Bachelor thesis
Computing Science

Radboud University

Creating a Formal Model of the Game 2048

Author:
Johan Sijtsma
s4793676

Supervisor/assessor:
dr. Nils H. Jansen
`N.Jansen@cs.ru.nl`

Second assessor:
Prof. dr. Frits W. Vaandrager
`F.Vaandrager@cs.ru.nl`

June 26, 2020

Contents

Chapter 1

Introduction

Recent years have showed increased attention to machine learning (ML) for various purposes. ML is an efficient way to solve problems that would take too long or take too much resources otherwise. However, there are safety concerns. The resulting policy from a ML algorithm is a black box, and as such holds no guarantees of safety. Take for example a self-driving car. The people in the car would want to know that the car is guaranteed to not make certain decisions that could lead to a crash.

A way to guarantee certain properties has been proposed by Alshiek et al. in the form of *shielded decision making* [1]. Shields offers a guarantee that an agent does not reach certain unwanted states by shielding it from bad decisions. To implement shielding we need formal definition of the environment, a model. With a model we can calculate which actions need be avoided in which states. One source for are games. Games are a great source for finding environments to trying out machine learning and formal verification techniques. Games can have a simple environments where agents can reach goals and avoid pitfalls. For example, both Jansen et al. [9] and Hasanbeig et al. [8] used PAC-MAN for their research in Reinforcement Learning.

The game 2048 has complete information, the player can see the entire board, but the consequences of the player's actions are random. Additionally, there are very few possible actions, but many different board states. Additionally, the game is very popular, which might attract attention to this field of study, increasing the amount of research of formal verification techniques. To implement shields or other techniques, however, a model of 2048 has to be created first. Such a model would not only help with testing shields, but also optimal strategy finding, statistical model checking, formal verification and more. To make this model useful, to describe 2048 as a Markov Decision Process (MDP). An MDP encodes all possible states, actions and probabilities of those actions leading to certain states. A partial MDP for 2048 has been created already by Lees-Miller for his blog about 2048 [13]. This model only goes as far as the 64 tile. We want to create a full MDP. This is unrealistic because the size for a full 2048 model of the entire game would be too large for modern computers to handle. Any of the sixteen cells can have one of 12 different values (0, 2, 4, ..., 1024, 2048). At most we have $16^12 = 281.474.976.710.656$, or about 281 trillion different states. Some of these are unreachable, but this does number does show the scale. Doing formal verification on this level is impossible with current hardware, so instead we can use statistical model checking.

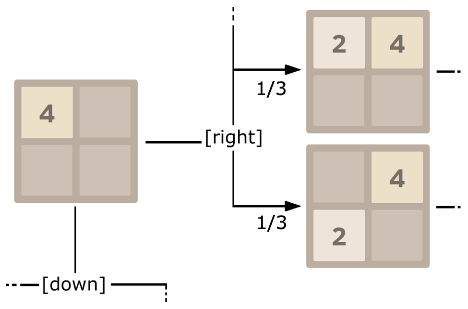


Figure 1.1: Graphical representation of how part of a 2048 model could look

MDPs can be created by describing the model in a language such as PRISM. With the PRISM you can describe a model's states and transitions between those states using variables and commands. A program like PRISM model checker [11] or Storm [6] can then be used to convert the model to an MDP.

In this thesis we will create a PRISM model of 2048. We also create a Python script that can generate N by N 2048 models. We also perform statistical model checking to compare different strategies of playing 2048. We do this by extracting the rules of 2048 from the source code, structuring those rules and then creating a small model representing a simplified version of 2048. Then we will expand on this prototype to model the full game. We then improve this model to minimize the amount of commands used by iteratively combining commands with similar behaviour until no more commands can be combined. This provides a robust 2048 model that can be used in future research. To show that this model can be used for statistical model checking, we will be comparing a few strategies and see which strategy has a higher chance of leading to a game over screen.

Chapter 2

Preliminaries

2.1 Markov Decision Processes

As described in the book Principles of Model Checking [2], a Markov Decision Process (MDP) is a 4-tuple (S, A, P_a, R_s) where S is the set with all states, A the set with all actions. $P_a(s, s')$ is a function $Pr(s_{t+1} = s'|s_t = s, a_t = a)$ which defines the probability of arriving in state s' after action a was used in state s. $R_a(s, s')$ is the expected reward from reaching state s' from s with action a. In other words, when in state s, a user chooses an action a after which random chance lands them in s'. MDPs are a good fit for single player games without a hidden board and where user input has unpredictable results.

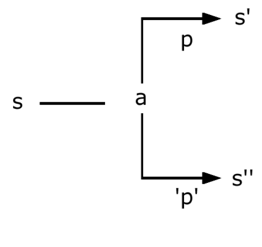


Figure 2.1: General shape of an MDP

2.2 PRISM

PRISM is a probabilistic model checker. With PRISM, you are able to build and analyse discrete-time and continuous-time Markov Chains, Markov Decision Processes, as well as probabilistic (timed) automata [11]. It is also possible to write and test properties of the model with temporal logic such as CTL or LTL. To write models you use the PRISM language. We will give a short summary of relevant features and concepts of the language.

2.2.1 The PRISM language

The PRISM language is used to describes the state-space and transitions between those states.When making a model, you split it in multiple modules, one module for each separate process. A module contains its own internal variables and commands. The commands describe how and when these variables change. Different modules can see each others variables, but can only change their own. Global variables can also be defined outside of a module, but these can never be changed. The full state-space of the model is described by all values the variables can take. The commands describe the transitions between the states. We will be discussing a few key functionalities of PRISM using the following example of a simple model in the PRISM language:

```
mdp

module Example1

    i : [0..2] init 0;
    j : bool;

    [up] i=0 -> 0.5 : (i'=0) + 0.5 : (i'=2);
    [] i>0 -> (i'=i-1);
    [] true -> 0.5 : (i'=0)&(j'=true) + 0.5 : (i'=1)&(j'=false);

endmodule

module Example2

    k : [0..2] init 0;
    l : bool;

    [up] k=0 -> 0.5 : (k'=0) + 0.5 : (k'=2);
    [] k>0 -> (k'=k-1);
    [] true -> 0.5 : (k'=0)&(l'=true) + 0.5 : (k'=1)&(l'=false);

endmodule
```

This specific example does not do anything useful, but does show all aspects of PRISM that we will discuss. In this model, when counter i and k are zero, there is a fifty percent chance of those variables becoming 2. Every step where i or k is larger than zero, the counter subtracts one from itself. There is also a chance of the booleans j and l to change along with their counter.

First we define the model type. This is an MDP, so we put `mdp` on at the top. Other options include `ctmc`, `dtmc` or `pta`. Every module describes a process. The module contains variables and commands between the `module` and `endmodule` tags. After `module` comes the name of the module, here `Example1` and `Example2`.

The variables define in which states the module can be. Variables can be either integers or booleans. `i` and `k` are initiated to zero. If no initial value is given, PRISM defaults to zero or `false`.

Commands

```
[up] i=0 -> 0.5 : (i'=0) + 0.5 : (i'=2);
[] i>0 -> (i'=i-1);
[] true -> 0.2 : (i'=0)&(j'=true) + 0.8 : (i'=1)&(j'=false);
```

The commands define what transitions have a non-zero chance of occurring. A command consists of an action (between square brackets), a guard (before the arrow) and one or more updates (after the arrow). The action is the 'name' of the transition. It used to sync commands between modules. If the `Example1` module chooses the `[up]` command, then PRISM enforces that module `Example2` will choose a command with `[up]` as well or vice versa. This is called parallel composition. If you do not want to use parallel composition, then no action is necessary for a command.

The guard determines whether a command can be used. If the expression in the guard evaluates to true, the command can be chosen. In the first command of `Example1`, `[up]`, we check in the guard whether `i` is 0. This way, this command can only occur when `i` is zero. Guards can be as complicated as any expression can be.

After the `->` in the command come one or more updates. Every update has a probability. If only one update is defined, the probability can be omitted like in the second command of either module. If multiple variables need to be changed, you can chain the variable updates with `&` as in the third command of either module.

```
[] true -> 0.2 : (i'=0)&(j'=true) + 0.8 : (i'=1)&(j'=false);
```

The variable that is updated is denoted with a `'` to distinguish the post transition variable from the pre transition variable. This is to avoid ambiguity.

Other relevant operators for expressions include the addition (`+`) and subtraction (`-`), relational (`=, !=. >, <=`), disjunction (`|`) and conjunction (`&`), as well as the condition evaluation (`condition ? true-expression : false-expression`) and build-in functions, such as (`min(...)`), (`floor(...)`) and (`pow(x, y)`)

One other useful feature of the PRISM language is that we can easily create a new module by renaming a previously defined module. This renaming is on a textual level, meaning the renaming happens before parsing. We can replace the entire second module in the example module with the following line:

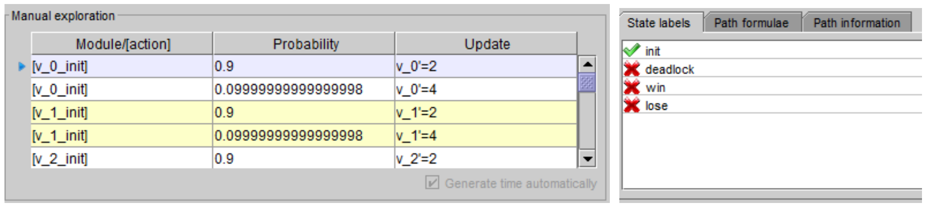
```
module Example2 = Example1 [ i=k, j=l ] endmodule
```

This produces the exact same Example2 module as in the first model.

PRISM Simulator

We use the PRISM simulator to try out the model as we are making it. By simulating we can detect errors and unwanted behaviour. The PRISM simulator offers various features that make simulating convenient.

