# **2048**

# 20 (Slightly) Different 2048 VersionsProject Summary

The idea of the game of 2048 is that you move and merge tiles with equivalent values to generate tiles with larger values. The grid on which you can move the tile is a 4 \* 4 grid. Every time you can move in one of the four directions: up, down, left, right. Once a tile with value 2048 is being created, you win; if there is a deadlock, which means there are no possible ways to further make a move or merge, you lose the game. We would like to model the logic behind this game. Particularly we would like to model the mechanics of movements. We will be diving into the status of each tile before and after movements, whether they can move, and toward which direction they are moving.

# Propositions

# Location (tile, loc): tile t is at location loc, the location is where on the board that things exist

# empty (tile, orientation): check if adjacent places are empty

# EmptyTile (tile, loc): check if the tile at a location loc is empty

# CanMerge (tile, orientation): check if the tile can merge

# able\_2\_move (tile, orientation): check if at least 1 move can be made on the given orientation

# random (loc): after a move, a random new tile is generated at an location

# NextTile (tile, orientation): check the next tile in the given orientation, if the next tile is of the same value, return true

# row\_or\_column\_can\_move(orientation): check if a row or column can move

# Constraints

* Location(t1, loc) Location(t2, loc): No two tiles can be put to the same location, for tiles t1 and t2 with location loc
* NextTile (tile, orientation) able\_2\_move (orientation)CanMerge (tile, orientation): If a move can be made and there the next tile in the given orientation is as of the same value. then they can merge
* ¬ EmptyTile (loc) ¬ random (loc): no new tile can be generated on a not empty location
* get (nextTile, orientation) empty (tile, orientation) able\_to\_move (tile, orientation): If the next tile in the given orientation is as of the same value, if the next tile in the direction is empty, then we can move.
* able\_to\_move (t1, orientation) able\_to\_move (t2, orientation) able\_to\_move (t3, orientation) able\_to\_move (t4, orientation) row\_or\_column\_can\_move (orientation): If at least 1 out of 4 tiles on the same row or column (depending on the orientation of movement, i.e. we use row if move left/right, use column if move up/down) can move, we are able to make at least 1 move in that orientation
* Only 1 tile is generated randomly among the all the empty locations.

# Model Exploration

## Refining Our Propositions and Constraint

In our previous proposition, we used propositions U and L to declare the direction we are moving. However, we noticed that these propositions may be too hard to be used in the constraints and make things more complicated. Therefore, in our new propositions, we removed U and L. Instead, we used the parameter orientation to declare the direction.

In our previous proposition, to determine if two tiles can merge, we must check if, in the given direction, the adjacent next tile has the same value as the original one or the tile that has the same value on the row/column (depending on the direction) it is on, and the spaces between them are empty. This expression over here already seems too long and complicated. Therefore, we try to simplify the proposition by splitting it down. We introduce a new proposition called row\_or\_column\_can\_move (orientation) and we know that this is a result of some manipulations of our propositions. We then have able\_to\_move (t1, orientation) able\_to\_move (t2, orientation) able\_to\_move (t3, orientation) able\_to\_move (t4, orientation) row\_or\_column\_can\_move (orientation), which gives us the same result as we had before but using simpler combinations of propositions.

## Figuring Out the Data Type of the Tiles

Before we tried to create our tiles using types, i.e. we have type “2”, type “4”, … type “2024” tiles. However, it came to us that we need to deal with at most 16 tiles on the grid/board, therefore we could have a name for each of the 16 tiles and then try to model their moving mechanics. We thought at first we could use a dictionary of 16 entries to store the name of each tile as keys and then the values they bear as items. However, it came to us that it is difficult to create a new key and assign value to that key, and then update the value of the corresponding tile when a merge happens. Therefore, we have tried to figure out a new way to model the tiles. We are still progressing.

A computer screen with text

Description automatically generated

## Differentiating Each Tile from Another

Previously, we were using a dictionary to model our grid with the coordinates as keys and their numerical values as values. This turned out to be problematic as many of the constraints and propositions require us to check the coordinates. It would be hard to check the coordinates, so instead of them directly being the keys, we are creating tiles as objects, which would then be inserted as keys into the dictionary.

# Jape Proof Ideas

We haven’t started this yet but we have some rudimentary ideas on some proofs we may try later.

* Location (tile i, loc) Location (tile j, loc)
* New tile generation is only at vacant locations

# Requested Feedback

1. *How are the propositions and constraints of our model? We have been told that we need to focus more on the mechanics of tile movements on the feedback we received earlier, and we have tried hard to modify our propositions and constraints to make them more valid.*
2. *What data structure would you suggest us to use modeling our tiles? How can we properly model our tiles?*
3. *What would be a recommended way for us to implement the “merge” feature in the game 2048, if you had played it before?*
4. *We have difficulties determining which of our propositions should be made as objects using classes and which not.*
5. *We have difficulties trying to model our constraints. Specifically, we are not sure when to add a constraint in our code, and for some constraints (e.g. only 1 tile is in each location; this is particularly difficult for us because we do not know what data structure to use for tiles), we do not know how to add them (e.g. we have created classes such as NextTile and CanMerge but we do not know how to use them in creating constraints in python).*
6. *We know that when the key-item pair of a location and the tile on it changes, we have made a move. At this point, we have some idea how to model when tiles CAN move, however, we do not know how to write the code implementing the movement.*
7. *Halllllp!*

# First-Order Extension

We have not started this yet.

# Useful Notation

*Feel free to copy/paste the symbols here and remove this section before submitting.*