Final Year Project Report

Full Unit - Final Report

Solving 2048

Daniil Kuznetsov

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science (with Artificial Intelligence)

Supervisor: Dr Argyrios Deligkas

Department of Computer Science
Royal Holloway, University of London

March 25, 2022

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14657

Student Name: Daniil Kuznetsov

Date of Submission: March 25, 2022

Signature: Kuzy

Table of Contents

Abstract

This project makes use of various AI and Machine Learning strategies and techniques to design algorithms, which either directly or via a prior trained model, solve the game 2048 in an optimal under specified constraints (if any). The game itself entails a stochastic game environment contained within a 4x4 grid, which differs from deterministic games such as Sudoku, by having a set of potential states which can be generated after each move each sampled from a uniform distribution. Thus this problem cannot be simply modeled and solved solely using the theory from the Constraint Satisfaction Problem domain, but can however be described as a Markov Decision Process given that the player moves have an impact on the set of potential states, which can be generated after each move made by the player. The goal of the game is to reach the number 2048 by combining the numbered squares on the grid, which can be achieved via methods such as Depth-First Search (DFS) with the Bellman equation, Lookahead, Monte Carlo Tree Search and Reinforcement Learning approaches. Some of the strategies will play the game optimally, such as DFS with Bellman equation, but are computationally infeasible for the complete specification of 2048, hence approximating methods such as Lookahead and Reinforcement Learning have to be used to process the massive state spaces generated by the rules of the game. The performance of algorithms is evaluated and compared when possible, as not all will be able to play equal versions of the game without additional constraints to simplify 2048 and hence reduce its state space, which may be unique to the algorithm in question. The goal of this project is to find the best computationally feasible algorithm, which can solve 2048 optimally whilst obtaining the highest win rate among all other candidate algorithms.

Chapter 1: Rules of the game

1.1 Brief overview

2048 is a simple game to play, but that does not imply that it is easy to solve. This is because of the immense state spaces which can be generated via the application of the 4 actions the player can execute. The merging of nodes after each action along with the procedural generation of a new node into the grid, allows the game system to terminate after a sufficient number of moves within the restricted 4x4 grid space, but the merging property makes it difficult to precisely estimate the volume of possible states which can be generated by the game.

This chapter focuses on going over all of the components which make up the specification of 2048, as well as the impacts they will have on creating an algorithm that is able to solve the game.

1.2 Nodes

The simplest attribute of the game is a single node with a value, this value is traditionally a power of 2 and hence numbers such as 2, 4, 8, 16... are all valid values that a single node can possess. The absence of a node can be thought of as a node with a null value, which obeys slightly altered game logic or instead can be instead pictured as empty space which can be filled by a value.

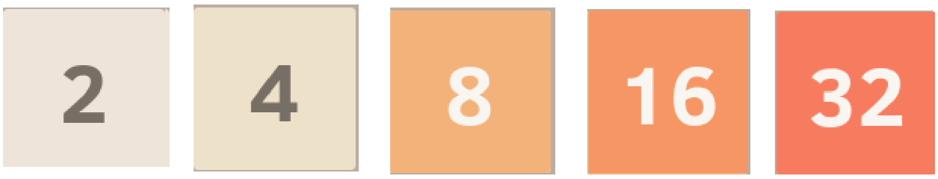


Figure 1.1: Nodes from 2048, usually denoted via use of different colours

1.3 Grid and states

The nodes are stored within a 4x4 Grid in the default version of the 2048 game, these can be thought of as numbers within a 2 dimensional matrix where empty nodes are represented by a value of 0. Hence each value node has a unique position, within the grid represented via indexing of the corresponding grid matrix.

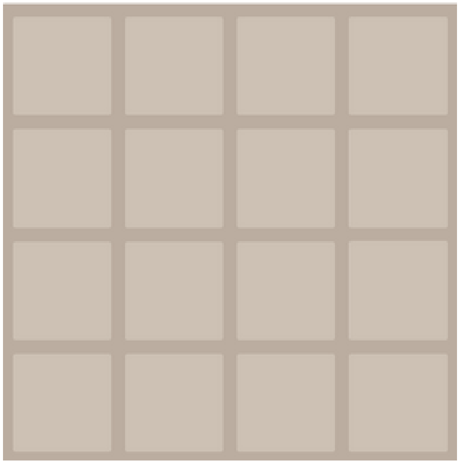
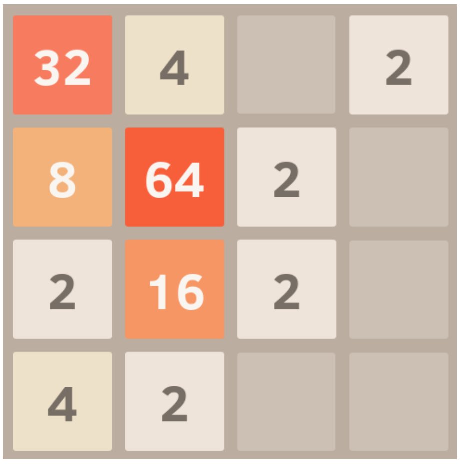


Figure 1.2: An empty grid, can be filled with value nodes

A state is a possible instantiation of the game grid, more precisely a grid of nodes with potential assignments of values. Two states are considered equal if at each position within the grid, the nodes satisfy an equality i.e. they have the same value. A state space is a set of all the possible states that can be generated from a given starting state, by applying every combination of moves a player can make, these will be discussed in the upcoming actions section.

In the project implementation each type of node has a unique representing class, that inherits from an abstract base class Node. This allows the node classes to override the abstract class methods with their own implementations most suitable for each node, hence a software engineering technique of delegation is used to avoid now unnecessary type checks for each type of node. This also makes the project implementation simpler to scale, by only requiring to creation of a new class and inheriting from the abstract Node class, when adding new types of nodes without having to modify any of the already existing logic of the other node classes as they are independent.



32	4		2
8	64	2	
2	16	2	
4	2		

Figure 1.3: A possible game instance within 2048

1.4 Actions

There a 4 moves a player/agent can make: slide up, slide right, slide down, slide left. These will from now on be referred as actions, and were originally represented by swiping the screen of the device on which the game was being played. The follow subsection discuss the semantics of each action and the effects they have on the generation of the following game states.

1.5 Movement

Each slide shifts every value node within the grid in the corresponding direction, as long as the position being moved into is an empty node or until a halt condition is met, the details of these conditions will be clarified in an upcoming section. Each action can be described as:

- **Slide UP:**

  Shift every node along the positive y-axis, or decreasing row if representing the grid as a matrix.

- **Slide RIGHT:**

  Shift every node along the positive x-axis, or increasing column if representing the grid as a matrix.

- **Slide DOWN:**

  Shift every node along the negative y-axis, or increasing row if representing the grid as a matrix.

- **Slide RIGHT:**

  Shift every node along the negative x-axis, or decreasing column if representing the grid as a matrix.

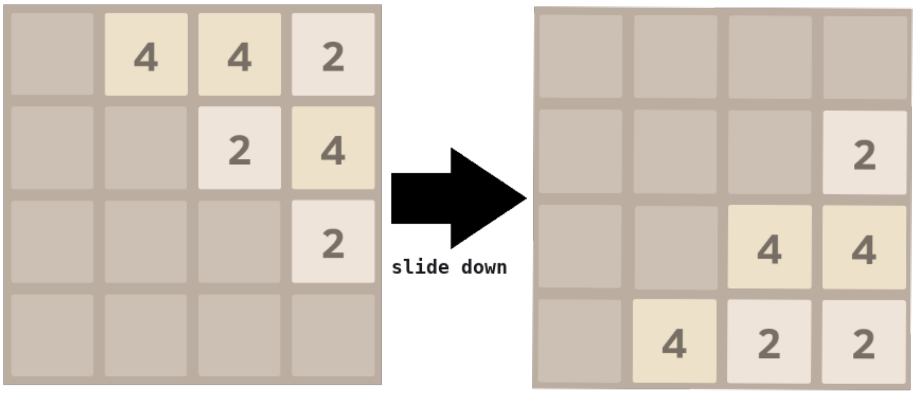


Figure 1.4: Transformation of nodes after performing a slide **DOWN**, note that a new node was not generated for simplicity

1.6 Generation of nodes

This is simple component of 2048 transforms it into a non-trivial to solve stochastic system, after every move a player/agent takes a new value node will be generated at random in an available, empty square within the game grid following a uniform probability distribution.

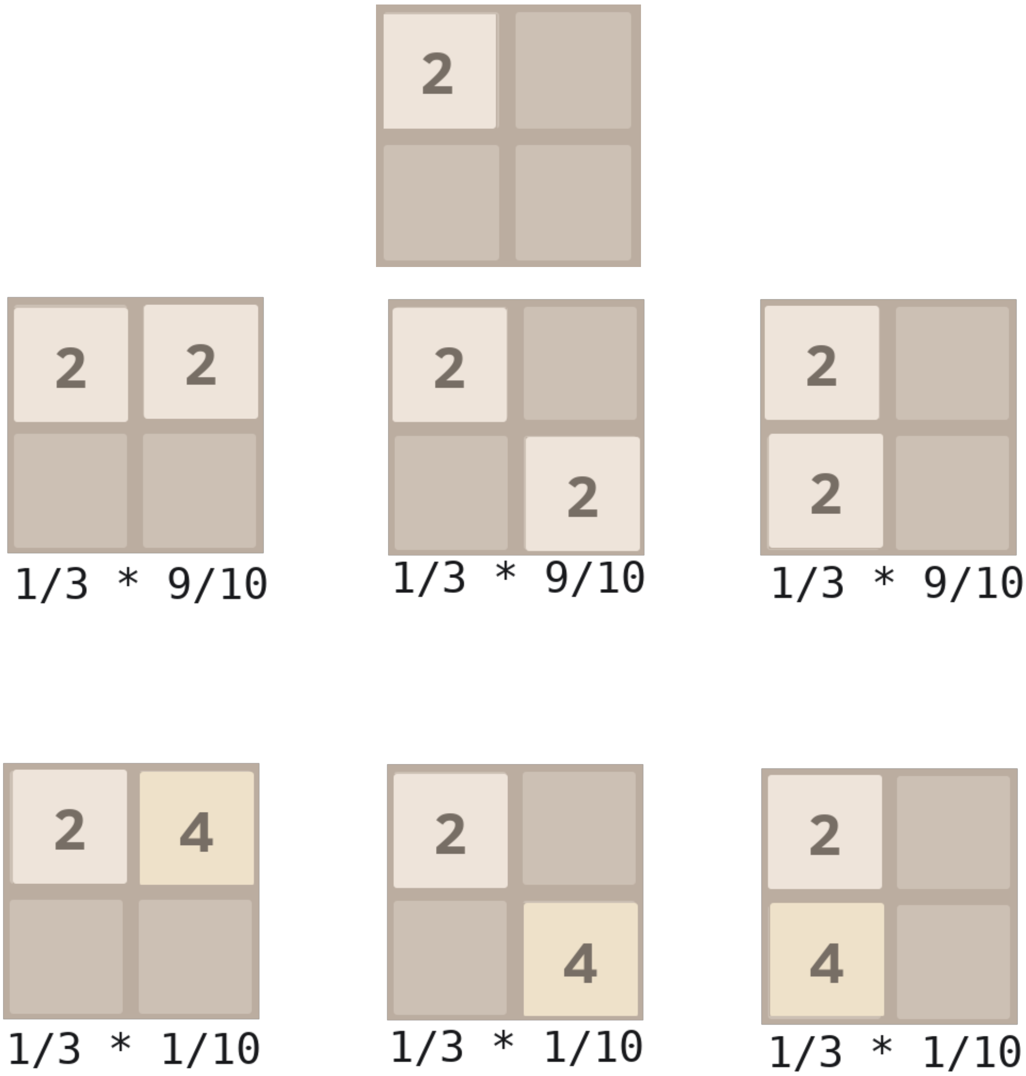


Figure 1.5: Probabilities of each possible outcome of node generation, from the start state given at the top

The value this node possesses in the original iteration of 2048, is 2 with probability of 90% and 4 with the probability of 10%. Of course these are parameters that can be modified, such as setting the probability to 100% for either of the values to essentially half the number of possibilities that could be generated, resulting in a smaller overall state space. This has been used in the early versions of the optimal algorithm, which could not at the time process the random generation of value nodes 4's correctly.

1.7 Merging

The merging procedure is the only way to reduce the number of existing nodes within the grid, this property is one of the key reasons that result in the generation of immense state spaces.

When two identical nodes collide, a new node is created at the point of collision whose value is the sum of values of the colliding nodes. The colliding nodes will be destroyed and removed from the game grid, leaving a single node after the merging.

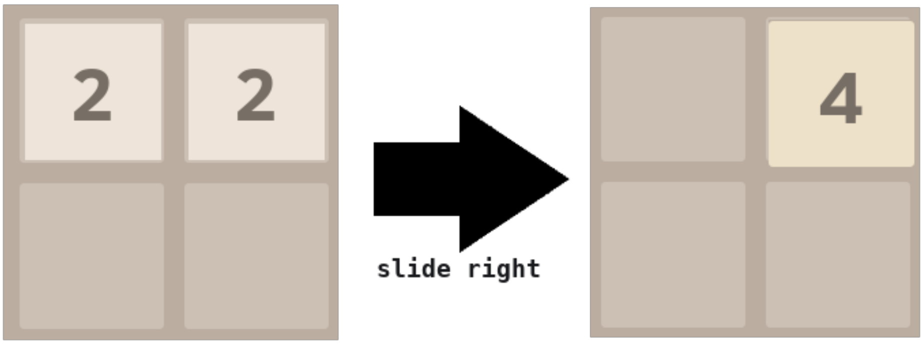


Figure 1.6: Merging

An important property of the merging procedure is that only a single merge can be performed per action, even if it would have been possible to perform additional merges from the newly merged nodes. Multiple merges are not allowed by the game logic, an additional action will have to be made to merge newly created matching value nodes.

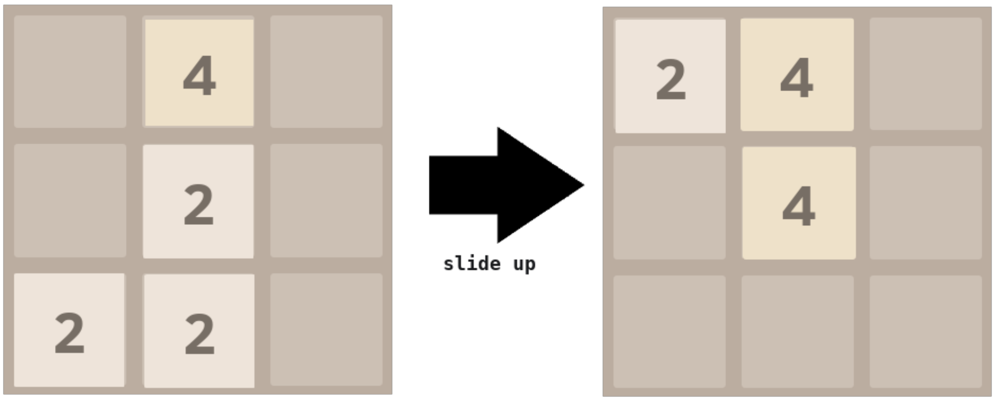


Figure 1.7: The result is not 8, as may be expected

1.8 Halt conditions

The node will keep shifting into empty nodes as long as it is able to in the specified direction, until one of halt conditions is met. Then the node will either collide or merge with another node as clarified below. Each of the halt conditions is stated, along with how it is handled by the game:

1. **Condition:** Shifting the value node in the given direction will place it out of bounds of the grid. This can be thought of as an invalid index being assigned to the node, which is no longer be representable on the grid in matrix form.