The PRISM simulator offers step by step execution of the model. We can either choose which transitions are taken or let the simulator take one randomly. This transition can be chosen from a list of all possible transitions. This list also shows the probability of each transition. This list can be used to see if more transitions are possible than intended. This helps detect

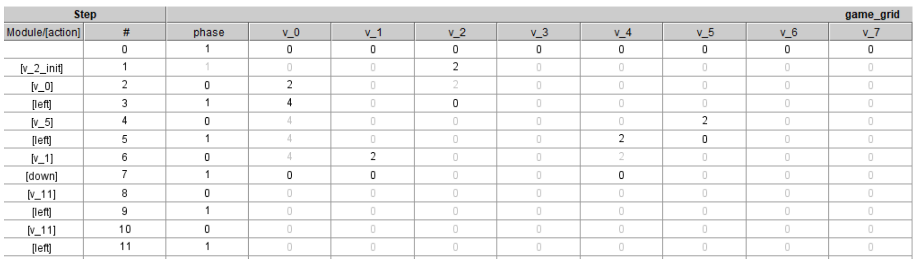


Module/[action]	Probability	Update
[v_0_init]	0.9	v_0'=2
[v_0_init]	0.09999999999999998	v_0'=4
[v_1_init]	0.9	v_1'=2
[v_1_init]	0.09999999999999998	v_1'=4
[v_2_init]	0.9	v_2'=2

✓ init
✗ deadlock
✗ win
✗ lose

Figure 2.2: List of possible transitions and list of labels in the PRISM simulator.

unintended behaviour. Steps can be reverted, allowing easy traversal of the state space to test multiple similar situations quickly. The simulator shows the values of all variables (per module) in the model. It is also possible to make formulas for the model. You can then see whether a formula evaluates to true or false in the current state. One very important feature, perhaps the most useful, is that the simulator can be used without building the whole model. PRISM only needs to parse the model before the simulator can be used. This allows us to use the simulator even when working with a large model like a full 4x4 2048 game, as long as we make a model that can be parsed in a reasonable amount of time.



Step		game_grid								
Module/[action]	#	phase	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
	0	1	0	0	0	0	0	0	0	0
[v_2_init]	1	1	0	0	2	0	0	0	0	0
[v_0]	2	0	2	0	2	0	0	0	0	0
[left]	3	1	4	0	0	0	0	0	0	0
[v_5]	4	0	4	0	0	0	0	2	0	0
[left]	5	1	4	0	0	0	2	0	0	0
[v_1]	6	0	4	2	0	0	2	0	0	0
[down]	7	1	0	0	0	0	0	0	0	0
[v_11]	8	0	0	0	0	0	0	0	0	0
[left]	9	1	0	0	0	0	0	0	0	0
[v_11]	10	0	0	0	0	0	0	0	0	0
[left]	11	1	0	0	0	0	0	0	0	0

Figure 2.3: List of steps the simulator took in the current path.

2.2.2 PRISM Property Checker

PRISM's main purpose is to verify properties of a model. You can write these properties in PRISM's property specification language (PSL). PSL combines aspects of many probabilistic temporal logics, like LTL and PCTL. An example of such a property would be `[F fail]`. In natural language, this would translate to 'The model eventually fails.' Other symbols are `X` (next) and `G` (Globally).

To check a property, we first create all paths through the model. A path is a list of all states the agent travelled through. Some of these states are labeled. In our case, a game over state is labeled with `'fail'`. Say a path existed with two states, first an initial state, and the next being a game over state. The property `'fail'` would not be true for this path, as the first state is not a game over, but `X 'fail'` would be, as the next state from the first state would be true. Multiple paths are possible in the models, so we do not just calculate if a path property is true, we want to calculate over all possible paths. As such, PRISM properties usually look like `P=1[F fail]`, or in natural language: The probability of a path ending in failure is one.

We can also ask PRISM to calculate the probability with `P=?[F fail]`. This would return the probability of an execution of the model ending in failure. A property we often used was `Pmin=?[F fail | success]`, or in natural language: what is the minimum probability of a path resulting in either failure or success. If the answer was 1, we did not have loose ends in our model.
To calculate this, PRISM has to generate every path of the model. This is clearly not a feasible solution for large models such as our model for 2048. For such cases, you can use the simulate function. Using the simulate function, PRISM instead generates a finite amount of paths and calculates the

probability from those. PRISM also calculates a number that tells us how likely that probability is accurate.

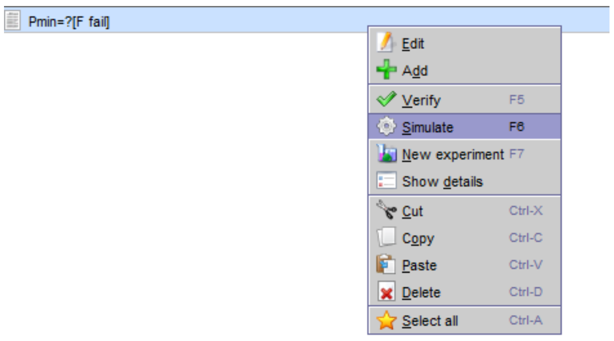
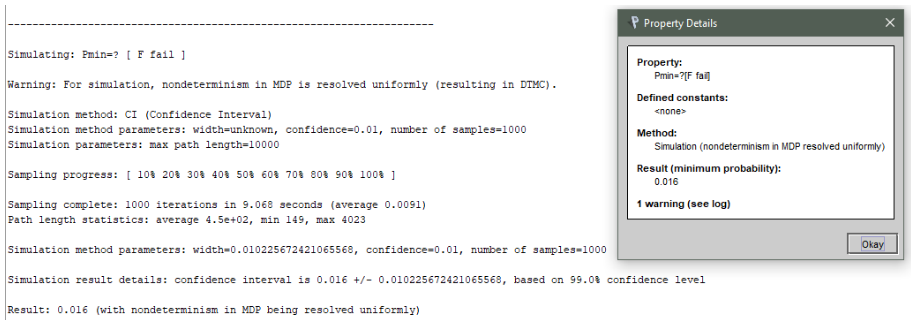


Figure 2.4: Al the options you have when clicking on a property.

We will be using `Pmin=?[F fail]` in this thesis to find the minimum chance of losing in the model. For the statistical analysis we will be using Confidence Interval (CI). The specifics of CI are better explained in this paper [15], but the short version is as follows.
Let the calculated probability be P. Using the variables confidence α and width w, and amount of samples, CI can determine that the real probability has a $100 \times (1-\alpha)\%$ chance of being in the range $[P-w, P+w]$. CI only needs two of those variables, the missing one can be calculated from the other two.



```
---------------------------------------------------------------------

Simulating: Pmin=? [ F fail ]

Warning: For simulation, nondeterminism in MDP is resolved uniformly (resulting in DTMC).

Simulation method: CI (Confidence Interval)
Simulation method parameters: width=unknown, confidence=0.01, number of samples=1000
Simulation parameters: max path length=10000

Sampling progress: [ 10% 20% 30% 40% 50% 60% 70% 80% 90% 100% ]

Sampling complete: 1000 iterations in 9.068 seconds (average 0.0091)
Path length statistics: average 4.5e+02, min 149, max 4023

Simulation method parameters: width=0.010225672421065568, confidence=0.01, number of samples=1000

Simulation result details: confidence interval is 0.016 +/- 0.010225672421065568, based on 99.0% confidence level

Result: 0.016 (with nondeterminism in MDP being resolved uniformly)
```

Property Details

Property:
Pmin=?[F fail]

Defined constants:
<none>

Method:
Simulation (nondeterminism in MDP resolved uniformly)

Result (minimum probability):
0.016

1 warning (see log)

Okay

Figure 2.5: Output of the simulate function with CI.

Chapter 3

Research

In this chapter we create the 2048 model and perform an experiment with it. First we analyse the source code to determine the rules of 2048. Then we create a prototype based on these rules by hand. Afterwards we iteratively improve the model by decreasing the amount of commands in the model. We create a Python script that generates N by N models. Then we create a policy manager that enacts a certain strategy of playing the game. Using this policy manager we compare a few strategies of playing the game and show that our model can be used for statistical analysis.

3.1 Rules of 2048

This section describes the rules of 2048. We based these rules on the source code [4]. The game starts with a four by four grid of empty cells. At the start of the game two of these cells will be filled with a tile. When a cell is fated to be filled, the tile will either have a value of 2 or 4. Every empty cell has equal chance to become filled. The chance for a new tile to have a value of 4 is 10%, otherwise it will have a value of 2. The player then chooses to swipe up, left, down or right. Actions that do not change the state of the grid are not allowed and disabled. All tiles will move towards the specified direction until they either hit another tile or if they reach the edge of grid. If a tile hits another tile with the same value, they 'merge' and become one tile with double value. There is a bit more nuance to the moving and merging of the tiles which will be explained in a bit.

Once all tiles are moved, one of the empty cells becomes filled with a tile of either value 2 or 4 with the same odds as before. The player loses when the grid is full and they can no longer perform any actions. The player wins when a tile reaches a value of 2048. Players are usually allowed to play after this until they lose, but we will not be modeling after the player wins.

Once the player has specified a direction, the moving and merging of

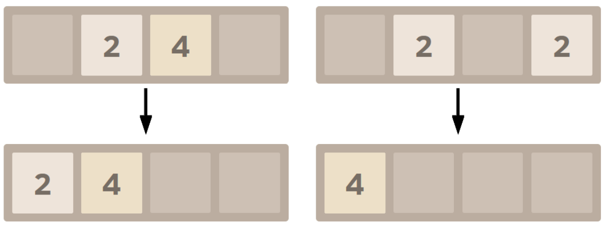


Figure 3.1: With a swipe to the left, all tiles move to the left in this manner. When two equal tiles hit, they combine into one.

all tiles does not happen simultaneously. The tiles closer to the side of the chosen direction are moved first. E.g. with a left action: the tiles on the leftmost column will be handled first (and not moved as they are already at the edge), then the tiles in the column to the right of it will move to the leftmost row if possible, then the tiles on the next column, etc. Take a look at figure 3.2, 3.3 and 3.4 for examples.

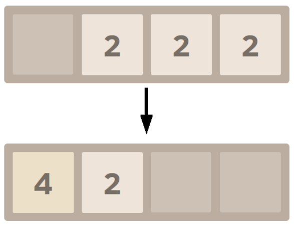


Figure 3.2: Here you can see that, with a swipe to the left, the two leftmost tiles combine.

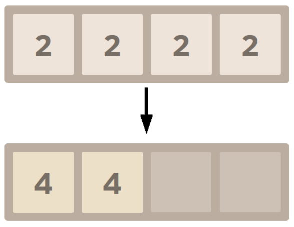


Figure 3.3: In this example, the two rightmost tiles merge as well, but the two newly combined tiles do not merge.

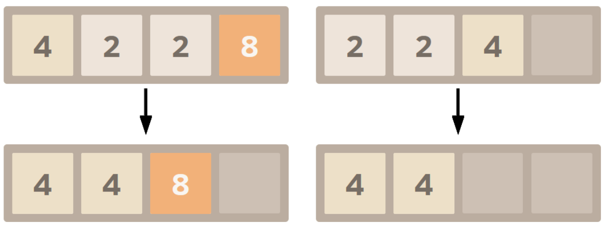


Figure 3.4: Likewise, the newly combined 4-tiles do not merge with the old 4-tile.

