Opleiding Informatica

Generating and Solving

PushPush Puzzles

Liva van der Velden

Supervisors:
Walter Kosters & Luc Edixhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 05/07/2023

Abstract

This paper investigates PushPush puzzles. The puzzle consists of a robot that aims to reach the target with blocks obstructing its way. The robot can push a block out of its way, after which it will keep sliding until it hits another block or the edge of the board. We discuss how these puzzles can be generated and solved, and what could be a fitting difficulty measure. All puzzles of a small size are generated and analysed. We also investigate how neural networks could be used to determine the solvability, the minimum number of moves to solve, or the difficulty level of a puzzle.

Contents

1 Introduction

In the field of game complexity there are many interesting questions that can be asked. How can we determine the complexity? How can we design and solve puzzles? How can we determine the difficulty of a puzzle?
For many puzzles these questions have been extensively researched, for example for Rush Hour or Lunar Lockout. Such games and their research will be discussed in Section 3.

For this project we investigated PushPush puzzles. This is a puzzle in which a robot has to reach a target which it can get to by moving blocks. The robot can push blocks to move them, where the blocks slide as far as they can. An example board is shown in Figure 1. The solution of this puzzle requires six block pushes.
This puzzle has not been researched much before, so there are still many unanswered questions. One thing that we know is that the puzzle is PSPACE-complete [Demaine et al., 2004]. This makes other questions regarding the complexity more interesting since they become harder to answer.
In this project we investigate how to generate and solve PushPush puzzles and determine the difficulty, while (efficiently) computing the search space.

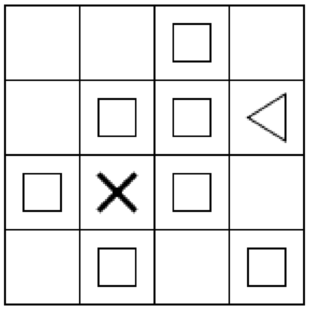


Figure 1: An example of a PushPush puzzle. The triangle represents the robot, the $\times$ is the target and the squares are blocks.

1.1 Thesis overview

This paper is ordered in the following way. First, in Section 2, we set some definitions to clarify the research question and precisely define how we approach the research question to get an answer. In Section 3 we consider other work that has been done in the same field of research. Section 4 discusses how PushPush puzzles can be generated and then solved through algorithms, and how this can be done more efficiently when considering many puzzles together. In Section 5 we explore a possible difficulty measure and in Section 6 we apply neural networks to determine several properties of a puzzle. Finally, Section 7 contains the conclusion and discussion. In this section we summarise our findings and give a future outlook of the research that can be done as a follow-up.

This paper is a bachelor thesis for the Data Science and Artificial Intelligence program at the Leiden Institute for Advanced Computer Science (LIACS), Leiden University, supervised by Walter Kosters and Luc Edixhoven.

2 Definitions and problem description

The game PushPush-1 is played on a (usually) square grid of cells. From now on we will write PushPush to indicate PushPush-1. There are two types of objects on the grid: a robot and blocks.
There is one robot, which takes up one cell and can move around over the empty cells immediately horizontally and vertically adjacent. The other object type is a block. There can be multiple blocks in the grid which take up one cell per block. These blocks keep the robot from passing over the cell. The blocks can be moved. The robot can push horizontally and vertically adjacent blocks if there is an empty cell on the other side of the block. If the robot can get behind a block, so the cell behind it is empty, and the cell in front of the block is empty as well, the robot can push the block forward. Then the block will slide as far as it can until it hits another block or the edge of the grid. There is one cell in the grid that is the target, which is where the robot is aiming to get to. This cell is still available for all objects and the target does not move. The target position differs from the start position of the robot.
The rules are demonstrated in Figure 2.

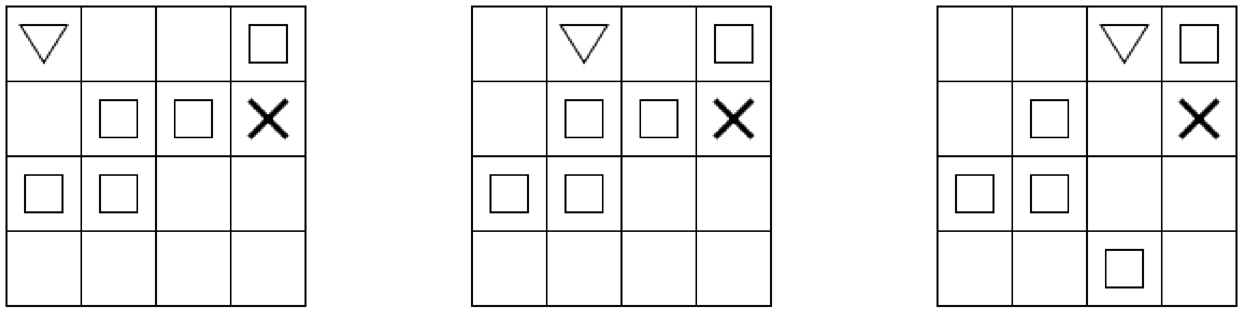

(a) The initial state of the board of a PushPush puzzle.
(b) The robot attempts to push the block below, which it cannot.
(c) The robot pushed the block below, creating a path to the target.

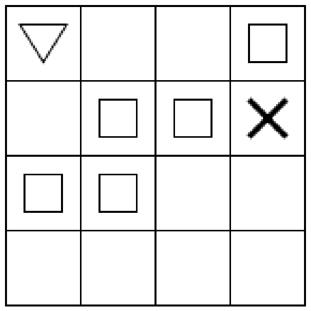
Figure 2: A demonstration of the rules of PushPush. The triangle represents the robot, pointing in the direction it is pushing. The squares represent blocks and the cross represents the target.

There are other versions of the game in which the number of blocks that can be pushed at once is varied. In PushPush-k with $k \geq 1$ the robot can push k blocks at a time. In PushPush-$*$ the robot can push an unlimited number of blocks at the same time.
The principle of PushPush-k is visualised in Figure 3.

In this paper we represent the cells of the grid with squares. In these squares the robot is represented by a triangle, the target by an $\times$ and blocks by smaller squares.

2.1 Problem description

The research question is "How can we generate and solve PushPush puzzles and determine the difficulty, while (efficiently) computing the search space?"
Generating puzzles is in general not hard, but creating "interesting" puzzles is a different story. In


(a) The initial state of the board of a PushPush−k puzzle, the same as in Figure 2a.

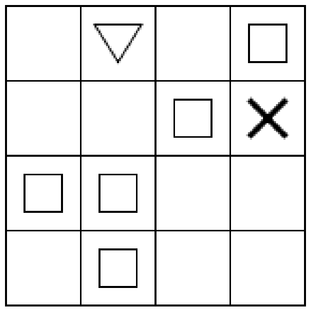

(b) If $k \geq 2$ in PushPush-k, the robot can push both blocks below in one move.

Figure 3: A visualisation of another type of PushPush.

this case an "interesting" puzzle is defined as being solvable and not trivial (trivial meaning solvable without having any options to make a mistake). As noted before, the solvability of a puzzle is a PSPACE-complete problem, which was proven in [Demaine et al., 2004]. This makes it challenging to create solvable puzzles. To determine whether the puzzle that was generated is solvable and interesting the puzzle has to be solved.
Solving PushPush puzzles is usually not very hard if we know that the puzzle is solvable beforehand. It can take some time, but one can find a solution.

Determining the difficulty of a PushPush puzzle can be done in many ways. The challenge is to find a difficulty measure that either matches the difficulty for a person to solve the puzzle, or matches the difficulty for a computer program to solve the puzzle, or even both.

We attempt to solve these questions by computing the search space. We expect that a relevant difficulty measure could be related to the size of the search space.

3 Related work

An important aspect that makes PushPush puzzles and many other games interesting is the complexity. As mentioned in Section 2.1, PushPush is PSPACE-complete.
The complexity classes that are relevant for the current project are shown in Figure 4 [Hearn and Demaine, 2009].
The class P contains all problems that are solvable by a deterministic Turing machine in polynomial time. NP contains all problems that are solvable by a nondeterministic Turing machine in polynomial time, which makes P a part of NP. Problems in NP may be hard to solve, if they are in NP $\setminus$ P, but their solutions can be verified in polynomial time. PSPACE contains all problems that are solvable in polynomial space. This includes all problems in NP.
The class of NP-complete problems contains the hardest problems in NP. Assuming that P $\neq$ NP (which we can not be sure of), this means that the class NP-complete does not contain any problems from P. NP-hard problems are problems for which there is an NP-complete problem (such as SAT) which is reducible to the NP-hard problem in polynomial time. PSPACE-complete problems are the hardest problems in PSPACE. The class PSPACE-hard contains all problems for which there is a PSPACE-complete problem that can be reduced to it in polynomial time.