   Handling: The node retains its current position and the traversal is considered terminated for the action. This can be thought of as the node colliding with the boundary of our game grid.

2. **Condition:** Shifting the value node in the given direction will collide it with another node of a different value.

   Handling: The node retains its current position and the traversal is considered terminated for the action. This can be thought of as the node colliding with another node, with the merge unable to take place.

3. **Condition:** Shifting the value node in the given direction will collide it with another node with the same value.

   Handling: The merging procedure takes place, creating a new node as specified previously. However if possible, the newly created node will continue to travel in the direction specified by the action until one of the other halt conditions is satisfied, remember that the merge procedure can only occur once for each node per action.

1.9 Termination conditions

2048 will always terminate after a sufficient number of moves has been made, assuming that the grid size specified at the start of the game remains constant. This is true due to the generation of new nodes into the game grid. As a consequence of this 2048 can terminate in two distinct states, a win or a loss.

1.9.1 Win condition

A win condition is satisfied when any one of the nodes in the grid has obtained the desired target value, which we specified at the beginning of the game. A path, sequence of actions, which leads from the start state to a state which satisfied the win condition is considered to be a successful one however, because of the stochastic nature of 2048 the traversal path will naturally have an associated probability of being realized. An optimal path is one that leads to the win condition being satisfied while possessing the highest probability of occurring, out of the rest of potential successful paths.

1.9.2 Loss condition

A loss condition occurs when there are no more actions that can be taken, which would transition the game state to a new one. This can be represented as a graph node that points to itself for every possible action and hence the game is 'stuck' in this state and cannot proceed further. Unless this stuck state contains a value higher or equal to the win condition, the game is considered to be lost and the loss condition is satisfied which terminates the game.

1.10 Variants and modifications

Despite offering no modifications to the game environment in the original version of the 2048, there is no reason why rules such as actions, win conditions, grid shapes, probabilities and merging procedures cannot be modified to form new variants of the game. This of course would result in the definition of a new game, so this may seem redundant at first given that we are trying to solve 2048 specifically, but there are things that could be changed that would drastically simplify the game without steering it away too far from its original specification.

This is very useful for constructing and experimenting with 'theoretical' algorithms, that would not be able to parse the state space of the default 2048, but could handle a simpler, smaller version of it. Such algorithms, when proven to have potential, can then be improved with optimizations both from the software and hardware side and hence be able to work with more sophisticated state spaces.

1.10.1 Transforming 2048 into a deterministic game

The main reason for the huge state spaces generated by 2048, is because any given algorithm cannot consistently predict where new nodes will 'spawn'. This forces any optimal procedure to construct an exponentially growing tree of possibilities to remain optimal. A natural modification is to remove determinism and have a consistent set of rules for generation new nodes, which would then be predictable by the algorithm and hence instead of leading to sets of possibilities, each action would result in a single, deterministic transition state.

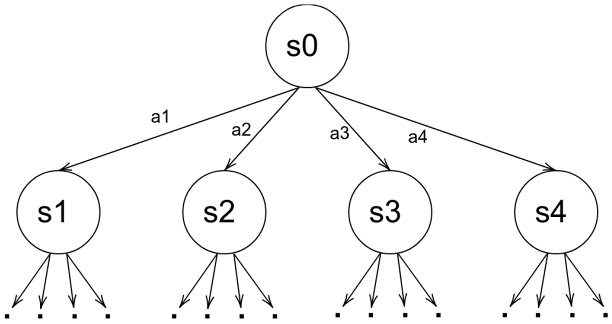


Figure 1.8: Example of how we could represent 2048, if it was deterministic

Having a deterministic game would allow for the generation of the action tree with a constant, exponential growth factor at each height, with the formula being 4^d, where d is the depth of the tree. A simple calculation of the states in the generated tree can then be performed, that could be used as part of the algorithm verification process. More details about the exact procedure of generating the action tree will be explained later, this small section just highlighted the potential applications of making 2048 deterministic.

1.10.2 2x2 grid

The most straightforward, solvable version of 2048 that retains its square board structure, is on a 2x2 grid with a terminating win condition. Because of the finite grid size, it is easy to realize that there exists an upper bound on the win condition, past which the game becomes unsolvable.

What such numbers are for a particular grid is usually difficult to find, due to the merging procedure which allows for many possibilities of combinations. However, one possible way to go about this is to run the optimal algorithm which was not yet discussed, but will be in a later chapter. Using the optimal algorithm, it is possible to keep increasing the win condition until the probability of winning a game from every possible start state is 0. This would then provide us with the maximal win condition, for the given set of rules and the grid. Note that just because it is possible to solve the game by playing optimally, does not mean that the probability of this happening will be high as well. Hence the optimal algorithm will only satisfy the win condition in some iterations of the game.

1.10.3 3x3 grid

The next step in increasing the size of the state space from a 2x2 grid, but still remaining in solvable, is to increase the grid size to a 3x3 grid. This is a significant increase in terms of state space, making algorithms which solve the game by generating the complete tree of all possibilities infeasible to execute. Optimizations and tricks have to now be applied in order to parse the decision trees in reasonable time and not to run out of memory during execution of the program.

Going with the pattern, the next logical step would be to increase the grid size to 4x4, that would bring us back to the original 2048. One final note however, is that for the correct specification of the game, the win condition has to be the number 2048, but of course we could create algorithms which attempt to go higher than that.

1.10.4 More grid shape possibilities

Lastly it is important to realize that the grid square shape can be altered to form different ones such as circular or triangular, composed of the nodes. The laws of the game will still hold as any shape can be drawn in a sufficient large square with the grid boundaries represented by modified nodes, known as collision blocks.

This could be used for further experimentation or personal interest, by seeing how the algorithms cope with more unconventional grids. Perhaps the default square grid is much easier to solve than a circular one or vice versa, more testing is required.

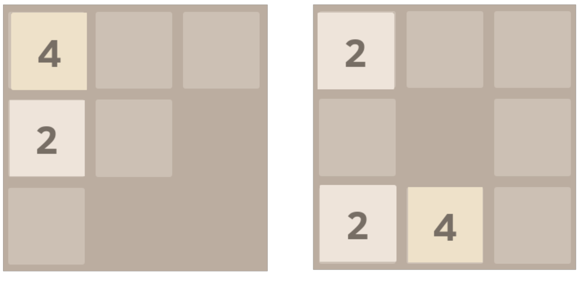
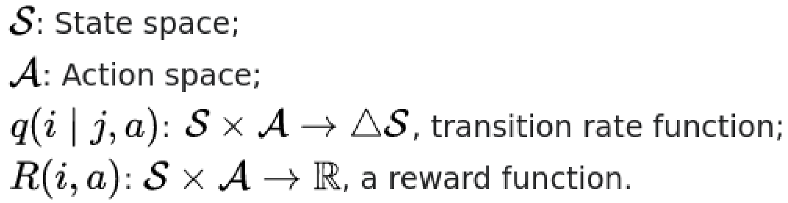


Figure 1.9: Possible unconventional grid shapes

Chapter 2: **Markov Decision Processes (MDPs)**

Deterministic games such as sudoku, chess, tic-tac-toe, etc... can be modeled as finite state automaton[1]. Each of the mentioned games naturally possesses a set of all the possible states the game instance could be realized and in each given state there are established rules or actions which the game agent may choose to take. Such actions will transition the game instance between the available transitionable states. There exist deterministic games, such as chess, which can be represented by a finite state automaton, may contain cycles (think about moving a bishop to a square and then back again to the original square), which results in exhaustive graph search [2] not guaranteeing convergence, hence finding an optimal algorithm would be complicated and require heuristics. However, some games can be instead represented as more constrained directed acyclic graphs (DAGs) like tic-tac-toe or rather trees if every path leads to a unique state. In such cases of FSA (finite state automaton) it is not possible to transition back to the original state after taking an action.

2048 is not a deterministic game, but rather a stochastic one. Actions do not transition the game instance to a predetermined state, but rather a set of possible states the game instance could potentially transition into. In which exact transitionable state the game transfers to, directly depends on the pseudorandom number generator utilized by the game instance. It is not easy to represent probabilistic transitions, between states using the standard finite state automaton however, there exists a mathematical formulation that is able to model it very naturally. It is known as the Markov Decision Process [3], and will allow us to reduce a simplified version of 2048 into a finite state automaton[1] which supports probabilistic transitions between states, and can then be solved using already existing methods, such as assigning optimal actions via the Bellman equation [4].



$\mathcal{S}$: State space;

$\mathcal{A}$: Action space;

$q(i \mid j, a)$: $\mathcal{S} \times \mathcal{A} \to \triangle\mathcal{S}$, transition rate function;

$R(i, a)$: $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$, a reward function.

Figure 2.1: Sets and mappings which make up the Markov Decision Process on finite space and action states [source]

2.1 Problem Reduction

By using the understanding of 2048 game rules, we can reformulate and reduce the algorithmic problem of winning the game, into a Markov Decision Process (MDP) via its property sets. We shall do this by specifying the sets and functions necessary to model an MDP after which we can use the Bellman equation to solve the game by choosing to take actions, that maximize the expected reward [5]. This will result in an optimal algorithm in terms of winning the game, but does not imply that our solution will be achieved within a minimal number of moves/actions. This would lead to a different problem to be solved and required additional semantics of integrating DAG depth into the procedure, which is not the focus of this project.

2.1.1 Actions

The action set, is the set of all possible moves that can taken, within the rules of the game from a specific state/game instance.

For 2048 specifically, we have four actions that can be taken from any state, these are to swipe: up, right, down and left. Hence our action set contains all of these actions and is of cardinality 4.

2.1.2 State space

The state space in an MDP, is the set of all the possible states that the game we are modelling could exist in, having taken a sequence of actions that transitioned the game instance into new states from the specified original start state.

2.1.3 Probability transition function

Taking an action from a given state may result in the game instance transitioning to different possible states over multiple iterations, due to the stochastic nature of 2048.

The reason this can occur is because new nodes generated at different available locations on the grid, will result in varying possibilities of resulting states.

The probability transition function maps states, actions and transition states to probabilities of moving from the start state to the transition state after choosing to take a given action. The resulting transition state is dependent on the chosen action and the state we are transitioning from, which is expressed by the probability via the probability transition function.

2.1.4 Reward function

The reward function specifies the 'reward' obtained by the agent having performed an action and transitioning to the new game instance state [6]. In the project implementation, the reward is specified as the expected reward of the transition state for the non-terminal case, 1 for the terminal case which results in a win condition and 0 for every other terminal case.

2.2 Policy

An optimal policy can be viewed as a game agent that chooses to take the action from the actions set, which maximizes the reward obtained via the reward function.

This can be seen as the 'model' of our optimal solution, each state will be assigned the optimal action which the agent can take when it is in the particular state. After transitioning to the following new state, the process repeats until game termination which will result in a win or a loss.

2.3 Markov property

The Markov property states that the probabilities generated by the probability transition function only depend on the current state the agent is located in, the previous actions and states do not affect the outcome of the probability function. This property is also known as memorylessness [7], the future from the given state is not dependent on the past.

This is an important factor in creating a computationally efficient method of representing the MDP model, as we do not concern ourselves about the probabilistic dependencies obtained as a consequence of executing the sequence of actions, which lead the game agent to the state it is located in. All that matters is the reward obtained from each state, not what sequence of actions lead to the state being realized. This is what later on allows the model to be represented as a hash table indexed by states, which gives the optimal action for each state. More details on this will be given in the next chapter.

2.4 Bellman equation

To solve an MDP system we shall use the stochastic version of the Bellman equation[4], which states that our optimal action is one that maximizes the expected reward from the possible transition states that the agent could end up in by taking the action. Note that because this project is not concerned with finding the optimal solution in the least number of moves, but rather finding a solution with the highest probability of success, the discount factor in the Bellman equation is always set to 1.

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s'|s)\left(R_a(s,s') + \gamma V_i(s')\right) \right\}$$

Figure 2.2: The stochastic Bellman Equation. $P_a(s'|s)$ denotes the transition probability, $R_a(s, s')$ is the immediate reward obtained from the transition state, discount factor will always be set to 1. [source]

Chapter 3: **Trees, DAGs, searches and models**

Now that 2048 has been successfully formulated as an MDP, that can subsequently be solved using the Bellman equation, it is possible to find the optimal algorithm for playing 2048.

The next logical question is how to represent the MDP as a computational program, preferably as some form of a data structure that is efficient and simple to implement. It should then be possible to iterate through such a data structure, using a conventional search algorithm with the optimal solution calculated and stored as some sort of a model for our game agent to utilize, when playing and be reused.

3.1 Trees

At first glance a tree data structure [8] provides a very sensible representation for the MDP, where each node can be seen as a set of possible transition states, followed by four child nodes that are the sets of possible transition states reachable from the parent node, via the four actions 2048 defines. The reason we have sets of states is because we have no guarantee in what exact state the game agent will end up in after performing an action due to the probabilistic nature of 2048, hence we will have to account for every possibility when performing searches. This is defined recursively, with the leaf nodes being the terminal states: win, loss or not being able to execute one of the actions, this can happen when taking the particular action would not transition the game to a new state.

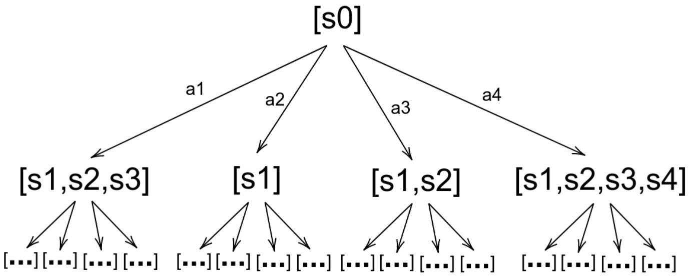


Figure 3.1: Tree representation, showing the sets of nodes. Note that the number of elements within each set is arbitrary.

The reason we may assume to first investigate a possible tree representation instead of a more general graph one, is because we know that 2048 has an acyclic property. Representing the MDP as a tree simplifies the search algorithms which will be used later to solve the game. This will give rise to an opportunity to design them more efficiently by making assumptions such as not having to check if we visited a prior node already for example, since it is not possible. Having simpler code makes it easier to follow and is also important when checking for correctness of our algorithms, which is crucial when experimenting as we always assume that they are working as intended.

3.2 Directed Acyclic Graphs

However, despite this there is still another significant optimization we can implement in the design stage of our data structure, but instead of constraining as we have done with the general graph, we will instead choose to relax it by representing the MDP as a directed acyclic graph (DAG) [9][10]. The reason this will greatly increase the speed of the search algorithms is because in 2048 it is possible to reach already visited states via different actions and pathways as we traverse down the tree. A tree representation results in each node being treated identically, by evaluating its actions and recursively computing the sub-trees formed via the traversal, despite the fact that we may have already processed some of the nodes previously. This is clearly very inefficient in terms of representation of the MDP as we are storing duplicate sub-trees, and by using a DAG representation instead we will avoid this and hence decrease the space complexity of the data structure and the running time of the search algorithms. In the project implementation, the DAG is represented as a Tree structure, but with a hash caching optimization implemented into the algorithm model. This will retrieve and re-use any already processed nodes from the model by indexing with the hash of the node, calculated via an implemented hash function, effectively turning the Tree into a DAG.

3.3 Search algorithms

Having implemented the abstract MDP within the program, we now need a way to traverse the DAG in order to find the optimal solution via the Bellman equation as intended. This will entail iterating over every node in the given DAG and calculating its reward via the reward function, specified as part of our MDP definition, and because this will depend on the node's children we will need to parse down through them as well recursively, until reaching a terminal/leaf node.