3.2 Reduction

To define the 2048 game as an MDP we need to describe all aspects of the MDP as 2048 equivalents. That is, we need to define (S, A, P_a, R_s).

Reducing 2048 to MDP

S is the set of states. We define a 2048 MDP state as the collection of values in the grid. A state will have 16 variables, each corresponding to a single cell. If a cell is empty the value will be zero. If it is not empty, the value will be the same as the value of the tile filling the cell, so either 2, 4, 8, etc.

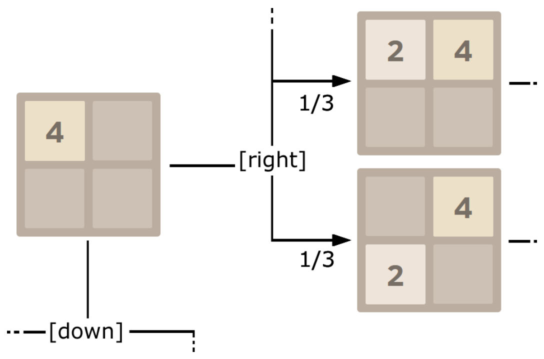


Figure 3.5: General shape of a 2048 MDP

A is the set of actions. There are four actions in 2048, so A in our MDP will consist of the four actions up, down, left and right. P_a is the function that defines the probability of every transition. The probability of a transition will be higher than zero if the transition is possible in the 2048 game. The exact probabilities depends on the action and the amount of empty cells. While the action moves existing tiles in predictable location, afterwards a random empty cell is filled with a 2- or 4-tile. This could grow to 30 different transitions from a single state. We will explain how we handle this complexity in the next subsection. R_s is the function that defines the reward for doing a specific action to a specific state. Defining the reward depends on the goal one has using the model. As such we will not give a definition.

Random and Move-Merge phase

The probabilities for the transitions are quite complex as can be seen in Figure 3.6. With 14 empty cells we get 28 different transitions. Describing such a transition in a single command would require us to make very long and difficult commands. To simplify modeling we split the transition into

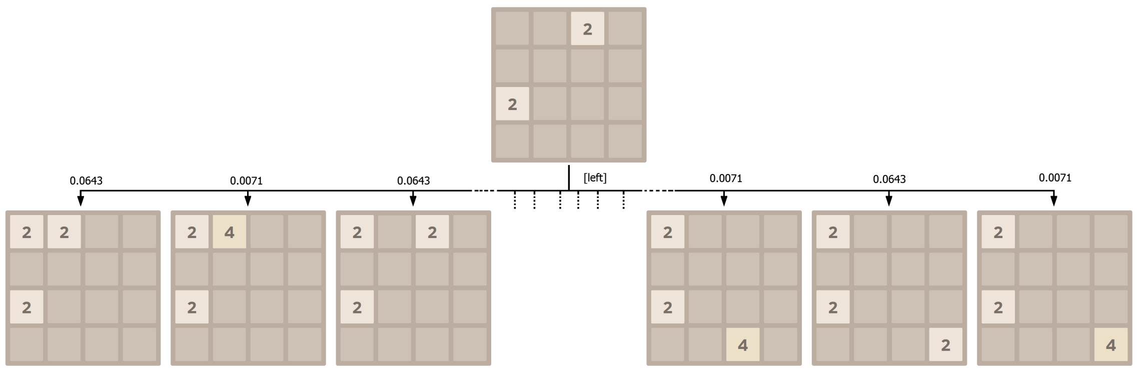


Figure 3.6: How the probabilities of all transitions from a single state could be described.

two distinct phases. These steps can be seen in Figure 3.7. One transition is the Move-Merge phase, where the tiles move and merge in accordance with the rules lined out in the previous section. The second phase is the Random phase, where a random empty cell is filled with either a 2- or a 4-tile. As can be seen in the figures, the probabilities become much simpler. Every single arrow represents a single command. PRISM will choose any available command with equal chance. This choice determines which cell is filled. We then define the probabilities of that cell receiving a 2- or 4-tile. To keep track of the phase, we add a single boolean variable to the state.

PRISM

To create this model we use PRISM model checker and the PRISM language. To write a model, we need to define the state-space and the transitions between the states. The model needs to parse and build quickly so it can be used for formal verification and statistical analysis. We expect the state space to be large, so to improve build time we would like to be able to only partially build the model. We can debug the model using PRISM's simulator. With this simulator we can try out multiple scenarios to find problems with the model. The simulator only builds the model one step at a time, making it possible to use even when building the whole model would take a long time.

3.3 Prototype

To make an MDP with PRISM, we first need to define the complete state space. Following that, we need all the transitions between those states. The goal with this first model was to get familiar with PRISM and to get a

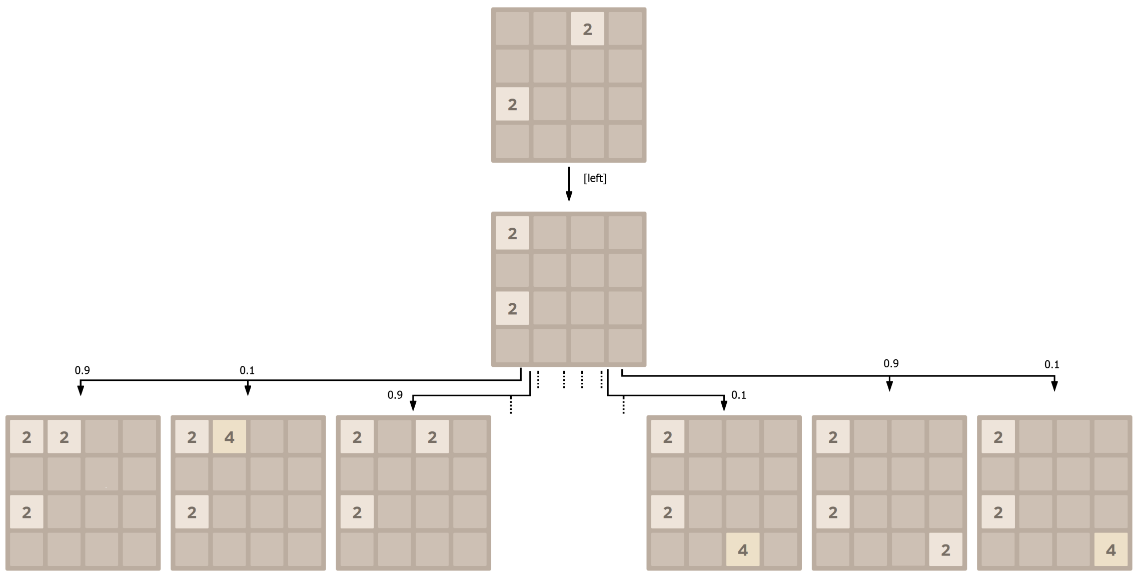


Figure 3.7: Probabilities after splitting the previous transition into two separate phases.

working prototype. To make the design process simpler we decided to make a model of a two by two 2048 game. After completing this model we work to scale this model up by making a generator for two by two and four by four 2048 models.

Modeling choices

There are two main advantages to modeling a two by two first compared to modeling a four by four. First, there are less variables to consider. Modeling all cells would only take 4 variables instead of 16. Second, there are less interactions between tiles. A single row has at most two tiles. The tiles either combine, or they do not. This reduces complexity.

Recall that it is possible to instantiate multiple modules from the same code using the renaming feature.

Because of this renaming feature, we decided to put the behaviour of every cell in its own module. This makes reading and understanding the behaviour of each single cell easier, but does require us to synchronize all modules. Our hope at the time of this decision was that it was possible to rename a single module such that we only need to make one. We will explain later why this was not possible in the end.

Modeling

We begin the model with declaring constants: The maximum value for the tiles, the chance of a new tile having a value of 2 and the chance of a new tile having a value of 4. The maximum value for a two by two is 32. The chance for a new tile to have a value of 2 will be 0.9.

Next we build the modules. We give each module two values: `v_xy` and and `phase_xy`. Where the xy stands for the location of the cell (either 0 or 1, so v_00 for the upper left cell). The phase variable keeps track of the phase the game is in, either the random phase or the move-merge phase. Every module can access all variables from other modules. However, they can only update their own variables, so every module has its own phase variable.

To force synchronization between modules, each command needs to be labeled with a unique action. Take for example the random phase. Only a single cell will get a new tile, but if no synchronization happens, and every module just had a flat percentage chance of getting a new tile, then it would be possible for both no new tiles, all cells filled and everything in between. To stop that from happening, we need to add one action plus command for every possible configuration of filled and non-filled cells in the random phase. The action contains the information of which cell will be filled.

Commands

The module then needs to have the commands describing transitions for the random and move-merge phase. For the random phase we need one command per configuration. In the move-merge phase we need at most four commands (for all directions) per configuration. To keep things organised, we group the commands on the amount of cells filled in the configuration. In the next sections we will describe the different kinds of situations and how we made commands to accommodate them.

Random Phase Commands

The random phase transitions look as follows. This command is from the module of the top left cell. Here the model decides whether an empty cell becomes filled with either a 2 (N=0.9, 90% chance), or a 4 (M=0.1, 10% chance).

```
[f0000_1100] (field_0000 & phase_00=1)}  ->
  N : (v_00'=2) & (phase_00'=0) +
  M : (v_00'=4) & (phase_00'=0);
```

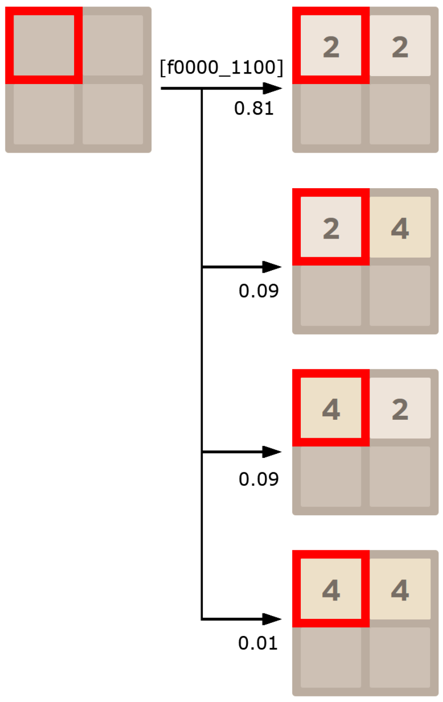


Figure 3.8: All possible transitions with probabilities from this command, the cell in question is marked with the red square

The action [f0000_1100] describes what the action will do. The first four digits describe an empty grid, after the underscore, the top row is filled. An empty grid only happens at the start of the model. However, two random cells are filled at the start of the game. This is one example of the commands that bridges this gap. In this command, the top row will be filled with tiles. This module describes the top left cell, as such we need a command with an update to change its variable v_00. After the arrow are the updates, including the probabilities of each occuring. N is a constant defined as 0.9, M as 1-N. As such, we define that if this action is chosen, all modules will use the command with this same action. When that happens, there is a 90% chance of this cells value to become 2 or a 10& chance of it becoming 4. It's phase variable becomes 0 in both transitions. This command combines with the commands with the same action in all other modules to create the full range of probabilities that can be seen in figure 3.8.