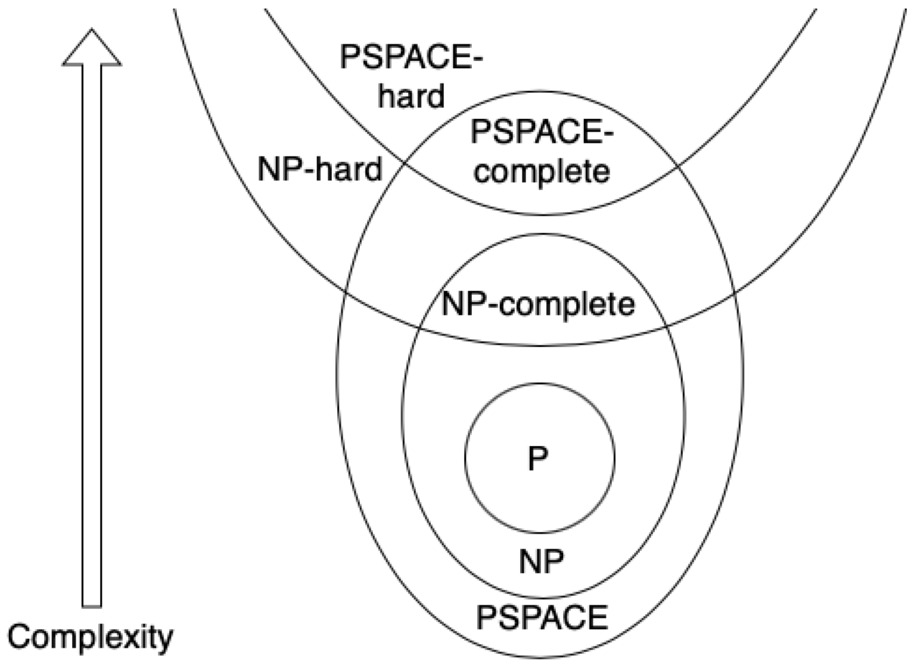


Figure 4: The relation between complexity classes.

O'Rourke and Demaine have researched the theoretical complexity of PushPush. They first proved the puzzle to be NP-hard in 3D, then in 2D by reducing from SAT [O'Rourke and The Smith Problem Solving Group, 1999, Demaine et al., 2000]. As a follow-up, Demaine et al. proved PushPush to be PSPACE-complete [Demaine et al., 2004]. O'Rourke and Demaine also gave a general proof for the NP-hardness of a whole class of similar block-pushing puzzles [Demaine et al., 2003]. However, solving methods and difficulty measures for PushPush have not yet been investigated.

There is more research on other block-pushing puzzles that are similar to PushPush, but with slightly different rules. One of those games is the ice sliding puzzle [Dorbec et al., 2018]. In this game the setting is similar. It is played on a grid (not necessarily a square) with robots and blocks. However, in this game the blocks cannot move and the robot slides all the way in the direction it moves until it hits a block or the edge of the grid. In an ice sliding puzzle the aim is to place as few blocks as possible to enable the robot to visit all cells of the grid or to visit a specific location. Different solving methods and difficulty measures for this puzzle have been investigated [Kluiver, 2020].

Another puzzle similar to PushPush is Sokoban, in which the blocks move only one square at a time when they are pushed. In this case, blocks must be pushed to specific targets. Puzzles in which the pushed objects only move one cell are called Push games. This puzzle has been extensively investigated and it has been determined to be NP-hard [Dor and Zwick, 1999]. It has also been proved to be PSPACE-complete, using a technique to prove it which also works for Rush Hour and other similar games [Hearn and Demaine, 2005].

There has been quite some research done on Rush Hour. In this puzzle the aim is to get the red block out of the grid by moving it and the other blocks around. The blocks in this puzzle are not square as in PushPush but rectangles, which can only move in the direction in which they are positioned with the short side. An instance of this game is shown in Figure 5. The complexity has

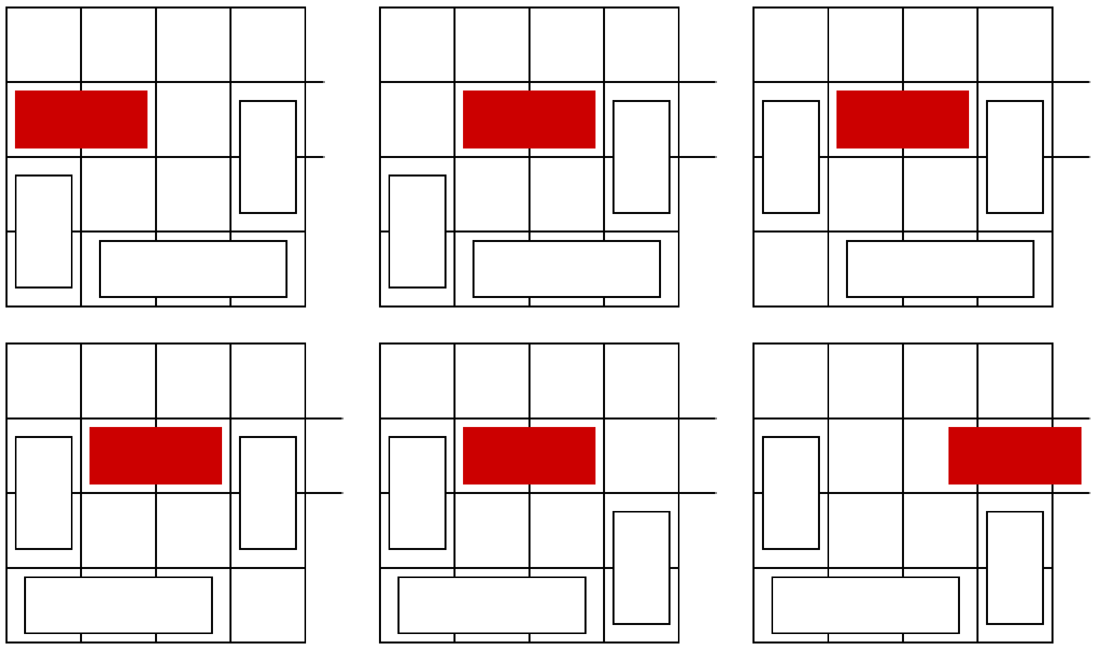


Figure 5: An example of the puzzle Rush Hour. The aim is to move the red block through the gate.

been determined to be PSPACE-complete [Flake and Baum, 2002] and many solving algorithms have been written using different approaches like breadth first search [Stamp et al., 2001] and genetic algorithms [Hauptman et al., 2009].

Some PushPush-like puzzles are made into games like Lunar Lockout and Rasende Roboter

[Dorbec et al., 2018]. In Lunar Lockout multiple robots are placed on a 5×5 grid. There is one target cell and one target robot. The robots can move up, down, left and right but only if another robot is on its path in that direction. When moving a robot keeps sliding until it hits another robot. The objective of the game is to get the target robot to the target cell. A generalised version of Lunar Lockout in which the grid is square but not necessarily 5×5 and there can also be unmovable robots is called GLLV. This has been proved to be PSPACE-complete [Hartline and Libeskind-Hadas, 2003]. This proof relies heavily on the property of stationary robots in the grid, so they did not reach a conclusion for the version of the game in which there are no stationary robots.

4 Generating and solving PushPush puzzles

In this section we discuss how PushPush puzzles can be generated and solved by implemented algorithms.

4.1 Generating puzzles

A PushPush puzzle requires one robot, one target and a certain number of blocks placed in a way that allows the puzzle to be solved. To determine whether a puzzle can be solved we use the solving algorithm that will be discussed in Section 4.2. We can generate all different boards for a certain board size by providing the size and the number of blocks which leads to a set containing a robot, a target, a number of blocks and $\mathit{boardsize} - \#\mathit{blocks} - 1$ empty cells. This set can then be ordered in every possible way to generate every possible board for that number of blocks.

The number of boards grows exponentially with the board size, which makes it hard to explore them all. The number of possible boards is given by Equation 1 with n being the number of cells in the grid and not considering boards in which the robot and the target are in the same cell:

$$\#\mathit{boards} = n \times (n-1) \times 2^{n-1} \tag{1}$$

The robot can be placed in one of n cells. Then the target can be placed in all cells except for the cell the robot is in, so there are $n-1$ options left. Every cell except the one containing the robot either has a block in it or it does not, which explains the factor 2^{n-1}. This leads to the total number of boards for n cells on the board as stated in Equation 1.
The number of possible boards of a certain size with a known number of blocks k on the board is described by Equation 2:

$$\#\mathit{boards}_k = n \times (n-1) \times \binom{n-1}{k} \tag{2}$$

The robot can be in any of the n cells. Then the target can be in any of the other $n-1$ cells. There are k blocks left to put in the grid. Then there are $\binom{n-1}{k}$ ways to put k blocks in the $n-1$ possible cells, which leads to Equation 2.

Table 1 shows the number of possible boards for 0 to 16 blocks for 3×3, 4×4 and 5×5 boards. As expected considering Equation 2, the value increases until the number of blocks is half of the number of cells on the board. Then it decreases again with exactly the same values, the pattern is symmetrical. Indeed, $\#\mathit{boards}_k = \#\mathit{boards}_{n-1-k}$.