Considering each action leads to sets of transitionable states the reward will be calculated as the expectation of the rewards of the possible transition states, after which the action with the biggest expected reward will be selected as the optimal action for that particular state. How this is stored as part of the model will be discussed in the next section, but for now we will simply assume that we have a way of remembering the optimal action for a given state in the DAG.

Because we do not utilize any heuristics (our DAG is not weighted) when choosing which action to take first, this limits our selection to a few viable searching algorithms. Breadth First Search [11] and Depth First Search [12] are very simple algorithms which can be implemented to parse the abstract representation of the MDP and either one would be correct however, there are several properties of the problem we are solving that make DFS far more favourable than its BFS counterpart, which is related to the fact that each node can only be processed after it's children have been as well.

3.3.1 BFS

If we were to use BFS we would be processing each node layer by layer however, because every node is dependent on the expected reward of it's children it cannot processed it until it has processed all of its children. This means that before the DAG can be collapsed, it would have to compute and store it in memory and only after it has finished traversing it, can the optimal actions for the nodes be calculated. This is perfectly acceptable for small DAGs however, because the state space of 2048 is huge (even for constrained version of the game using a 3x3 grid) it is infeasible to construct such a DAG and hence would only work for a 2x2 constrained version of 2048.

3.3.2 DFS

DFS allows the DAGs to be processed in a more sensible way, by evaluating it's individual branches separately as opposed to every single branch at once (when the entire tree is constructed). This reduces the running space complexity from the size of the DAG to worst case running space complexity of the longest branch (Most complicated sequence of actions we can take until termination). This improvement allows calculation of optimal models for the constrained 2048 version of 2x2 and 3x3 grids up to a reasonable win condition.

The recursive DFS implementation makes heavy use of the call stack and hence may quickly lead to a stack overflow error to be generated for large state spaces, even in 3x3 grids. Hence an iterative implementation was made which is more efficient and handles larger sets of state spaces by not relying on the call stack.

3.4 Model representation

Given that each unique node has an optimal action, there needs to be a way to represent this information. This representation needs to be indexed by state, to give the optimal action and this alone is enough to create a game agent that solves the game. This is possible because the MDP has the memorylessness property, which was briefly discussed in the previous chapter.

There are several approaches to implement this such as: sorted arrays of tuples with binary search [13], databases via SQL and hash tables [14]. For simplicity and ease of experimentation, hash tables are used in the current implementation to provide a constant time indexing of the states with a relatively efficient space complexity. In practise hash tables store the action represented by an integer or an enum, which is then indexed hash values of states.

However, this approach is limited by the size of the RAM available on the system the algorithm is running on and hence is not scalable with the primary memory. An SQL implementation would avoid this constraint and is more favourable for computing larger state spaces and thus solving bigger and more complicated versions of 2048.

3.5 Calculating optimal solution from the DAG

Finally it is possible to find and store the optimal solution, using DFS and the Bellman equation the DAG can be parsed and the expectation of each node calculated. The action which leads to the largest expected value is selected as the optimal value and stored in the hash table, which represents our AI agent model that plays the game optimally.

Chapter 4: Optimal Algorithm Results

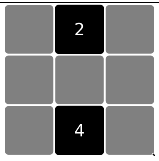
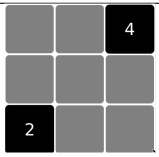
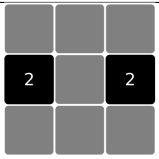
Because the algorithm is optimal, there is no need to empirically find the expected win rate. Note that whenever the win condition is 64, our algorithm is unable to find a solution because it does not exist.

4.1 2x2 experiments

Initial State	Win condition	# of unique states	Expected win rate (%)
□ 2 □ 2	8	33	100%
2 □ □ 2	8	29	100%
□ □ 2 2	16	173	96.26%
2 □ 2 □	16	173	96.26%
□ □ 4 2	16	112	96.14%
4 4 □ □	32	275	8.36%
□ 2 2 □	32	301	8.29%
2 □ 4 □	32	273	8.28%
□ 2 2 □	64	429	0%
□ 4 □ 4	64	403	0%
2 □ □ 4	64	418	0%

Using some analysis it is relatively trivial to realize that the value of 64 cannot be computed within a 2x2 grid. In the best-case scenario, there would need to be a mergeable sequence of squares in the following order 32, 16, 8, 4, 4. This would then be collapsed/merged to obtain 64, but clearly there need to be at least 5 squares for this to occur and hence it is proven to be impossible.

4.2 3x3 experiments

Initial State	Win condition	# of unique states	Expected win rate (%)
	8	967	100%
	8	919	100%
	8	1698	100%

At this point our experiments cannot proceed further due to speed and memory constraints, our optimal algorithm cannot at this point in time compute the win probability for a win condition of 16 for a 3x3 grid. Potential ways to amend this have already been discussed, as well as the fact that such fixes do not scale well with larger state spaces because of the brute force nature of the optimal algorithm. 4x4 grid with the win condition of 8 also does not terminate without raising an out of memory exception.

Chapter 5: **What comes next**

This chapter gives a brief overview on the topics that will be addressed next and the reasoning behind looking into them as potential solutions to the problem of 2048. They will be covered in the second term of the project, and will hopefully achieve a better results than what so far has been seen via the optimal algorithm or more concretely, be computationally feasible.

5.1 Memory efficient model representation for the optimal algorithm

The optimal algorithm that was implemented in this project is at the moment, bottle-necked by memory or more precisely the primary memory on the computer system, the RAM. This is because of how the current model is represented as a growing hash table, which was very convenient, efficient and simple to implement. However, a more practical is to create an SQL database which would be indexed by game states and store optimal actions, identically to the hash table. This will give a major improvement by allowing the program to make use of the secondary memory instead, which tends to be much larger (terabytes instead of gigabytes). Hence the algorithm will be able to process much larger state spaces, but due to the exponential growth of the state spaces this increase will not be proportional with respect to the increase of memory.

5.2 Approximators

As mentioned previously classic search based algorithms are no longer suitable for the task, hence a combination of heuristics and tricks will have to be utilized to solve the beat 2048. Huge state spaces so far have been the limiting factor, because the optimal algorithm is forced to process all of the elements of the state space. However, what if instead the algorithm did not have to evaluate every possible state and the possibilities formed from it. What if given a game state, there was a way to approximate a value for it? This would remove the requirement to form a sub-tree for every instance in the state space and hence remove the exponential growth which occurs when traversing every possibility.

5.3 Lookahead

One of the easiest to explain approximators performs limited lookahead in the DAG formed by the game states. It works by traversing the child nodes of the current state, in the formed sub-tree up to a specified maximum depth, giving each non-terminal leaf node a reward value determined by a heuristic function. Then we apply the Bellman equation on these heuristic values to calculate the expected reward and propagate it up the DAG in exactly the same manner as the optimal algorithm. In fact with an infinite max depth this algorithm is identical to the optimal algorithm that was developed previously. The tricky part is choosing a good heuristic reward function, note that in doing so we break the optimality property. One such function could perhaps be evaluating the number of squares on the given grid and assigning highest reward values to grids with the least number of squares. This is natural as grids with the least amount of squares, after identical number of moves, have to have most successful number of merges and this should be directly proportional to win success. Of course this is a hypothesis, built entirely on intuition, but the good news is that this can be easily empirically tested and evaluated. Note that this is a simple example of a heuristic function, as more insight into the problem is gained more sophisticated reward functions could be designed which should perform better in practise, however all of this would need careful verification.

5.4 Monte Carlo Tree Search

Monte Carlo Tree Search[15] intrigues me as it is certainly not as obvious or intuitive, as the other proposed approximators. It works by running several potential iterations of the game for each possible action, while taking random actions subsequently as it travels down each realized branch. After a parameterized number of descents down the instance sub-tree the algorithm terminates and evaluates the value obtained having taken this path using a heuristic function. This is then repeated several times for each current instance action, and then the action to take is chosen as the argmax of the expected reward from each simulation. The approach certainly has a lot of configuration that can be altered and hence have a significant effect on the results of the algorithm. Despite seeming rather arcane, this method avoids exponential evaluation of the entire sub trees, same as lookahead, but maintaining a much deeper depth than the lookahead algorithm even thou not every possible branch is evaluated. In this regard lookahead behaves more like BFS, and Monte Carlo Tree Search like DFS, both algorithms avoiding exponential sub-tree processing by limiting the search space.

5.5 Reinforcement Learning

Despite its use of neural networks [16], this method is relatively straight forward to implement as its formulation is very close to an MDP (perhaps it was even derived from it). The only difference between the optimal algorithm and the Reinforcement Learning [17] implementation is that a neural network is used as the approximating function, which is trained over many simulated 2048 games after which it hopefully captures a pattern and learns to exploit it.

Chapter 6: **Increasing available memory for models**

As explained in more detail previously, the optimal algorithm demands discovery and evaluation of every possible state, building up the correct expectations of reward for each action bottom-up. This requires a procedure to store all the possible states along with the optimal action and its corresponding maximum expected reward that can be obtained. Due to the immense state spaces that can be generated by 2048, memory is a significant constraint for this approach and hence we at the very minimum require a scalable in terms of memory and performance data structure that will be able to store and query millions of states.

This chapter will begin by looking at the hash table solution and explain why it does not meet the scalable criteria, after which a more viable implementation will be presented in form of a database. This will utilize secondary memory and hence the storage of states will not be bottle-necked by RAM availability.

6.1 Hash table model and its problems

The Hash table approach is trivial to implement, as it is already a well defined programming construct that supported in most popular programming languages, including Java. All that needs to be done is a wrapper built around the standard library HashMap Java class that implements a general Map interface to support the various types of Map implementations that exist, such as ConcurrentMap that will be used and elaborated in a later chapter on threaded algorithm implementations.

A good analogy of this approach is the symbol table from the field of parsers, where the identifiers' are the possible states and its optimal actions are the 'values'.

6.1.1 There isn't a lot of Random Access Memory

The Java HashMap is created in primary memory and is very fast at inserting (Average O(1)) and fetching (Average O(1)) values from the table, this is important as we need to frequently check if a state has been processed by the algorithm so we could utilize the hash cache optimization.

Unfortunately the HashMap is bottle-necked by the availability of RAM on the given operating system running the algorithm. From experiments on my machine with 8GB of RAM, the HashMap could fill out approximately 50 million different states before raising a memory exception. This is nowhere near enough for the full representation of 2048, and therefore I tried to increase the available memory to 64GB of RAM but to no avail.

Scaling up RAM is tricky, as we will now have to resort to some sort of cloud computing system since my local computer motherboard ran out of DDR4 slots and even if we do there is no clear amount of RAM that will be sufficient. Thus for now we can conclude that this procedure is not viable for our purposes, and it is time to move on and try to tackle the problem from a slightly different angle. In order to alleviate the RAM bottle-neck we could instead try utilize secondary memory, an external hard-drive, as the memory model storage of states.

6.2 SQL database

Secondary memory is more abundant than primary memory, and more importantly easier to scale up, since we are now limited by the amount of USB and not DDR4 slots. However, the process of storing states and their corresponding actions is no longer as straightforward as an already implemented programming hash table construct. There is now a requirement for a proper database set-up that must efficiently handle storage and querying.

Using the SQLite Java Database Connectivity (JDBC) driver, I created a primitive database with an interface to the 2048 Optimal Algorithm (refer to UML if needed) that uses it to store state hashes and actions, represented as integers, in exactly the same way as with the HashMap, only now we were no longer constrained by primary memory.

6.2.1 External hard-drive

After acquiring a 5TB hard-drive the 'training' was commenced and it was very slow compared to the HashMap model, this is because an SQL select statement is being called at least once per newly discovered node in the DAG. Therefore the database is being queried millions of times for single rows of data among millions of rows within the database. This is obviously not efficient and creates a huge slow down in performance. Having to send requests through the JDBC driver and SQL, without batching the statements, is also a significant factor loss in performance that we will attempt to address where batching is possible to implement.

6.2.2 Optimizations

Because of the SQL database implementation performance drawbacks, I decided to combine the database and the HashMap into a single, better performing model storage. The HashMap acts as a local buffer, that will write to the database once it has reached a certain threshold of elements. The flush from the HashMap to the SQL database could then be batched as a collection of insert statements, making it much more efficient than previous standard database implementation. However, the issue of single select queries is not solved, but at least the insertions are efficient now.

The buffer is of course queried first when fetching from the database, before calling an SQL select statement, if the corresponding node is not found. This occasionally helps if we get lucky, but most of the time there is still a big performance loss of having to query the database.

Another small optimization is the storage of the latest node pulled from the database. This is to help avoid issues such as when the same node that is not in the local buffer, but in the database and is subsequently queried multiple times in a row. This avoids calling the select query on the same row multiple times and hence improves performance.

6.2.3 Discovery of a latent problem

The optimizations of the database model helped improve the overall performance, and should allow us to processes a much larger quantity of states that would be limited by secondary memory as opposed to RAM.

In practise however, this was unfortunately not the case. The experiments on my Ubuntu

Server system showed that something was still using up the RAM as the optimal algorithm trained. My guess is that it is some form of caching, likely performed by the JDBC. It seems from my observations that the select queries are being cached by something external to my Java application, and hence it appeared that the database backend was performing well when in reality it was the caching of queries greatly improved the performance. Once I manually cleared the cache, the speed of the training was greatly reduced to previously unacceptable levels, as it would if the cache would fill up all 64GB of RAM. Hence unfortunately, the bottleneck of RAM is still present constrained by the millions of select queries the optimal algorithm has to execute as part of its caching optimization, which is a shame.

At this point I decided to move on to alternative solutions of solving 2048, it seems to be computationally infeasible at least with the resources I currently have to further pursue an effective Optimal Algorithm implementation. Furthermore, I knew from the start that this approach was unlikely to work because of the exponential growth behaviour of the possibilities generated by rules of 2048. There would be a need for an immense amount of memory, most likely far more than 5TB. However, I still wanted to have a go and see if something came out of it as I thought it would be also interesting to set up a database to act as a model of solutions.

Chapter 7: **Lookahead**

You should be convinced by now that whilst being optimal, the Optimal Algorithm is very difficult to implement without some sort of compromises that will most likely break it's optimal property. This chapter will look at what kind of corners can be cut in an attempt to bring the algorithm within the range of computational feasibility, while hoping that we will not sacrifice the algorithm's performance too much in the process.

Now that we have accepted for now the fact that evaluating every possible node is impractical for our purposes, what kind of DAG branches should we choose to traverse? We have to create a procedure that ideally selects the more promising paths/branches in terms of reaching 2048, but with limited information in order to keep the performance acceptable.

7.1 How to choose branches?

There are two viable algorithms that answer this question, Lookhead [18] and Monte Carlo Tree Search (MCTS)[19]. In this project I decided to implement lookahead as I found it to be much more intuitive and hence straightforward to implement. In fact I'd argue that it is the next logical algorithm to try after the failure of optimal algorithm, because of how similar they are.

7.2 Lookahead algorithm

Lookahead stops the search upon reaching a certain depth in the possibility branch, after which the next branch is evaluate down to the same depth and so on. Hence we can now control how many nodes are processed, allowing us to fine-tune the depth to our computational requirements. Unfortunately there is a problem, the optimal algorithm knew the exact best expected reward of each node, but that is not the case in lookahead's leaf nodes of the cut DAG at the given depth. The leaf nodes have not been evaluated and thus we do not know the expected reward of each action and hence we cannot propagate any reward up the DAG as was previously done in the execution of the Optimal Algorithm.

