capability to learn of DreamCoder different sets of primitives were supplied and it was analyzed how they performed. It was also analyzed how many tasks were solved after each iteration and how many new primitives were found. All experiments were performed with a recognition timeout of 30'000 seconds, 15 iterations, and a testing timeout of 2'000 seconds. The experiment were performed with 4 different sets of primitives: Elementary, Basis ARC, All and Selected. In addition, DreamCoder was run with the primitive selection "All" and the merging of the training set and the evaluation set as task set. This resulted in a larger training set and no evaluation set. In Table-2.1 a summary of the performed experiments can be found.

Primitive set name	Number of training tasks	Number of evaluation tasks	Iterations	Recognition timeout (s)	Testing timeout (s)
Elementary	340	255	15	30'000	2'000
Basis ARC	340	255	15	30'000	2'000
All	340	255	15	30'000	2'000
Selected	340	255	15	30'000	2'000
All	595	0	15	30'000	-

Table 2.1: Summary of the performed experiments on the ARC dataset.

2.3.2 Sliding puzzle

As seen in Subsection-2.1.2 there were 2 different sets of tasks. On the one hand, 351 tasks requiring to solve the puzzle as a whole, on the other hand, 45 tasks trying to guide the process requiring to solve sub-problems. For the first collection of tasks, different sets of primitives were used. From the most general one, with which it was theoretically possible, but very difficult, to solve the problem, to more specific sets with higher-level primitives with which it would be easier to solve the puzzle. For the second collection a more specific subset of primitives was used, with which it wouldn't be possible to solve the whole problem but at least the tasks of this collection. In Table-2.2 a summary of the performed experiments can be found.

Primitive set name	Taskset name	Number of training tasks	Iterations	Recognition timeout (s)
Basis sliding puzzle	General Tasks	351	15	30'000
Basis + Help_1	General Tasks	351	15	30'000
Basis + Help_2	General Tasks	351	15	30'000
Basis + Help_3	General Tasks	351	15	30'000
Tasks and Primitives engineered	Engineered Tasks	45	15	30'000

Table 2.2: Summary of the performed experiments on the sliding puzzle.

Chapter 3

Results and Discussion

In this chapter first, the results of the experiments are presented and then discussed. The ARC dataset is covered first followed by the sliding puzzle. In the next chapter, a general discussion is presented in which both experiments are considered together.

3.1 ARC

3.1.1 Results

In this subsection, the results of the experiments listed in Table-2.1 are presented. The results are given separately for each experiment.

Elementary

Running DreamCoder with the primitive selection "Elementary" resulted in 0 training tasks solved and 0 evaluation tasks solved. This is reasonable and was also expected. Although it is possible to create any kind of image with these primitives, the issue is that there is no way to read and react to the input image. Therefore few imaginable tasks exist that could be solved. An example could be a task where the output is the same image for any input. But even in this case, this kind of task would be difficult to solve because it would require many function compositions with many input parameters to get to the final output image. Other solvable tasks could be tasks where it is simply removed a row or column from the input image. However, in the ARC dataset already the entry difficulty tasks are often more complex than those described here.

Basis ARC

When running DreamCoder with the primitive selection "Basis ARC" 53 primitives were in the initial library. After the first iteration, 19 training tasks and 1

evaluation task were solved. In addition, 4 new primitives could be found. The following primitives were newly found:

```
1. #(lambda (mirrorVer (mirrorHor $0))) = ''Mirror along
   diagonal''
2. #(lambda (_union $0 (mirrorHor $0))) = ''Union of the image
   and its horizontally mirrored image''
3. #(lambda (stack_h $0 (mirrorVer $0))) = ''Flip right''
4. #(lambda (stack_v $0 (mirrorHor $0))) = ''Flip down''
```

It can be seen can see how the first new primitive, mirror an image horizontally and vertically, was created out of the primitives mirrorVer and mirrorHor. These are primitives that can be found in the initial set. This newly created primitive was for example used to solve Task 6150a2bd. The visual representation of this task can be found in Figure-3.1. Another task that has been solved is for example task F25FFBA3. This has been solved using the second newly found primitive and can be found in Figure-3.2. When a task is solved using various primitives, it does not necessarily mean that a new primitive is created. For example task 2dee498d has been solved using the primitives "take_left_half_with_border" and "addColLast" in the following way:

```
(lambda (take_left_half_with_border (addColLast (
    take_left_half_with_border $0))))
```

Where $0 is the input argument of the lambda function. Nevertheless no new primitive has been created out of this solution. This is because a new primitive is only created when a part of a function composition is used in different places in the process of solving tasks. Example 2dee498d can be found in Figure-3.3.
In the second iteration, 6 more training tasks and 2 more evaluation tasks could be solved. All new solved tasks, except for 1, were solved using new primitives found in iteration 1. For example, one newly solved task is 62c24649 and can be found in Figure-3.4. This task could be solved thanks to the use of the new primitives 3 and 4 found in iteration 1. It is solved as follows:

```
(lambda (#(lambda (stack_h $0 (mirrorVer $0))) (#(lambda (
    stack_v $0 (mirrorHor $0))) $0)))
```

This function has also been used to create a new primitive. The result of this primitive is an image that is first flipped to the right and then flipped down.
After iteration 2 no more tasks have been solved and no new primitives have been found. This leads, after 15 iterations, to a final result of 25 out of 340 training tasks and 3 out of 255 evaluation tasks solved, and 5 newly found primitives. In the end, the library contained 58 primitives.

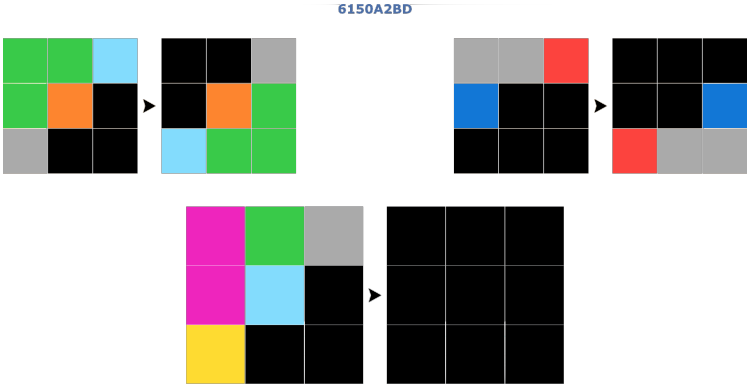


Figure 3.1: Task 6150a2bd of the ARC dataset. The task was solved in the following way: (lambda (#(lambda (mirrorVer (mirrorHor $0))) $0)). [3]

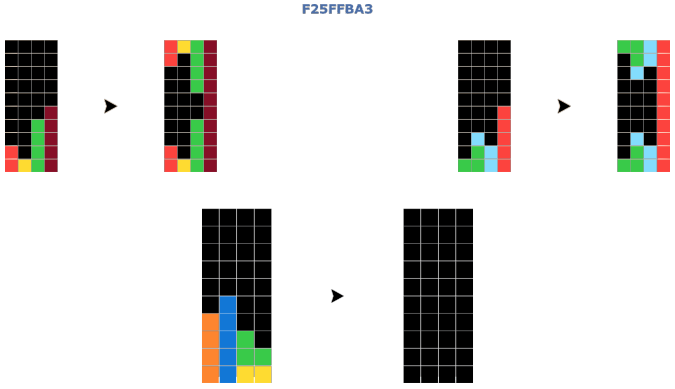


Figure 3.2: Task F25FFBA3 of the ARC dataset. The task was solved in the following way: (lambda (#(lambda (_union $0 (mirrorHor $0))) $0)). [3]

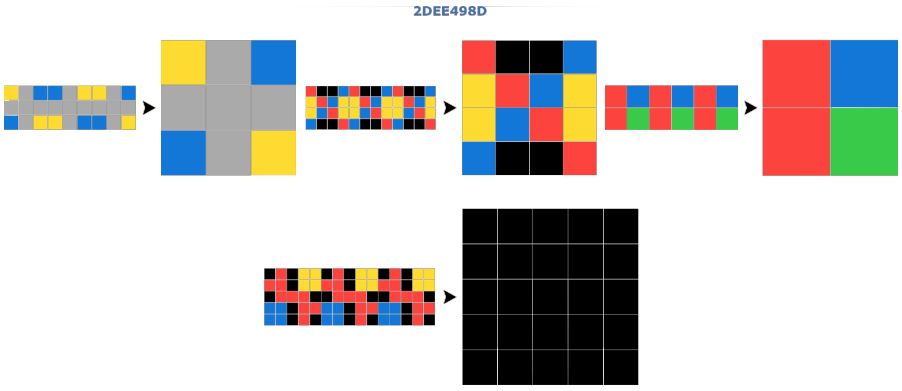


Figure 3.3: Task 2dee498d of the ARC dataset. The task was solved in the following way: (lambda (take_left_half_with_border (addColLast (take_left_half_with_border $0)))). [3]

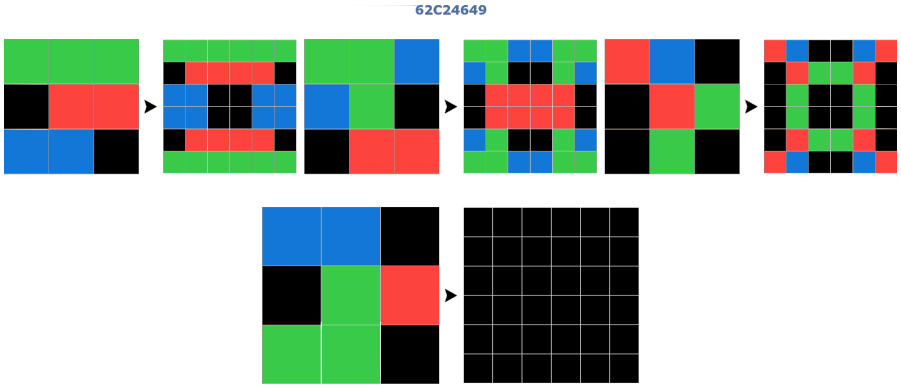


Figure 3.4: Task 62c24649 of the ARC dataset. The task was solved in the following way: (lambda (#(lambda (stack_h $0 (mirrorVer $0))). (#(lambda (stack_v $0 (mirrorHor $0))) $0))). [3]

All

When running DreamCoder with the primitive selection "All" the starting point was 72 primitives. The additional primitives, with respect to the set "Basic ARC", are primitives that could be achieved through composition using only primitives in the set "Basic ARC", but in this case were already given at the start. One example is the primitive "rotate90degCCW" which is the same as "(mirrorHor (transpose $0))". "rotate90degCCW" was given in "All" but not in "Basic ARC". Thanks to more primitives, after the first iteration already 26 training tasks and 2 evaluation tasks were solved. One example, of an evaluation task, that was solved thanks to additional primitives is task 32E9702F. This task can be found in Figure-3.5. To solve this task the primitive "move_left" was used. This was not present in "Basic ARC" but could have been emulated with "(remFirstCol(addColLast $0))".
The newly found primitives were very similar to the previous experiment. The only primitive that was found with this starting set and was not with the set "Basis ARC" was the following:

```
#(lambda (lambda (negative (row_of_size_n $1) (
    first_color_ecountered $0)))) = ''1x1 image of the first
    color encountered in a row-major traversal''
```

This new primitive has for example been used to solve task 445EAB21. This task can be found in Figure-3.6.