4.2 Solving puzzles

To solve a PushPush puzzle we implemented a backtracking algorithm, considering paths depth-first, in combination with iterative deepening.
A backtracking algorithm is a brute-force method. It tries all possible solutions and chooses the best one. The search space can be represented as a graph with a tree structure. Every node is a game state and the branches represent the actions to get from one game state to the next. Depth-first search is a method to go through the tree, starting at the root and considering a path to the final

# blocks	3×3 board	4×4 board	5×5 board
0	72	240	600
1	576	3,600	14,400
2	2,016	25,200	165,600
3	4,032	109,200	1,214,400
4	5,040	327,600	6,375,600
5	4,032	720,720	25,502,400
6	2,016	1,201,200	80,757,600
7	576	1,544,400	207,662,400
8	72	1,544,400	441,282,600
9	-	1,201,200	784,502,400
10	-	720,720	1,176,753,600
11	-	327,600	1,497,686,400
12	-	109,200	1,622,493,600
13	-	25,200	1,497,686,400
14	-	3,600	1,176,753,600
15	-	240	784,502,400
16	-	-	441,282,600
total	18,432	7,864,320	10,066,329,600

Table 1: The number of possible boards for different board sizes and different numbers of blocks.

node (leaf) before backtracking and taking the next path.
To avoid the algorithm getting slow by checking all paths to the final node, depth-first search is combined with iterative deepening. In iterative deepening we cut off the search at a certain depth in every iteration, going deeper each time until a solution is found. This ensures that the found solution is the one taking the least moves. The first solution it comes across will always be one with the least branches on its way through the graph.
This combination of methods works, but an upper bound for the depth in the iterative deepening process has to be determined. This problem has not been solved yet, so far we used $\#blocks \times 3$ as an upper bound, which appears to work for puzzles on a small board up to 5×5. However, it has not yet been determined whether this is an upper bound for all board sizes.

Now that a method to solve a PushPush puzzle has been established, we try to solve all possible puzzles of a specific board size. All boards of a certain size are generated and then solved by the previously explained algorithm to determine how many moves it takes to solve.
There are two approaches for solving multiple boards of the same size. The first option is simply considering each board on its own and solving it with the depth-first backtracking implementation. The second way to solve all boards of a certain size is by dynamic programming. We store the boards and their solutions once they are solved. After that, when we pass the same board again while solving another board, we can stop exploring that branch and assign that known solution to it. This method of cutting off the search is faster than considering each board again. Searching for a solution through a tree takes more time than searching for the value of a board in a table. This

method saves time, but it requires much more RAM.
Another way to optimise the process of solving boards is by putting equivalent boards together. There are two types of equivalence that we consider.
The first type of equivalence is about the location of the robot. We consider a move to be pushing a block. All cells that the robot can reach without moving a block are part of an equivalence class. All boards that are the same, but the robot being in a different cell in the equivalence class, can be considered equivalent. These boards can be grouped, so only one of them has to be solved to know the solution to all. In Figure 6 an example of this type of equivalence is shown.

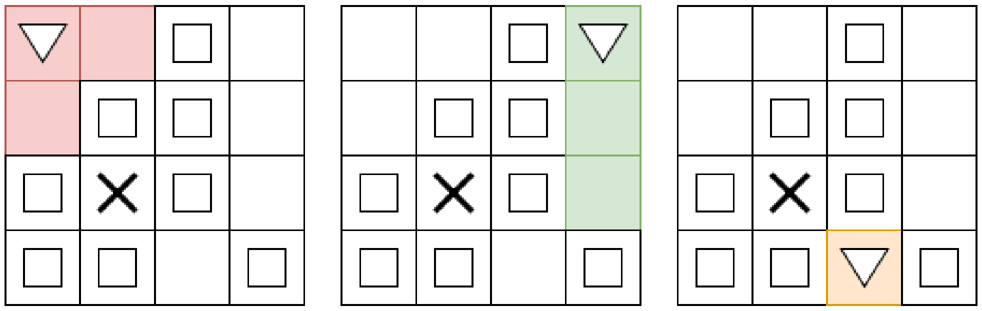


Figure 6: This board has three equivalence classes with the equivalence relation of the robot being able to move between the cells without having to move a block. Each colour represents an equivalence class. Note that we do not take the class into account where the robot is on the target, since the puzzle would already be solved in that case. Also, the position and direction of the robot within a class do not matter.

The other type of equivalence is symmetry. On a square-shaped board there are four rotational symmetries, and a horizontal, vertical and diagonal reflection symmetry. These are shown in Figure 7. Combining these symmetries in the implementation of the solving algorithm for all boards of one size makes the program much more efficient, having to explore fewer boards.

In our implementation of the solving algorithm we use a combination of the equivalence of boards from Figure 6 and the symmetries from Figure 7. We still generate all boards, but when solving one, we first check whether it is already solved. If it is not, it is solved and the solution is stored for the board itself and its equivalent boards. This sped the run time up with a factor of at least four compared to the version in which all boards are considered separately.

4.3 4×4 boards

We ran an experiment where we checked for all 7,864,320 4×4 boards whether they are solvable, and if so, how many moves it takes to solve them. An interesting statistic we found is that the puzzles that need the most moves to solve take only seven moves, and there are only 28 puzzles for which this is the case. Of those puzzles four contain seven blocks, and the other 24 contain eight blocks. These puzzles are shown in Figure 8. The puzzles that take seven moves and contain seven blocks, are symmetrical in one of the diagonals, which leads to only four different puzzles instead of the eight different puzzles that would usually follow from the symmetries.
In Figure 9 the process of solving the first puzzle in Figure 8 is shown. In this puzzle, the player could make a mistake after which the puzzle becomes unsolvable. This is shown in Figure 10.

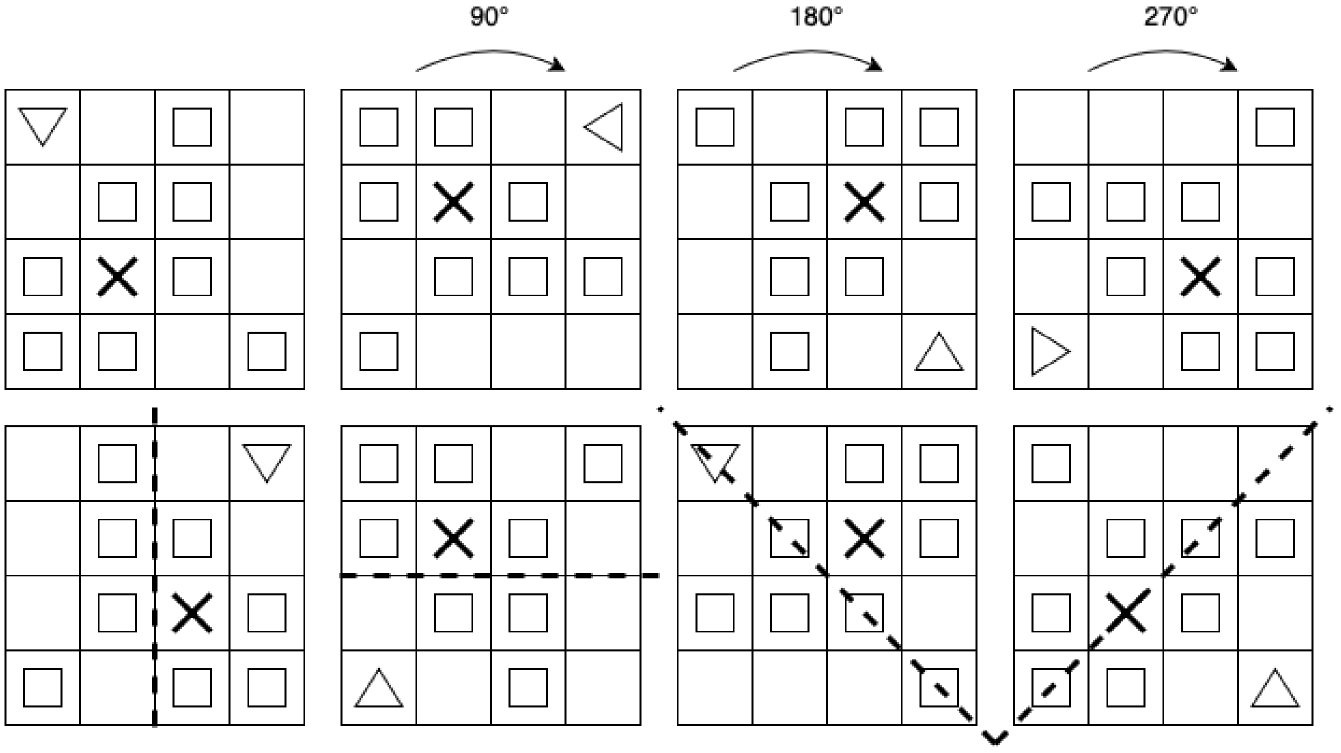


Figure 7: The board in the top left has three equivalent boards from rotational symmetry and one equivalence from horizontal reflection symmetry, one from vertical reflection symmetry, and two from diagonal reflection symmetry, one for each diagonal.

In Figure 11 all possible puzzles on a 4×4 board are sorted by the number of blocks in the grid, distinguishing the solvable from the unsolvable puzzles. The total number of puzzles for each number of blocks appears to be a binomial distribution. There seems to be a clear correlation between the number of blocks and the proportion of puzzles that are unsolvable. When there are more blocks, more puzzles are unsolvable. This makes sense, since more blocks are more objects keeping the robot from the target.