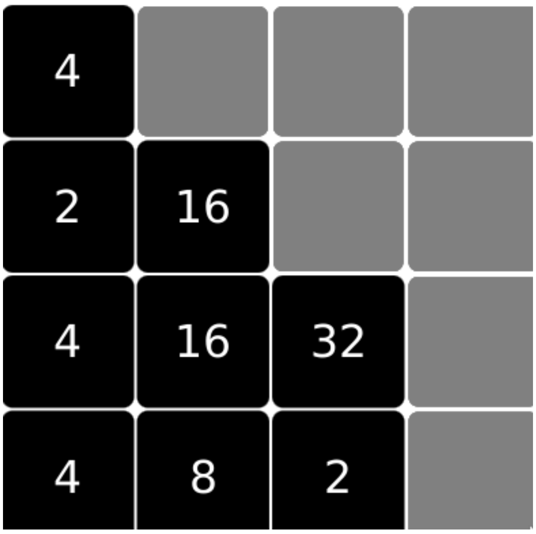
7.2.1 Compromises

The time has come for compromises to be made as we cannot proceed without assigning some sort of reward to the node, even if we do not know exactly what this reward should be. A heuristic function will need to be created that will input a node and assign some corresponding reward to it, how we define such a function is up to us and should be able to assign higher rewards for better states and lower rewards for less favourable states. The quality of the heuristic function will directly affect the performance of the lookahead algorithm, hence great care should be taken to ensure it is as general as possible or we may suffer from over-fitting the heuristic function to favourable states, causing poor general performance.

Such heuristics can be complicated, such as favouring states with high valued nodes around the edges of the board, however for simplicity's sake I created two basic heuristic methods that have been successful in practise.

7.2.2 Empty nodes heuristic

This heuristic function counts up the quantity of empty nodes present in the game grid. The intuition behind this, is after an identical number of moves (the chosen depth in our case) nodes that possess more empty nodes must have combined and merged more, which is what we want as it transitively gets us closer to the goal. A useful property of this heuristic method is that it is bounded, assuming that we start the game with two nodes, as is default, the highest reward that could exist is 15 (if we combined the two starting nodes together). This allows to provide a reward for states that won the game, in other words have a node with a value 2048. The main issue with this approach however, is that it does not take more significant merges into account. Combining two nodes with a value of 2 is identical to combining nodes with a value of 512. Clearly we would favour the second combination as those nodes are usually much more difficult to combine (by that point we would have a lot of other nodes getting in the way), whereas nodes with a value of 2 are much more common and hence should be easier to merge. This also inevitably results in many states with the same reward value, even if we would favour some far more than others as demonstrated in the previous example. This gave motivation to find a more suitable heuristic function that would address this problem.

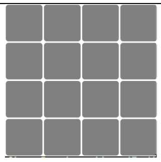
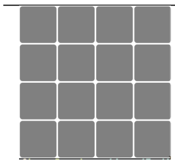
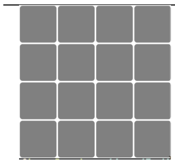


4			
2	16		
4	16	32	
4	8	2	

Figure 7.1: This particular state would have a heuristic score of 7

7.2.3 Lookahead empty nodes heuristic results

A larger lookahead value could have been used that should have won a larger number of games on average, but would have required a lot more time to execute the experiments. For practical reasons, lookahead of 2 is commonly used in this project as a comprise between effectiveness and performance, when playing out many games.

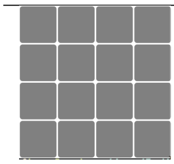
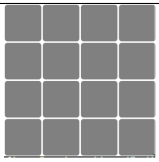
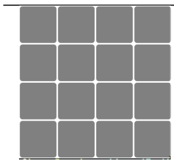
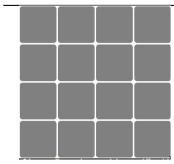
Grid shape	Win condition	# of lookahead steps ahead	# games played	# games won	Win rate (%)
	512	2	100	100	100%
	1024	2	100	85	85%
	2048	2	100	15	15%

7.2.4 High score heuristic

Perhaps the more natural heuristic method is to simply use the score of the entire game reached at each node, where the score is calculated as the cumulative sum of all the values of combinations after each previous action. This gives a more fine-tuned reward metric, assigning higher values to states with more substantial merges in terms of the value magnitude. However, there is a new problem the function is now unbounded, unlike with the previous approach. In practise this should not be a significant issue, as the reward gained for combining more significant nodes grows exponentially, but it is plausible that a state could exist that has not reached the terminal value, but has a higher reward value than a state that did. To reiterate this is only an issue for lowered valued rewards where the exponential 'leap' has not yet taken place, try it for example for a win condition of 16 on a 4x4 grid. One could relatively easily combine enough nodes to achieve a higher score without reaching a node with a value 16, than that of a grid that has. This of course requires some effort and will only become harder to do as the win condition increases, up to values of 2048 for example. A potential fix could be to perhaps provide an increased reward (such as 10000) if a node reaches the win state, however I refrained from doing as it could introduce bias into the calculation of expectation, that could make some actions excessively optimistic.

From practical experimentation, lookahead with depth of 3 has the best trade off between win rate of 75% (with the default game parameters) and speed performance of around 1 second of making a decision. Depth of 4 increases the speed performance of around 30 seconds, which is far above the desired threshold.

7.2.5 Lookahead high score heuristic results

Grid shape	Win condition	# of lookahead steps ahead	# games played	# games won	Win rate (%)
	512	2	100	100	100%
	1024	2	100	97	97%
	2048	2	100	29	29%
	2048	3	100	69	69%

7.3 Monte Carlo Tree Search algorithm

I mentioned that there were two algorithms I am aware of that have very different ways of evaluating the subset of possible branches. While I have not implemented MCTS (Monte Carlo Tree Search) I will lightly touch on the theory behind them.

The limiting factor of lookahead is it cannot dive very deeply down without becoming computationally exhausting. We cannot pre-compute the model as we did with the optimal algorithm. We must re-run the lookahead and MCTS procedures after taking each action on the new state. Because we are interested in algorithms that have good performance we have only a given number of nodes that can be evaluated within a suitable time frame, based on the system we are running the algorithms on.

Lookahead evaluates nodes in a Breadth First Search manner, but MCTS is more of a probabilistic approach that seeks to dive down to deeper depths at the cost of not evaluating every possible branch, unlike lookahead. This subset of branches is chosen by taking random (from a uniform distribution) actions down to a certain path, whose depth will be much larger than could be possible with lookahead. The expected reward is then calculated in exactly the same way as we would have done with lookahead with the limited information we obtained from these stochastic dives, the experiment is then repeated a predetermined number of times (more experiments should give a less biased result), until we take the action that has obtained the highest expected reward from the experiments.

It is an interesting approach that would certainly be worthwhile to investigate and compare against lookahead performance wise, however I decided to move on to other potential solutions as I wanted to implement a Reinforcement Learning algorithm first and if given time, have a go with Monte Carlo Trees.

Chapter 8: **Threaded Lookahead**

It is not difficult to realize that the procedure of the optimal algorithm and hence lookahead is very prone to parallelization. Every branch of the DAG is evaluated, without any data dependencies between them, with exception of hash caching optimization. However, if we for now put it aside, we could process every required branch evaluated on a different core and hence speed up the algorithm linearly with respect to how many available CPUs are on the system.

The trivial implementation is to take every single possible start state and put it on the 'to be processed' stack, after which each thread would take the available starting state and process it individually until all possible starting states have been evaluated. There is a problem with the existing hash caching optimization, as it is a data dependency that is shared between the evaluating branches. If a thread begins processing a state, and another thread begins processing the same state (since we are dealing with a DAG, not a Tree) in sync (it is not in the hash map as it has not been fully evaluated) then both threads will be processing the same state branch and hence doing the identical work, which is inefficient. We can use the already existing ConcurrentMap implementation available in the standard Java library to act as a thread-safe hash table, but this does not solve the mentioned issue of race conditions between the thread processing identical branches. To reiterate, the solutions will be correct and processed faster than a non-thread implementation, but we are not exploiting the entire potential of parallelism for this task with the trivial version.

A more correct implementation, would include the ability to 'reserve' states for processing that would cause the other threads to wait until the state has been evaluated by the thread that reserved the said state. Meanwhile the other threads could attempt to process other states as to not remain idle and not waste CPU time.

I was successful in implementing the naive version, and observed increases in performance by factor of 2, which is great but not sufficient enough to process even a single increase in depth. Hence it does not improve win rate performance in comparison to non-threaded lookahead, but it does execute faster. I did not attempt to implement the more appropriate version, as I do not have sufficient evidence to believe that it will increase the depth of the lookahead algorithm to 4, while still maintaining an acceptable speed performance. Instead I chose to pursue a reinforcement learning solution, that will utilize a neural network to act as a predictor for the best action to take at a given state, that will be trained through experience on many thousands of games.

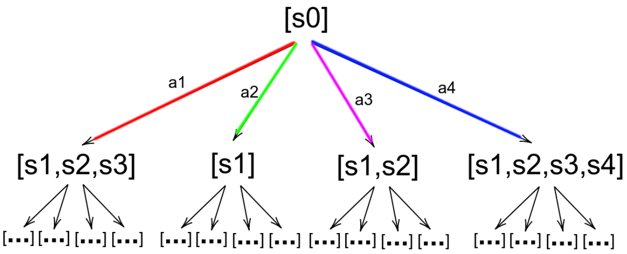


Figure 8.1: Each colour in the picture represents a parallel branch that can be independently processed

Experiment	Non-threaded execution time (seconds)	Threaded execution time (seconds)
100 games of 1-step lookahead	44	16
100 games of 2-step lookahead	1378	296
5 games of 3-step lookahead	1279	290

Chapter 9: **Neural Networks**

All of the heuristic methods for solving 2048 discussed so far are limited by the approximating methods they use, using the high score as a heuristic for estimating the desirability of a state can be potentially misleading. There is a plethora of examples that have very good high scores after a given number of moves, but are dangerous as they can be difficult to merge and hence a win may not be possible to achieve. Of course increasing the depth of lookahead would mitigate this issue, but as already discussed it is not feasible in terms of computational performance. Hence if we want the algorithms to improve we need better heuristic methods of evaluating how good a certain game instance is.

While it is possible to come up with complex rules, backed up by empirical evidence and optimism that could potentially work in practise, I think it will be more productive to instead use a more deterministic approach to this problem. Neural networks are well known universal approximators, and are able to learn the underlying desired functions between the input and the desired output, after being trained on sufficient quantity of data. Thus it should be possible to train a neural network that acts as the required heuristic method and is able to predict not just the immediate reward, but potential future reward as well, without having to resort to complete evaluation of every possibility as we would have done with the optimal/lookahead algorithm. In principle, even if we have to train a relatively large network it should still be much more computationally efficient to run it than evaluate the branches up to large depths.

Neural networks typically require large quantities of data to train on in order to be sufficiently good approximators in terms of accuracy. Fortunately in this case we can choose how much data is generated by controlling how many games are played out using the neural network approximator. Every game that is played out generates states that can be used as input into the neural network, after which it attempts to predict what action is most favourable to execute by taking the argmax of the output vector of length 4 (representing each of the swipes that the algorithm can execute). This procedure will be the foundation of the Reinforcement Learning algorithm we will use in order to train the approximator, that will be discussed in the next chapter.

The implementation of the neural network is my own and I will briefly elaborate what kind of neural network I have created, more specifically how the Stochastic Gradient Descent procedure will be carried out and the regularization techniques I have applied in the process. All of the gradients for each activation method and weights have been computed manually, without the use of any automatic differentiation engines.

9.0.1 Stochastic Gradient Descent

I will not describe in detail the intricacies of the SGD procedure, there are other resources available for that[20][21]. My implementation of the neural network supports the standard subtraction of error derivatives with respect to the corresponding weights after applying a learning rate scaling, this is the very basic iteration of gradient descent that will converge to a local minimum after sufficient training time and correct fine tuning of the learning rate. At this stage I have verified that the neural network is able to learn after achieving an accuracy of 97% on the MNIST training set, which I used as a benchmark and sanity verification that my neural network is functioning as intended.

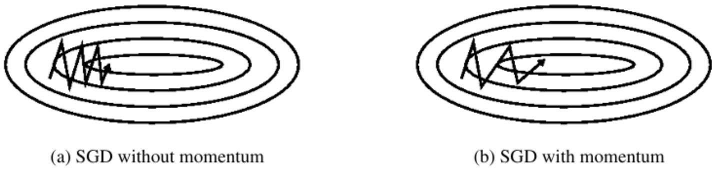
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Figure 9.1: Stochastic Gradient Descent formula, source:[22]

9.0.2 Momentum

After this, I modified the SGD procedure to also support momentum [23] based steps as in practise this tends to lead to faster converge to a local minima. The idea behind this is very simple, instead of just calculating the gradient step and subtracting the weights by it, the previous gradient step is scaled up and added as well.

This results in gradient steps 'accelerating' if there are persistent steps in terms of magnitude allowing for faster convergence as long as the network does not accelerate too much which could lead to an explosion of gradients. This modification usually requires a much smaller learning rate than normal, but will allow for bigger leaps if necessary.



(a) SGD without momentum

(b) SGD with momentum

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned}$$

Figure 9.2: Momentum SGD formula, source:[22]

9.0.3 L2 regularization

This technique is copy and paste from linear ridge regression[24], that penalises models with large weight values in hopes of achieving a simpler model that does not over-fit the training data. An over-fit 2048 would be able to handle the common states very well, but fail to generalise to the edge cases and hence not perform as well as it should.

Dropout is also a very effective regularization procedure that is surprisingly simple to implement, however I will not do so as it usually requires larger neural networks to be effective. Because the neural networks that will be used in this project shall be relatively small, for now I decided to stick with L2 regularization for simplicity's sake.

Thus in conclusion there are 3 hyper-parameters to fine tune: learning rate, momentum rate and L2 regularization rate. Of course there will also be a need to test various neural network sizes, larger networks will take longer to train, but smaller ones have the danger of under-fitting and hence performing poorly. Combinations of all these factors will need to be experimented with, fortunately everything is set up and ready to formulate the problem of solving 2048 as a Reinforcement Learning challenge that can be solved with proper fine-tuning.

Chapter 10: **Reinforcement Learning (RL)**

10.0.1 Almost MDP formulation

The RL formulation is very similar to that of the Markov Decision Process (MDP), the state space, action space, and transition rate function are identical, as previously discussed and elaborated. Of course the reward function will no longer be based on human made heuristics nor optimally, instead a neural network will be used. The neural network will attempt to learn through experience a sufficiently good reward assignment function for each action, given an input instance. Using this, the best action to take according to a neural network is one with the highest value in the neural network final layer output vector. It will then make the action and repeat the processed, until termination, with a hope that it will perform better than the lookahead algorithm. Because 2048 has the Markovian property, the past actions should not affect the present choice of action to take and thus a standard feed-forward neural network, should have all the information available needed to learn to predict and determine good and bad actions for each state.

10.0.2 Neural Network Input and prepossessing

The feed-forward neural network expects a vector as an input to propagate signals throughout the network, so there needs to be a procedure to convert a 2048 game state into a corresponding vector representation. The most straightforward way to go about this, is to convert the game instance into a matrix and flatten it along it's rows or columns (implementation is along rows). Therefore for a 4x4 grid, we would have a 16 element vector representation filled with value node's values and 0's for empty nodes. For simplicity's sake brick nodes will not be taken into account for now, we will only deal with empty nodes and value nodes.

Another, perhaps more appropriate representation for the neural network would be to take logs with base 2 instead of direct values, because they will be distinct and their magnitude will not be taken into account, as it is with the raw representation. Therefore it is a form of normalization that will help neural network process exponentially large values by transforming the possible values onto a linear plane. The node with the value of 1048576 is much more difficult to process than 20 for example, despite the fact that it does have the capacity to do it. For small value nodes up to 2048 this is most likely not going to be a significant issue, and hence I only implemented only the raw representation for now, however if the requirements needed to run the algorithm past 2048 up to higher values it is most likely that the learning rate would be impacted.