After 15 iterations 29 out of 340 training tasks were solved and 4 out of 255 evaluation tasks were solved and the library increased from 72 to 78 primitives.

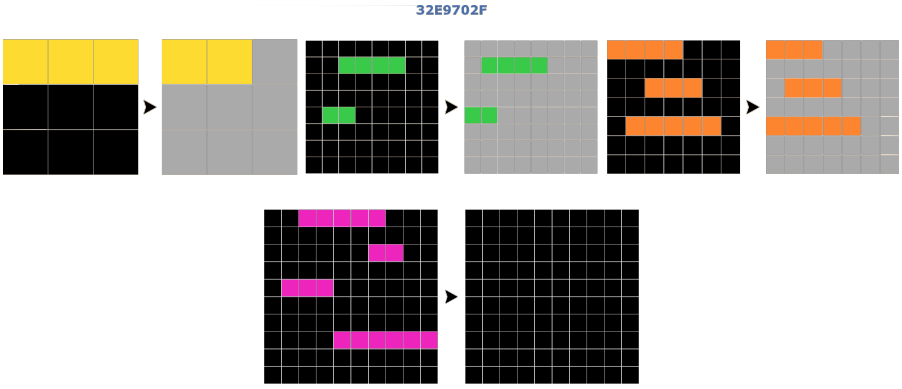


Figure 3.5: Task 32E9702F of the ARC dataset. The task was solved in the following way: ((lambda (negative (move_left $0) 5)). [3]

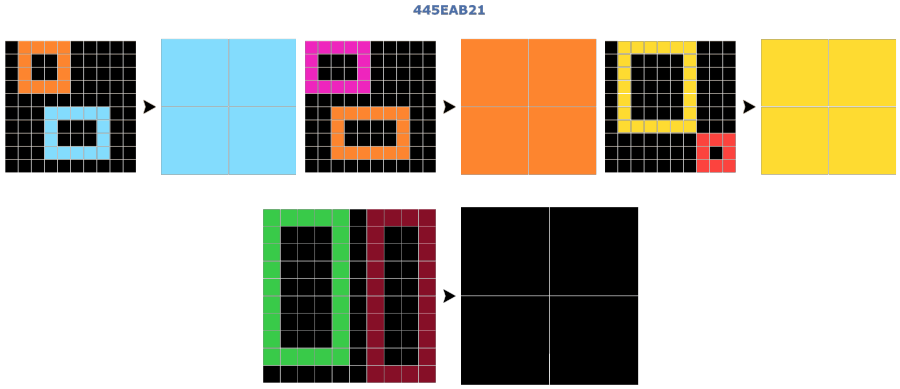


Figure 3.6: Task 445EAB21 of the ARC dataset. The task was solved in the following way: (lambda (flip_down (#(lambda (lambda (negative (row_of_size_n $1) (first_color_ecountered $0)))) 2 (take_bottom_half_with_border $0)))). [3]

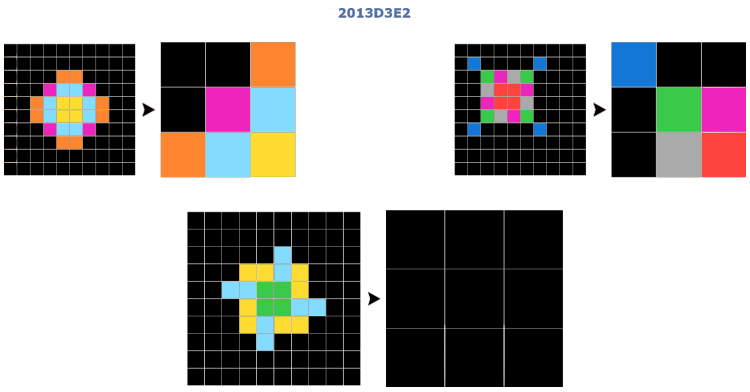


Figure 3.7: Task 2013d3e2 of the ARC dataset. The task was solved in the following way: (lambda (copyPart (#(lambda (remove_bottom_black_rows (remove_left_black_rows (remove_top_black_rows $0)))) $0) 0 5 2 3)). [3]

Selected

As explained in Subsection-2.2.1, for this subset only primitives were used that seemed to be strictly necessary to solve those tasks that were solved by the primitive set "All". The aim was to see if the right function compositions could be made using the given primitives to solve the tasks. The starting point for this experiment was 29 primitives. After 15 iterations 27 out of 340 training tasks were solved and 6 out of 255 evaluation tasks were solved. In total 4 new primitives were found. One primitive is interesting as it was not found in the other experiments. The primitive is the following:

```
#(lambda (remove_bottom_black_rows (remove_left_black_rows (
    remove_top_black_rows $0)))) = ''remove external black rows
    except the right black rows ''
```

Thanks to this primitive, for example, task 2013d3e2 could be solved which was not solved in "Basis ARC" and "All". Task 2013d3e2 can be found in Figure-3.7. The fact that fewer primitives were given in the beginning is interesting as in this way the search space was reduced. This can be seen in the fact that more evaluation tasks and nearly as many training tasks as in the experiment "All" were solved. One evaluation task that was solved in this case and not in the experiment "All" is be03b35f. This task can be found in Figure-3.8.

"All" on unified training and evaluation tasks

As explained in Subsection-2.3.1 for this experiment there were no evaluation tasks and therefore there were only training tasks. These were composed of all training and evaluation tasks together. Interestingly after the first iteration, 33

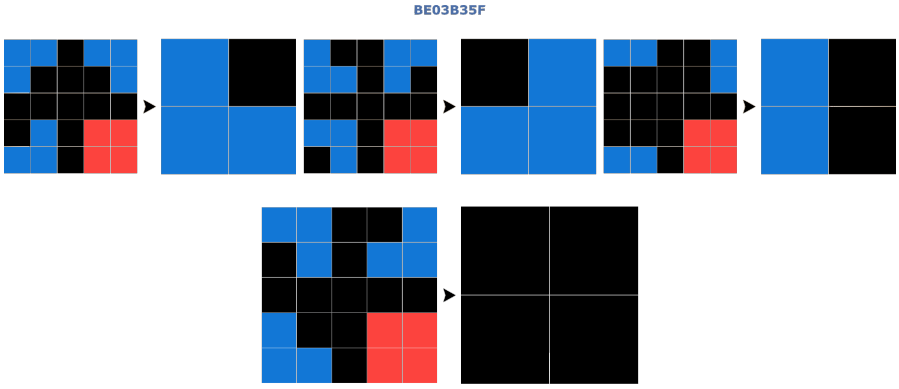


Figure 3.8: Task be03b35f of the ARC dataset. The task was solved in the following way: (lambda (mirrorHor (take_top_half_without_border (transpose (take_top_half_without_border $0))))). [3]

out of 595 training tasks were solved but in the following iterations, no more tasks were solved. This was despite the fact that 6 new primitives were found in the first iteration. No different primitive, with respect to the newly found primitives in the previous experiments, was found. The total number of tasks solved is equal between this experiment and the experiment "All". Interestingly, in this experiment, the tasks bc4146bd and be03b35f were solved, whereas in the experiment "All" those were not solved. At the same time the tasks, d9fac9be and 1cf80156, were solved in this experiment but not in the experiment "All". These four tasks were solved in the following way:

```
Solved by ''''All'' on unified training and evaluation tasks''
    but not by ''All'':
be03b35f:
        (lambda (#(_union (addRowFirst (row_of_size_n 2))) (
            rotate90degCW $0)))
bc4146bd:
        (lambda (stack_h (#(lambda (rotate90degCW (flip_down (
            transpose      $0)))) (#(lambda (rotate90degCW (
            flip_down (transpose $0)))) $0)) $0))
```

```
Solved by ''All'' but not by ''''All'' on unified training and
    evaluation tasks'':
d9fac9be:
        (lambda (negative empty1x1 (first_color_ecountered (#(
            lambda (_union $0 (mirrorHor $0))) $0))))
1cf80156:
        (lambda (remove_bottom_black_rows (
            remove_left_black_rows (remove_right_black_rows (
            remove_top_black_rows $0)))))
```

Name	Solved training tasks / Total training tasks	Solved evaluation tasks / Total evaluation tasks	Number of primitives at the beginning	Number of new found Primitives
Elementary	0/340	0/255	23	0
Basis ARC	25/340	3/255	53	5
All	29/340	4/255	72	6
Selected	27/340	6/255	29	4
All on unified training and evaluation tasks	33/595	-	72	6

Table 3.1: Summary-table of the experiments performed on the ARC dataset.

Different explanations are possible, surely it has to be noted that if on the one hand more training tasks means more data on the other hand it means that it becomes also more difficult to solve the single tasks.

3.1.2 Discussion

From the results of the experiments it can be seen that some tasks were solved, new primitives were found and the newly found primitives were used to solve more difficult tasks. DreamCoder was also able to find some primitives that were missing in the primitives set "Basis ARC" compared to the primitive set "All". For example the primitive "flip down" was found using the primitives "stack vertically" and "mirror horizontally". These results show that DreamCoder works on this domain.
The results found on the ARC dataset, which are summarized in Table-3.1, are in line with the results of the paper from *Banburski et al.* [5]. Using a small set of primitives including "vertical flip", "rotate counter-clockwise", "overlay of two images", "stack vertically" and "take left half" *Banburski et al.* run DreamCoder on a subset of tasks all involving symmetries. They solved 22 out of 36 tasks. Their success rate of solved tasks, number of solved tasks/total number of tasks, is so much higher because they only tackled an accurately chosen selection of tasks. In addition, it is very intuitive to find the right primitives to tackle symmetry problems. This statement is underlined by the fact that many symmetry-related tasks were also solved in the conducted experiments of this work. It can be expected that by using a bigger subset of ARC tasks, and using the same set of primitives as *Banburski et al.* used, no more tasks would be solved.
The paper "On the Measure of Intelligence" by François Chollet [1], in which the ARC dataset is presented, describes what prior knowledge is necessary to solve the ARC tasks. Four different categories are listed: objectness priors, goal-directedness priors, numbers and counting priors, and basic geometry and topology priors. It is broadly explained what is meant by these categories and some examples are given, but it is left to the reader to try to figure out exactly what might fit into this category.
Thinking about these categories, it is no surprise that most of the tasks solved by the used primitives were tasks related to symmetries and basic geometric opera-

	Type of tasks:							
	All 2×2 instances together in one task	Single tasks (1 instance per task)		5er tasks (5 instances per task)				
	Dimension:							
	2×2	3×3	4×4	3×3	4×4	5×5	6×6	7×7
Number of tasks:	1	100	50	60	50	40	30	20

Table 3.2: Summary of task-distribution in task set "General Tasks" of the slider puzzle.

tions and that these tasks were also the ones they focused on in the publication of *Banburski et al.*[5]. This is because primitives regarding basic geometric operations and symmetries are probably intuitively the easiest to think of and also the easiest to implement.