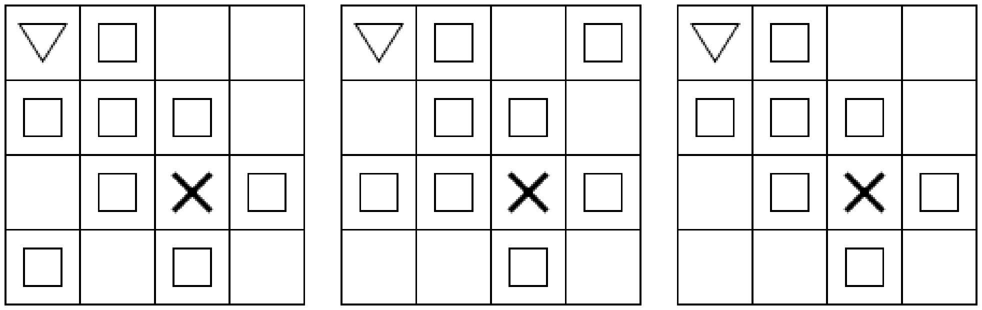


Figure 8: All 4×4 boards that are solvable and require seven blocks to be pushed to be solved. All these boards can be turned one, two and three quarters or be mirrored horizontally, vertically and diagonally. The second board also has an equivalent in which the robot is one step below its current position. The boards that are created by applying these equivalences take seven moves to solve as well.

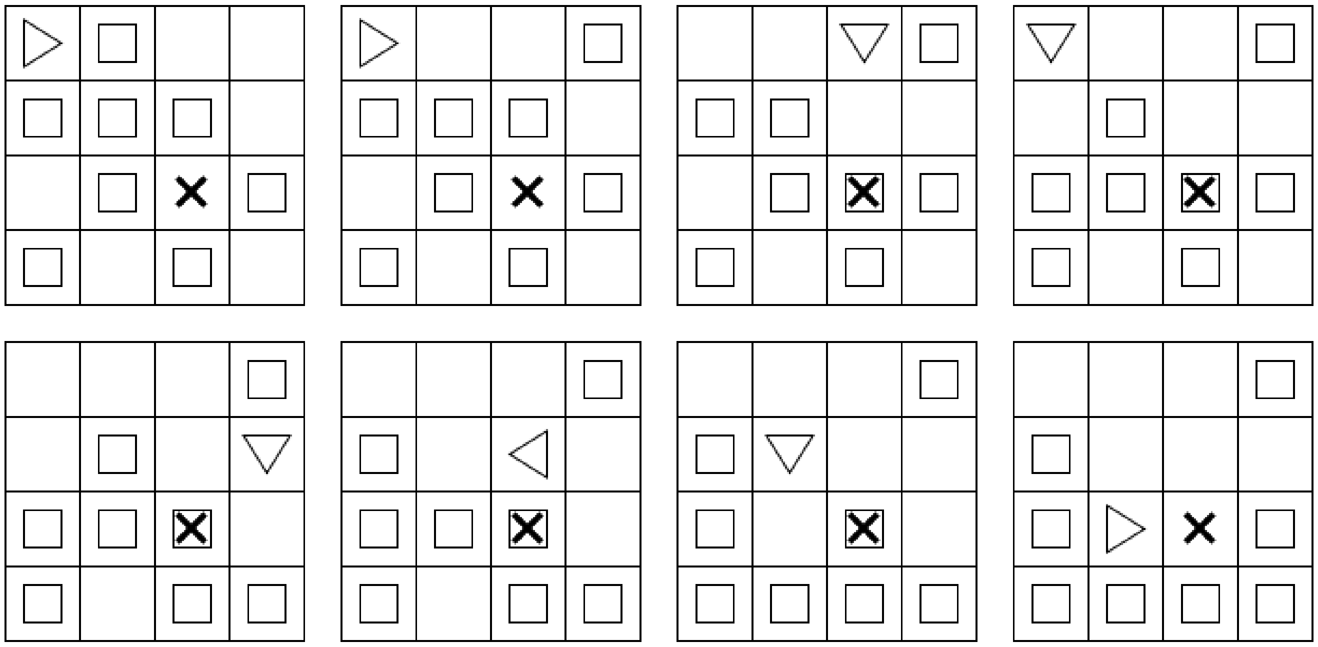


Figure 9: Reading the images from left to right, top to bottom, this figure shows a way to solve the puzzle in the first image. This is a hard puzzle, it takes seven moves to solve, and mistakes can be made which render the puzzle unsolvable.

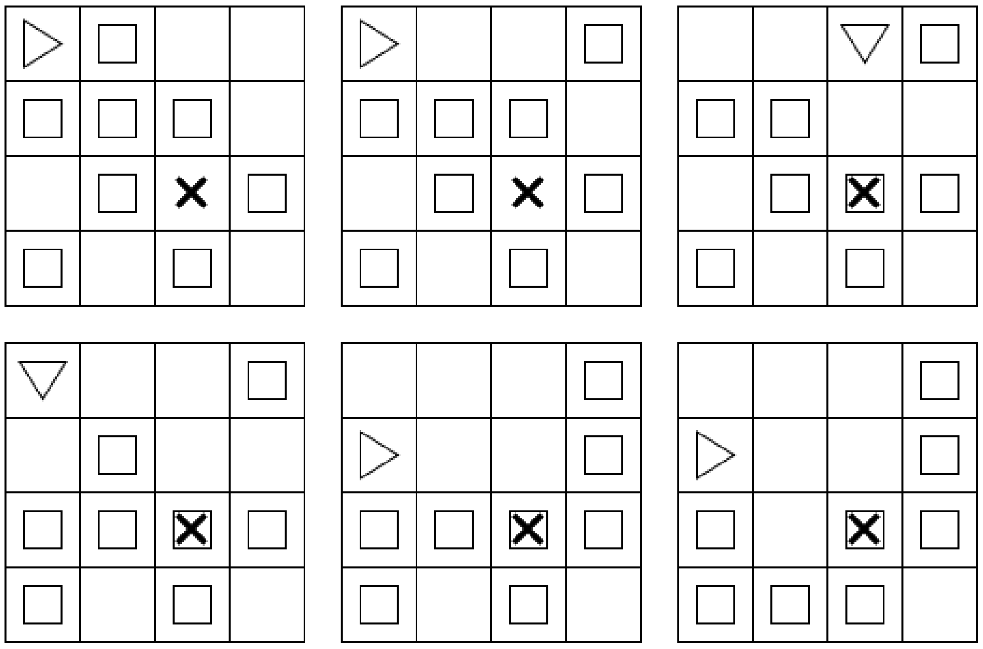


Figure 10: Reading the images from left to right, top to bottom, this figure shows a way to get from a solvable board (this is the same puzzle that is being solved in Figure 9) to an unsolvable board in the second to last image. From there on there is only one more possible move, which does not lead to a solution. This shows that a player can make irreversible mistakes.

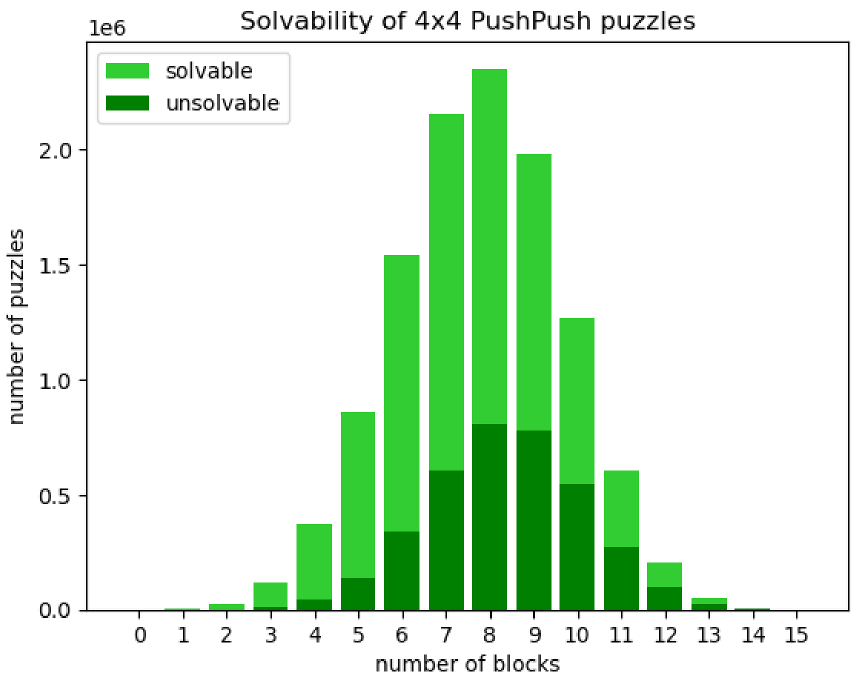


Figure 11: The plot shows how many of the 4 × 4 boards containing a certain number of blocks are solvable.

5 Difficulty measure

Some PushPush puzzles are harder to solve than others. There are multiple ways in which the various levels can be distinguished. In this section we explore two ways to rank the puzzles.