10.0.3 Immediate and future reward

In order to train the neural network, we have to back-propagate[25] a difference between what we want the neural network to output, and what it did output. In order for the network to learn, it has to have first produced an output that we are not satisfied with, but what should such an ideal output be? This is most likely the most difficult part about the entire reinforcement learning procedure, we must choose a reward such that the network is able to learn how to play the game through experience, but in processes without making it biased so that it overfits [26] and only performs well on particular states. Of course, another major problem is that we do not know what the correct reward for each action is, for that we would require the optimal algorithm, which is infeasible to compute on our system (and would make the RL procedure pointless).

We can start with simply setting the immediate reward the network obtains as the reward, however this will only encourage the network to act greedily and selecting the most short term profitable action. It would be no better than simply using one step lookahead, as the training does not involve any rewards based on the network's future performance. Thus a better reward function would entail how well the network performed immediately and later on, we can obtain the needed data by playing out the entire game and creating the label reward vectors in reverse order of the actions taken. To clarify our reward formula would be 'reward = immediate reward + future reward', the future reward on the last action we took before termination would be 0, then for the penultimate action it would be the immediate reward of the last action plus 0, because the last action does not have a future reward. Finally we may wish to discount future rewards with some factor between 0 and 1, due to the stochastic uncertainty of 2048. Because there is no guarantee this is the exact reward the network shall obtain in the future the discount factor will penalize potential future rewards to make them not as significant.

Note that, at least initially, it will be very difficult for the agent to reach 2048, in fact it will most certainly lose most of its games made in the initial stages of training. If the network lost a game, we do not want to propagate a positive reward as the network carried out an undesirable sequence of actions, which we do not wish to repeat in order to play better. To encode this in the learning procedure, whenever the network loses, a reward of -1000 is propagated throughout the sequence. The value is a hyper-parameter, and may not be optimal, higher punishment values should encourage caution while softer negative rewards may encourage the network to take more risks. Of course too high of a negative reward may cause the network to not converge to any local solution, or greatly decrease its learning time.

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s', a)$$

Figure 10.1: Modification of the Bellman equation to use rewards instead of expectation SOURCE IMAGE

10.0.4 Exploration

There is a potential problem with taking the argmax of the neural network output vector as the next action of choice, without sufficient negative reward the network might continuously reinforce the action it arbitrarily chose to execute never trying out more fruitful alternatives. To clarify if by random chance the neural network chose to shift the nodes to the right, and doing so resulted in a positive overall reward it will reinforce that move as a 'good' one. While this is the expected behaviour of the reinforcement algorithm, because of the argmax behaviour it will never try the alternative moves even if perhaps through repeated experience they will obtain better rewards.

The current algorithm is missing an exploration factor, something to encourage it to explore different possibilities even if sometimes it would not be a wise move to make. Otherwise our model becomes very confident in its decisions, you could even say arrogant because it is not encouraged to learn alternative possibilities. This is where the epsilon-greedy [27] modification, that will introduce a little stochastic behaviour into our deterministic neural network model. Every time the neural network makes a choice there will be a 10% chance that instead of taking the action according to the neural network model, a random action will be taken instead sampled from a uniform random distribution. This will result in the algorithm taking sometimes good and bad actions, but overall it will be able to process a larger set of possibilities and hence hopefully provide a less over-fitted predictor.

10.0.5 Training

The training procedure is relatively simple to explain now that the input vectors to the FNN have been defined, and the reward methodology. The network will take in the start state, predict an action and repeat until termination. Note that the state of the network after each move is saved, and used in backpropogation [25] to find the derivatives of cost with respect to network parameters at each stage of the game being played out. Once these derivatives have been computed, they will be aggregated together via a sum operation and normalized by dividing each one by the number of moves made throughout the game. Hence what we have in the end is an averaged weight update, which is then used update the neural network and apply momentum along with regularization as part of the post processing procedure, before at long last changing the network parameter to minimize the cost.

This is repeated for 100,000 games before the network is evaluated for performance, although 10,000 games are first used to make sure the hyper-parameters do not lead to explosion of network gradients (learning rate set too high for example).

10.0.6 Results

Unfortunately, despite extensive experimentation, I was unable to produce a neural network that was able to achieve even minimal performance using this technique. There are many potential issue that could be causing this, input vector normalization, incorrect cost function definition, lack of dropout, insufficient network size, small learning rate that could be improved with more advanced gradient step procedures such as Adam. To pinpoint the issue a significant amount of time would be required, this at this stage in the project I do not have. However, while developing the reinforcement learning procedure I turned once again to the lookahead algorithm and its heuristic methods, and realizing how we could improve it so that it consistently performs better than 75%. This is what the next and final chapter shall be dedicated to.

Chapter 11: **Returning to Lookahead and its heuristic methods one final time**

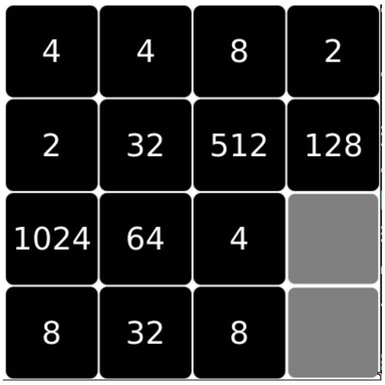
My failure with applying reinforcement learning was disappointing, I believed that neural networks would have performed much better than they did. After taking a look at papers that attempted to do something similar [28][29], the result is not surprising, this appears to be a difficult neural network problem to solve in practise.

However, I wasn't quite satisfied with the 69% win rate of 3-step high score heuristic lookahead and decided to investigate further on what exactly caused the algorithm to lose games. After analyzing several examples of where 2048 was not reached, I noticed that were was a general pattern occurring that caused the game to become very difficult to play. The agent would try to greedily combine nodes that lead to the largest 3-step high score, as intended, however it would not take any account of where the resulting nodes would end up. To clarify, as the agent continued to play the game and naturally create nodes of larger values (such as 256, 512, 1024) it would not attempt to keep them at any particular part of the grid in particular. They would be shuffled about, until a new possibility emerged where the nodes could be combined.

11.1 Dangerous states

This is still all under the expected behaviour of the lookahead algorithm, however this can lead to states that achieve a great high score, but are dangerous. This is because if there are two high valued nodes on the grid that cannot be combined within 3 steps, the algorithm will make no attempt to merge them. Hence they become an obstacle, as their presence makes it more challenging to merge the other value nodes. If we were to use the Optimal Algorithm, obviously every possible node would be processed and most likely the agent would perform a specific sequence of slides that merge them together. This is very important not only to beat the game, but to also make sure that the number of empty nodes on the grid is maximized, which is what usually causes lookahead to lose games.

Hence here lies the problem, by only using the high score as the reward metric for the lookahead we are ignoring how dangerous a board is or could be after the merges takes place. Some of the best 2048 human made strategies involve trying to keep the high valued nodes in the corner of the grid, because that way they do not 'get in the way', and hence cause the board to end up in a dangerous state that could lead to a systematic loss.



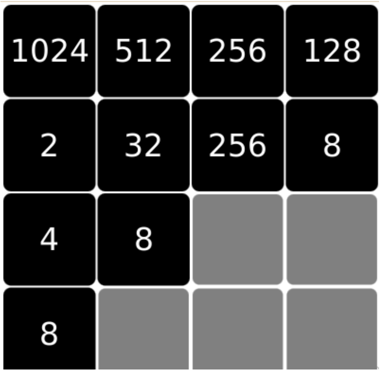
4	4	8	2
2	32	512	128
1024	64	4	
8	32	8	

Figure 11.1: Example of a 'dangerous' state

11.2 Order heuristic

Based on this knowledge, the lookahead reward metric should somehow decide which possible game states are more ordered and therefore safer, and which ones contain more scrambled nodes that could quickly cascade into an unwinnable instance, so as to avoid them.

This can be achieved by enticing the algorithm to place its biggest valued nodes in a particular part of the grid, in this case one of the corners. Note on the importance of one corner, and not all the corners as that would lead to states that are difficult to merge as high valued nodes would be most likely distributed far away from each other.



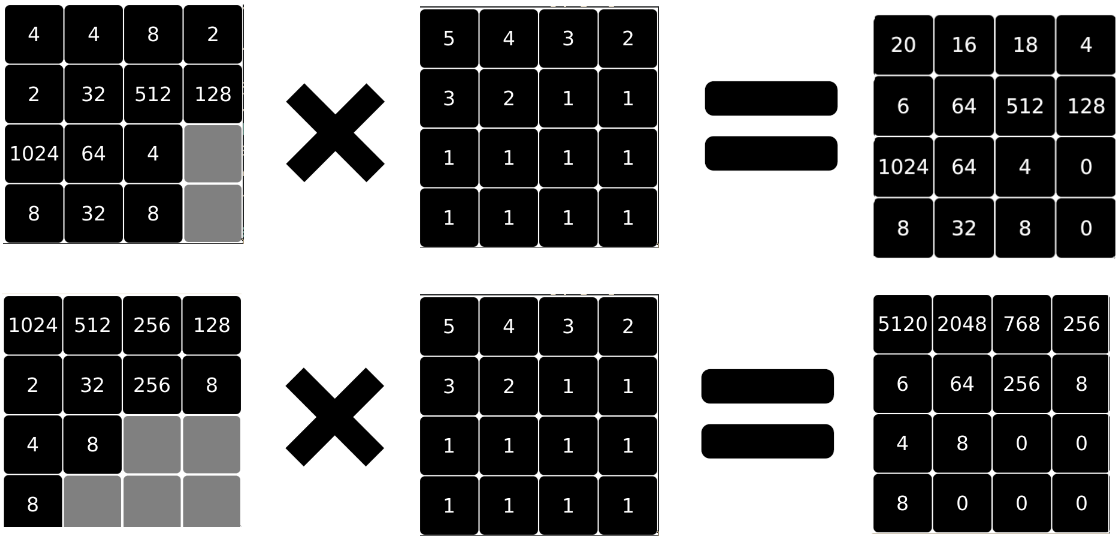
1024	512	256	128
2	32	256	8
4	8		
8			

Figure 11.2: Example of an 'ordered' state

This is implemented by generating a matrix identical in shape to the grid and populating it with positive integers. What combinations of integers and in what pattern is a heuristic, at the moment I am not aware if there is a best possible combination of values, but as will be shown there are some very effective ones. The idea is to have a flow throughout the grid that will prioritise states with large values to be located close together, near one of the corners. This is inspired by the human made technique discussed earlier, it encourages more ordered states that have the benefit of possessing nodes that do not become obstacles and are easier to 'chain' merge if the lookahead is successful in moving the nodes to the specified favourable locations on the grid.

After constructing such a matrix, an element-wise multiplication can take place with the grid's matrix representation. The result is then summed up into a single value, that will be from now labelled as the 'order metric'. This metric will be larger in magnitude for states with a similar organization of nodes, and less so for more chaotic instances.

This metric can then be added to the high score heuristic to improve its performance, by not only taking the high score into account, but also the ordering of the nodes withing a given state.



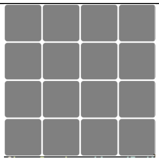
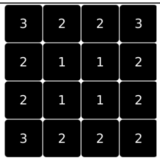
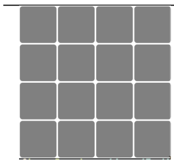
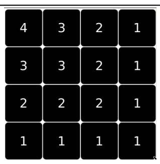
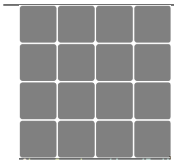
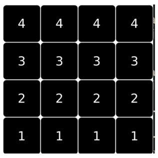
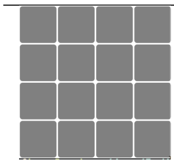
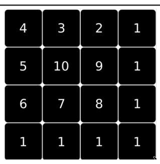
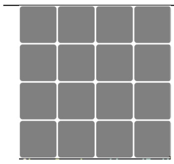
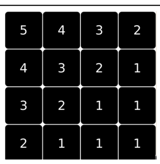
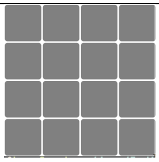
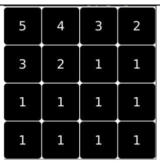
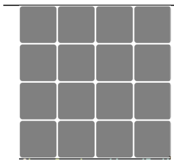
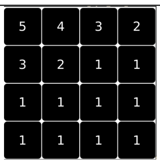
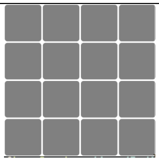
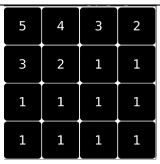
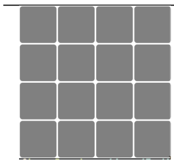
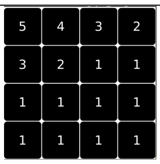
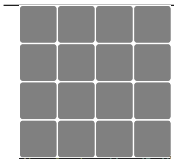
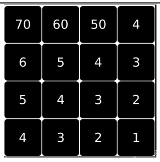
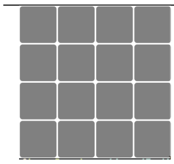
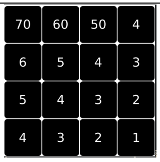
$$\begin{pmatrix} 4 & 4 & 8 & 2 \\ 2 & 32 & 512 & 128 \\ 1024 & 64 & 4 & \\ 8 & 32 & 8 & \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 & 2 \\ 3 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 20 & 16 & 18 & 4 \\ 6 & 64 & 512 & 128 \\ 1024 & 64 & 4 & 0 \\ 8 & 32 & 8 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1024 & 512 & 256 & 128 \\ 2 & 32 & 256 & 8 \\ 4 & 8 & & \\ 8 & & & \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 & 2 \\ 3 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 5120 & 2048 & 768 & 256 \\ 6 & 64 & 256 & 8 \\ 4 & 8 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{pmatrix}$$

Figure 11.3: Using the ordered heuristic to process a given state, note that the 'dangerous' state results in a score of **1908**, while the 'ordered' state attains a score of **8546**, after summing up all the elements in each resulting matrix

11.3 High score with order heuristic results

All experiments were empirically verified against a 100 games for each specified configuration.

Grid shape	Order matrix	Win condition	# of lookahead steps ahead	Win rate (%)
	 3 2 2 3 2 1 1 2 2 1 1 2 3 2 2 2	2048	2	46%
	 4 3 2 1 3 3 2 1 2 2 2 1 1 1 1 1	2048	2	15%
	 4 4 4 4 3 3 3 3 2 2 2 2 1 1 1 1	2048	2	80%
	 4 3 2 1 5 10 9 1 6 7 8 1 1 1 1 1	2048	2	0%
	 5 4 3 2 4 3 2 1 3 2 1 1 2 1 1 1	2048	2	92%
	 5 4 3 2 3 2 1 1 1 1 1 1 1 1 1 1	512	2	100%
	 5 4 3 2 3 2 1 1 1 1 1 1 1 1 1 1	1024	2	98%
	 5 4 3 2 3 2 1 1 1 1 1 1 1 1 1 1	2048	2	91%
	 5 4 3 2 3 2 1 1 1 1 1 1 1 1 1 1	2048	3	94%
	 70 60 50 4 6 5 4 3 5 4 3 2 4 3 2 1	2048	2	**97%**
	 70 60 50 4 6 5 4 3 5 4 3 2 4 3 2 1	2048	3	**97%**

Chapter 12: **Professional issues: Potential impact of Artifical Intelligence**

Computer programs are useful because they are able to automate a sequence of tasks from all kind of fields, statistics, robotics, management, etc. Such tasks used to be performed by human counterparts, but have since been replaced with technological advances, however I will not discuss the ethical issues of Artificial (AI) Intelligence replacing jobs and causing potential harm by such an abrupt change.