In terms of the ARC dataset, it can be said that some basic primitives are easy to find and that these also solve some tasks. However, to go beyond the simplest primitives and tasks and solve a larger proportion of tasks, a lot of reasoning on the knowledge priors and the implementation of more difficult primitives is needed. Also non-trivial is the question of how generic/low-order the primitives can be and how many are needed as starting point to solve the tasks. In general, solving the majority of tasks will likely require a large number of primitives across all 4 categories of needed prior knowledge. This means that the primitives must be very diverse, as they must cover 4 very different knowledge categories.

3.2 Sliding puzzle

3.2.1 Results

In this subsection, the results of the experiments listed in Table-2.2 are presented. Also in this case the results are presented separately for each experiment. As said in Subsection-2.2.2 the primitives of "Basis sliding puzzle" were extracted from the program in Subsection-B.1.3 which solves the puzzle. The list of the included primitives can be found in the appendix in Subsection-B.2.1. In the later experiments, increasingly high-level primitives were added to facilitate the solution of the puzzle. As explained in Subsection-2.3.2 and summarized in Table-2.2 the first 4 experiments were run on a more general task set whereas the last experiment was run on a more specialized one. In Table-3.2 a summary of the first task set can be found.

Basis sliding puzzle

Running DreamCoder with this set of primitives resulted in the solution of only a few tasks. This was despite the fact that with the primitives of this set the

problem could be solved. Only the 2×2 task and two single 3×3 tasks were solved. In addition, no new primitive was found.

Basis + Help_1

The results running DreamCoder with the primitive set "Basis sliding puzzle" and some additional primitives, "Help_1", were very similar to the experiment "Basis sliding puzzle". In "Help_1" the two primitives "move_to _right_position_row" and "move_to _right_position_col" could be found. More detail about those can be found in Subsection-2.2.2. In this case, only the 2×2 task and one single 3×3 task were solved. Also in this case no new primitives were found. It has to be noted that the primitives in "Help_1" were still quite low-level.

Basis + Help_2

In this experiment, DreamCoder was run with the primitive set "Basis sliding puzzle" and two additional primitives of "Help_2". Those were different from the ones in the previous experiment. These two additional primitives were higher-level than the ones in Help_1. The primitives in "Help_2"' were "solve_one _number_of_a_row" and "solve_one _number_of_a_col". More detail about those can be found in Subsection-2.2.2. These primitives are already very high level as can be seen in Subsection-B.1.4 where the definition of these primitives can be found. They can solve one number of a row/column. Both are function compositions of many many primitives of the "Basis sliding puzzle" set. After the first iteration, 3 new primitives are found. These primitives are:

```
1. #(lambda (solve_one_number_of_a_col (
   solve_one_number_of_a_row $0)))
    = ''Solves the  first number that is not in order of the
       first row that is not ordered followed by solving the
       first number that is not in order of the first column
       that is not ordered''
2. #(lambda (solve_one_number_of_a_col (
   solve_one_number_of_a_col $0)))
    = ''same as above but twice regarding the number of a
       coulmn''
3. #(lambda (move_number_left_with_zero_below (move_slice_down
   $0)))
    = ''move 0 one up and then move the number above the zero
       one to the left''
```

The first primitive was for example used to solve task "p_size_3_74". In Figure-3.9 the solution process of "p_size_3_74" can be found including an intermediate step. The intermediate step is after "solve_one_number_of_a_row" has been applied. It has to be noted that it is a "lucky" coincidence that after applying

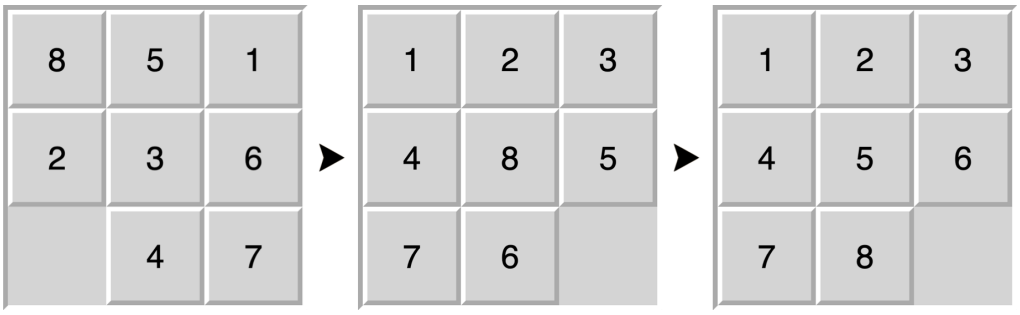


Figure 3.9: Task p_size_3_74 of the slider "General Tasks" dataset. The task was solved in the following way: (lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))). On the left the input instance can be seen. In the middle the result after the application of solve_one_number_of_a_row. On the right the final result can be seen. [4]

"solve_one_number_of_a_row" the whole first row is in order and after applying "solve_one _number_of_a_col" the whole problem is solved. In other tasks, more function applications were needed.

In following iterations more primitives were found, mainly using compositions of "solve_one _number_of_a_row" and "solve_one _number_of_a_col" and newly found primitives.

Thanks to these high-level primitives and the newly discovered primitives, nearly all single instance tasks were solved and in addition all 5er tasks of size 3×3. Nevertheless, no 5er tasks of dimension 4×4 or higher were solved. This means that DreamCoder, with this set of primitives, was able to learn how to solve general instances of size 3×3. Also, many instances of size 4 can be solved but, as the 5er tasks of size 4×4 were not solved, DreamCoder did not learn to solve general instances of this, and bigger sizes. The exact number of tasks and newly found primitives can be found in the summary Table-3.3. In addition in the appendix a complete table, Table-B.5, with the number of solved tasks per iteration and all newly discovered primitives can be found.

Basis + Help_3

"Help_3" contained the most high-level primitives. In this experiment, the primitive set "Basis sliding puzzle" and "Help_3" were given. The set "Help_3" contained the primitives "solve_row" and "solve_col", which solve an entire row/column (see also Subsection-2.2.2 for more detail). With these high-level primitives, DreamCoder was able to solve all tasks involving instances of sizes 2×2, 3×3, and 4×4. Also, it solved 37/40 of the 5er tasks of size 5×5. Interestingly the function compositions used to solve some of the solved 5er tasks of size 5×5

would also have been able to solve the 3 unsolved 5er tasks of size 5×5. One explanation for why those tasks were not solved, could be that DreamCoder was overwhelmed by the size of the primitives, which are very large as can be seen in the appendix in Subsection-B.1.4, or by the size of the tasks. This could also be the explanation for why tasks of higher dimensions were not solved.
In this experiment only 2 new primitives were found:

```
1. #(lambda (solve_2x2 (solve_col (solve_row $0))))
    = ''solve first row that is not in order
    followed by solve first column that is not in order
    followed by solve the 2x2  square at the bottom right
       corner''
2. #(lambda (solve_col (solve_row (solve_row $0))))
    = ''solve first row that is not in order
    followed by solve next row that is not in order
    followed by solve first column that is not in order''
```

The first is a primitive able to solve directly a 3×3 instance of the problem. Alternatively, if the problem is bigger than 3×3, it orders the first row and column that are not ordered and additionally orders the 2×2 square in the bottom right corner. The second primitive orders the first two not ordered rows and the first not ordered column. Applying the first primitive many times one after the other would solve also bigger instances of the puzzle. This underlines the questions raised about the overwhelmingness of DreamCoder with demanding and extensive tasks. A summary of the first four experiments on the slider puzzle can be found in Table-3.3.

Primitive set name:	Type of tasks:								Newly found primitives:
	All 2×2 instances together in one task	Single tasks (1 instance per task)		5er tasks (5 instances per task)					
	Dimension:								
	2×2	3×3	4×4	3×3	4×4	5×5	6×6	7×7	
Basis sliding puzzle:	1/1	2/100	0/50	0/60	0/50	0/40	0/30	0/20	0
Basis + Help_1:	1/1	1/100	0/50	0/60	0/50	25/40	0/30	0/20	0
Basis + Help_2:	1/1	100/100	47/50	60/60	0/50	0/40	0/30	0/20	18
Basis + Help_3:	1/1	100/100	50/50	60/60	50/50	25/40	0/30	0/20	2

Table 3.3: Summary of the number of solved tasks for the different primitive sets on the "General Tasks" set of the slider puzzle.

Task and Primitives engineered

In this experiment, the tasks and primitives were different from the experiments above. There were 45 carefully designed tasks and 31 primitives that should allow solving the tasks. More details about the tasks can be found in Subsubsection-2.1.2, about the primitives in Subsection-2.2.2 and a full list of the primitives

can be found in the appendix in Subsection-B.2.2. The general aim was to make DreamCoder learn to bring the 1 in the upper left corner. This is by learning intermediate steps such as moving the 0 to the bottom right corner or moving the 0 below the 1. These steps were further split into smaller sub-steps. After 10 iterations DreamCoder learned, among other things, to move the 0 to the bottom right corner, to move the 1 away from the last row, and to move the 0 below the 1. However, DreamCoder did not learn to move the 1 to the upper left corner. A total of 9 new primitives were learned. These newly discovered primitives were solutions or intermediate steps of the tasks. For example, the following two newly learned primitives are very interesting:

```
1. #(lambda (#(lambda (repeat_x_left $0 6)) (#(lambda (
   repeat_x_up $0 4)) $0)))
   = ''Move the 0 four times down and 6 times to the right''
2. #(lambda (#(lambda (move_slice_up (repeat_x_down $0  (
   n_movesdown_bring_zero_in_row_below_1 $0)))) (#(lambda (#(
   lambda (#(lambda (repeat_x_right $0 (
   n_movesright_bring_zero_in_col_of_1 $0))) (
   if_1_is_in_last_row_move_one_up_else_no_move_in_order_to_get
    _it_away_from_last_row $0))) (#(lambda (#(lambda (
   repeat_x_left $0 6)) (#(lambda (repeat_x_up $0 4)) $0))) $0)
   )) $0)))
   = ''Move 0 below 1''
```

The first brings the 0 down to the bottom right corner whereas the second moves the 0 below the number 1. DreamCoder did not solve tasks related to the movement of the 1. This could be because more intermediate steps would be needed or better-engineered primitives. A table visualizing the full results of this experiment can be found in the appendix in Subsection-B.4.1.