The first method of assigning a difficulty level is simply counting the minimum number of moves it takes to solve a puzzle. In general, puzzles get harder when the solution requires more moves. Puzzles that can be solved in one move are easier to solve than puzzles that take four moves. More moves make it more difficult since it requires more planning to reach the solution.
However, one could argue that a puzzle that requires more moves to be solved is not always more difficult. For instance, consider a puzzle which takes four moves to solve, but at each step on the way there is only one move that can be made. This could be considered easier than a puzzle that takes one move to solve, but where there are also two other moves you can make which do not lead to a solution. This leads us to the second way we can define a difficulty measure.
Not only the number of moves required to solve the puzzle is a factor, but also the number of alternative moves. A new difficulty measure is defined in Equation 3:

$$\left(1 - \frac{\#solutions}{\#possible\ moves}\right) * 5 \tag{3}$$

By $\#possible\ moves$ we mean the number of actions in the search space with a cut-off at the number of moves it takes to solve the puzzle. This is visualized in Figure 12. Since there are solutions after two moves, we don't consider any moves after that, i.e., below the dashed line, for determining the difficulty of the puzzle.
The fraction leads to higher values for easier puzzles, which is unintuitive for a difficulty measure, so this is turned around by substracting the fraction from one. Then the difficulty measure increases for harder puzzles and is bounded between zero and one. Usually the difficulty measure of games is between zero and five, like a star system, so we multiply the value by five to fit this scale.
For the puzzle that corresponds to the search space in Figure 12 $\#solutions = 2$, since there are two paths that lead to a solution above the dashed line. $\#possible\ moves = 10$, since there are 10 actions in the search tree above the dashed line. This leads to a difficulty of $\left(1 - \frac{2}{10}\right) * 5 = 4$.

For all puzzles on a 4×4 board this difficulty measure has been implemented, which leads to the results in Figures 13, 14 and 15.

From Figure 13 we can conclude that there is a correlation between the number of moves it takes to solve a puzzle and the difficulty of the puzzle. The puzzles that take more moves to solve are on average more difficult. The variation in difficulty between the puzzles that require a certain number of moves goes down when the number of moves goes up.
Most puzzles that are solvable in one move have a difficulty of 0. These are for example the puzzles that contain only one block, which is on the target. Then there is no wrong move to make, so the number of solutions is equal to the number of possible moves. The puzzles that take two moves to solve all have a difficulty between three and five, with most puzzles having a difficulty of three. In all puzzles that require three moves there are options to take other paths than one leading to the solution. When the number of moves required to solve the puzzle increases, the number of possible

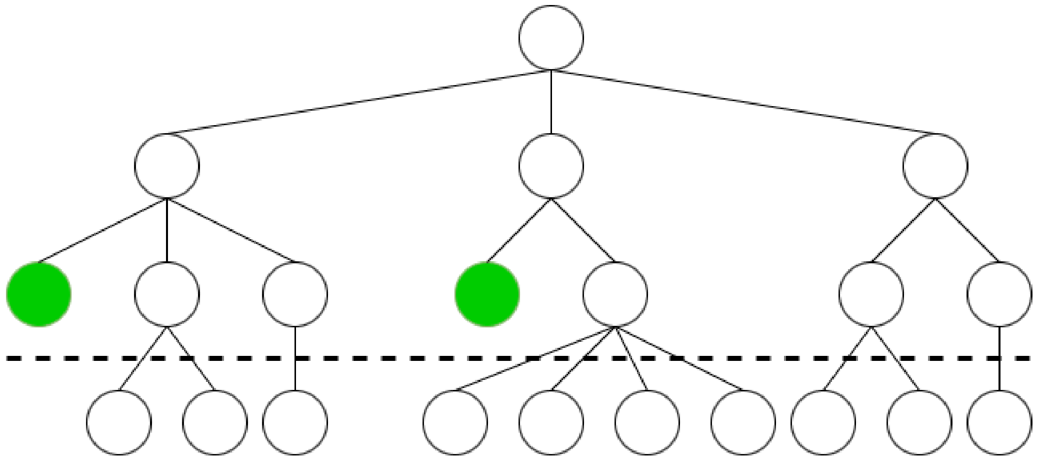


Figure 12: A visual representation of a possible search space for a PushPush puzzle. The nodes represent game states, the branches are moves. The green nodes are nodes that lead to a solution.

moves increases with it. This leads to a higher difficulty for these puzzles.

In Figure 14 we see that the puzzles that require more moves to be solved are the puzzles that have a number of blocks for which the board has approximately as many cells with blocks as cells without blocks. This can be explained by considering how many blocks can be moved. If there are fewer blocks on the board, fewer blocks can be moved, so puzzles will usually take fewer moves to solve. If there are more blocks on the board, more blocks will be unmovable because there are other blocks in their way. With few movable blocks, puzzles cannot require many moves to be solved. The most movable blocks are on boards with half of the cells empty and half of the cells filled. Then there are many possible moves and therefore puzzles can take many moves to solve.

The correlation between the number of moves and the difficulty and the correlation between the number of blocks and the number of moves together lead to the result in Figure 15. The most difficult puzzles have a number of blocks for which half of the board is empty, since this leads to the most possible moves. When the board contains very few or very many blocks, there are only a few possible moves, so the puzzle will not be difficult.

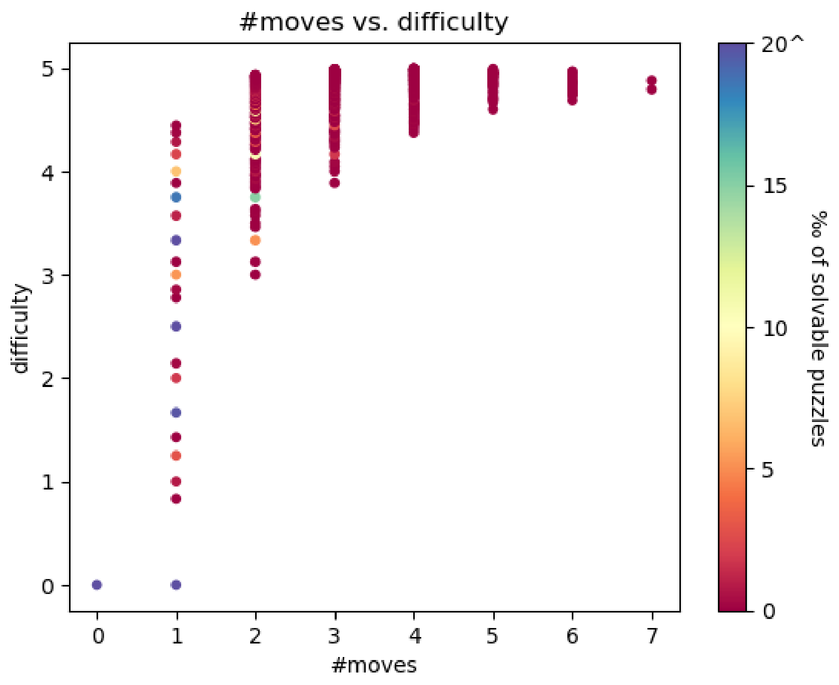


Figure 13: In this plot the difficulty of the puzzles is set out against the number of moves required to solve the puzzles. The colours represent how many ‰ of all solvable puzzles on a 4×4 board are represented by the dot in the plot.

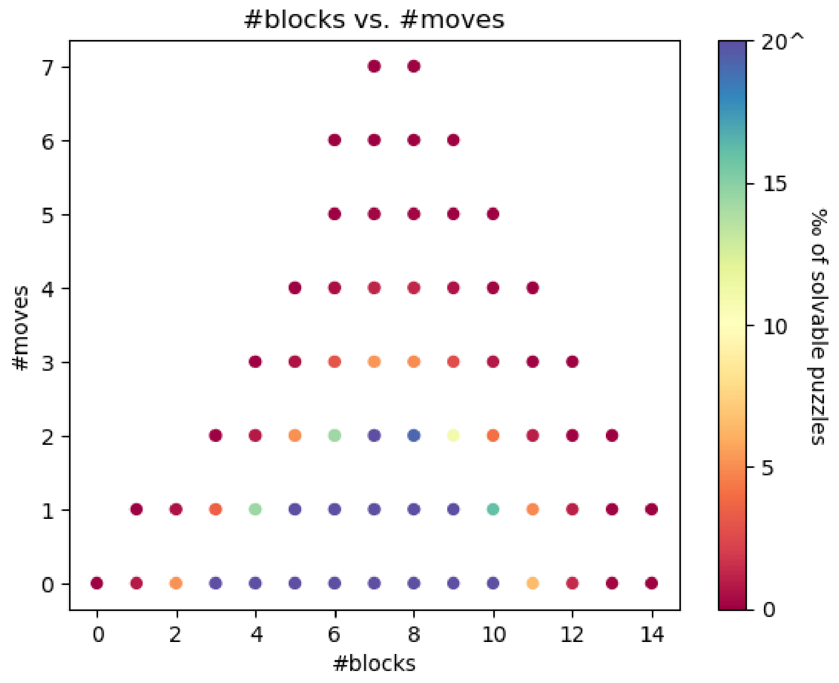


Figure 14: In this plot the number of moves required to solve the puzzles is set out against the number of blocks the puzzles contain. The colours represent how many ‰ of all solvable puzzles on a 4×4 board are represented by the dot in the plot.