Next is another example from the start of the model.

```
[f0000_0011] (field_0000 & phase_00=1) ->
(phase_00'=0);
```

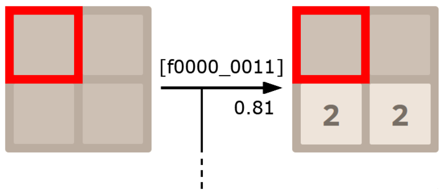


Figure 3.9: All possible transitions with probabilities from this command, the cell in question is marked with the red square

We enforce that this command can only happen at the start of the model with the guard. At the bottom of the file is a long list of formulas defining all configurations of filled and non-filled cells as short names. In this example, `field_0000` only evaluates to true at the start of a model, when all cells are empty. Every digit can be seen as a boolean defining if a cell is filled. The order of the digits is top-left, top-right, bottom-left, bottom-right. Thus we see that the guard only evaluates to true when all cells are empty and we are in the random tile phase. The action before the guard once again ensures that this module is synced with the other modules, even if the value in this specific module does not change in this module. Only the bottom row cells will be filled, as such the only value that is updates here is the phase variable.

A non-start command can be seen in Figure 3.10.

```
[f0100_0110] (no_MAX & field_0100 & phase_00=1) ->
phase_00'=0);
```

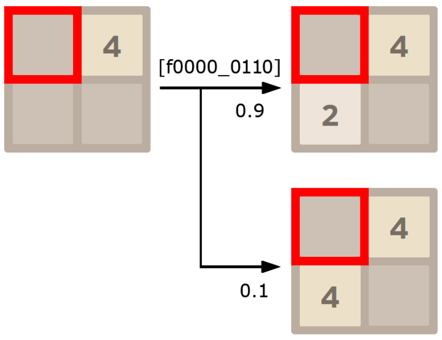


Figure 3.10: An example of a transition for this command, the cell in question is marked with the red square

The `no_MAX` checks if no cell holds the maximum value. If this check is not done, the MDP grows endlessly. `f_0100` checks if only the top right cell is filled. From this configuration three possible target configurations can be made (a new cell at top-left, bottom-left or bottom-right). This command describes the transition that creates a new tile at bottom left. The value of the top-left cell does not matter, as long as it is not 0. This means that the cell could hold 5 different values, which means that this command supports 10 different transitions. This greatly reduces the amount of commands needed. In the model the commands are sorted by initial amount of cells filled. This is because behaviour is very similar between configurations with the same amount of cells filled. This will be much clearer when we look at the move-merge transitions. Grouping commands with similar behaviour will be a recurring theme.

Move-Merge Phase Commands

Now we describe the Move-Merge phase transitions. These are grouped for organisational purposes by amount of filled cells.

```
[down]  (field_1000 & phase_00=0) ->
  (v_00'=0) & (phase_00'=1);
```

Here we first check for configuration and phase. As any action will cause this cell to become empty, we set the variable to 0 and move the phase to 1. Let's look at the command that gets used at the same time as this one from another module:

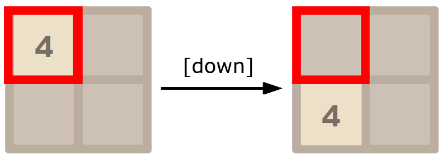


Figure 3.11: An example of a transition for this command, the cell in question is marked with the red square

```
    [down]  (field_1000 & phase_10=0) ->
      (v_10'=v_00) & (phase_10'=1);
```

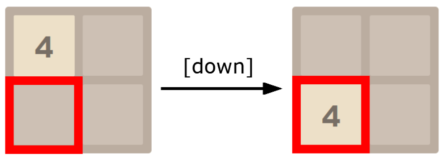


Figure 3.12: An example of a transition for this command, the cell in question is marked with the red square

This is from the bottom-left module. Here its value (v_10) becomes the top-left value (v_00). We know that only these two commands can be used at the same time, the action + field combination is unique for every possible field and command. This results in at most four commands being possible in the move-merge phase, and never more than one with the same action. For every extra command that is possible in any module, the amount of possible transitions double. In the move-merge phase every action should only have one transition. Let us look at yet another module.

```
  [down]  (field_1000 & phase_11=0) -> (phase_11'=1);
```

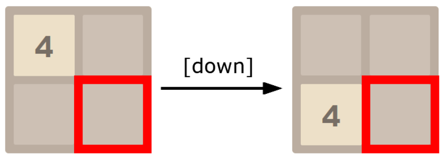


Figure 3.13: An example of a transition for this command, the cell in question is marked with the red square

In the bottom right module, the value does not change with this configuration and action. We still need a command however, as an action can only be chosen if it is possible in all modules. Thus, to prevent desynchronization, we need a command for this specific configuration-action combination as well, even when this cell is not affected. We only change the phase variable here.

Move-Merge Commands with two filled cells

The commands for two filled cells can be split in two groups based on behaviour. One group has the two filled cells next to each other, the other in a diagonal. With the former, there should be two commands that cause the tiles to merge, one that moves the tiles. One direction will not lead to a legal move, so it will be commented out. With the latter, all actions are legal, but none lead to two tiles merging.

```
    //[up]    (field_1100 & phase_00=0) ->  (phase_00'=1);
```

The command above is an example of a command that is commented out because it can never be called. The top row is filled, and cannot move further up. Even though the configuration-action combination is unique for this module, this move is not legal in 2048. By not having this move be a command, it cannot happen.

```
    [left]  (no_MAX & field_1100 & phase_00=0 & v_00=v_01)  ->
        (v_00'=2*v_01) & (phase_00'=1);
```

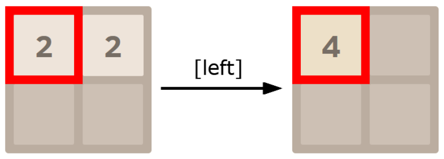


Figure 3.14: An example of a transition for this command

In this command, the top row is filled once again, but the tiles will combine in the top-left cell with a left action. The new value of v_00 becomes two times that of v_01. The no_MAX prevents a merge that would exceed the maximum value a cell can have. This differs from the no_MAX use in the random phase, this no_MAX prevents moves that cannot be made in the move-merge phase. The previous one prevents the game from continuing after the goal has been reached.

```
    [left]  (field_0110 & phase_00=0)  ->
        (v_00'=v_01) & (phase_00'=1);
```

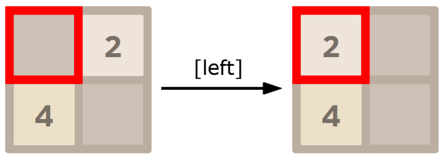


Figure 3.15: An example of a transition for this command

The behaviour of this command is the same as the one filled cells ones. No action can lead to a merge, and all actions are valid moves.

Move-Merge Commands with three filled cells

The three cells filled group is more complex, but can be grouped as one using conditional expressions. With one cell left open, there are four possible commands per action. For every configuration of the grid, all actions are possible under certain conditions. Either the move is valid to move a tile to the one empty cell, or a move is valid because two tiles can merge. This means we get commands like the following:

```
    [up]    (no_MAX & field_1011 & phase_00=0)  ->
        (v_00'=(v_00=v_10?2*v_10:v_00)) & (phase_00'=1);
```

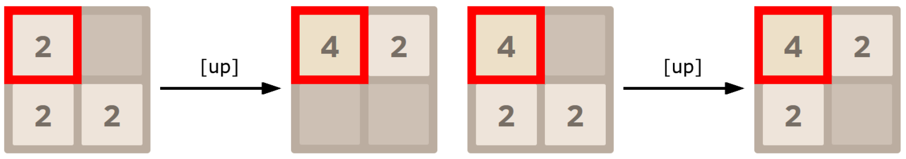


Figure 3.16: Two examples of a transition for this command

There is a possibility of merging, so we invoke no_MAX again. In the update we have a tertiary expression, an if statement. If the values in this column are equal, then this cell's value doubles. Else it stays equal. We need this conditional expression inside of the update instead of the guard for a reason. The up action here is always valid, because the bottom-right cell is filled, but the top-right cell is not. The guard only defines when an action is valid. Seeing as this action is valid, there are still two different behaviours possible, either the top- and bottom-left tiles are equal, or they are not. Thus, we use the conditional expression to describe these possibilities.

Move-Merge Commands with four filled cells

The four tile group needs very few commands. Only one command per action per module.

```
    [up]    (no_MAX & field_1111 & phase_00=0 &
            (v_01=v_11 | v_00=v_10)) ->
                (v_00'=v_00=v_10?2*v_10:v_00)
                & (phase_00'=1);
```

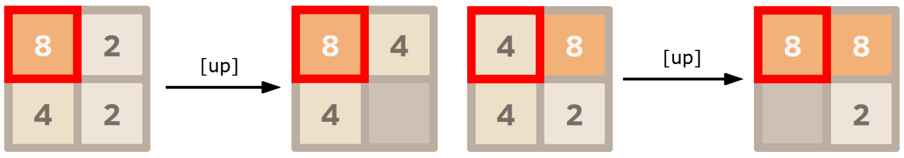


Figure 3.17: Two examples of a transition for this command

This command combines every technique used for the move-merge phase. We need to check for no_MAX because any move can merge. We need to check in the guard if a merge is possible. We use a tertiary expression once more to describe differing behaviours within a single valid configuration+action combination. This is because from passing the guard, we do not know on which side the tiles were equal, only that at least one was. As such we need to describe both once again.

Fail and Win states

```
     [loss] fail -> true;
     [victory] success -> true;
```

To end the module, we put two self loops at the end, this prevents intended dead ends to be marked a deadlock. This also signals to PRISM that this is indeed an end point, it will not model further than this. fail and success here are formulas that are defined at the end of the file as follows:

```
    formula fail = (v_00!=v_01 & v_01!=v_11 &
                    v_11!=v_10 & v_10!=v_00 &
                    field_1111 & phase_00 = 0);
    formula success = ((v_00 = MAX | v_01 = MAX |
                       v_10=MAX | v_11=MAX) & phase_00 = 1);
```

`fail` is when no neighbours are equal and all cells are full, so no merging or moving can happen, losing the player the game. The `success` formula is true when one or more cells has the max value, this only happens when the game is won, ending this model.