3.2.2 Discussion

Starting with general, straightforward primitives and generic tasks involving solving different instances of different sizes of the slider problem, it quickly became clear that DreamCoder would not be able to learn how to solve the problem. Therefore, the initial approach had to be reconsidered, towards a careful development of primitives that would allow solving the problem and tasks that would allow learning the necessary intermediate steps to solve the problem.
In the experiments, it can be seen that DreamCoder, although it is theoretically possible, was unable to solve the problem with the "Basis sliding puzzle" set of primitives. As said, it would theoretically be possible, although the program is extremely long, as shown by the program in the appendix in Subsection-B.1.4. This program solves the puzzle using the same primitives and is solving it in a way as DreamCoder could learn it. With the set "Basis sliding puzzle" it was only able to solve the size 2×2 instances of the problem, for which a primitive existed,

and a single instance of size 3×3 that was already nearly solved. This shows that it is not enough to have primitives that can solve the problem in order to learn how to solve it. It was expected that it would not learn to solve the whole problem but nevertheless, the results were not very promising. Also with some additional help, in form of higher-level primitives derived from parts of the solving program, DreamCoder still struggled to solve the puzzle. Help_2 primitives include primitives capable of solving the next number, not in order of a row or a column. These were already quite-high level primitives that require numerous function compositions of the basic primitives. Nevertheless, DreamCoder got to solve only tasks up to size 3×3 when considering the 5er tasks. These 5er tasks are particularly interesting as, to solve them, it is required to be able to solve general instances of the problem. This is due to the fact that the same program needs to solve 5 different instances. This means no general solving skills were learned with those primitives.
Only with the highest level primitives, which include solving the first unordered row/column, which again means that the problem was almost solved, it was able to learn to solve general instances of the problem up to a size of 5×5. However, interestingly, it is not able to solve sizes of 6×6 or more. One possible explanation is that the model is overwhelmed by the size of the primitives or the complexity of the tasks.

Learnings from the findings above are that instead of, or better at the same time as, developing primitives, it is necessary to develop tasks that represent intermediate steps in the solution of the problem. Adding too many general tasks would not help to learn to solve the problem as the number of possible tasks is huge already with a small 3×3 problem and the needed steps to solve most of these small problems is already large. Both aspects enlarge the possible search space to find a general solution to the problem immensely. This means that finding intermediate steps in the solution of the problem and creating tasks that require solving these smaller sub-problems is a necessity. As can be seen in Subsection-3.2.1 in "Task and Primitives engineered" or in Table-B.4, DreamCoder slowly, step by step, learns intermediate steps such as bringing the 0 in the bottom right corner or bringing the 0 below the number 1. The steps in between the tasks, for it to learn, had to be very small, the smaller the better. Furthermore, to create new primitives there also had to be some sort of similarity between the tasks. Despite the careful design of primitives, and tasks to solve the problem of putting the 1 in its place, DreamCoder was unable to learn this ability. The engineering of tasks and primitives requires time and in-depth knowledge of the problem. You have to understand it so thoroughly that you are practically solving it on your own whereas the objective would be to give general primitives, without too much and in-depth reasoning, and straight forwards tasks and leave DreamCoder to do the magic/work.

Chapter 4

Conclusion

From the two different problems that have been studied and the experiments carried out, it can be seen that a large number of tasks are needed which do not differ too much from each other and which are increasing in difficulty. Also, basic primitives are needed, that can solve the problems that are wanted to be solved. This does not seem too far from the expectations of what needs to be provided to a model like DreamCoder for it to be able to solve the task sets. However, finding primitives that are capable of solving a problem and finding the right tasks that allow DreamCoder to learn has proven to be much harder than expected. Although, in theory, one could simply provide a set of primitives that implement a Turing complete language and allow one to find any computable answer, the results in Subsection-3.2.1 in "Basis sliding puzzle" have shown that having the tools to be able to solve something is not enough in order to learn how to solve it. In general, it seems reasonable to assume that some domain knowledge is needed to be able to write the tasks and primitives. However, it should not be the case that to write the tasks and primitives you need to have such a deep knowledge that you have basically solved the problem yourself. The problem is that it is not easy to guess what primitives suffice to solve the problem unless you solve it yourself. In addition, creating tasks of increasing difficulties seems easy when thinking about the slider problem: simply take tasks of increasing size. This has though proven to be far from what is needed to permit DreamCoder to be able to learn to solve the problem. Also here it is difficult to guess the needed intermediate steps without solving the problem yourself. As well in the case of the ARC dataset, where the tasks were already given, the work behind the creation of the primitives turned out to be much bigger than what was imagined. This is because the step from knowing what prior knowledge is needed to knowing which primitives are needed requires a lot of in-depth reasoning.

Returning to the opening discussion about an AI/model's ability to learn like a child, based on prior knowledge, I think DreamCoder shows some promising results, but there is still a long way to go. Especially because DreamCoder struggles with an increased number of tasks and an increased number of primitives. This means that even though the approach of DreamCoder seems very abstract, mixing different domains can lead to difficulties. Therefore it can only tackle one

problem at a time and not use knowledge of other previously solved problems. DreamCoder can not "think outside the box". To conclude, I would say that for humans, learning to solve puzzles and logical challenges, is also ”learning by doing”, there is a sort of artificial generation of tasks and an implicit division into sub-problems. This is a capability also needed in such models in order to solve abstract reasoning problems and logic puzzles based on general primitives and tasks. In DreamCoder, although in the Dreaming phase (see Figure-1.1) new tasks are generated by sampling from the primitive library, this capability is not sufficient. The task generation is a challenging and fundamental step in the solution of such problems which needs further attention in future work.

Bibliography

[1] F. Chollet, "On the measure of intelligence," *CoRR*, vol. abs/1911.01547, 2019. [Online]. Available: http://arxiv.org/abs/1911.01547

[2] K. Ellis, C. Wong, M. I. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. B. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum, "Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning," *CoRR*, vol. abs/2006.08381, 2020. [Online]. Available: https://arxiv.org/abs/2006.08381

[3] F. Chollet. Arc dataset. [Online]. Available: https://volotat.github.io/ARC-Game/?

[4] S. Tatham. Fifteen, from simon tatham's portable puzzle collection. [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/fifteen.html

[5] A. Banburski, A. Gandhi, S. Alford, S. Dandekar, S. Chin, and tomaso a poggio, "Dreaming with {arc}," in *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. [Online]. Available: https://openreview.net/forum?id=-gjy2V1ko6t

[6] C. S. Princeton. Slider puzzle assignement, course 226 in spring 2020. last accessed: 02.04.2022. [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring20/cos226/assignments/8puzzle/specification.php

Appendix A

ARC

A.1 Primitives

A.1.1 Primitives "All"

Primitive name:
flatten
first_row_is_equal
first_col_is_equal
image_equality
is_horizontically_simmetric
is_vertically_simmetric_
8
6
2
5
7
0
1
3
4
9
color_at
list_at_index
count_colored_pixels
first_color_ecountered
flip_down
colormapping_from_c1_to_c2
transpose
take_top_half_with_border
take_right_half_with_border
rotate90degCW

Table A.1 continued from previous page

Primitive name:
mirrorHor
_union
remove_right_black_rows
remove_left_black_rows
remove_bottom_black_rows
duplicate_h
remove_top_black_rows
take_bottom_half_with_border
mirrorVer
rotate90degCCW
take_left_half_with_border
remFirstRow
move_down
move_left
negative
addColLast
copyPart
addColFirst
duplicate_v
identity
remFirstCol
stack_h
tilev2
tileh2
row_of_size_n
addRowLast
empty1x1
addRowFirst
addPixelAllAround
remLastRow
remLastCol
cropByOne
colpixijc
colrow
colcol
tilev3
tileh3
_intersection
move_up
move_right
take_first_row

Table A.1 continued from previous page

Primitive name:
take_first_col
stack_v
color_if_was_c2
fkt_if

Table A.1: Names of primitives in set "All".

Appendix B

Sliding puzzle

B.1 Code

B.1.1 Is Solvable

In the following python code, the definition of the function "is_solvable" can be found. This function returns whether an instance of the sliding puzzle problem is solvable or not. The idea behind the code was taken from [6].

Listing B.1: Python code with function that is able to discover if an instance of the sliding puzzle is solvable or not.

```
def find_zero(im):
  for i in range(len(im)):
    for j in range(len(im[0])):
      if (im[i][j]==0):
        return (i,j)

def count_inversions(lss):
  ls=[item for sublist in lss for item in sublist]
  ls.remove(0)
  count=0
  for i in range(len(ls)-1):
    for j in range(i+1,len(ls)):
      if(ls[i]>ls[j]):
        count+=1
  return count

def is_solvable(lss):
  if(len(lss)%2==0):
    return (0!=((count_inversions(lss)+(find_zero(lss))[0])%2))
  else:
    return (0==(count_inversions(lss)%2))
```