Instead I want to point out that most programs not relating to AI, are written by programmers that attempt to use their logic to construct a solution to the problem. Sometimes that logic can be flawed and result in bugs, which subsequently can be fixed with time and testing but in the end there is an explanation to why they wrote the code the way they did. This can be analyzed and therefore the expected result to be somewhat predicted, we know how the program works, what kind of objects it creates and how they interact with each other and hence we know what to expect from it.

This is not always the case with Artificial Intelligence.

There are deterministic AI solutions, usually in the field of graph search as have been discussed in great detail during the development of this project, that provide an intuitive explanation to how they function and work. It is possible to know and understand the process through which the argument makes it's decisions. The next step up could be perhaps linear regression where we attempt to fit a line of best fit by minimizing the least squared distances between all the points. The weights associated with the input vector could be arcane, but we will still have a deterministic model (that can be interpreted via the graph) even if we apply a non-linear function after performing the linear regression task.

Neural networks however are incredibly difficult to interpret, some of them obtaining 175 billion weights like in the GPT-3 model. These weights will then tend to have complex interactions between one another that have been learned through the training process, but may not even have a clear explanation. This is a significant problem as it means that our Neural Network models, that have been achieving state of the art scores on a large majority of machine learning tasks, are arcane. We do not understand HOW they work, the only reassurance we have is its performance on test data and even if we did manage to dissect a neural network and explain its reasoning this would have to be done on every single iteration of it. Let alone have some assurance that our understanding of it was correct in the first place. This is the price we pay for data driven models.

The reason this could be a dangerous ethical problem, is these neural networks could potentially be applied to areas with a risk to human life. Because they do not have an understanding of our real world, like humans do, but only what they can infer from the data provided (which is fair to say an incredibly limited aspect of the real world if at all), we cannot guarantee them to make logical decisions. They will only make decisions based on what they have learnt during their training, hence in rare cases where the network potentially encounters an edge case it could predict something very illogical and even worse dangerous. Thus despite being some of the best solutions to a large set of problems, we must be incredibly careful that these networks are applied in appropriate areas where they can pose no danger or direct harm to human life at the very least. It is also important that these developed neural network models are thoroughly tested and validated, not just for performance but to see how it performs on the rare and unexpected cases.

However, it is important to note that there are cases where neural networks are very adequate stochastic models and outperform any carefully thought out human heuristics and so it makes sense to apply them to tackle such problems. A safe system could perhaps entail using the neural network only for cases where it is applicable, falling back onto well though out logic to make the important and potentially dangerous decisions that will minimize the risk to human life. Careful evaluation of the data the networks are trained on must also take place, to ensure that they do not overfit and even perform well on the validation datasets while in reality it could generalize poorly in real use cases.

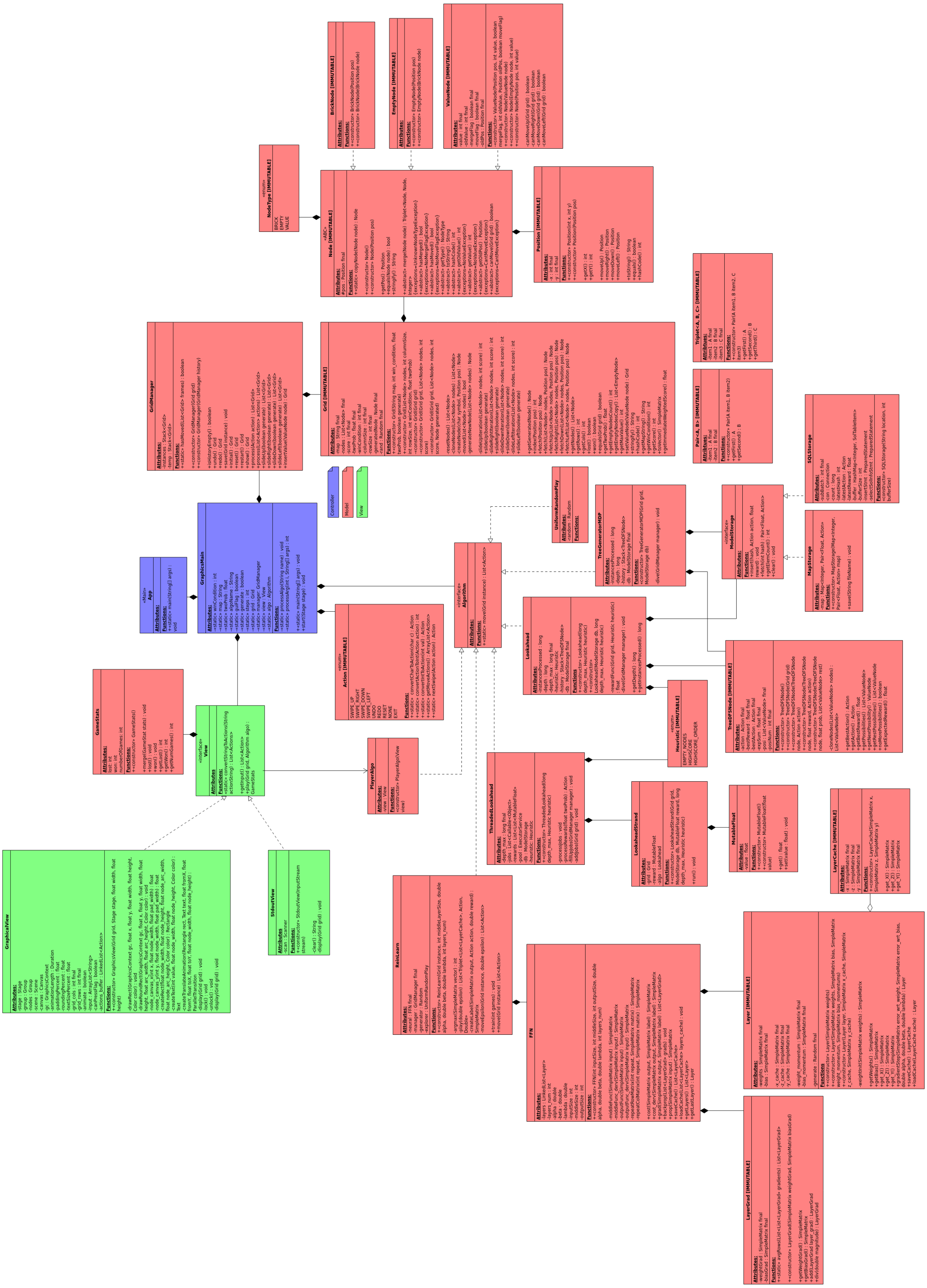
These are some of the issues I have been thinking about throughout the development of the project, especially when I begun the work on the reinforcement learning procedure, which involves the use of neural networks. Fortunately it poses no risk to human life, although the reinforcement learning procedure could be applied to potentially malicious tasks, due to the sheer power of neural networks being able to approximate so well, it is difficult to determine what kind of nefarious tasks they could perform. Generation of fake data in text, image, audio and other formats is likely to be the most realistic issue in the coming future, as enormous models such as GPT-3 have become incredibly good at writing coherent text and other technologies like Deepfake that are based on convolutional neural networks are able to modify existing images and alter them very well. This could lead to exchange of fake data and information that will be incredibly difficult to disprove and identify, and will most likely result in us using neural networks to classify what is real and what has been artificially generated. Of course the potential catastrophic damage of this is evident, as most of us in the current state of the world rely on the internet to obtain information and knowledge with a hope that it is not fake. Such hopes have already begun to become less justifiable to have.

Chapter 13: **Overview of Software Engineering design choices**

13.1 UML

The Unified Modeling Language diagram provides a broad overview of all the Java classes within the project and their relationships between each other. Several design patterns, such as delegation via polymorphism, can be seen taking place that will be discussed at length in this chapter. Throughout the development of the project the UML diagram was used as part of my planning for developing new features and functionality, as well as being part of the overall code documentation.

For an easier viewing of the diagram please consult the UML .png file found within the project 'images/' directory.

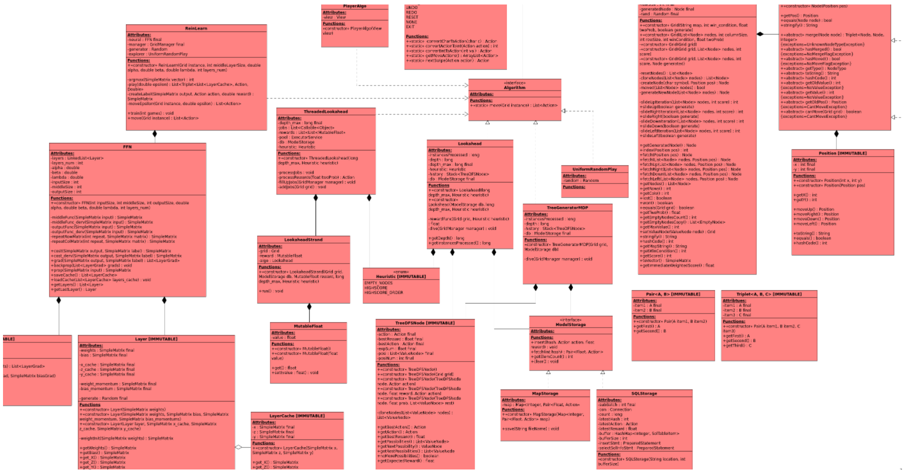


13.2 MVC

At the highest level of abstraction the code is split up into three distinct groups, 'Model', 'View' and 'Controller'. Every class in the project is assigned to one of the three groups, and possesses particular functionality that categorises it as such. This design allows for a very modular approach when implementing new features, allowing recycling of already existing code as will be demonstrated in the next section. By having a clear separation between each group, they can therefore be swapped in and out as required. For example the GridManager class could be rewritten so that it contains a new set of features and logic behind it. However, as long as all the methods are satisfied the new code will work both with the 'Controller' and the 'View' logic, because of the implementation disconnect between the groups, making them independent.

13.2.1 Model

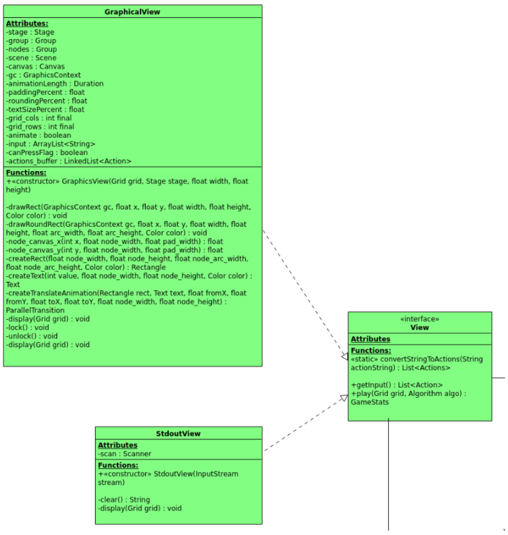
The backend of the code, classes in this group are responsible and focus on implementation of functionality, such as the logic behind moving of the nodes within the grid as well as the merging and how 2048 is represented internally. Algorithms such as optimal, lookahead and reinforcement learning are also under in this group, and are all tied together under the 'Algorithm' interface that allows trivial addition of new algorithms for solving 2048, without modification to other parts of the code, by overriding the required methods.



13.2.2 View

The human interface to the application, used to view the output of the backend ('Model') via a text or graphical interface. User actions are also processed via the keyboard input, allowing for operations such as sliding, undo and redo to take place. While the graphical interface may be more appealing with the animations and a scalable window, the text interface was predominantly used throughout the project as it allows for representation of 2048 without the strictly unnecessary features the graphical interface possesses.

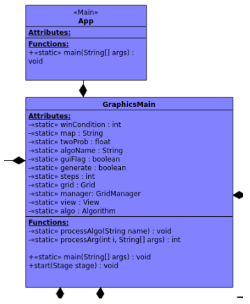
Introduction of new viewing interfaces, such as a mobile application for example, would be required to override the 'View' interface class with the logic of 2048 abstracted away via the 'Grid' class. Hence no code in the 'Model' group would need to be changed to support it, unless new logic functionality is required.


GraphicalView
Attributes:
-stage : Stage
-group : Group
-nodes : Group
-scene : Scene
-canvas : Canvas
-gc : GraphicsContext
-animationLength : Duration
-paddingPercent : float
-roundingPercent : float
-textSizePercent : float
-grid_cols : int final
-grid_rows : int final
-animate : boolean
-input : ArrayList<String>
-canPressFlag : boolean
-actions_buffer : LinkedList<Action>
Functions:
+«constructor» GraphicsView(Grid grid, Stage stage, float width, float height)
-drawRect(GraphicsContext gc, float x, float y, float width, float height, Color color) : void
-drawRoundRect(GraphicsContext gc, float x, float y, float width, float height, float arc_width, float arc_height, Color color) : void
-node_canvas_x(int x, float node_width, float pad_width) : float
-node_canvas_y(int y, float node_width, float pad_width) : float
-createRect(float node_width, float node_height, float node_arc_width, float node_arc_height, Color color) : Rectangle
-createText(int value, float node_width, float node_height, Color color) : Text
-createTranslateAnimation(Rectangle rect, Text text, float fromX, float fromY, float toX, float toY, float node_width, float node_height) : ParallelTransition
-display(Grid grid) : void
-lock() : void
-unlock() : void
-display(Grid grid) : void
«interface»
View
Attributes
Functions:
«static» convertStringToActions(String actionString) : List<Actions>
+getInput() : List<Action>
+play(Grid grid, Algorithm algo) : GameStats
StdoutView
Attributes
-scan : Scanner
Functions:
+«constructor» StdoutView(InputStream stream)
-clear() : String
-display(Grid grid) : void

13.2.3 Controller

The smallest and the final group is the 'Controller', and it is responsible for tying everything together into a working application by centralizing and initializing the appropriate procedures with the required arguments. To do this, it makes use of the class interfaces that have been discussed as a way to abstract the functionality to allow for multiple different implementations to be specified under a single class, the use of this will be elaborated in the next section.

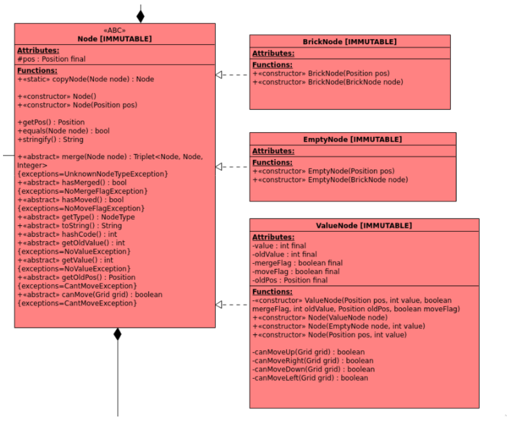
Additionally it is also responsible for processing command line arguments and initializing the respective classes accordingly, such as what algorithm to execute or what view interface to launch based on user input.