These commands may look (and are) simple, but these commands have an important function in the model. These are the states players and agents will want to either avoid or reach. To accommodate this we added labels for these states. This makes it easier to detect these states and makes property expression simpler.

There are problems with this model. As mentioned before, this model makes use of multiple modules without any purpose. Currently the amount of commands can be greatly reduced by putting multiple updates in a single transition instead of distributing them over multiple modules. This will reduce readability, but readability will be less of a problem because we will be generating the model with a Python script. With the next model we aim to put everything in a single module, as well as scale up to the four by four 2048.

3.4 Combining Modules

For our model to be useful, we need it to parse and build in a reasonable amount of time. The simplest way to do this is by reducing the amount of commands. A two by two 2048 game has 4 cells. A four by four game has 16 cells. So far, the model has been made by hand, with commands based on all different configurations for filled-empty cells. The amount of such configurations is 2^N, where N is the amount of cells. For a two by two, that is 16 different configurations. A four by four has 65,536 configurations. Making commands for all these configurations by hand is not viable. We need to find ways to automate command creation. One way is to find groups with similar behaviour and describing them with a single command. First we look at general improvements to the model for the two by two grid. Then we look at how we change the random and move-merge phase commands. Afterwards we will scale this model to the four by four grid.

Improvements from the first prototype

The first improvement from the prototype is that all commands will be in one single module called game_grid. Here we define a single phase variable, rather than the previous four. This alone reduces state complexity by a factor of 8.

Readability is less important now that we generate the model with a python script. As such we won't be declaring `v_00`, `v_01`, `v_10` and `v_11`, but `v_0` to `v_3`. This makes defining the value variables easier for the generator and also makes scaling this operation easy, as we just loop N times, where N is the amount of cells.

We now use a similar loop for the commands that can be called in the random phase. Let N be the amount of cells. The previous model had 2^N filled or non-filled configurations for the grid, described with binary numbers at the bottom of the model with formulas. For example, the formula `f_0100` would evaluate to true when all cells are empty except the top-right cell, as cells are ordered starting top-left, left-to-right. The n'th digit would define the status of the n'th cell.

We want to describe all random phase transitions. With the previous system, that would mean that we would need to have 2^N starting configurations. From any position we could fill one cell, so we need to multiply every instance with the amount of empty cells, further enlarging the amount of commands. Furthermore, we need to handle the starting state of the board, where two cells are filled. Thus we would need $N * N - 1$ different additional starting configurations. We will not need this many commands now that we put all updates in a single module. We will discuss how in the next section.

Random Phase Commands

Because we now use only one module, we can greatly reduce the amount of commands to $2N$. N for the general transitions and N for the start of the game, so two commands per cell. One of these is the following example:

```
[v_1] !( v_0=0 &  v_1=0 &  v_2=0 &  v_3=0) &
        (v_1=0) & no_MAX & (phase=1) ->
        N : (v_1'=2) & (phase'=0) + M : (v_1'=4) &
        (phase'=0);
```

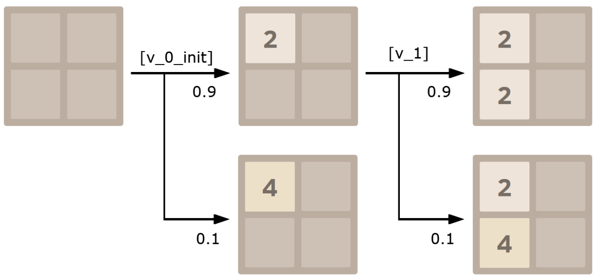


Figure 3.18: Transitions these commands define

While this command has an action, it only uses it for tracking purposes, no synchronization is needed until we start making a policy manager. The guard only evaluates to true if the state is not an initial empty grid, no cell has max value (to prevent the state space from growing endlessly) and the phase is 1. `(v_0=0)` here is the only part that differs between all the commands for every cell, it checks if a single cell is empty. This command is repeated for every cell. Because of this guard, only commands that will fill an empty cell will be considered by the simulator. PRISM chooses between one of these commands with equal probability for each. The only thing left is to begin the game with two random tiles on the grid, which we can do with only one extra command per cell.

```
[v_0_init]  ( v_0=0 &  v_1=0 &  v_2=0 &  v_3=0) &
(phase=1) ->
                 N : (v_0'=2) + M : (v_0'=4);
```

The first part line ensures that the guard only evaluates to true if all cells have a value of zero, a state that is only possible at the very start of the game. On the last line we do not change the phase, thus another random tile will be added by a normal random phase command.

Move-Merge Commands

The commands for the move-merge phase are generated with two different systems. One system defines the guard, one defines the updates. Both of these systems base their output on the filled-empty configuration of the grid and the action. For a two by two, this means at most $2^4 = 16$ commands per action. Some of these commands never lead to a valid transition. Take for example the grid with a single cell filled on the top-left corner, no left or up action will be valid. This lowers the total amount of commands to 12 commands per action. Some transitions are purely possible because of the contents of the filled tiles. Think of a grid with the left side filled. An up or down command will only be possible if both tiles are equal. Thus our guards will need to consist of two important parts, one part that evaluates to true if the grid is equal to the configuration the updates are made for, and one possibly empty part that evaluates to true if the contents of the grid allow for a transition. Thus our guard will contain the grid configuration with a disjunction of extra demands. How these demands are generated will be discussed in the section Model Generation.

Generating updates for the commands

The updates will be generated based on the position of the cell in the row and the amount of cells filled in that row. We first made updates based on a left action on the top most row. We can later convert the updates and guards to different directions and rows. All of these updates are made with the assumption that an exact amount of cells are filled. The updates can be seen in Table 3.1

	cell 0	cell 1
0,1 filled	(c0=0?(c1=0?0:c1):c0)	(0)
2 filled	(c0=c1?2*c0:c0)	(c0=c1?0:c1)

Table 3.1: Cell updates

In this table, the values of the cells are held in the c0 and c1 variables. These are placeholders for the in-model variables. The 0 and 1 cell filled rows are combined into one, as the combining both in a single expression is easy and reduces the total amount of such updates we need to make. In these conditional evaluations we first check if the first cell is empty, then we check if the second cell is empty. If both are empty, then the row had zero filled cells. In all other branches, we know which cell has the filled cell, and can then fill the leftmost cell with that value. For the second cell in this case, we know that it will always be empty afterwards, thus it is only a zero. When we know that there are two filled cells, we get a conditional evaluation very similar to the one used in the prototype. When the cells have equal value we double the value in the first cell and set the value to zero in the second cell. If not, they both remain as they were. One could think that unchanging variables would be against the rules, as an action that doesn't change any values should be illegal. However, because of our guard, we know that either the values *are* equal, or a move is legal because the state of the grid in another row allows the move.

Move-Merge Command Examples

The generator can determine which of the updates to use for each variable based on the guard by calculating how many cells are filled in each row. Take this command as an example:

```
[left] ((!v_0=0 &  v_1=0 & !v_2=0 & !v_3=0) &
    (v_2=v_3)) & no_MAX & phase=0 ->
    (v_0'=(v_0=0?(v_1=0?0:v_1):v_0)) &
    (v_1'=0) &
    (v_2'=(v_2=v_3?2*v_2:v_2)) &
    (v_3'=(v_2=v_3?0:v_3)) & (phase' = 1);
```

The generator is tasked with making a command for a left action where all cells are filled except for an empty top-right cell. The extra condition

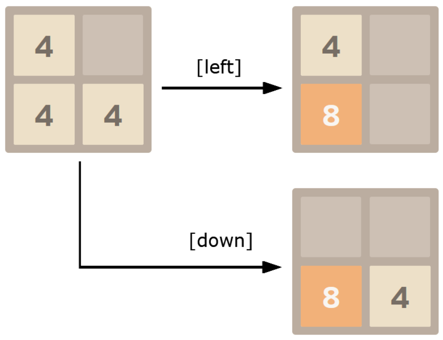


Figure 3.19: Examples of transitions for this and the next command

v_2=v_3 is added because this action can only be legal if this condition is true. This technically does make one conditional evaluation in the updates redundant, as we will check for equality there as well. However, this redundancy is fairly minor and simplifies the model generation substantially. After the arrow, we have one update for every cell, as well as one that changes the phase.

The generator determines which cells are in which rows, the first has v_0 and v_1, the second one has v_2 and v_3. For the first row, it knows that one or zero cells are filled. Thus it uses the top-left formula from Table 3.1 for v_0 and top-right for v_1. For the second row it knows both are filled. Thus it uses the bottom-left for the v_2 update and the bottom-right for the v_3 update. In all of these updates, the correct variables need to be put in place instead of the placeholders. The generator calculates which cells are in which row, and in what order. Take a look it this down action command that stems from the same grid configuration.

```
[down] ((!v_0=0 &  v_1=0 & !v_2=0 & !v_3=0) &
    (v_2=v_0)) & no_MAX & phase=0 ->
    (v_2'=(v_2=v_0?2*v_2:v_2)) &
    (v_0'=(v_2=v_0?0:v_0)) &
    (v_3'=(v_3=0?(v_1=0?0:v_1):v_3)) &
    (v_1'=0) & (phase' = 1);
```

Here we can see that the guard is mostly the same, only the additional requirement part is different. This is because the action has a different direction, which requires different variables to be equal. At first glance one can see that the order of the updates is different. The order matters little and is a result of the generation process. In this case, the first row (now a column) contains v_0 and v_2, with v_2 as the 'leftmost' cell. The updates themselves are from the bottom row of Table 3.1, as both cells are filled. you can see that the variables have changed to reflect the rotated row. The v_1 and v_3 updates follow the same procedure.

To scale this process to a four by four 2048 game, we need 16 different updates for the new grid. The amount of commands that will be generated will also grow because of the large amount of configurations. In the next section we discuss the problems the large amount of commands cause, and how we minimize the amount of commands.

3.5 Merging Commands - Final Model

In this section we will first describe how the updates work for the four by four grid. Then we will discuss the improvements that made the model practical to use. The following updates were made similarly as the updates in the previous two by two model. With placeholder names as described in figure 3.20. The updates are grouped by amount of cells filled. We start with zero or one cell filled.

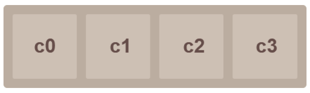


Figure 3.20: Placeholder variable names used in the updates.