B.1.2 Functions/Primitives for solving slider problem

Listing B.2: Python code with functions needed to solve the slider problem.

```
#coordinates of element n in the image
def find_(im,n):
  row=0
  column=0
  for i in range(len(im)):
    for j in range(len(im[0])):
      if (im[i][j]==n ):
        row=i
        column=j
  return (row, column)

#move a tile right and zero left
def move_right(im1):
  im=copy.deepcopy(im1)
  row=find_(im,0)[0]
  column=find_(im,0)[1]
  if(column==0):
    return(im)
  else:
    temp=im[row][column]
    im[row][column]=im[row][column-1]
    im[row][column-1]=temp
  return im

#move a tile left and zero right
def move_left(im1):
  im=copy.deepcopy(im1)
  row=find_(im,0)[0]
  column=find_(im,0)[1]
  if(column==(len(im[0])-1)):
    return(im)
  else:
    temp=im[row][column]
    im[row][column]=im[row][column+1]
    im[row][column+1]=temp
  return im

#move a tile up and zero down
def move_up(im1):
  im=copy.deepcopy(im1)
  row=find_(im,0)[0]
  column=find_(im,0)[1]
  if(row==(len(im)-1)):
    return(im)
  else:
    temp=im[row][column]
    im[row][column]=im[row+1][column]
    im[row+1][column]=temp
  return im

#move a tile down and zero up
def move_down(im1):
  im=copy.deepcopy(im1)
  row=find_(im,0)[0]
  column=find_(im,0)[1]
  if(row==0):
    return(im)
  else:
    temp=im[row][column]
    im[row][column]=im[row-1][column]
    im[row-1][column]=temp
  return im

#repeat fktion x times
def repeat_x (im1, fkt, x):
  im=copy.deepcopy(im1)
  for i in range(x):
    im=fkt(im)
  return im

#If n is in last row return 1 else 0
def n_of_needed_move_down_to_get_n_away_from_last_row(im1,n):
  rown=find_(im1,n)[0]
  if (rown==(len(im1)-1)):
    return 1
  else: return 0

#number of movements to the right needed to get 0 in the same column as n
```

```
def n_of_needed_move_right_to_get_0_in_same_col_as_n(im1,n):
  coln=find_(im1,n)[1]
  return ((len(im1[0])-1)-coln)

#number of movements down needed to get 0 in the row below the row where n is
def number_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(im1,n):
  rown=find_(im1,n)[0]
  return  ((len(im1)-1)-rown-1)

#sequence of moovements to move the number above the 0 one up, passing with the zero
    ↪ from the right
def move_up_on_right(im1):
  im=copy.deepcopy(im1)
  return move_up((move_right(move_down(move_down(move_left(im))))))

#sequence of moovements to move the number above the 0 one up, passing with the zero
    ↪ from the left
def move_up_on_left(im1):
  im=copy.deepcopy(im1)
  return move_up((move_left(move_down(move_down(move_right(im))))))

#sequence of moovements to move the number above the 0 one to the left
def move_left_with_zero_below(im1):
  im=copy.deepcopy(im1)
  return move_right((move_up(move_left(move_down(move_right(im))))))

#returns move_on_left if the 0 is located in the last column, else returs
    ↪ move_on_right
def choose_move_up_fkt(im1):
  im=copy.deepcopy(im1)
  coln=find_(im,0)[1]
  if(coln==(len(im1[0])-1)):
    return move_up_on_left
  else:
    return move_up_on_right

#list of correct order how to proceed in the solution of the problem
def list_correct_order(im):
  rows=(len(im))
  cols=(len(im[0]))
  correct_list=[0]*rows*cols
  counter=0
  counter_for_first_element_in_colum=1
  for i in range(rows):
    for k in range((rows-i)):
      correct_list[counter]=(i*(rows+1)+1)+k
      counter+=1
    if (counter<(rows*cols)):
      correct_list[counter]=correct_list[counter-1]+counter_for_first_element_in_colum
      counter_for_first_element_in_colum+=1
      counter+=1
      for k in range((rows-i)-2):
        correct_list[counter]=correct_list[counter-1]+rows
        counter+=1
  correct_list[rows*cols-1]=0
  return correct_list

#coordinates where to postition each element, the last 2 elements of each row and
    ↪ column
#represent a special case. The second last element has to be placed as last and the
#last element below or to the right of it (depending if we are talking about a row or
    ↪ col)
def right_position_of_n(im,n):
  counter=1
  for i in range(len(im)):
    for j in range(len(im[0])):
      if(counter==n):

        #special case for last two in row
        if (((len(im[0])-1)-j)<2):
          if(j==(len(im)-2)):
            return(i,j+1)
          else: return (i+1,j)

        if (((len(im)-1)-i)<2):
          if(i==(len(im)-2)):
            return(i+1,j)
          else: return (i,j+1)

        return (i,j)
      counter+=1

#in which row is element n in the end
def row_in_definite_end_order(im,n):
```

```
  ls=[0]*(len(im)*len(im[0]))
  index=0
  for i in range((len(im)*len(im[0]))-1):
    ls[i]=i+1
    if(i+1==n):
      index=i
  row=(index)//(len(im[0]))
  return row

#in which col is element n in the end
def col_in_definite_end_order(im,n):
  n=n+1 #because n was the index and not the actual number
  ls=[0]*(len(im)*len(im[0]))
  index=0
  for i in range((len(im)*len(im[0]))-1):
    ls[i]=i+1
    if(i+1==n):
      index=i
  col=(index-1) % (len(im[0]))
  return col

#check if it is needed to solve a row
def rows_are_ok_up_to_last_2(im):
  counter=1
  for i in range(len(im)):
    for j in range(len(im[0])):
      if ((i<(len(im)-2)) and (im[i][j]!= counter)): return False
      counter+=1
  return True

#check if it is needed to solve a column
def cols_are_ok_up_to_last_2(im):
  counter=1
  for i in range(len(im)):
    for j in range(len(im[0])):
      if ((j<(len(im[0])-2)) and (im[i][j]!= counter)): return False
      counter+=1
  return True


#last number of the row working on
def last_number_of_row_of_first_row_not_in_order(im):
  for i in range(len(im)):
    first_el=len(im)*i+1
    if (im[i][0]!=first_el):
      return (i+1)*len(im)
    for j in range(len(im[0])):
      if (im[i][j]!=first_el+j):
        return (i+1)*len(im)

#last number of the currently column working on
def last_number_of_col_of_first_col_not_in_order(im):
  for i in range(len(im[0])):
    first_el=i+1
    if (im[0][i]!=first_el):
      return i + ((len(im)-1)*len(im))+1
    for j in range(len(im)):
      if (im[j][i]!=first_el+j*(len(im))):
        return i + ((len(im)-1)*len(im))+1

#number of steps down  needed to get 0 in the same row as n
def number_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im1,n):
  rown=find_(im1,n)[0]
  return  ((len(im1)-1)-rown)

#number of steps right needed to get 0 in the same col as n
def n_of_needed_move_right_to_get_0_in_col_rigth_of_n(im1,n):
  coln=find_(im1,n)[1]
  return max(0,((len(im1[0])-1)-coln-1))


#sequence of moovements to move the number above the 0 one down, passing with the zero
    ↪ from the right
def move_down_on_right(im1):
  im=copy.deepcopy(im1)
  return move_right((move_up(move_up(move_left(move_down(im))))))

#sequence of moovements to move the number above the 0 one down, passing with the zero
    ↪ from the left
def move_down_on_left(im1):
  im=copy.deepcopy(im1)
  return move_left((move_up(move_up(move_right(move_down(im))))))
```

```
#returns move_down_on_left if the 0 is located in the last column, else retunrs
    ↪ move_down_on_right
def choose_move_down_fkt(im1):
  im=copy.deepcopy(im1)
  coln=find_(im,0)[1]
  if(coln==(len(im1[0])-1)):
    return move_down_on_left
  else:
    return move_down_on_right

#number of steps right needed to get n to the bottom
def number_of_move_down_sequences_to_get_number_to_bottom(im1,n):
  rown=find_(im1,n)[0]
  return  ((len(im1)-1)-rown)

#sequence of moovements to move the number above the 0 one to the right, passing with
    ↪ the zero from above
def move_right_above(im1):
  im=copy.deepcopy(im1)
  return move_up((move_left(move_left(move_down(move_right(im))))))

#sequence of moovements to move the number above the 0 one to the right, passing with
    ↪ the zero from below
def move_right_below(im1):
  im=copy.deepcopy(im1)
  return move_down((move_left(move_left(move_up(move_right(im))))))

#returns move_right_above if the 0 is located in the last orw, else retunrs
    ↪ move_right_below
def choose_move_right_fkt(im1):
  im=copy.deepcopy(im1)
  rown=find_(im,0)[0]
  if(rown==(len(im1)-1)):
    return move_right_above
  else:
    return move_right_below

#number of steps right needed to get n to the right
def number_of_move_right_sequences_to_get_number_to_right(im1,n):
  coln=find_(im1,n)[1]
  return  max(0,((len(im1[0])-1)-coln))


#check if row is finished
def row_is_finished(im,n):
  first_el=len(im)*n+1
  test=(first_el==im[n][0])
  for i in range(len(im[0])):
    test=test and (im[n][i]==first_el+i)
  return test

#check if numbers of a row are placed in such a way that you can perform the swap
def row_is_ok(im,n):
  first_el=len(im)*n+1
  test=(first_el==im[n][0])
  for i in range(len(im[0])):
    if(i<len(im[0])-2):
      test=test and (im[n][i]==first_el+i)
    else:
      if(i==len(im[0])-2):
        test=test and (im[n][i+1]==first_el+i)
      else: test=test and (im[n+1][i]==first_el+i)
  return test

#check if col is finished
def col_is_finished(im,n):
  first_el=n+1
  test=(first_el==im[0][n])
  for i in range(len(im)):
    test=test and (im[i][n]==first_el+(i*(len(im))))
  return test

#check if numbers of a col are placed in such a way that you can perform the swap
def col_is_ok(im,n):
  first_el=n+1
  test=(first_el==im[0][n])
  for i in range(len(im)):
    if(i<len(im[0])-2):
      test=test and (im[i][n]==first_el+(len(im))*i)
    else:
      if(i==len(im[0])-2):
        test=test and (im[i+1][n]==first_el+(len(im))*i)
      else: test=test and (im[i][n+1]==first_el+(len(im))*i)
  return test
```

```
#find out on wich row we are working
def working_on_row(im):
  for i in range(len(im)):
    first_el=len(im)*i+1
    if (im[i][0]!=first_el):
      return i
    for j in range(len(im[0])):
      if (im[i][j]!=first_el+j):
        return i

#find out on wich col we are working
def working_on_col(im):
  for i in range(len(im[0])):
    first_el=i+1
    if (im[0][i]!=first_el):
      return i
    for j in range(len(im)):
      if (im[j][i]!=first_el+j*(len(im))):
        return i

#find the next number that is not in order
def next_number_not_in_order_row_col(im):
  rows=(len(im))
  cols=(len(im[0]))
  correct_order_in_filling=list_correct_order(im)
  for i in range(rows*cols):
    row=right_position_of_n(im,correct_order_in_filling[i])[0]
    col=right_position_of_n(im,correct_order_in_filling[i])[1]
    if((im[row][col]!=correct_order_in_filling[i])
        and (not(row_is_finished(im,row_in_definite_end_order(im,
            ↪ correct_order_in_filling[i]))))
        and (not(col_is_finished(im,col_in_definite_end_order(im,
            ↪ correct_order_in_filling[i])))):
      return correct_order_in_filling[i]

#number of steps up needed to get n to the right position
def needed_steps_up_to_get_n_in_rigth_position_(im1,n):
  final_row_n=right_position_of_n(im1,n)[0]
  rown=find_(im1,n)[0]
  return (max(0,rown-final_row_n))

#number of steps down needed to get n to the right position
def needed_steps_down_to_get_n_in_rigth_position_(im1,n):
  final_row_n=right_position_of_n(im1,n)[0]
  rown=find_(im1,n)[0]
  if(final_row_n==(len(im1)-1)):
    return (max(0,final_row_n-rown-1))
  return (max(0,final_row_n-rown))

#number of steps left needed to get n to the right position
def needed_steps_to_left_to_get_n_in_rigth_position_(im1,n):
  final_col_n=right_position_of_n(im1,n)[1]
  coln=find_(im1,n)[1]
  return (max(0,coln-final_col_n))

#number of steps right needed to get n to the right position
def needed_steps_to_right_to_get_n_in_rigth_position_(im1,n):
  final_col_n=right_position_of_n(im1,n)[1]
  coln=find_(im1,n)[1]
  return (max(0,final_col_n-coln))

#special move sequence only for col to bring number one down if the right position is
    ↪ the last row
def if_needed_move_up_move_left_to_bring_interested_element_in_last_row(im,n):
  final_row_n=right_position_of_n(im,n)[0]
  rown=find_(im,n)[0]
  if((max(0,final_row_n-rown))==1):
    return move_left(move_down(im))
  else: return im

#move number above 0 one to the right
def move_right_with_zero_below(im1):
  im=copy.deepcopy(im1)
  return move_left((move_up(move_right(move_down(move_left(im))))))

#move number left of 0 one to the left
def move_left_on_last_row(im1):
  im=copy.deepcopy(im1)
  return move_up((move_right(move_right(move_down(move_left(im))))))

#returns move_left_on_last_row if the 0 is located in the last row, else retunrs
    ↪ move_left_with_zero_below
def choose_move_left_fkt(im1,n):
```

```
  im=copy.deepcopy(im1)
  rown=find_(im,n)[0]
  if(rown==(len(im1)-1)):
    return move_left_on_last_row
  else:
    return move_left_with_zero_below

#number of steps up needed to get 0 from the last row to the row yoz are working on
def steps_to_get_up_to_row_where_to_do_the_swap(im1):
  row=working_on_row(im1)
  return (len(im1)-row-1)

#number of steps left needed to get 0 from the last col to the col you are working on
def steps_to_get_left_to_col_where_to_do_the_swap(im1):
  col=working_on_col(im1)
  return (len(im1)-1-(col))

#solve the 2x2 square in bottom right corner, if 0 is there
def solve_2(im1):
  im=copy.deepcopy(im1)
  ul=im[(len(im1)-2)][(len(im1[0])-2)]
  ur=im[(len(im1)-2)][(len(im1[0])-1)]
  dl=im[(len(im1)-1)][(len(im1[0])-2)]
  dr=im[(len(im1)-1)][(len(im1[0])-1)]
  if((dl==(max(ul,ur,dl,dr))) ):
    if ((dr==(min(ul,ur,dl,dr)))):
      return im
    if ((ur==(min(ul,ur,dl,dr)))):
      return move_up(im)
    else:
      return move_up(move_left(im))
  if((dr==(max(ul,ur,dl,dr))) ):
    if ((ul==(min(ul,ur,dl,dr)))):
      return move_left(move_up(im))
    if ((dr==(min(ul,ur,dl,dr)))):
      return move_left(im)
    else:
      return move_left(move_up(move_right(im)))
  if((ur==(max(ul,ur,dl,dr))) ):
    if ((ul==(min(ul,ur,dl,dr)))):
      return move_up(move_left(move_down(move_right(move_up(move_left(im))))))
    if ((dr==(min(ul,ur,dl,dr)))):
      return move_left(move_up(move_right( move_down(im) )))
    else:
      return move_left(move_up(move_right( move_down(move_left(im)))))
  if ((dl==(min(ul,ur,dl,dr)))):
    return (move_up(move_left(move_down(im))))
  if ((dr==(min(ul,ur,dl,dr)))):
    return (move_up(move_left(move_down(move_right(im)))))
  else:
    return (move_up(move_left(move_down(move_right(move_up(im))))))


#return the fkt resulting of the function composition of fkt1 and fkt2
def two_fkt_in_sequence(fkt1,fkt2):
  def fkt3(im):
    return fkt2(fkt1(im))
  return fkt3
```