«Main»
App
Attributes:
Functions:
+«static» main(String[] args) : void
GraphicsMain
Attributes:
-«static» winCondition : int
-«static» map : String
-«static» twoProb : float
-«static» algoName : String
-«static» guiFlag : boolean
-«static» generate : boolean
-«static» steps : int
-«static» grid : Grid
-«static» manager: GridManager
-«static» view : View
-«static» algo : Algorithm
Functions:
-«static» processAlgo(String name) : void
-«static» processArg(int i, String[] args) : int
+«static» main(String[] args) : void
+start(Stage stage) : void

13.3 Polymorphism and delegation

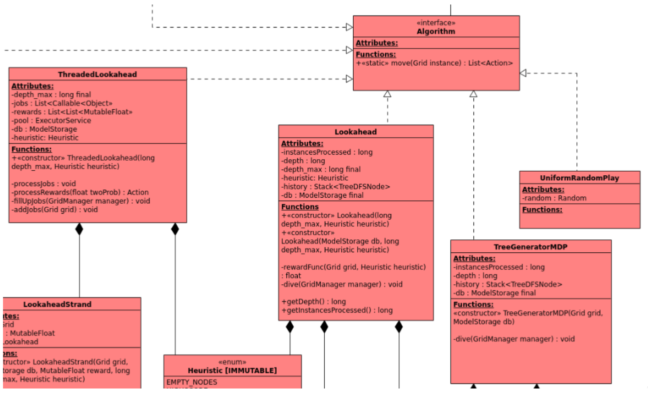
One of the software engineering design questions that needed to be addressed was how to represent the nodes, the grid and the algorithms performed on 2048 from a programming perspective. The choice of data structures would impact not only the efficiency of the code, but in my opinion more importantly, the simplicity of it. An intuitive and straightforward representation is very important when it comes to writing code, as it helps avoid latent bugs that occur when dealing with models and their interactions that are difficult to mentally processes.

Hence as opposed to creating a single 'Node' class that could perform the functionality of every type of node, I used class inheritance to split the node class into three possible sub-classes: 'BrickNode', 'EmptyNode' and 'ValueNode', that override the Node abstract class and implement their own functionality.


«ABC»
Node [IMMUTABLE]
Attributes:
#pos : Position final
Functions:
+«static» copyNode(Node node) : Node
+«constructor» Node()
+«constructor» Node(Position pos)
+getPos() : Position
+equals(Node node) : bool
+stringify() : String
+«abstract» merge(Node node) : Triplet<Node, Node, Integer>
{exceptions=UnknownNodeTypeException}
+«abstract» hasMerged() : bool
{exceptions=NoMergeFlagException}
+«abstract» hasMoved() : bool
{exceptions=NoMoveFlagException}
+«abstract» getType() : NodeType
+«abstract» toString() : String
+«abstract» hashCode() : int
+«abstract» getOldValue() : int
{exceptions=NoValueException}
+«abstract» getValue() : int
{exceptions=NoValueException}
+«abstract» getOldPos() : Position
{exceptions=CantMoveException}
+«abstract» canMove(Grid grid) : boolean
{exceptions=CantMoveException}
BrickNode [IMMUTABLE]
Attributes:
Functions:
+«constructor» BrickNode(Position pos)
+«constructor» BrickNode(BrickNode node)
EmptyNode [IMMUTABLE]
Attributes:
Functions:
+«constructor» EmptyNode(Position pos)
+«constructor» EmptyNode(BrickNode node)
ValueNode [IMMUTABLE]
Attributes:
-value : int final
-oldValue : int final
-mergeFlag : boolean final
-moveFlag : boolean final
-oldPos : Position final
Functions:
-«constructor» ValueNode(Position pos, int value, boolean mergeFlag, int oldValue, Position oldPos, boolean moveFlag)
+«constructor» Node(ValueNode node)
+«constructor» Node(EmptyNode node, int value)
+«constructor» Node(Position pos, int value)
-canMoveUp(Grid grid) : boolean
-canMoveRight(Grid grid) : boolean
-canMoveDown(Grid grid) : boolean
-canMoveLeft(Grid grid) : boolean

Doing it this way avoids having unnecessary if/switch statements that would need to evaluate the type of 'Node' class before executing the appropriate methods, however using delegation it is possible to call the methods without any checks. This is done via polymorphism of the Node object, which can be one of the three classes mentioned previously, because of inheritance. Therefore not only is this an intuitive representation, it is more efficient in terms of control flow operations, which reduces branching of code and hence its complexity.

The 'Algorithm' class also uses this idea to create an interface for all the agent procedures that can be executed in the framework, by using delegation after initializing the corresponding sub-class of the interface. This philosophy also allows trivial extension of new nodes and algorithms by simply inheriting the appropriate interface/abstract class and overriding the required methods, without any modification needed to the rest of the code.


«interface»
Algorithm
Attributes:
Functions:
+«static» move(Grid instance) : List<Action>
ThreadedLookahead
Attributes:
-depth_max : long final
-jobs : List<Callable<Object»
-rewards : List<List<MutableFloat»
-pool : ExecutorService
-db : ModelStorage
-heuristic: Heuristic
Functions:
+«constructor» ThreadedLookahead(long depth_max, Heuristic heuristic)
-processJobs : void
-processRewards(float twoProb) : Action
-fillUpJobs(GridManager manager) : void
-addJobs(Grid grid) : void
Lookahead
Attributes:
-instancesProcessed : long
-depth : long
-depth_max : long final
-heuristic: Heuristic
-history : Stack<TreeDFSNode>
-db : ModelStorage final
Functions
+«constructor» Lookahead(long depth_max, Heuristic heuristic)
+«constructor» Lookahead(ModelStorage db, long depth_max, Heuristic heuristic)
-rewardFunc(Grid grid, Heuristic heuristic) : float
-dive(GridManager manager) : void
+getDepth() : long
+getInstancesProcessed() : long
UniformRandomPlay
Attributes:
-random : Random
Functions:
TreeGeneratorMDP
Attributes:
-instancesProcessed : long
-depth : long
-history : Stack<TreeDFSNode>
-db : ModelStorage final
Functions:
«constructor» TreeGeneratorMDP(Grid grid, ModelStorage db)
-dive(GridManager manager) : void
LookaheadStrand
tes:
rid
: MutableFloat
ookahead
ons:
tructor» LookaheadStrand(Grid grid,
torage db, MutableFloat reward, long
nax, Heuristic heuristic)
«enum»
Heuristic [IMMUTABLE]
EMPTY_NODES

13.4 Immutability

It took me some time to realize that my project was starting to grow more and more complex as the development continued, before it started to impact my ability to diagnose and find bugs. Majority of issues I was starting to face at the start of the second term were related to unintended or unaccounted for side effects of my objects, the order in which I would interact with them by calling their methods would result in different outputs of the program. While this is still valid behaviour, it became more dangerous as time went on, I had to be aware of not only the functionality behind each method call but also what potential side effects it would create. Obviously this made it very difficult to verify if my programs were working correctly and refactoring existing code to introduce new features was challenging.

Realizing that this problem needed to be addressed, I decided to get rid of as many side effects as possible, which can be achieved by making objects immutable. While some could argue that doing this may decrease the efficiency of my code, as immutable objects need to be recreated every time they are altered, I think that it is a price worth paying as it improves my ability to verify programs by making them simpler.

Immutability minimizes side effects by disallowing the state of a given object to be changed at all, if a change is needed the object need to be recreated. Already this solved any potential bugs related to accidental referencing, as well as force me to alter any classes that relied on use of boolean flags, that changed program output whenever the order of method calls was altered creating bugs that were obnoxious to debug, as they relied on the programmer remembering the intricate interactions between the classes throughout code execution. By using immutability, such fragile relationships were no longer possible, as classes could no longer modify each others state without recreating each other making them more independent from each other.

In some cases immutability is too expensive to be practical, as was the case when I was altering the weights of my neural networks. Since each layer was an immutable object, all the existing weights would need to be reinitialized whenever they were updated and because this is expensive to do it on every propagation it tremendously decreased performance. In those scenarios I came up with the term 'partially immutable', that implies that such an object only has necessary side effects and tried to minimize them when possible.

13.5 Enums

The algorithms need a way to interact with the grid, to slide the nodes or to perhaps undo an action. Usually such operations would simply call the corresponding functions directly using the controller class, however in this case this is not possible because the procedures to be called are not known at compile time, they are dynamic. Therefore, I created a representation of all the possible actions an algorithm could perform on the grid by using enums, which are then processed by the 'Controller', calling the appropriate procedure. 'Magic numbers' could have been used, but enums provide a much clearer picture of what kind of procedures the 'Grid' class supports. In order to support compatibility with classes that may not be aware of the 'Action' class, there are several static methods that convert Actions into corresponding integer/char type and back.

Once again, extending more possible actions would simply involve creating a new enum element and handling it in the 'process' method of the 'GridManager', which gets called in the 'Controller'.

Chapter 14: **Appendix**

14.1 Project Diary

September 30th

- Completed project plan.
- Made a start on the UML diagram of the 2048 simulator.
- Had a meeting with my supervisor where we discussed how to tackle the project ¿ Read about MDPs and thought how to apply them to my project
- Approximated the possible number of states 2048 has, $10^1$6 as a lower bound.

October 25th

- Finished implementation of a text based 2048, with all the features including proper merging of tiles, detecting if we lost the game as well as ability to construct any game map via a specified string (which was completed last week).
- Made a start on creating an MDP representation of the game, as well coming up with an algorithm to construct the tree of all the possible game states for a 2×2 game. Several assumptions have been made, such as starting with an empty grid, and only having 2's be generated to simplify the game and hence reduce the size of the tree. After completion of this basic representation of all the possible game states I will look into expanding the tree to handle a full 2×2 2048 game instance.
- Begun trying to come up with ways to compute the reward for the MDP process, for now I shall simply count up the number of wins and losses for each node if we were to descend down that node, using this ration as the reward function. This should be suitable for now as it this reward function would focus on maximizing winning the game (albeit in perhaps not the optimal amount of moves) but for now this shall be suitable, at least for experimentation and perhaps benchmarking. I do not believe this method/approach will be able to handle 3×3 or higher dimensions of the game due to the rapidly growing state space, but I will simply have to experiment and see for myself.

November 11th

- Created a recursive DFS algorithm to traverse through the 2048 tree of all the possible states, this however failed to compute due to stack overflow (which later I realized was due to a bug in my program, it is very possible that a recursive procedure would have been sufficient).
- Upon the failure of the recursive DFS I created an iterative implementation which is more efficient by avoiding flooding of call stack with thousands of recursive method calls. This would also be much easier to parallelize if it will come to this.
- Applied the Bellman equation to calculate the optimal choices at each possible game state, this is then stored in a hash table which can be seen as the 'model' for the game solver.

- Applied caching optimization which reuses already computed states instead of recomputing the entire sub tree again.
- During the project meeting it was pointed that it was redundant to calculate the sub trees of the other actions once we have reached the win terminal state. This reduces the total size of the game tree which needs to be processed.
- Currently running experiments in an attempt to get the DFS to converge within a reasonable time frame, which I have not succeeded in doing yet, but time will tell. I am thinking about potential compromises/parallelizations such as threads which could potentially be applied to speed this process up, as well as debugging information which could be used to judge the current progress of the DFS algorithm.
- At this moment the main bottleneck for the algorithm seems to be speed as opposed to memory, even with optimizations such as caching.

November 15th

- Made a start on the interim report, currently at 1k words having written the abstract and gone into great details explaining the rules of structure of the game 2048.
- Fixed the DFS algorithm for traversing the 2048 game tree, which previously had a fault which caused optimizations to not work as intended. This lead to the algorithm working correctly and successfully generation a model to solve 2048 under the constraints of a 3×3 grid up to 256, higher goal numbers do not posses possible win paths however more experimentation is required to prove this.
- Having completed 3×3 version of 2048 we can proceed to start thinking of how to tackle 4×4 grid. Unfortunately the DFS algorithm is constrained by memory which grows exponentially with the size of the grid and estimations from current observations hint at simply not being possible for our given hardware to compute the optimal DFS algorithm for a 4×4 grid with no constraints on the game. We are talking about thousands of terabytes of states.
- Thus we shall need to think of alternative methods such as Monte Carlo trees, Reinforcement Learning, Neural Networks, lookahead and other approximating procedures. Ideally we shall implement them all and deduce through experimentation those with the highest win rates.

November 24th

- Integrated support for the probabilistic generation of the node with a value of 4, which completes the specification of 2048 rules not relating to grid size. Using this I can proceed with experimenting to find the limits of the currently developing optimal algorithm, so far it seems that the main bottleneck is primary memory. This issue could be fixed by replacing the current hash table model with an SQL database, but it is difficult to predict if this will be sufficient for a 4×4 grid. Currently, the hash table cannot be made large enough to accommodate the immense state space generated by a 4×4 grid even for small win targets. It seems at this point in time that a 3×3 grid up to a reasonable value is the best the optimal algorithm is able to handle within reasonable time and memory. It is time to start looking into approximations and alternatives, as was predicted to happen this is not a surprise.
- This week however was mostly dedicated to the report, whose deadline is fast approaching. I have described every I have done and achieved so far in it, currently at 4k words

so only 1k more to go before I have reached the required criteria. However, referring and images will also need to be included, which will be the focus of the upcoming week as well as making the presentation slides.

November 30th

- Finished the interim report, adding several diagrams and a chapter on the next steps for this project.
- Almost completed cleaning up the code/documentation so it is at a professional standard.
- Made several optimizations in the code after careful analysis, such as merging two loops when copying Grid states which results in simpler, more efficient implementation.
- Created several diagrams for the report, as well as made sure the UML diagram is up to date.
- Ran and documented several experiments of the optimal algorithm as proof of work and personal interest.

January 19th

- Integrated an SQLite database to act as state storage, which greatly increases the capacity of the optimal algorithm as opposed to the previous pure hash table implementation, which was bottlenecked by primary RAM memory. Now the model is able to processes up to 4 billion states, whereas previously it could only reach 50 million. However there is still difficulty in pushing it even further due to the need for disk caching in order for the SQL entries to be retrieved in reasnable time. I am currently not sure how this can be solved without comprimises, none of which look favourable.
- Created a GUI view for 2048 game, this is to replace the previous text interface that is primitive, although effective. Animations have been added along with a custom palette so that the important information about the game's current state is well conveyed to the user. This makes visualizing the running algorithms much more beautiful and easier to interpet for humans.
- Performed a lot of refactoring of the older code which made some assumptions, that increased its complexity. The codebase now is much simpler and more intuitive, which will hopefully make bug fixing and maintenance easier.
- UML has be re-synced with the codebase so that it accurately reflects the dependencies between the classes as well as their attributes and functions.

February 2nd

- Rewrote the project to use immutable objects for its data classes, this is important as it tremendously simplifies the ceonption model of the code, removing the need for bit flag switching as all the logic for state manipulation can be built into the constructors for the appropriate tasks; removing the need for setters. The reason I believe this is a good idea is that we now have to initialize the object completely during its creation or not create the object at all. Hence all potential bugs due to improper copying/referencing have now been removed entirely albeit at the cost of efficiency of not requiring to create the object, however I think is is a good trade off. It does not matter how efficient a program is if it does not perform correctly.

- These immutable objects needed new code logic to be supported and hence many algorithms such as grid manipulation have been re-written to support it, it is much cleaner and simpler now and the refactoring allowed me to make some new optimizations both in terms of performance and conceptualization, which should not be underestimated. Furthermore the new system simplifies the optimal/lookahead algorithm and because of my refactoring allowed me to realize that my initial implementation was not fully correct, as there were heuristic function evaluation steps I was not performing correctly due to my negligence. This has now been fixed and resolved, evaluation of the new algorithm shall be the next steps in the project as alongside I attempt to implement an efficient multi-threaded solution.