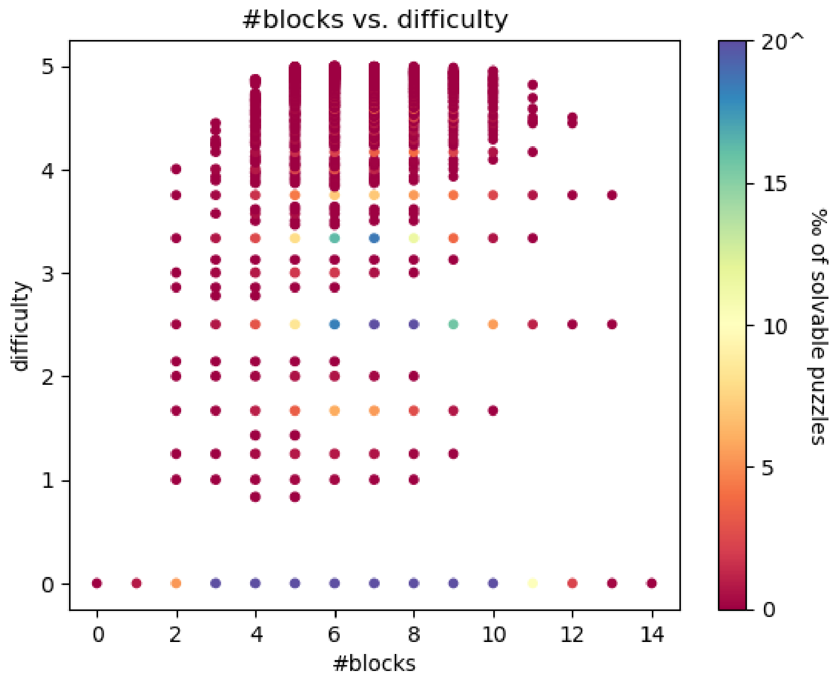


Figure 15: In this plot the difficulty of the puzzles is set out against the number of blocks the puzzles contain. The colours represent how many ‰ of all solvable puzzles on a 4×4 board are represented by the dot in the plot.

6 Applying neural networks

In this section we discuss how neural networks can be applied to determine the solvability, how many moves it takes to solve, and the difficulty of a PushPush puzzle.

6.1 Preliminaries on neural networks

Artificial neural networks (ANNs) are problem-solving models based on biological neural networks [Goodfellow et al., 2016]. In the human brain neurons transmit information to each other very fast, each neuron performing a simple task, but together completing a harder one. The speed makes it compelling to build a similar system to perform hard tasks in a computer. In ANNs the neurons are called nodes. A node receives input from multiple nodes and this input is used in combination with weights and a threshold as the input for a function f that determines the output. This is formulated as an equation in Equation 4:

$$y = f\left(\sum_{i=1}^{n} w_i x_i - T\right) \tag{4}$$

Here y represents the output of the neuron, f is the activation function, w_i are the weights for each input, x_i are the input, n is how many inputs the neuron gets, and T is the threshold.
The simplest example of a function f is the step function in Equation 5:

$$f\left(\sum_{i=1}^{n} w_i x_i - T\right) = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} w_i x_i - T > 0 \\ 1, & \text{otherwise} \end{cases} \tag{5}$$

Other functions that are often used are sigmoid, RELU or tanh.
In an ANN the nodes are put in layers. The nodes between one layer and the next are connected. There can also be a loop in the connections, which makes it a recurrent neural network (RNN). In the case of only one-way connections it is called a feedforward network, which is often referred to as an FNN. An example of an FNN is shown in Figure 17.
In this project we used a classifying FNN, so this is what will be explained further. An FNN is trained on data, by feeding input data to the model. An input is put through the network, which then produces an output. This is compared to the previously known correct label and the weights in the network are adjusted in the so-called backpropagation to get closer to the correct label for this instance. This is done for all training data, so the weights are based on many different inputs. After this training phase, the network is fed data from a validation set, which is used to tune the hyperparameters such as the learning rate. This process of training and validating is repeated a number of times (epochs), until the results converge to a certain accuracy and loss. The accuracy of a network is the percentage of instances it classifies correctly out of all input. The loss is a measure of the errors that were made by the model in classifying the data. There are several different measures for this, like the residual sum of squares or the cross entropy loss. The loss function we used in this project is the sparse categorical cross entropy loss as described in Equation 6, which is often used in classification tasks.

$$\mathit{CEloss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \tag{6}$$

Here n is the number of classes, y_i is the binary truth label for class i, and $\hat{y}_i$ is the probability of the output for class i.
When the network has learned for a number of epochs that lead to convergence, the model is tested using the test data. This set contains input that the network has never processed before, so it shows whether the network can generalise its results and does not contain a bias. A network can have a high variance when it is overfitted to the training data, which leads to the model not generalising to the test data. If the model did not have enough training data it can have a high bias. This means that it did not have enough training data to find the patterns connecting the input to the labels.

The second type of neural network that we will apply, is a convolutional neural network (CNN). A CNN is a type of network that is often used for pattern recognition in image data [O'Shea and Nash, 2015]. An example of this type of network is shown in Figure 20. A CNN takes image data as input, which keeps its shape through the process. A convolutional layer moves a filter over its input data and produces output based on the values that come out of the filter. These filters base the new value of a pixel on its current value and the values of the pixels around it. This allows it to find patterns in the data.
Usually pooling layers are used in CNNs to reduce the size of the data. In a pooling layer the features of the input are summarised. An example of a pooling layer that is used often is max pooling. This method divides the input in zones and determines the highest value in each zone. It puts these maxima together and produces that as output. However, since our input consists of only 16 pixels, these reductions are not necessary.
After a number of convolutional layers and pooling layers the data is flattened and put through one fully-connected layer leading to the output, which works the same way as in an FNN.

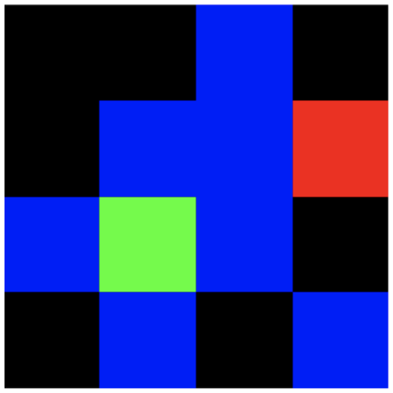
After this introduction into neural networks we can apply them to our problems.

6.2 Input data

We used both feedforward neural networks and convolutional neural networks, which require different input shapes. In all our implementations of neural networks we used a randomly generated data set of 1,600,000 different 4×4 boards. Boards that are selected for the data set are taken out of the set to choose from. This method does not take into account the equivalences between boards, so there could be equivalent boards in the data set. This set was split in a training set containing 0.8 of the total set, a validation set of 0.1, and a test set of 0.1. We generated four of those random sets and tested all of them to check if the results were not caused by chance.
The training data is used to set the weights of the network, the validation data to tune the hyperparameters, and the test data to check if the final model gets accurate results on new data.

The first model we considered was an FNN. The input is a board in the shape of an array of 18 numbers: 16 zeroes and ones, representing respectively empty squares and squares containing a block, and the location of the robot and the location of the target as integers between zero and 15. The label corresponding to a board is one or zero for being solvable or not. For example, the board shown in the top right corner of Figure 9 is represented as `[0,0,0,1,0,1,0,0,1,1,1,1,1,0,1,0,0,10]`. The robot is in the top left corner of the grid, which is cell 0, and the target is in cell 10.

We also implemented a CNN, which is mainly used for classifying image data. To use a CNN the data had to be transformed into images. Instead of an array of integers, an image of the board is provided. This is an image of 4×4 pixels, one for each cell in the grid of the puzzle. The empty cells are represented by black pixels, the blocks are blue, the robot is red, and the target is green. If a block is on the target, the colour is the combination of blue and green, resulting in a light blue pixel. An example of a board represented this way is shown in Figure 16.


(a) The board in the shape that was provided as input for the CNN.

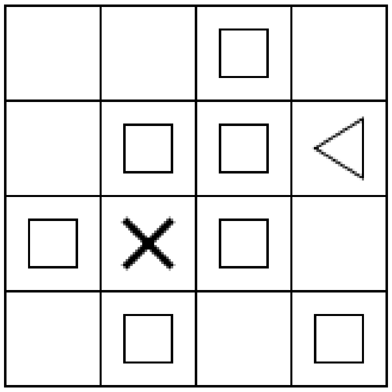

(b) The board represented in the way explained in Section 2.

Figure 16: An example of a board provided as input for the CNN.

6.3 Predicting the solvability

We investigated whether a neural network can distinguish the solvable from the unsolvable puzzles. The first model we considered was a feedforward neural network. The neural network itself has had many different shapes in the attempts to optimise it. Different numbers of layers and different numbers of nodes per layer have been used (see Figure 17). However, none of the attempts led to an accuracy higher than 0.7, which means it correctly classifies 70% of the instances. The results of the network in Figure 17 are shown in Figure 18. In the first epochs of training the network learns quickly, but then it stabilises at a low accuracy and does not improve any further.

This led us to consider a convolutional neural network (CNN). The CNN is much more effective in determining the solvability of PushPush puzzles than the FNN. The network we use has three convolutional layers after which the image data is flattened. Then one dense layer follows and lastly the output layer with two nodes, one for the input being a solvable puzzle and one for it being an unsolvable puzzle.
The results of this model are shown in Figure 19. The network learns quickly and keeps improving over the epochs, with less improvement at each epoch, but still getting better. The model performs well on the test data, reaching an accuracy of 0.98.