B.1.3 Solving the slider puzzle

The function ”solve” that solves the slider puzzle is based on the functions “solve_row” and “solve_col”. The function “solve” brings first the 0 down to the right bottom corner. And then applies the functions ”solve_row” and ”solve_col” one after the other 10 times. At this point, only the 2x2 square in the right bottom corner needs to be solved. This is done using the ”solve_2” function. ”Solve_row” and ”solve_col” are very similar, they are based on an inner loop that brings one number at a time to the correct place. The last two elements of the row are brought into the last position and in the last position of the row below. In this way, with a simple series of movements, called the swap, the row

can be completed. The function begins to solve a row only if there is a row left to order that is not one of the last 2. Also in the inner loop numbers are brought to the right position only until the current row is not finished. Two important remarks, before beginning to solve a row/col the last element has to be brought away the furthest possible to not end in a bad position before being able to place it. In addition after each step, the 0 is brought back to the bottom right corner.

Listing B.3: Python code that solves the slider puzzle.

```
def solve_row(im1):
 im=copy.deepcopy(im1)
 #if it happens that all rows except the last 2 are already solved no need to continue
      ↪ (would only distroy the other rows again)
 if(not rows_are_ok_up_to_last_2(im)):

  #last element of the row, because you need to move it away from the top row before
       ↪ continuing
  last_el_of_row_working_on=last_number_of_row_of_first_row_not_in_order(im)

  #move 0 below the interested number
  im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
       ↪ last_el_of_row_working_on))
  im=repeat_x(im,move_down,
       ↪ number_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(im,
       ↪ last_el_of_row_working_on))

  #move the number down
  im=repeat_x(im,choose_move_down_fkt(im),
       ↪ number_of_move_down_sequences_to_get_number_to_bottom(im,
       ↪ last_el_of_row_working_on))

  #move 0 back to right lower corner
  im=repeat_x(im,move_up,8)
  im=repeat_x(im,move_left,8)

  for i in range(10):
   if(not(row_is_ok(im,working_on_row(im)))):
    #next number working on
    next_number=next_number_not_in_order_row_col(im)

    #move the intrested number away from the last row
    im=repeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
         ↪ next_number))
    im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
         ↪ next_number))
    im=move_up(im)

    #move 0 back to right lower corner
    im=repeat_x(im,move_up,8)
    im=repeat_x(im,move_left,8)

    #move 0 below the interested number
    im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
         ↪ next_number))
    im=repeat_x(im,move_down,
         ↪ number_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(im,
         ↪ next_number))

    #move  to right position
    im=repeat_x(im,move_right_with_zero_below,
         ↪ needed_steps_to_right_to_get_n_in_rigth_position_(im,next_number))
    im=repeat_x(im,choose_move_up_fkt(im),needed_steps_up_to_get_n_in_rigth_position_(
         ↪ im,next_number))
    im=repeat_x(im,move_left_with_zero_below,
         ↪ needed_steps_to_left_to_get_n_in_rigth_position_(im,next_number))

    #move 0 back to right lower corner
    im=repeat_x(im,move_up,8)
    im=repeat_x(im,move_left,8)

  #the swap(bring last 2 elements into right postion)
  im=move_right(im)
  im=repeat_x(im,move_down,steps_to_get_up_to_row_where_to_do_the_swap(im))
  im=move_left(im)
  im=move_up(im)

  #move 0 back to right lower corner
  im=repeat_x(im,move_up,8)
  im=repeat_x(im,move_left,8)
```

```
 return im

def solve_col(im1):
  im=copy.deepcopy(im1)

  if(not cols_are_ok_up_to_last_2(im)): #if it happens that all cols except the last 2
      ↪  are already solved no need to continue (would only distroy the other cols
      ↪ again)

    #last element of the col, because you need to move it away from the top row before
        ↪  continuing
    last_el_of_col_working_on=last_number_of_col_of_first_col_not_in_order(im)

    #move 0 to the right of the interested number
    im=repeat_x(im,move_down,
        ↪ number_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im,
        ↪ last_el_of_col_working_on))
    im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        ↪ last_el_of_col_working_on))

    #move the number to the right
    im=repeat_x(im,choose_move_right_fkt(im),
        ↪ number_of_move_right_sequences_to_get_number_to_right(im,
        ↪ last_el_of_col_working_on))

    #move 0 back to right lower corner
    im=repeat_x(im,move_up,8)
    im=repeat_x(im,move_left,8)

    for i in range(10):
      if(not(col_is_ok(im,working_on_col(im)))):
        #next number working on
        next_number=next_number_not_in_order_row_col(im) #next_number_to_work_on

        #move the intrested number away from the last row
        im=repeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
            ↪ next_number))
        im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
            ↪ next_number))
        im=move_up(im)

        #move 0 back to right lower corner
        im=repeat_x(im,move_up,8)
        im=repeat_x(im,move_left,8)

        #move 0 below the interested number
        im=repeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
            ↪ next_number))
        im=repeat_x(im,move_down,
            ↪ number_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
            ↪ im,next_number))

        #move  to right position
        im=repeat_x(im,choose_move_up_fkt(im),
            ↪ needed_steps_up_to_get_n_in_rigth_position_(im,next_number))
        im=repeat_x(im,choose_move_down_fkt(im),
            ↪ needed_steps_down_to_get_n_in_rigth_position_(im,next_number))
        im=repeat_x(im,choose_move_left_fkt(im,next_number),
            ↪ needed_steps_to_left_to_get_n_in_rigth_position_(im,next_number))
        im=if_needed_move_up_move_left_to_bring_interested_element_in_last_row(im,
            ↪ next_number)

        #move 0 back to right lower corner
        im=repeat_x(im,move_up,8)
        im=repeat_x(im,move_left,8)

    #the swap (bring last 2 elements into rigjt postion)
    im=move_down(im)
    im=repeat_x(im,move_right,steps_to_get_left_to_col_where_to_do_the_swap(im))
    im=move_up(im)
    im=move_right(im)

    #move 0 back to right lower corner
    im=repeat_x(im,move_up,8)
    im=repeat_x(im,move_left,8)
  return im

def solve(im1):
  im=copy.deepcopy(im1)
  #move 0 to bottom right
  im=repeat_x(im,move_up,8)
  im=repeat_x(im,move_left,8)
  #solve row then col until 2*2 square is left down right
```

```
  im=repeat_x (im, two_fkt_in_sequence(solve_row,solve_col), 10)
  return (solve_2(im))
```