Updates for 0 and 1 filled cell in a row

cell 0	(c0=0?(c1=0?(c2=0?(c3=0?0:c3):c2):c1):c0)
cell 1	0
cell 2	0
cell 3	0

Table 3.2: Cell updates for rows with no or one filled cell

The updates in table 3.2 are used for variables where we know that their row has one or zero filled cells. A visual example can be seen in Figure 3.21. Only a single cell is filled, so a left action will only be able to the leftmost cell. We also know that all other cells will be empty after the action. We do need to know what value the cell needs to be. Thus the largest part of this conditional expression is a series of c_=0 conditionals that determine which cell is filled. The value of c0 is then changed into that value. If no cell is filled, c0 stays zero.

Updates for 2 filled cells in a row

The updates in table 3.3 are for two filled cells. The largest part of these updates is in discovering which two cells are empty, for which there are six

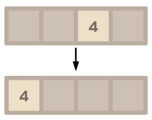


Figure 3.21: Examples of transitions the updates in table 3.2 are made for

cell 0	(c0=0?(c1=0?(c2=c3?2*c2:c2):(c2=0?(c1=c3?2*c1:c1):(c1=c2?2*c1:c1))): (c1=0?(c2=0?(c0=c3?2*c0:c0):(c0=c2?2*c0:c0)):(c0=c1?2*c0:c0)))
cell 1	(c0=0?(c1=0?(c2=c3?0:c3):(c2=0?(c1=c3?0:c3):(c1=c2?0:c2))):(c1=0?(c2=0?(c0=c3?0:c3):(c0=c2?0:c2)):(c0=c1?0:c1)))
cell 2	0
cell 3	0

Table 3.3: Cell updates for a full 2048 grid with two filled cells

options total. After finding the filled cells, the expression evaluates whether the tiles have equal value. If so, c0 becomes the doubled value, and c1 zero. If the two tiles are not equal, c0 gets the value of the first filled cell, c1 becomes the value of the second found filled cell. c2 and c3 always become 0.

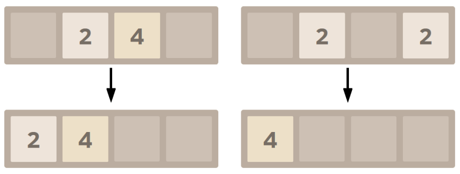


Figure 3.22: Examples of transitions the updates in table 3.3 are made for

Updates for 3 filled cells in a row

Table 3.4 contains the updates when one cell is empty. In these expressions we once again try to find where the empty cell is. After that we determine if the first and second filled cell are equal, and if not whether the second and third filled cell are equal. Depending on these factors, the outcome of each cell changes. For c0 we do not need to know anything about the third filled cell, as such it is a much shorter update. It only needs to know if the first and second filled cell are equal. If these are equal, c1 gets the value of the third filled cell. If the first and second filled cell are not equal, then c1 depends on the second and third filled cell. If they are equal value, c1 becomes double that value. If not, c1 becomes the value of the second filled cell, and c2 becomes the value of the third filled cell. c3 always becomes 0.

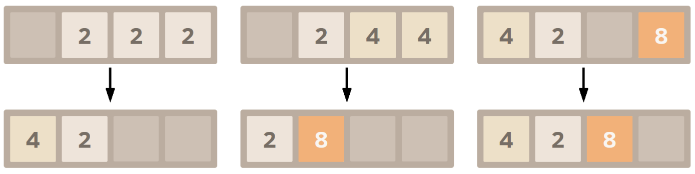


Figure 3.23: Examples of transitions the updates in table 3.4 are made for

cell 0	(c0=0?(c1=c2?2*c1:c1):(c0=c1?(c0=c2?2*c0:c0):(c0=c1?2*c0:c0)))
cell 1	((c0=0?(c1=c2?c3:c2):(c1=0?(c0=c2?c3:(c2=c3?2*c2:c2)):(c2=0?(c1=c0?c3:(c1=c3?2*c1:c1)):(c1=c0?c2:(c1=c2?2*c1:c1)))))
cell 2	(c0=0?(c1=0?(c2=c3?0:c3):(c2=0?(c1=c3?0:c3):(c1=c2?0:c2))):(c1=0?(c2=0?(c0=c3?0:c3):(c0=c2?0:c2)):(c0=c1?0:c1)))
cell 3	0

Table 3.4: Cell updates for a full 2048 grid with three filled cells

Updates for 4 filled cells in a row

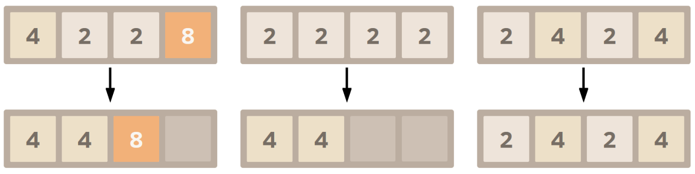


Figure 3.24: Examples of transitions the updates in table 3.5 are made for

cell 0	(c0=c1?2*c0:c0)
cell 1	(c0=c1?(c2=c3?2*c2:c2):(c1=c2?2*c1:c1))
cell 2	(c0=c1?(c2=c3?0:2*c3):(c1=c2?c3:c2))
cell 3	(c0=c1 — c1=c2 — c2=c3?0:c3)

Table 3.5: Cell updates for a full 2048 grid with four filled cells

When we know that all cells are filled, we only need to know what neighbouring cells are equal to each other. `c0'` only depends on the equality of `c0` and `c1`. There are two branches in the expression for `c1` and `c2`, splitting on the equality of `c0` and `c1`. `c3`'s update is very different. We know that it either stays the same value if no cells have equal value (save for `c0 = c2` and `c1 = c3`) or becomes empty. As such we only check if *a* value is equal to its neighbour. If so `c3` becomes 0, otherwise it stays `c3`.

Combining commands

It is not feasible to describe all transitions with the same technique as in section 3.4 with a 4x4 grid. There are $2^{16} = 65536$ different possible filled/non-filled configurations, each of which would need four commands; one per action. Some of these configurations cannot be reached and not all actions are valid in every configuration. This calculation does show the scale we need to reduce. We will now discuss some solutions and how they led us to a solution that worked well.

Merging commands with similar updates

If we create all the commands in the way we have been doing in section 3.4, then we create multiple commands with identical updates. This happens because the updates do not care which cells are filled in a row, only how many cells are filled in a row. We can see this in the following (abbreviated) example:

```
[left] f0010_0000_0000_0000 -> (v_0'=0or1filledcellupdate & (v_1'=0) & (v_2'=0) & (v_3'=0) & 12 other updates;
[left] f1000_0000_0000_0000 -> (v_0'=0or1filledcellupdate & (v_1'=0) & (v_2'=0) & (v_3'=0) & 12 other updates;
```

Despite the difference of the location of the single cell, these two commands have the exact same updates. We can combine these by merging the guards as follows:

```
[left] f0010_0000_0000_0000 | f1000_0000_0000_0000 -> (v_0'=0or1filledcellupdate & (v_1'=0) & 14 other updates;
```

Merging all commands where the updates are the same greatly reduces the amount of commands needed. This way we only need commands for every configuration of every amount of cells filled for every row. Because we combined zero and one we have four possibilities for every row (0
1, 2, 3 or 4). With four rows, we get $4^4 = 256$ updates per action, a massive improvement compared to Combining Modules. However, if you generate this model you might notice that it still takes a long time to parse this model in PRISM. This is because while the amount of updates have decreased, all definitions for the configurations are still in the guards, causing very long commands, which take a very long time to parse. For example, let us take a command for a state with 2,3,4 and 3 cells filled. There are four different ways to make a row with all of these amounts. That means that there are $4^4 = 256$ different configuration definitions in the guard of that command. For every 0 or 1 in the configuration of rows, this number doubles. The first (1111) command contains 4096 configurations in its guard. Thus, we need to find a way to rewrite the guards in a shorter way as well.

Merging commands with similar guards

Our goal is to make an expression that encompasses exactly and nothing more than all possible ways to describe a row configuration. That is, if we want to describe the row configuration 2, 3, 4, 1 (top row has two filled cells, second row three, etc.), we need to find a sufficiently short expression that evaluates to true if and only if the grid has the row configuration 2, 3, 4, 1.

To do this we would like to have an expression that counts how many cells are filled. We cannot use the variables themselves, like in `c0 + c1 + c2 + c3`, because these variables' value ranges from 0 to 32 or higher. We can use the `min(...)` function however. `min(c0, 1)` returns 1 when c0 is filled and 0 when the cell is empty. `min(c0,1) + min(c1, 1) + min(c2, 1) + min(c3, 1)` returns the amount of cells filled in a row with cells c0 to c3. Henceforth this function will be referred to as`count(c0, c1, c2, c3)` to save space. However, in a model it would still be written as the long version. This means that any guard for row configuration a, b, c, d, would look like the following example:

```
count(c0, c1, c2, c3) = a & count(c4, c5, c6, c7) = b &
count(c8, c9, 10, c11) = c & count(c12, c13, c14, c15) = d &
        -> {updates};
```

This is great, but still leaves us with the 256 commands per action. With the new count function, we can do much better.

Merging commands with the same action

Using the same `count()` function we can reduce the amount of commands from 256 to 1 per action. The current amount of commands stems from the fact that we have essentially four different variable updates to choose from four times, $4^4 = 256$. If we can merge two of the four different updates (leaving three), we leave $3^4 = 81$ different commands per action. Say that we want to merge the 3-cells-filled and 4-cells-filled expressions using `count()`. Using `count()` we could make the following expression. Recall that `count()` here is short for the actual expression in the model.

```
[action] guard -> (c0'=(count(c0, c1, c2, c3=3)? 3filledcellupdate : 4filledcellupdate))
```

This can be repeated for all different updates. The model implements the following flowchart

```
                    count()>2?
    no                             yes
  count()=2?                     count()=4?
  no           yes          no           yes
0or1filled     2filled      3filled      4filled
```

This leaves the generator with a single choice for an update. There are still four different updates, one for every cell in a row, but which are determined by the location of the variable in the row. This means that we now only need a single command per action. The guard for this action would be very long if we combined every previous guard, but with only one command, we can rewrite our guard. Instead of an expression evaluating to true if one of many grid states is true, we can evaluate whether the action is allowed, which is much shorter. For a left action, we need to know two things to be true for the action to be legal: Whether a single tile has an empty neighbouring cell on its left, or whether a tile has an equal value tile on its left. Either of these conditions is enough to allow for the move to be made.

```
[left] no_MAX &  (left_movable | left_mergeable ) & phase=0 -> updates;
```

This leaves us with a model with 38 commands. 32 in the Random phase, 4 in the Move-Merge phase, one fail loop and one win loop.