6.4 Predicting the minimum number of moves

We performed another experiment on the same CNN with a different output layer. Instead of only determining the solvability of a puzzle the model also determines the number of moves it takes

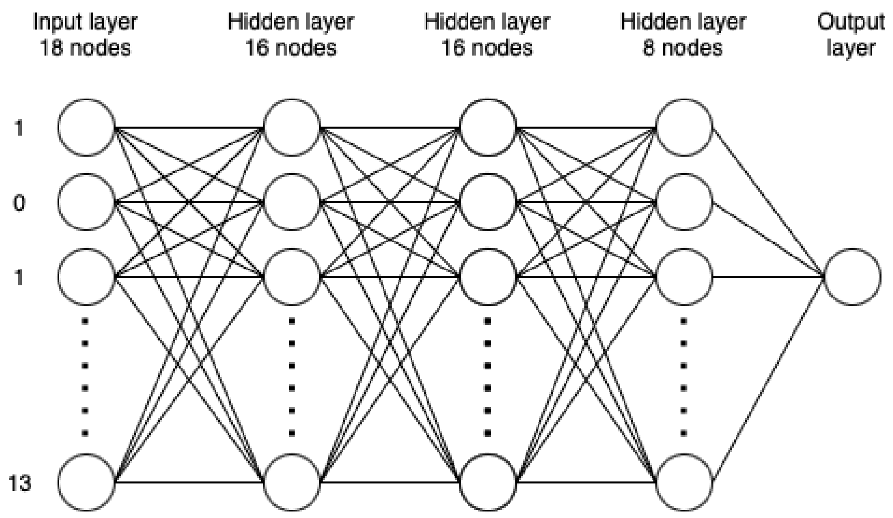
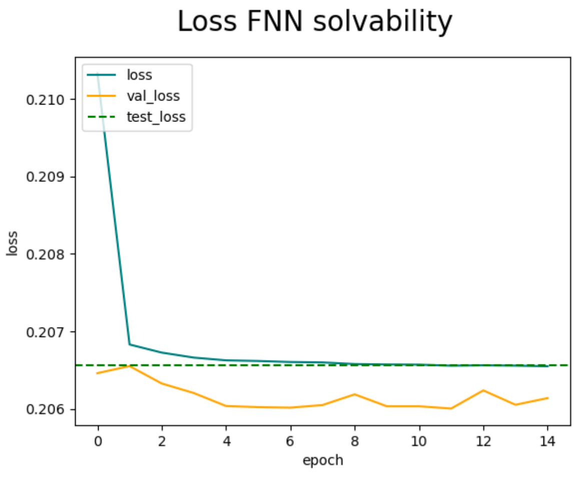


Figure 17: The architecture of an FNN that attempts to determine whether a PushPush puzzle is solvable. The input only moves forward through the network from the input to the output layer, it cannot encounter a cycle.

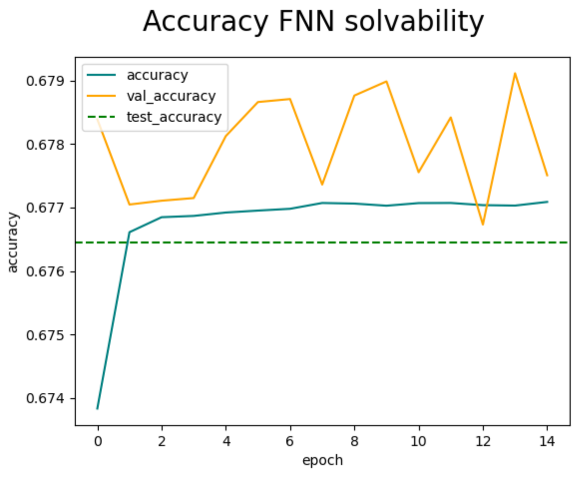
to solve a puzzle if it is solvable. The number of nodes in the output layer of this network is the maximum number of moves it takes to solve a puzzle and one extra for the unsolvable puzzles. For a 4×4 board, this is $7 + 1 = 8$ nodes. A schematic representation of this model is shown in Figure 20.
This model was provided with input in the same way as the earlier model. The labels are now not only zero and one, but integers in the range of minus one to seven, where minus one represents the unsolvable puzzles and the other numbers are the number of moves it takes to solve the puzzle.
The model gets accurate results, as shown in Figure 21. The network learns at a similar rate as the CNN determining the solvability, reaching only a slightly lower accuracy after the same number of training epochs: 0.97.

6.5 Predicting the difficulty

The number of moves it takes to solve a puzzle can be determined accurately by a neural network, so the next step is to attempt to get a neural network to determine the difficulty of a puzzle.
We used the same network as the one in Figure 20 except for the output layer. For this network, the input data was not sampled out of all 4×4 puzzles, but only the solvable ones, since the difficulty of unsolvable puzzles is undefined. The number of difficulty classes is six, zero to five. The difficulty of a puzzle was determined as a real number, but for this classifying network the values were rounded to the nearest integer.
The results of fitting the model are shown in Figure 22. The model learns less quickly on this data than it did on the solvability and the number of moves. After the same number of epochs the learning curve has not flattened yet, so it could benefit from more training. After 15 epochs the accuracy on the test data is 0.90. This leads to the conclusion that with this particular neural network the difficulty of a puzzle can not be determined accurately, since it misclassifies 10% of the instances.

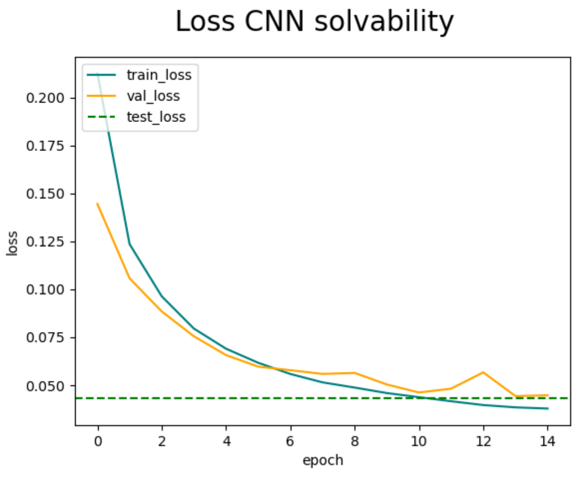


(a) Loss of the FNN over the training epochs.

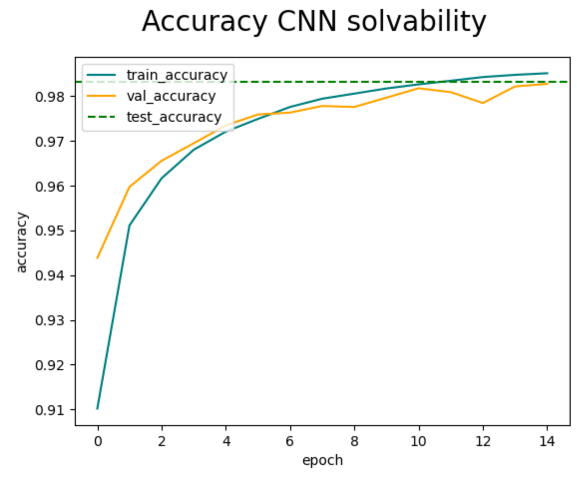


(b) Accuracy of the FNN over the training epochs.

Figure 18: The results of the FNN for determining the solvability of PushPush puzzles.



(a) Loss of the CNN over the training epochs.



(b) Accuracy of the CNN over the training epochs.

Figure 19: The results of the CNN for determining the solvability of PushPush puzzles.

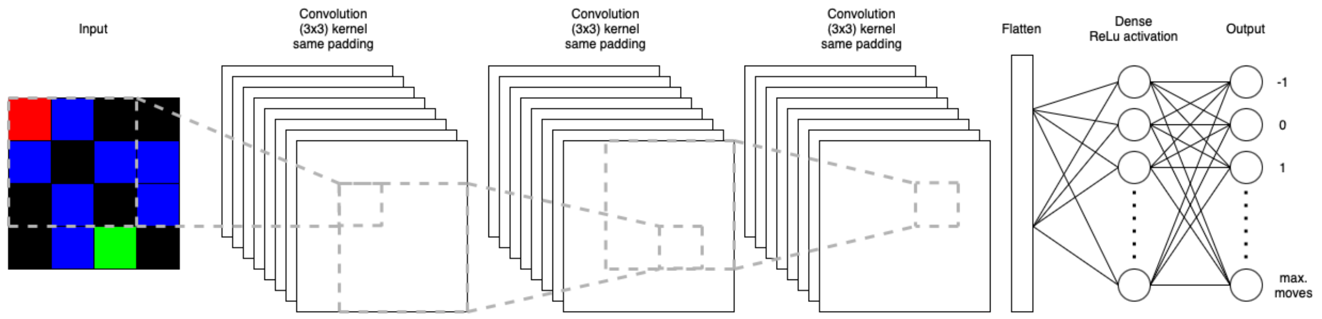
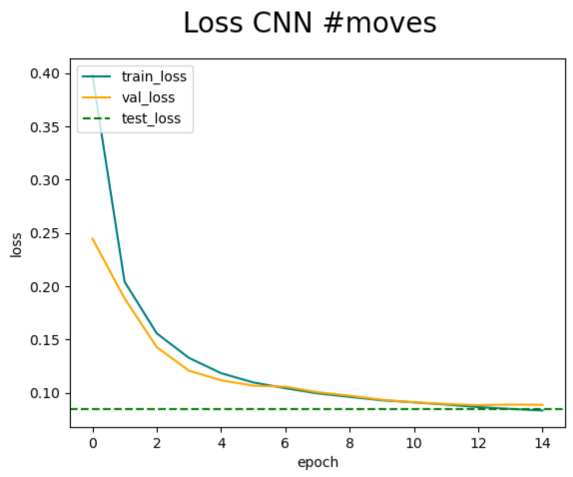
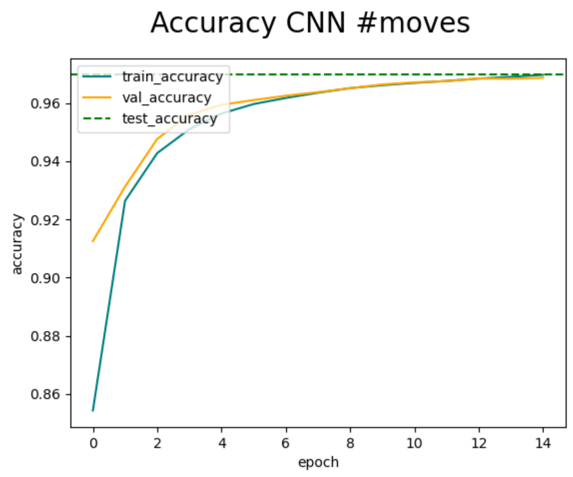

Input
Convolution (3x3) kernel same padding
Convolution (3x3) kernel same padding
Convolution (3x3) kernel same padding
Flatten
Dense ReLu activation
Output
-1
0
1
max. moves

Figure 20: The architecture of the CNN that predicts the number of moves it takes to solve a PushPush puzzle.