B.1.4 Solving the slider problem like DreamCoder

Listing B.4: Python code that solves the slider problem as DreamCoder could.

```
def function_composition(fkt1,fkt2):
  def fkt3(im):
    return fkt2(fkt1(im))
  return fkt3

def if_true_i1_else_i2(booll,im1,im2):
  if(booll):
    return im1
  else:  return im2

def function_application_x (im1, fkt, x):
  im=copy.deepcopy(im1)
  for i in range(x):
    im=fkt(im)
  return im

def move_to_right_position_col(im, next_number):
  im=repeat_x(im,choose_move_up_fkt(im),needed_steps_up_to_get_n_in_rigth_position_(im
      none,next_number))
  im=repeat_x(im,choose_move_down_fkt(im),
      noneneeded_steps_down_to_get_n_in_rigth_position_(im,next_number))
  im=repeat_x(im,choose_move_left_fkt(im,next_number),
      noneneeded_steps_to_left_to_get_n_in_rigth_position_(im,next_number))
  im=if_needed_move_up_move_left_to_bring_interested_element_in_last_row(im,
      nonenext_number)
  return im

def move_to_right_position_row(im, next_number):
  im=repeat_x(im,move_right_with_zero_below,
      noneneeded_steps_to_right_to_get_n_in_rigth_position_(im,next_number))
  im=repeat_x(im,choose_move_up_fkt(im),needed_steps_up_to_get_n_in_rigth_position_(im
      none,next_number))
  im=repeat_x(im,move_left_with_zero_below,
      noneneeded_steps_to_left_to_get_n_in_rigth_position_(im,next_number))
  return im


def solve_one_number_of_a_col(im):
  return if_true_i1_else_i2((col_is_ok(im,working_on_col(im))),im,repeat_x(repeat_x(
      nonemove_to_right_position_col(repeat_x(repeat_x(repeat_x(repeat_x(move_up(
      nonerepeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))))),move_up,8),move_left,8),
next_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,
     nonemove_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
     nonenext_number_not_in_order_row_col(im))),move_right,
     nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
     nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
     nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
     nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
     nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8))),
     nonemove_down,
     nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(repeat_x
     none(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,move_down,
     nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
     nonenext_number_not_in_order_row_col(im))),move_right,
```

```
    nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
    nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
    nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),move_right,
    nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(repeat_x(move_up(
    nonerepeat_x(repeat_x(im,move_down,
    nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),move_right,
    nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
    nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
    nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
    nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x(
    noneim,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),move_right,
    nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
    nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
    nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8)))),
    nonenext_number_not_in_order_row_col(repeat_x(repeat_x(repeat_x(move_up(repeat_x(
    nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),move_right,
    nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
    nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
    nonenext_number_not_in_order_row_col(im))),
  next_number_not_in_order_row_col(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))))))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
      nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
      none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8))))),
      nonenext_number_not_in_order_row_col(repeat_x(repeat_x(repeat_x(repeat_x(move_up
      none(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
      nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
      none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
  move_down,number_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
```

```
        nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
        none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8)))),
    next_number_not_in_order_row_col(repeat_x(repeat_x(repeat_x(move_up(repeat_x(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
        nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
        nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),
    next_number_not_in_order_row_col(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
        none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im)))))),move_up,8),move_left,8)))))))),
        nonemove_up,8),move_left,8))



def solve_one_number_of_a_row(im):
  return if_true_i1_else_i2((row_is_ok(im,working_on_row(im))),im,repeat_x(repeat_x(
        nonemove_to_right_position_row(repeat_x(repeat_x(repeat_x(repeat_x(move_up(
        nonerepeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
        nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
        nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
        none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),
    next_number_not_in_order_row_col(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8)))),
        nonemove_down,
        nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
        nonerepeat_x(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),
    next_number_not_in_order_row_col(repeat_x(im,move_down,
```

```
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonenext_number_not_in_order_row_col(repeat_x(move_up(repeat_x(repeat_x
      none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),
  next_number_not_in_order_row_col(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8)) )),
      nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x
      none(repeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
  next_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,
      nonemove_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),8) ))) ))
      none,next_number_not_in_order_row_col(repeat_x(repeat_x(repeat_x(repeat_x(
      nonemove_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),
  next_number_not_in_order_row_col(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
      none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8)) ),
      nonemove_down,
      nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(repeat_x(repeat_x(move_up(repeat_x(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
      nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
      nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
      nonenext_number_not_in_order_row_col(im) ))) ))),move_up,8),move_left,8),
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
      nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
```

```
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
   n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )) )),move_up,8),move_left,8),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
        none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )) )),move_up,8),move_left,8) )),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(repeat_x(move_up(repeat_x
        none(repeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) ),
   next_number_not_in_order_row_col(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )) )),move_up,8),move_left,8),
        nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(
        nonerepeat_x(move_up(repeat_x(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )) )),move_up,8),move_left,8),
        nonenext_number_not_in_order_row_col(repeat_x(repeat_x(move_up(repeat_x(repeat_x
        none(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(repeat_x(im,move_down,
        nonen_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )),next_number_not_in_order_row_col(
        nonerepeat_x(im,move_down,n_of_needed_move_down_to_get_n_away_from_last_row(im,
        nonenext_number_not_in_order_row_col(im) )) )),move_up,8),move_left,8) )) ))
        none) ),move_up,8),move_left,8))


def solve_row(im):
 return (if_true_i1_else_i2(rows_are_ok_up_to_last_2(im),im,repeat_x(repeat_x(move_up(
      nonemove_left(repeat_x(move_right(repeat_x(repeat_x(repeat_x(repeat_x(repeat_x(
      nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
      nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),
      nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im)))))),choose_move_down_fkt(
      nonerepeat_x(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
      nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),
      nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im)))))),
      nonenumber_of_move_down_sequences_to_get_number_to_bottom(repeat_x(repeat_x(im,
      nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
      nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),
 last_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im)))))),
      nonelast_number_of_row_of_first_row_not_in_order(repeat_x(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
      nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
      nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im))),
      nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
      nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
      nonelast_number_of_row_of_first_row_not_in_order(im)))))))),move_up,8),move_left
      none,8),solve_one_number_of_a_row,10)),move_down,
      nonesteps_to_get_up_to_row_where_to_do_the_swap(move_right(repeat_x(repeat_x(
```

```
        nonerepeat_x(repeat_x(repeat_x(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
        nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
        nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),
        nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))))),choose_move_down_fkt(
        nonerepeat_x(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
        nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
        nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),
    last_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im)))))),
        nonenumber_of_move_down_sequences_to_get_number_to_bottom(repeat_x(repeat_x(im,
        nonemove_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
        nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
        nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),
        nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))))),
        nonelast_number_of_row_of_first_row_not_in_order(repeat_x(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),move_down,
        nonenumber_of_move_down_to_bring_zero_in_row_below_the_intereseted_number(
        nonerepeat_x(im,move_right,n_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im))),
        nonelast_number_of_row_of_first_row_not_in_order(repeat_x(im,move_right,
        nonen_of_needed_move_right_to_get_0_in_same_col_as_n(im,
        nonelast_number_of_row_of_first_row_not_in_order(im)))))))),move_up,8),move_left
        none,8),solve_one_number_of_a_row,10))))),move_up,8),move_left,8)))


def solve_col(im):
    return if_true_i1_else_i2(cols_are_ok_up_to_last_2(im),im,repeat_x(repeat_x(
        nonemove_right(move_up(repeat_x(move_down(repeat_x(repeat_x(repeat_x(repeat_x(
        nonerepeat_x(repeat_x(im,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im))),choose_move_right_fkt(
        nonerepeat_x(repeat_x(im,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im)))),
        nonenumber_of_move_right_sequences_to_get_number_to_right(repeat_x(repeat_x(im
        none,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im))),
    last_number_of_col_of_first_col_not_in_order(repeat_x(repeat_x(im,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im)))))),move_up,8),move_left
        none,8),solve_one_number_of_a_col,10)),move_right,
        nonesteps_to_get_left_to_col_where_to_do_the_swap(move_down(repeat_x(repeat_x(
        nonerepeat_x(repeat_x(repeat_x(repeat_x(im,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im))),
    choose_move_right_fkt(repeat_x(repeat_x(im,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im)))),
        nonenumber_of_move_right_sequences_to_get_number_to_right(repeat_x(repeat_x(im
        none,move_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im))),
        nonelast_number_of_col_of_first_col_not_in_order(repeat_x(repeat_x(im,
        nonemove_down,
        nonenumber_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number(im
        none,last_number_of_col_of_first_col_not_in_order(im))),move_right,
```

```
        nonen_of_needed_move_right_to_get_0_in_col_rigth_of_n(im,
        nonelast_number_of_col_of_first_col_not_in_order(im))))),move_up,8),move_left
        none,8),solve_one_number_of_a_col,10))))),move_up,8),move_left,8))

def solve(im):
  return (solve_2(repeat_x (repeat_x(repeat_x(im,move_up,8),move_left,8),
       nonefunction_composition(solve_row,solve_col), 10)))
```

B.2 Primitives

B.2.1 Primitives "Basis sliding puzzle"

Primitive name:
rows_are_ok_up_to_last_2
cols_are_ok_up_to_last_2
row_is_ok
col_is_ok
6
4
5
7
8
9
0
1
2
3
n_of_needed_move_down_to_get_n_away_from_last_row
n_movesright_bring_zero_in_col_of_n
n_movesdown_bring_zero_in_row_below_n
last_number_of_row_of_first_row_not_in_order
last_number_of_col_of_first_col_not_in_order
number_of_move_down_to_bring_zero_in_same_row_as_the_intereseted_number
n_of_needed_move_right_to_get_0_in_col_rigth_of_n
number_of_move_down_sequences_to_get_number_to_bottom
number_of_move_right_sequences_to_get_number_to_right
working_on_row
working_on_col
next_number_not_in_order_row_col
needed_steps_up_to_get_n_in_rigth_position
needed_steps_down_to_get_n_in_rigth_position
needed_steps_to_left_to_get_n_in_rigth_position
needed_steps_to_right_to_get_n_in_rigth_position
steps_to_get_up_to_row_where_to_do_the_swap
steps_to_get_left_to_col_where_to_do_the_swap
move_slice_down
solve_2x2
move_number_left_with_zero_below
move_right_with_zero_below
move_slice_right
choose_right_move_up_fkt
move_slice_left
move_slice_up
choose_move_down_fkt
choose_move_right_fkt
repeat_x
if_needed_move_up_move_left_to_bring_interested_element_in_last_row
choose_move_left_fkt
function_composition
if_true_i1_else_i2