3.6 Model Generation

We wanted an easy way to scale this model. To do this we made a script that generates the model file for us. The script was written in Python 3.7. By running the script in an IDE it generates a file that models an N by N 2048 game. The user can change some constants, including N, the chance of an empty cell getting a 2-tile, the filename of the model, and the maximum value a tile can reach. This script itself only supports 2x2 and 4x4 grids. Other values for N would also work with minor fiddling in update generation and by adding the necessary 3x3 updates.

Most of the generating is done in the `write_file()` function. It writes lines of text to the given filename using the `.write()` function.

Variable generation

Using a simple for-loop, we loop $N * N$ times to write a variable definition for every cell. We also write a definition for the phase variable, initiated on 1. 0 will stand for the Move-Merge phase, 1 will stand for the Random phase. We also create a list with all variable names for the upcoming code to use.

Random phase commands generation

The random phase commands are generated with two for-loops of length $N * N$. The first loop is for the initial commands at the start of the game, the second loop for the normal commands. The first loop does not change the phase variable, adding two tiles instead of one. The first loop c=can only happen at the start because the guard only evaluates to true if the grid is empty. In the second loop, the guard checks if the grid is *not* empty, like in the example below.

```
    file.write('\t//random transitions\n')
    for i in range(N * N):
        full_string = '\t[v_' + str(i) + '] !field_empty & (v_' + str(i) + '=0) & no_MAX & (phase=1) ->
            N : (v_' + str(i) + '\'=2) & (phase\'=0) + '
        full_string += 'M : (v_' + str(i) + '\'=4) & (phase\'=0);\n'
        file.write(full_string)
```

This creates commands such as:

```
[v_1] !field_empty & (v_1=0) & no_MAX & (phase=1) -> N : (v_1'=2) & (phase'=0) + M : (v_1'=4) & (phase'=0);
```

Rotating the board

The move-merge commands are made of multiple parts that are combined together at the end and then written to the file. This happens four times, once per action. We would like to avoid making four separate functions, one for every direction. So instead we 'rotate the board'. We program the functions to base their output on a list of variable names. From this list they can see where all variables are located on the board. For a concrete example, with $N = 2$ a list would look like `(v_0, v_1, v_2, v_3)`. The function assume that the variables are ordered left-to-right top-to-bottom and will use `v_0` and `v_1` for the top row in that order, but for a `[right]` action we would like this order to be reversed. If we rotate the board 180 degrees, and create a new list we would instead get the list `(v_3, v_2, v_1, v_0)`. Now the function would use `v_1` and `v_0` in the now correct order for it's command generation.

Move-Merge phase commands

The command is made of distinct pieces that are combined at the end. The pieces are the action, the guard and the updates. The guard is made of three parts that evaluate to true when, the end of the model has not been reached, tiles can move in the given direction and the model is in the correct phase. Tiles can move either when a tile can merge or when a tile can move. The updates contain the updates for all variables, including the phase variable. The updates in the generated command use the same updates from table 3.1 and table 3.2 and onwards, combined in a single command as described in section 3.5.

We create the guard by generating all neighbour pairs on the board. With these pairs we create the two different expressions. One that evaluates to true if one tile has an empty cell to it's left. The other if the two tiles are equal and not empty. If the neighbours would be `v_0` and `v_1`, then these expressions look as follows:

```
movable: (v_1!=0 & v_0=0)
mergeable: (v_0!=0 & v_0=v_1)
```

All these expressions will be chained in one massive disjunction, as only one has to evaluate to true to allow an action. These large disjunctions are printed at the bottom of the model with the other formulas. In the actual command it will look like `[left] no_MAX & can_left & phase=0 ->`.

Update generation

We need to create a string that contains all the updates in the command. We loop over all variables in the grid. For every variable we create a list that contains all the variables in its row, including itself. We also calculate what its position is in the row. With the row and position the large `generate_update()` function can determine what update the variable needs. `generate_update()` creates all the conditional expressions using a few helper functions that handle syntax to prevent most human errors. This functions contains the updates for all two by two and four by four games. The possibility to add three by three or other sizes is there as well. For a concrete example, here is the code for $N = 2$ if the position of the variable is 0:

```
if N == 2:
    if pos == 0:
        str_fill_1 = prif(preq(var[0], 0), prif(preq(var[1], 0), 0, var[1]), var[0])
        str_fill_2 = prif(preq(var[0], var[1]), prdb(var[0]), var[0])
        str_count = prcount(var)

        return prif(str_count + '=2', str_fill_2, str_fill_1)
```

Which creates the update

```
((min(1,c0)+min(1,c1))=2?(c0=c1?2*c0:c0):(c0=0?(c1=0?0:c1):c0))
```

where `c0` and `c1` are replaced with the actual variable names.

Formula and label generation

The bottom of the model contain formulas and labels. We have described most of these formulas already, but put these at the bottom of the model for readability. There are four formulas not specifically for the Move-Merge phase commands. One for an empty grid to be used by the random phase commands, one to ensure that no cell has maximum value, one for the win-condition, and one for the loss-condition. `field_empty` is simply a conjuction of `v_X=0` where x ranges from 0 to N-1. `no_MAX` is a conjuction of `v_X<=MAX/2`. We test for larger than MAX to ensure that if somehow MAX is skipped, the model is still stopped. the fail condition is created by a conjunction of `v_X != v_Y`, where `v_X` and `v_Y` are neighbours, and a reverse `field_empty`, ensuring that all cells are filled. The win-condition is a disjuntion of `v_X=MAX`, evaluating to true when one variable has reached the goal.

3.7 Policy Manager

We want to do an experiment to show that this model can be used for statistical model checking. Statistical model checking can not be done on MDPs [14], due to its non-deterministic nature. This is also seen in our model, where multiple commands can be chosen at once, e.g. when both left and right are valid actions or when multiple empty cells can be filled. PRISM does allow us to perform statistical model checking, by resolving non-determinism uniformly. This is in our favor, as that is is the exact distribution we want when filling empty cells and when choosing random actions.

We still want to play the model in a certain way to simulate different strategies. There might also be a need in the future to remove non-deterministic choices in our model. Both can be done by creating a new module. This new module will use parallel composition such that only a single action can be chosen in every state, thus removing non-determinism.

Module

When multiple modules have a command with the same action, these commands can only be used if both guards evaluate to true. This is called parallel composition. We can use parallel composition to prevent that multiple actions can be chosen at the same time. For example, say we have a module with a variable `b` that is true if and only if left actions are allowed. Next, the model is in a state where both a left and a down action are possible. We add the following command to the new module: `[left] b=true -> true;`. This command can only be chosen if `b` is true. But if `b` is false, only the down action can be chosen. This concept is used in a new module called `policy_manager` to remove non-determinism and to apply certain strategies.

Random Move-Merge Phase

For our purposes of testing different strategies, we actually do not need to remove non-determinism from the model. This is because PRISM interprets a choice between for multiple different commands as a single command with uniform probability between them. This means that the same path with the following module and without the following module have equal chance of being generated. But for some purposes it might be useful to remove non-determinism from the model.

In a successful implementation of a random strategy, only a single action between left, up, right and down is allowed at a time. To do this we add a single variable move_rn, that holds a random number between 0 and 3, that determines which move will be chosen. We then add four commands, one for each direction, with a guard that only evaluates to true on a certain value of move_rn. For example, only the left command is available when the random number is 0. The transition for this command changes move_rn to a new number, where each number has equal chance of being chosen. The random number can be a number for an illegal action, causing a deadlock. To prevent this we add a command that resets move_rn, but if and only if the chosen number corresponds to an illegal move. This way, every legal move has equal chance to be chosen.

```
module policy_manager
    m_rn : [0..3];
        [left]  m_rn = 0 -> 0.25: (m_rn'=0) + 0.25: (m_rn'=1) + etc.
        [up]    m_rn = 1 -> 0.25: (m_rn'=0) + 0.25: (m_rn'=1) + etc.
        [right] m_rn = 2 -> 0.25: (m_rn'=0) + 0.25: (m_rn'=1) + etc.
        [down]  m_rn = 3 -> 0.25: (m_rn'=0) + 0.25: (m_rn'=1) + etc.
        [rn_reset] !fail & phase = 0 &
            ((!can_left & m_rn=0) | (!can_up & m_rn=1) | etc. )) ->
         0.25: (m_rn'=0) + etc. ;
```

The !fail is there to ensure that this command can only be chosen if the game is in progress.

This implementation does have a downside. Although rare, it is technically possible that the move_rn reset loops infinitely, by choosing the same illegal move over and over. This also means that the total amount of steps in a path does not correspond exactly to the amount of moves by the player. This downside does not affect our experiment, as path length does not matter to us, and an infinite loop is very rare.

Listed Preference Move-Merge Phase

Creating a simple strategy that is not random is less complicated. For our experiment we want to have a strategy that favors one move over the others. The following strategy will always take a left action if the left action is available. Only when the left action is not available (the formula for this is !can_left) will the up action be chosen. Similarly for all action, when no action above them is available can the action be chosen. This strategy does not add extra steps and cannot become an infinite loop like the random strategy.

```
    [left]  true -> true;
[up] true & !can_left -> true;
```

```
[down] true & !can_left & !can_up -> true;
[right] true & !can_left & !can_up & !can_down-> true;
```

One other strategy is to alternate between two orthogonal actions. We can create this behaviour by using two preference lists and alternating between them. We add boolean variable to the module. One list is only available when the variable is true and the other when false. The boolean switches values when the top choice is taken. Technically, the bottom two actions from both lists can be combined.

```
    turn : bool init true;

    [left]  turn=true -> turn=false;
[up] turn=true & !can_left -> true;
[down] turn=true & !can_left & !can_up -> true;
[right] turn=true & !can_left & !can_up & !can_down-> true;

[left]  turn=false -> turn=true;
[up] turn=false & !can_left -> true;
[down] turn=false & !can_left & !can_up -> true;
[right] turn=false & !can_left & !can_up & !can_down-> true;
```

Random Phase

As with the random move strategy, this implementation does not affect our measurements in the following experiment. It can, however, be useful when non-determinism must be removed. For this we use the same technique as with the random strategy. We add a random number to the module and only allow a cell to be filled if the random number is equal to the cell's number. If the random number corresponds to a command that cannot be used, then the random number is reset until a check can be filled.

```
    rn : [0..N]
    [v_X]  seed=X -> 1/N: (seed'=0) +  1/N: (seed'=1) + etc.

    [reset_rn] !success & phase=1 &
    ((!v_0=0 & rn=0) | (!v_1=0 & rn=1) | etc. ->
    1/N: (rn'=0) +  1/N: (rn'=1) + etc.
```

The `!success` is there to ensure that this command can only be chosen if the game is in progress.