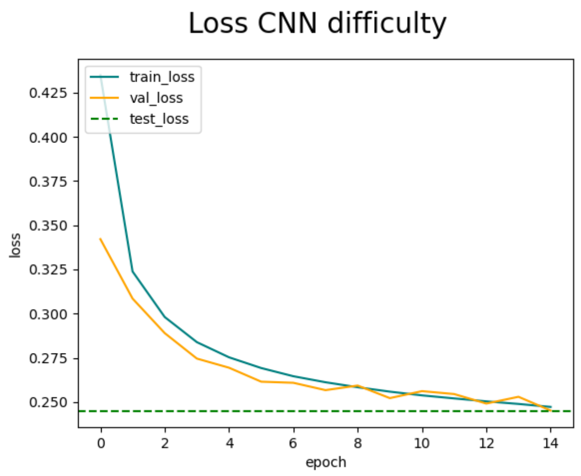


(a) Loss of the CNN over the training epochs.

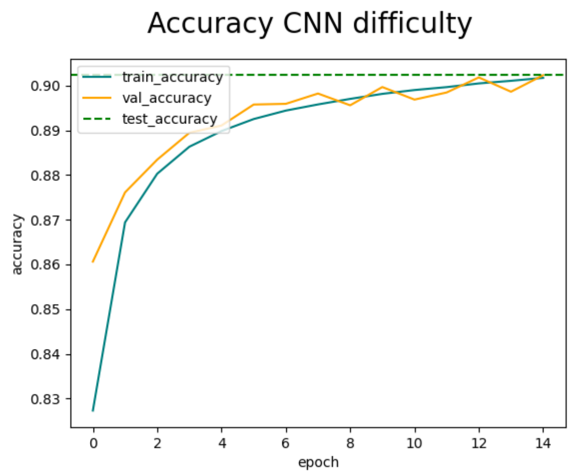


(b) Accuracy of the CNN over the training epochs.

Figure 21: The results of the CNN for determining in how many moves PushPush puzzles can be solved.



(a) Loss of the CNN over the training epochs.



(b) Accuracy of the CNN over the training epochs.

Figure 22: The results of the CNN for determining the difficulty of PushPush puzzles.

7 Discussion and conclusion

The aim of this project was to answer the question "How can we generate and solve PushPush puzzles and determine the difficulty, while (efficiently) computing the search space?"

To get an answer to this question we first wrote a program that generates and solves puzzles of a given size. In this program backtracking was combined in a depth-first search through the search space, combined with iterative deepening. Using this implementation, we generated all possible puzzles for small board sizes and had the algorithm solve them. This led to the conclusion that there is a correlation between the number of blocks in a puzzle and whether it is solvable or not. As suggested in Figure 11, more of the puzzles are unsolvable when there are more blocks on the board. This fits with the expectations: when more paths to the target are blocked, the chance is higher that the target is unreachable. Another conclusion we could draw from the same experiment is that the number of puzzles fits a binomial distribution, which follows from Equation 2.

For the next part of the research question we discussed and applied a method to define the difficulty of a PushPush puzzle. The resulting difficulty measure is given in Equation 3. With this difficulty measure there is a correlation between the number of moves it takes to solve a puzzle and the difficulty, and the number of other moves that could have been made, which do not contribute to the solution, are taken into account as well.

We performed experiments using neural networks to see if these can help identify the solvability of a puzzle, the number of moves it takes to solve it, or even the difficulty more efficiently. A conclusion from the experiments is that a feedforward neural network cannot learn to accurately distinguish solvable from unsolvable puzzles, illustrated by Figure 18. Using a convolutional neural network leads to better results. The CNN model built in this project can predict whether a puzzle is solvable and in how many moves it is solvable with a high accuracy of respectively 0.98 and 0.97. Determining the difficulty is a harder challenge, the network only reaches an accuracy of 0.9, so it is not a very reliable method for this purpose.

In short, generating and solving PushPush puzzles can be done by using a combination of depth-first backtracking and iterative deepening. The difficulty can be assessed by applying Equation 3. The solvability and the number of moves needed to solve a puzzle can be accurately determined by a neural network, but so far, the difficulty cannot.

7.1 Future work

In this thesis the research question has been answered, but many new questions came up during the project.
Since the time and resources for this project were limited, only the 4×4 boards are completely worked through. All results are based on this board size. This might be general and work for other board sizes as well, but this has not yet been researched. In the future, the methods from this project could be applied to larger boards to check if the results generalise.

The puzzle-solving method as described in Section 4.2 works well for solvable puzzles, but an upper bound to the depth one should search before concluding a puzzle to be unsolvable has not yet been determined. It could be interesting to investigate how this upper bound can be set as strict as possible to avoid unnecessary searching.

The difficulty measure that was determined in Section 5 was reached from a theoretical point of view. This measure has not been applied to the puzzles and no research has been conducted yet to decide whether people solving the puzzles agree with this difficulty measure, which is an important factor in determining whether the measure is good.

In the application of neural networks more work could be done to tweak the networks to lead to better results. Possibly, a CNN to determine the difficulty of a puzzle could perform much better, but in this project the right configuration of parameters has not been found. In the future, this could be researched further.
Another interesting question about the neural networks is why the feedforward neural network did not perform better. What makes the CNN so much better than the FNN? Maybe the FNN could have worked as well when the input data would be in another shape.

Lastly, the results of this project only apply to PushPush-1, but possibly some results could generalise to other types of PushPush, or other similar puzzles.

References

[Demaine et al., 2003] Demaine, E. D., Demaine, M. L., Hoffmann, M., and O'Rourke, J. (2003). Pushing blocks is hard. *Computational Geometry: Theory and Applications*, 26(1):21–36.

[Demaine et al., 2000] Demaine, E. D., Demaine, M. L., and O'Rourke, J. (2000). PushPush is NP-hard in 2D. *CoRR*, abs/cs/0001019.

[Demaine et al., 2004] Demaine, E. D., Hoffmann, M., and Holzer, M. (2004). PushPush-k is PSPACE-complete. In *Proceedings of the 3rd International Conference on FUN with Algorithms*.

[Dor and Zwick, 1999] Dor, D. and Zwick, U. (1999). SOKOBAN and other motion planning problems. *Computational Geometry*, 13(4):215–228.

[Dorbec et al., 2018] Dorbec, P., Duchêne, E., Fabbri, A., Moncel, J., Parreau, A., and Sopena, E. (2018). Ice sliding games. *International Journal of Game Theory*, 47(2):487–508.

[Flake and Baum, 2002] Flake, G. W. and Baum, E. B. (2002). Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895–911.

[Goodfellow et al., 2016] Goodfellow, I. J., Bengio, Y., and Courville, A. C. (2016). *Deep Learning*. Adaptive computation and machine learning. MIT Press.

[Hartline and Libeskind-Hadas, 2003] Hartline, J. R. and Libeskind-Hadas, R. (2003). The computational complexity of motion planning. *SIAM Review*, 45(3):543–557.

[Hauptman et al., 2009] Hauptman, A., Elyasaf, A., Sipper, M., and Karmon, A. (2009). GP-rush: Using genetic programming to evolve solvers for the Rush Hour puzzle. In *Proceedings of the 11th Annual Genetic and Evolutionary Computation Conference, GECCO-2009*, pages 955–962.

[Hearn and Demaine, 2005] Hearn, R. A. and Demaine, E. D. (2005). PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1):72–96.

[Hearn and Demaine, 2009] Hearn, R. A. and Demaine, E. D. (2009). *Games, Puzzles, and Computation*. CRC Press.

[Kluiver, 2020] Kluiver, S. (2020). Solving and Generating Ice Block Puzzles. Bachelor Thesis, Leiden University.

[O'Rourke and The Smith Problem Solving Group, 1999] O'Rourke, J. and The Smith Problem Solving Group (1999). PushPush is NP-hard in 3D. *CoRR*, abs/cs/9911013.

[O'Shea and Nash, 2015] O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. *CoRR*, abs/1511.08458.

[Stamp et al., 2001] Stamp, M., Engel, B., Ewell, M., and Morrow, V. (2001). Rush Hour® and Dijkstra's algorithm. *Graph Theory Notes of New York*, 40:23.