Table B.1: Names of primitives in set "Basis sliding puzzle"

B.2.2 Primitives "Tasks and Primitives engineered"

Primitive name:
0
1
2
3
4
5
6
7
8
9
n_move_left_to_bring_1_in_right_col
n_move_up_to_bring_1_in_right_row
n_movesright_bring_zero_in_col_of_1
n_movesdown_bring_zero_in_row_below_1
if_1_is_in_first_row_then_return_0_else_1
if_1_is_in_first_col_then_return_0_else_1
move_slice_down
move_slice_left
move_slice_up
move_up_on_left_side
if_1_is_in_last_row_move_one_up_else_no_move_in_order_to_get_it_away_from_last_row
move_slice_right
repeat_x
move_up_on_right_side
move_number_left_with_zero_below
choose_right_move_up_fkt
repeat_x_down
repeat_x_up
repeat_x_left
repeat_x_right

Table B.2: Names of primitives in set "Tasks and Primitives engineered"

B.3 Tasks

B.3.1 Engineered tasks

In the following Table-B.3 all tasks of the second set of tasks of the sliding puzzle are listed:

Task name:
p_move_0_to_right
p_move_0_to_bottom
p_move_0_to_bottom_right
p_move_0_to_right2
p_move_0_to_bottom2
p_move_0_to_bottom_right2
p_move_1_away_from_last_row_starting_with_0_bottom_right
p_move_1_away_from_last_row
p_move_1_away_from_last_row_and_0_to_last_row
p_move_zero_in_same_colum_as_1
p_move_zero_in_row_below_the_row_1
p_move_zero_in_row_below_the_row_of_1
p_move_0_same_col_as_1_but_0_was_not_in_bottomom_right_corner_when_tarting
p_move_zero_in_row_below_the_row_of_1_but_0_was_not_in_bottomom_right_corner_when_starting
p_move_zero_below_one_1_but_0_was_not_in_bottomom_right_corner_when_starting
p_move_0_same_col_as_1_but_1_could_be_in_last_row
p_move_zero_in_row_below_the_row_of_1_but_1_could_be_in_last_row
p_move_zero_below_one_1_but_1_could_be_in_last_row
p_move_0_same_col_as_1_but_0_was_not_in_bottomom_right_corner_when_starting_and_but_1_could_be_in_last_row
p_move_zero_in_row_below_the_row_of_1_but_0_was_not_in_bottomom_right_corner_when_starting_and_but_1_could_be_in_last_row
p_move_zero_below_one_1_but_0_was_not_in_bottomom_right_corner_when_starting_and_but_1_could_be_in_last_row
p_move_zero_in_same_colum_as_1_2
p_move_zero_in_row_below_the_row_1_2
p_move_zero_in_row_below_the_row_of_1_2
p_move_0_same_col_as_1_but_0_was_not_in_bottomom_right_corner_when_starting_2
p_move_zero_in_row_below_the_row_of_1_but_0_was_not_in_bottomom_rigt_corner_when_starting_2
p_move_zero_below_one_1_but_0_was_not_in_bottomom_right_corner_when_starting_2
p_move_0_same_col_as_1_but_1_could_be_in_last_row_2
p_move_zero_in_row_below_the_row_of_1_but_1_could_be_in_last_row_2
p_move_zero_below_one_1_but_1_could_be_in_last_row_2
p_move_0_same_col_as_1_but_0_was_not_in_bottomom_right_corner_when_starting_and_but_1_could_be_in_last_row_2
p_move_zero_in_row_below_the_row_of_1_but_0_was_not_in_bottomom_rigt_corner_when_starting_and_but_1_could_be_in_last_row_2
p_move_zero_below_one_1_but_0_was_not_in_bottomom_right_corner_when_starting_and_but_1_could_be_in_last_row_2
p_move_zero_above_1_if_1_is_in_last_col
p_move_zero_above_1_if_1_is_in_last_col_2
p_move_zero_above_1_if_1_is_in_last_col_zero_everywhere
p_move_zero_above_1_if_1_is_in_last_col_zero_everywhere_2
p_move_0_below_1
p_move_1_up_and_0_below_1_not_in_last_row_n1lr
p_move_1_left_and_0_below_n1lr
p_move_1_corner_up_left_n1lr
p_move_1_up_and_0_below_2_n1lr
p_move_1_left_and_0_below_2_n1lr
p_move_1_corner_up_left_2_n1lr
p_move_1_corner_up_left

Table B.3: In this table all names of the tasks in the set "Engineered tasks" can be found.

B.4 Results

B.4.1 Results of "Task and Primitives engineered"

Iteration:	Solved tasks:	Number of primitives*:	New primitives discovered:
0	-	31	
1	7/45	34	#(lambda (repeat_x_right $0 (n_movesright_bring_zero_in_col_of_1 $0))) = " Bring 0 in the same column as the number 1" #(lambda (repeat_x_up $0 4)) = " Move the zero down 4 times" #(lambda (repeat_x_left $0 6)) = " Move the zero right 6 times"
2	21/45	39	#(lambda (#(lambda (repeat_x_left $0 6)) (#(lambda (repeat_x_up $0 4)) $0))) ="move 0 4 time down and 6 times right" #(lambda (move_slice_up (repeat_x_down $0 (n_movesdown_bring_zero_in_row_below_1 $0)))) ="move down 0 until it is in the row below the row of 1" #(lambda (#(lambda (repeat_x_right $0 (n_movesright_bring_zero_in_col_of_1 $0))) (#(lambda (#(lambda (repeat_x_left $0 6)) (#(lambda (repeat_x_up $0 4)) $0))) $0))) ="move 0 4 time down and 6 times right and bring zero in column of 1" #(lambda (#(lambda (repeat_x_right $0 (n_movesright_bring_zero_in_col_of_1 $0))) (if_1_is_in_last_row_move_one_up_else_no_move_in_order_to_get_it_away_from_last_row $0))) ="if 1 is in last row move 0 one up an in same column as 1, else move 0 in same column as 1"
3	38/45	40	#(lambda (#(lambda (#(lambda (repeat_x_right $0 (n_movesright_bring_zero_in_col_of_1 $0))) (if_1_is_in_last_row_move_one_up_else_no_move_in_order_to_get_it_away_from_last_row $0))) (#(lambda (#(lambda (repeat_x_left $0 6)) (#(lambda (repeat_x_up $0 4)) $0))) $0))) ="move zero 4 down and then 4 right followed by if 1 is in last row move 0 one up an in same column as 1, else move 0 in same column as 1" #(lambda (#(lambda (move_slice_up (repeat_x_down $0 (n_movesdown_bring_zero_in_row_below_1 $0)))) (#(lambda (#(lambda (#(lambda (repeat_x_right $0 (n_movesright_bring_zero_in_col_of_1 $0))) (if_1_is_in_last_row_move_one_up_else_no_move_in_order_to_get_it_away_from_last_row $0))) (#(lambda (#(lambda (repeat_x_left $0 6)) (#(lambda (repeat_x_up $0 4)) $0))) $0))) $0))) ="move 0 below 1"
...			
10	38/45	40	

Table B.4: In this table the results of running DreamCoder on the engineered slider puzzle dataset using the appositely engineered primitive set can be found. DreamCoder was run for 10 iterations and with a recognition timeout of 30'000 seconds. The number of solved training tasks and the total number of primitives and the newly found primitives are listed after each iteration. For each newly discovered primitive, the action of the primitive is described. After iteration 3 no more changes were recorded and these results are therefore omitted. The last row represents the final result.

B.4.2 Results using primitive set "Basis + Help_2"

Iteration:	Solved training task: all 2 × 2 tasks together	Solved single tasks of size: 3 × 3	Solved single tasks of size: 4 × 4	Solved 5er tasks of size: 3 × 3	Solved 5er tasks of size: 4 × 4	Solved 5er tasks of size: 5 × 5	Solved 5er tasks of size: 6 × 6	Solved 5er tasks of size: 7 × 7	Number of primitives:	New primitives discovered:
0	-	-	-	-	-	-	-	-	50	-
1	1/1	30/100	0/50	0/60	0/50	0/40	0/30	0/20	53	#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) = " solve first number thta is not in order of the first row that is not in order followed by solve first number that is not in order of the first column that is not in order" #(lambda (solve_one_number_of_a_col (solve_one_number_of_a_col $0))) = " same as above but twice regarding the number of a coulmn" #(lambda (move_number_left_with_zero_below (move_slice_down $0))) = " move 0 one up and then move the number above the zero one to the left"
2	1/1	92/100	3/50	0/60	0/50	0/40	0/30	0/20	56	#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) #(lambda (repeat_x $0 (lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0))))) #(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_col $0))) (move_slice_right (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) $0)))))
3	1/1	100/100	29/50	3/60	0/50	0/40	0/30	0/20	60	#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) #(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_col $0))) (move_slice_right (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) $0))))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) #(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0))))) $0 6)) #(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0))))) $0 7))
4	1/1	100/100	38/50	41/60	0/50	0/40	0/30	0/20	65	#(lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0)))) $0))) #(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0))) #(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_col $0))) (move_slice_right (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) $0))))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0)))) #(lambda (repeat_x $0 (lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0)) 9)) #(lambda (#(lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) $0)))
5	1/1	100/100	42/50	60/60	0/50	0/40	0/30	0/20	66	#(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0)) 9)) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_col $0))) (move_slice_right (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) $0))))) $0)))
6	1/1	100/100	36/50	60/60	0/50	0/40	0/30	0/20	68	#(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0)) 9)) (#(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0))))) $0 6)) $0))) #(lambda (#(lambda (repeat_x $0 (lambda (#(lambda (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) (#(lambda (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (solve_one_number_of_a_col (solve_one_number_of_a_row $0))) (#(lambda (move_number_left_with_zero_below (move_slice_down $0))) $0)))) $0))) $0)) 9)) (move_number_left_with_zero_below $0)))
7	1/1	100/100	31/50	60/60	0/50	0/40	0/30	0/20	68	-
8	1/1	100/100	46/50	60/60	0/50	0/40	0/30	0/20	68	-
9	1/1	100/100	46/50	60/60	0/50	0/40	0/30	0/20	68	-
10	1/1	100/100	47/50	60/60	0/50	0/40	0/30	0/20	68	-

Table B.5: In this table the results of running DreamCoder on the general slider puzzle dataset using the primitive set "Basis + Help_2" can be found. DreamCoder was run for 10 iterations and with a recognition timeout of 30'000. The number of solved training tasks is listed separately according to the different dimensions and the number of problem instances found per task. The number of solved training tasks per category and the total number of primitives and the newly found primitives are listed after each iteration. The last row represents the final result.