You can see here that we need one command per cell. The transition also contains as many updates as there are cells. Luckily these can easily be generated by the model generator, by looping over the amount of cells.

3.8 Experiment

To show that our model can be used for statistical analysis, we will be doing a small experiment. In this experiment we will be comparing four different strategies of playing 2048, using the PRISM simulate function.

3.8.1 Method

We will compare four different strategies to complete a full 4x4 cells 2048 game. We will use the model created by the script with N = 4 and MAX = 2048, 1024, 512 and 256. The three strategies will be:

- Random, the manager picks a random move from all available moves. Every available move has equal chance to be chosen.
- List of preferences, the manager will always pick a certain move when that move is available. If that move is not available, then it will pick the second move from the list, etc. One order will be: up, down, left, right. The other order used is left, up, down, right. One has the top two choices be opposite directions. With the other, the top two directions are orthogonal.
- Alternating lists of preferences. This simulates how humans tend to play the game. The manager has not one, like in the previous example, but two preference lists, and alternates between them when the top choice is picked in either of them. The two lists used where: up, left, down, right and left, up, down, right.

We will calculate the probability of a certain strategy resulting in a fail state. That is, we will calculate the property `Pmin=?[F fail]`. The minimum probability of a path ending in a fail state. We cannot calculate this because of the large size of the model, but we can apply statistical model checking. The statistical model checking algorithm used will be Confidence Interval (CI) [15], with a confidence of 0.01 and 1000 samples. CI will calculate the width. Recall that a confidence of 0.01 means that there is a 99% chance that the calculated probability has a difference smaller than the width with the real probability.

3.9 Results

As can be seen from figure 3.25 and the table 3.6, the opposite preference strategy has the lowest chance to get far in the game. Both orthogonal preference and alternating have a decent chance of getting far, with orthogonal preference performing a bit better. None of the tested strategies had a solid shot at the finish line however. All strategies have trouble with reaching the 1024 mark, and perform about equal from this point.

	Random		Preference	(opposite)	Preference	(orthogonal)	Alternating	
Max	Pmin	w	Pmin	w	Pmin	w	Pmin	w
128	0.138	0.02811	0.722	0.03651	0.010	0.00811	0.014	0.00957
256	0.537	0.04063	0.980	0.01141	0.134	0.02776	0.193	0.03216
512	0.873	0.02714	0.999	0.00258	0.583	0.04018	0.671	0.03829
1024	0.971	0.01368	1.0	0.0	0.965	0.01498	0.989	0.00850
2048	0.997	0.00446	1.0	0.0	0.998	0.00364	0.999	0.00258

Table 3.6: Minimum chance of losing the game using various strategies before reaching a max value tile.

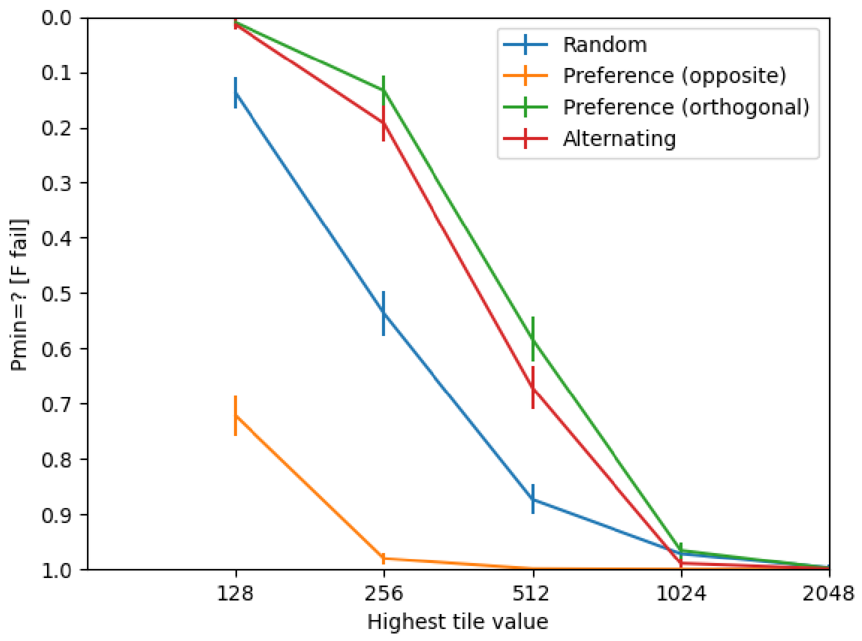


Figure 3.25: Minimum chance of getting a game over before reaching the highest value tile, with errorbars representing the width

Chapter 4

Related Work

In **Safe Reinforcement Learning via Shielding** [1] Alshiekh et al. introduces the concept of shielded decision making. The resulting policy of a reinforcement learning algorithm is good at maximising reward, but does not guarantee safety. To guarantee this safety they introduce the *shield*. They propose two different ways to implement this shield, either before the agent decides, or after the agent decides. When the implemented before, the shield removes all unsafe actions from the list of actions. In the case of after, the shield changes the action to a safe action after the agent chooses an unsafe action. The shield calculates which actions are safe or unsafe using a model of the environment the agent is in, usually an MDP. Using temporal logic, the shield calculates which states must be avoided, and which actions are safe/unsafe. Tested shielded agents performed at least as well as non-shielded agents in the performed experiments.

In **Shielded Decision-Making in MDPs** [9] Jansen et al. discuss the use of shielded decision making in the exploration phase of reinforcement learning algorithms. They constructed shields for PACMAN and a model about avoiding robots in a warehouse. To construct a shield an MDP is needed. This was done by first creating a behaviour model of the ghosts and robots. Then this behaviour model was converted to an MDP. Finally a shield is constructed that tries to prevent the agent fro choosing an action that leads to a game-over. Agents that learned with a shield performed much better than agents that learned without

Safety-Constrained Reinforcement Learning for MDPs [9] offers another way to compute safe and optimal strategies in an environment with "random choices, unknown cost, and safety hazards." S. Junges et al. use a *permissive scheduler* to allow multiple actions. These actions are calculated to be safe, after which exploration can happen within the bounds of this scheduler.

In a series of blog posts on his personal website, **John Lees-Miller** analyses 2048 using various mathematical techniques. In the first blog [12] he asks how many moves are needed on average to complete the game. To find this he created a simplified version (without tile position) of the game as a Markov Chain. Then he sampled one million paths from this chain to find the answer. In the fourth and final blog [13], Lees-Miller attempts to find optimal play using Markov Decision Processes. He manages to find an optimal policy for 2x2, 3x3 and 4x4 until tile 64. The creation of the final MDP "took roughly one week on an OVH HG-120 instance with 32 cores at 3.1GHz and 120GB RAM." The reason we made our own model is because our PRISM model can be used for a wider variety of purposes, including statistical analysis.

Chapter 5

Conclusions

This thesis presents a PRISM model of the game 2048. Furthermore, we present a Python script that can generate 2x2 and 4x4 2048 PRISM models. This script is also able to make NxN models when the cell updates for a given N are added. The model splits a usual 2048 turn in a 'move-merge' and 'random' phase. This split allowed us to create a model with few commands, allowing it to be parsed quickly. This makes it possible to perform statistical analysis on the model. The script also has the option to add a policy manager to the model. The policy manager played the game with a certain strategy We compared four different strategies using statistical model checking and found that some strategies, such as alternating orthogonal directions, can reliably get you to high values like 256 and 512, but none of the tested strategies were good enough to reach 2048.

Future Work

This model was made so that shielded decision making could be tested on the game 2048, but building the full MDP is infeasible for modern computers. So future research could look into implementing a shield on a partial MDP made using a program such as Storm model checker [6]. Future research could also investigate so-called counterexamples to point to faulty or bad behavior in the model. This could be done using research by T. Han et al. [7] or by N. Jansen et al. [10]. More specifically, C. Dehnert et al.'s research on PRISM debugging could be of use [5]. These require a model to be a DTMC instead of a MDP, but the techniques used in the policy manager could be used to reduce our model to DTMC.

More work could also be done by augmenting the model with partial observability to account for the fact that the next number is not known. Related research to model checking with POMDPs includes work by S. Carr et al' [3], L. Winterer et al. [17] and E. Walraven et al. [16].

Bibliography

[1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *CoRR*, abs/1708.08611, 2017.

[2] C. Baier and J. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[3] Steven Carr, Nils Jansen, Ralf Wimmer, Alexandru Serban, Bernd Becker, and Ufuk Topcu. Counterexample-guided strategy improvement for pomdps using recurrent neural networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5532–5539. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[4] Gabriele Circulli. 2048. `https://github.com/gabrielecirulli/2048`. Last Accessed: 17-12-2019.

[5] Christian Dehnert, Nils Jansen, Ralf Wimmer, Erika Ábrahám, and Joost-Pieter Katoen. Fast debugging of prism models. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 146–162, Cham, 2014. Springer International Publishing.

[6] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. *CoRR*, abs/1702.04311, 2017.

[7] T. Han, J. Katoen, and D. Berteun. Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering*, 35(2):241–257, 2009.

[8] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-coonstrained reinforcement learning. *CoRR*, abs/1801.08099, 2018.

[9] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in mdps. *CoRR*, abs/1807.06096, 2018.

[10] Nils Jansen, Ralf Wimmer, Erika Ábrahám, Barna Zajzon, Joost-Pieter Katoen, Bernd Becker, and Johann Schuster. Symbolic counterexample generation for large discrete-time markov chains. *Science of Computer Programming*, 91:90 – 114, 2014. Special Issue on Formal Aspects of Component Software (Selected Papers from FACS'12).

[11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[12] John Lees-Miller. The mathematics of 2048: Minimum moves to win with markov chains. `https://jdlm.info/articles/2017/08/05/markov-chain-2048.html`. Last Accessed: 10-6-2020.

[13] John Lees-Miller. The mathematics of 2048: Optimal play with markov decision processes. `https://jdlm.info/articles/2018/03/18/markov-decision-process-2048.html`. Last Accessed: 10-6-2020.

[14] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[15] V. Nimal. Statistical approaches for probabilistic model checking. MSc Mini-project Dissertation, Oxford University Computing Laboratory, 2010.

[16] Erwin Walraven and Matthijs T.J. Spaan. Point-based value iteration for finite-horizon pomdps. *The Journal of Artificial Intelligence Research*, 65:307–341, 2019.

[17] L. Winterer, S. Junges, R. Wimmer, N. Jansen, U. Topcu, J. Katoen, and B. Becker. Motion planning under partial observability using game-based abstraction. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2201–2208, 2017.