February 15th

- Created a standard feed-forward neural network implementation, achieves 96% accuracy on toy MNIST dataset, with manually computed backpropogation.
- Implemented momentum based gradient descent, arguably a naive optimizer however it is much simpler to implement than more complicated variants such as Adam, which I am planning to add after I have observed some evidence of learning from the RL 2048 system.
- Implemented He weight initialization, however with a slightly altered formula which results in the weights having mean zero and standard deviation of 1. This is very important, as it avoids gradient vanishing for more sensitive activation functions such as sigmoid or tanh.
- Implemented the Relu activation function, will stick with it for now however more sophisticated possibilities such as Leaky/Parametric Relu as possible, however for now I want to keep things as simple as possible.
- Implemented L2 regularization of weights, may implement Dropout if I decide to experiment with larger networks, but I do want to evaluate performance of L2 or perhaps even L1 first.

March 9th

- Integrated created neural networks into a reinforcement learning procedure, dedicated a week to experimentation whilst working on project report. After trying out a multitude of hyperparameter values and network sizes so far the network was not able to learn anything useful with the highest achieved value of 64. However, it is difficult to tell if this result was pure chance as there does not seem to be any kind of stradedgy which was the hope.
- Worked on the report, extending it with my findings throughout the second term including SQL database, lookahead and neural networks. Also wrote the section on professional issues here I discuss the ethics of using Artificial Intelligence and more specifically the potential danger to human life of using stochastic models such as neural networks to make decisions, as opposed to a deterministic, explainable solution.
- Improved the lookahead heuristic by instroducing an additional metric that attempts to pick favourable states based on where the high valued nodes are located. Instances with higher valued nodes located at the top of the grid are favoured to those with high valued nodes scattered around. The intuition is that it is a good idea to keep the harder to merge nodes in a corner of the board to prevent it from making states unmergable and hence losing the game.

- More thought and experimentation is required in creation of these stability matrix maps, at the current moment they are designed by hand using hopeful guess work. Perhaps there is a heuristic procedure one could follow.

March 17th

- Finalized the documentation of the codebase, patching the threading segement to be supported by the newer implemented immutable framework. This was delayed for a long time, but has finally been implemented, meaning that 2048 games are once again threaded for superior performance.
- Made progress on the report, reformatting the experiments section and adding more content throughout. Only two sections are left that should take me to the completion of the report.
- Begun work addressing the README file, that will be used to explain how to run and execute the required tests and programs.

March 24th

- After extensive experimentation discovered a map with a consistent 97% win rate on default 2048.
- Implemented more sophisticated order heuristics as the experiments went on.
- Finished off the report and the user manual in due for the final submission.
- Generated javadoc for the project and updated the README detailing how to run software.

14.2 Project Plan 1nd Term

Day	Goals & Deadlines	Description
01 October	Project Plan	Complete the plan for the project, including having an initial draft of the UML diagram for the 2048 simulator.
04 Oct	Started working on 2048 simulator	Initial start on a text based interface for the simulator, which will be upgraded to a fully-fledged GUI later down the line. Creating a fully tested backend early will allow for easier, more sophisticated GUI integration.
18 Oct	Formulate the game as an MDP on a 2x2 grid with the goal number 8	A toy proof of concept, which will be extended on to support games on larger grids and with higher goal numbers. If it does not work out, I will roll back to a deterministic environment which I will solve using CSP methods.
25 Oct	Completed the 2048 simulator (Text Based) & Extend MDP to support larger environments	Having a working simulation of 2048 means that algorithms to solve the game can be applied and experimented with. Attempt to determine up to which game configuration is using MDP's feasible.
01 Nov	Start working on a GUI implementation	Will make use of JavaFX to create the needed proper graphical implementation for 2048, with the text based interface serving as a back-up/additional interface.
08 Nov	Make a start on the Interim report	Should start early due to the 5,000 word expectation
15 Nov	Start preparing for the interim presentation	Having started on the report should allow for easier consolidation of ideas into a presentation
22 Nov	Finish the GUI	A generous deadline, due to potential difficulty in making it look good as well as work.
29 Nov	Begin investigating how to make mobile applications	Will need to understand what kind of tools are needed to build apps portable to android and IOS, will be something new and hence uncertain how long it will take.
03 December	Deadline for Interim Report	Make the submissions required for the interim report
13 Dec	Port the application to a mobile device	Changes will most likely need to be made to accommodate the smaller screen, libraries used to render the simulation.

14.3 Project Plan 2nd Term

Day	Goals & Deadlines	Description
17 January	Create a UML for the approximators that will be implemented and the GUI for 2048	Begin the work as the UML is being created to avoid delays.
24 January	Prepare to extend the interim report into the final report	Start making the necessary additions to the report, to meet the 15,000 word criteria. Good idea to start early.
31 January	Begin formulating the game as a Reinforcement Learning problem or another approximator	Should be simple enough as formulating the game as an MDP already allows us to apply a reinforcement learning algorithm with minimal changes.
28 February	Start preparing for the presentation of software	Need to come up with a poster and general idea of how and what I will be presenting during the showcase.
07 March	Confirm that everything works as expected	The project should be complete by now and simply needs polishing, documentation, comments, and fixing bugs
21 March	Make the final submissions	It is unknown when exactly we are supposed to submit the project, but I will assume it is at the end of the spring term

14.4 User Manual

Solving 2048

by Daniil Kuznetsov

'2048 is a simple game to play, but that does not imply that it is easy to solve. This is because of the immense state spaces which can be generated via the application of the 4 actions the player can execute. The merging of nodes after each action along with the procedural generation of a new node into the grid, allows the game system to terminate after a sufficient number of moves within the restricted 4x4 grid space, but the merging property makes it difficult to precisely estimate the volume of possible states which can be generated by the game. This chapter focuses on going over all of the components which make up the specification of 2048, as well as the impacts they will have on creating an algorithm that is able to solve the game.'

Dependencies:

- java >= 15
- maven >= 3.8.3 (Only to recompile, not necessary to run anything)

Getting started:

All the programs in this project are ran via a single .jar file called *'2048.jar'*, that accepts command line arguments.

`java -jar 2048.jar --help`

Example programs:

```
 1. java -jar 2048.jar --algo lookahead --map "####|####|####|####" --s 2 --gui --heuristic order
      'A good demonstration of the best algorithm created in this project solving the
      base 2048 game.'

2. java -jar 2048.jar --algo lookahead_threaded --map "####|####|####|####" --s 3 --gui --heuristic order
      'The threaded version that has an additional step of lookahead, should have
      slightly better performance but is slower in practise.'

3. java -jar 2048.jar --algo optimal --map "##|##" --win 16 --gui
      'Optimal algorithm executing on a very simple, downscaled version of 2048.
      Attempting to play at higher win conditions than 16 quickly becomes unfeasable.'

4. java -jar 2048.jar --algo lookahead_threaded --s 2 --heuristic order --n 100
      'Same as the first .jar file, but instead plays out the lookahead across 100
      games on a text interface. Summary of wins and losses is provided at the end
      of the experiment.'

5. java -jar 2048.jar --algo lookahead_threaded --map "xx#xx|x###x|#####|x###x|xx#xx" --s 2 --gui --heuristic highscore
      'A more unorthodox game grid, notice how the highscore heuristic is used instead.'

6. java -jar 2048.jar --algo player --gui
      'Simple iteration that allows the user to play the game'
```

Video examples:

Default, 2048, 2-step lookahead, GUI:https://youtu.be/HA_AGu2Lgzk

Unorthodox 1, 2048, 2-step lookahead, GUI:https://youtu.be/898grpU1WEY

Unorthodox 2, 2048, 2-step lookahead, GUI:https://youtu.be/OqV71RUjaBM

Default, 2048, 2-step lookahead, 5 games, ASCII:https://youtu.be/eQnTdfLYIVQ

Command line configuration:

The above command will give an overview of all the basic configuration parameters that can be specified into the application.

This project has two distinct interfaces, a text and a GUI. The GUI can be initialized by passing the *'--gui'*argument to 2048.jar. The text interface works by passing in user controls in a dynamic fashion, whilst the GUI directly executes the keypresses. Note that one is able to pass sequences of commands into the text interface.

```
Usage:
  --ARGUMENT <TYPE> [DEFAULT VALUE] {SUPPORTED INTERFACE}

Options:
  --map <STRING> ["####|####|####|####"]
        Construct a string map representation of the
        grid, on which to play out the game.

        Symbols:
            | == Separator of grid rows
            # == Empty node
            x == Brick node
            2 == Value node 2
            4 == Value node 4
            8 == Value node 8

        Example:
          "####|2###|#2##|####"

              |    |    |
          -------------------
             2|    |    |
          -------------------
              |   2|    |
          -------------------
              |    |    |


  --algo <player|optimal|lookahead|lookahead_threaded> [player]
        The algorithm to use, note that 'player' simply
        allows the user to play the game themselves.

  --win <INTEGER> [2048] {TEXT INTERFACE ONLY}
        The value of the victory node.

  --nogen <BOOL> [false]
        If true, will not generate the two starting
        nodes.

  --s <INTEGER> [2]
        Lookahead steps, will only take effect if any
        of the lookahead alrogrithms is specified.

  --n <INTEGER> [1] {TEXT INTERFACE ONLY}
        Number of games to play out.

  --heuristic <empty|highscore|order> [order]
        Lookahead heuristic to use when evaluating
        leafe states.

  --gui
        Enables GUI interface.
```

Algorithms:

Optimal

The best, but the least practical algorithm. It is only feasible for 2x2 instances of 2048, with the maximal possible win condition being 32. A 3x3 grid will also work with win condition of 8, but 16 it becomes computatinally impractical.

Lookahead

Achieves consistent 93% win rates for the base version of 2048, with 2 step lookahead and order heuristic. 3 step lookahead ahead does not seem to impact the average win rate, out of the samples that have been documented, and 4 steps ahead becomes computantionally impractical.

Player

Allows the player to play the game and make moves themselves.

Player controls:

Each command is entered via stdin (ascii within single quotes below) if using the text interface, or will be processed by the javafx UI thread if the GUI is used.

Movement:

- 'w' = Shift every node up
- 'd' = Shift every node right
- 's' = Shift every node down
- 'a' = Shift every node left

Miscellaneous:

- 'u' = Undo the most recent action, note that taking an action after an undo will overwrite the redo buffer.
- 'r' = Redo the most recent action
- 'x' = Restart the game

System:

- 'q' = Exit game

Custom map examples:

NOTE WHEN USING NON-STANDARD MAPS IT IS HIGHLY RECOMMANDED TO NOT USE THE 'order' HEURISTIC WHEN USING LOOKAHEAD, USE 'highscore' INSTEAD.

"xx#xx|x###x|#####|x###x|xx#xx"

```
 XXXX|XXXX|    |XXXX|XXXX
------------------------
XXXX|    |    |    |XXXX
------------------------
    |    |    |    |
------------------------
XXXX|    |    |    |XXXX
------------------------
XXXX|XXXX|    |XXXX|XXXX
```

"####|#xx#|#xx#|####"

```
     |    |    |
-------------------
    |XXXX|XXXX|
-------------------
    |XXXX|XXXX|
-------------------
    |    |    |
```

"x###x|#####|##x##|#####|x###x"

```
 XXXX|    |    |    |XXXX
------------------------
    |    |    |    |
------------------------
    |    |XXXX|    |
------------------------
    |    |    |    |
------------------------
XXXX|    |    |    |XXXX
```

"#x#|###|#x#"

```
       |xxxx|
--------------
      |      |
--------------
      |xxxx|
```

Small difference in text and GUI interface functionality:

Because GUI is inteded to not be used during large scale experimentation, but as a pretty feedback to the user, it will keep playing past the number 2048. Meanwhile the text interface will terminate at 2048, as this interface was used predominantely to run experiments for the project and it was important to finish running each game as soon as possible for the sake of running time.

Directory layout:

src/main/java/cs3822/

> Code for every class within the project, annotated with code comments.

javadoc/

> Generated javadoc, use a webrowser to open index.html or simply navigate to the code to read the documentation.

Documents/

> Contains all .pdf files, such as the user manual and the report.

misc/

> Contains UML image file, as well its corresponding UMLetino format.

This project uses the Maven Java Build Tool standard directory structure, whose documentation may be found here:
https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Bibliography

[1] Michael O Rabin and Dana Scott. “Finite automata and their decision problems”. In: *IBM journal of research and development* 3.2 (1959), pp. 114–125.

[2] Cyril Banderier et al. “Analysis of an Exhaustive Search Algorithm in Random Graphs and the nˆc\logn-Asymptotics”. In: *SIAM Journal on Discrete Mathematics* 28.1 (2014), pp. 342–371.

[3] Frédérick Garcia and Emmanuel Rachelson. “Markov decision processes”. In: *Markov Decision Processes in Artificial Intelligence* (2013), pp. 1–38.

[4] Brendan O’Donoghue et al. “The uncertainty bellman equation and exploration”. In: *International Conference on Machine Learning.* 2018, pp. 3836–3845.

[5] István Szita, Bálint Takács, and András Lorincz. “ε–MDPs: Learning in Varying Environments.” In: *Journal of Machine Learning Research* 3.1 (2003).

[6] Jacob Russell and Eugene Santos. “Explaining reward functions in markov decision processes”. In: *The Thirty-Second International Flairs Conference.* 2019.

[7] Joshua R Bertram and Peng Wei. “Memoryless exact solutions for deterministic mdps with sparse rewards”. In: *arXiv preprint arXiv:1805.07220* (2018).

[8] J Ross Quinlan. “Decision trees and decision-making”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 20.2 (1990), pp. 339–346.

[9] Malcolm Barrett. “An introduction to directed acyclic graphs”. In: *URL: https://cran. r-project. org/web/packages/ggdag/vignettes/intro-to-dags. html* (2018).

[10] Patrick Bahr and Emil Axelsson. “Generalising tree traversals to DAGs: Exploiting sharing without the pain”. In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation.* 2015, pp. 27–38.

[11] Jason J Holdsworth. *The nature of breadth-first search.* 1999.

[12] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[13] Robert Nowak. “Generalized binary search”. In: *2008 46th Annual Allerton Conference on Communication, Control, and Computing.* IEEE. 2008, pp. 568–574.

[14] Per-Ake Larson. “Dynamic hash tables”. In: *Communications of the ACM* 31.4 (1988), pp. 446–457.

[15] Cameron B Browne et al. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[16] Ben Kröse et al. “An introduction to neural networks”. In: (1993).

[17] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[18] Vincent Vidal et al. “A Lookahead Strategy for Heuristic Search Planning.” In: *ICAPS.* 2004, pp. 150–160.

[19] Guillaume Chaslot et al. “Monte-carlo tree search: A new framework for game ai”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* Vol. 4. 1. 2008, pp. 216–217.

[20] Shun-ichi Amari. “Backpropagation and stochastic gradient descent method”. In: *Neurocomputing* 5.4-5 (1993), pp. 185–196.

[21] Léon Bottou et al. “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 12.

[22] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[23] Yanli Liu, Yuan Gao, and Wotao Yin. “An improved analysis of stochastic gradient descent with momentum”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18261–18271.

[24] Donald W Marquardt and Ronald D Snee. “Ridge regression in practice”. In: *The American Statistician* 29.1 (1975), pp. 3–20.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[26] Shaeke Salman and Xiuwen Liu. “Overfitting mechanism and avoidance in deep neural networks”. In: *arXiv preprint arXiv:1901.06566* (2019).

[27] Hon Tik Tse and Ho-fung Leung. “Exploiting Semantic Epsilon Greedy Exploration Strategy in Multi-Agent Reinforcement Learning”. In: *arXiv preprint arXiv:2201.10803* (2022).

[28] Shilun Li and Veronica Peng. “Playing 2048 With Reinforcement Learning”. In: *arXiv preprint arXiv:2110.10374* (2021).

[29] Antoine Dedieu and Jonathan Amar. “Deep reinforcement learning for 2048”. In: *Conference on Neural Information Processing Systems (NIPS), 31st, Long Beach, CA, USA*. 2017.