Title: Probability and Statistics in 2048

Research question: How can the score in the game 2048 be maximized?

Subject: Mathematics

Word Count: 3762

Session: May 2022

Candidate number: jps156

Table of Contents

Introduction

2048 is a single player tile game that is played on a 4x4 grid, where players can choose to move the tiles up, down, left or right. By making a move, the player can slide all the tiles toward one side and can also combine tiles of the same value if they bump into each other. Once two tiles combine, they form into a single tile with the value of the original tiles added up. For example, if two tiles of eight combined, they would form a tile of 16. Every time a move is made, a new tile is spawned in a random empty location in the grid, with a 90% probability of being 2 and a 10% probability of being 4. There is also a score counter, which starts at zero and increases by the value of newly combined tiles each turn. The primary objective of the game is to get the 2048 tile, but increasing score is another possible objective. Additionally, the game ends once there are no legal moves, which is when all tiles are occupied, and no tiles can be combined.

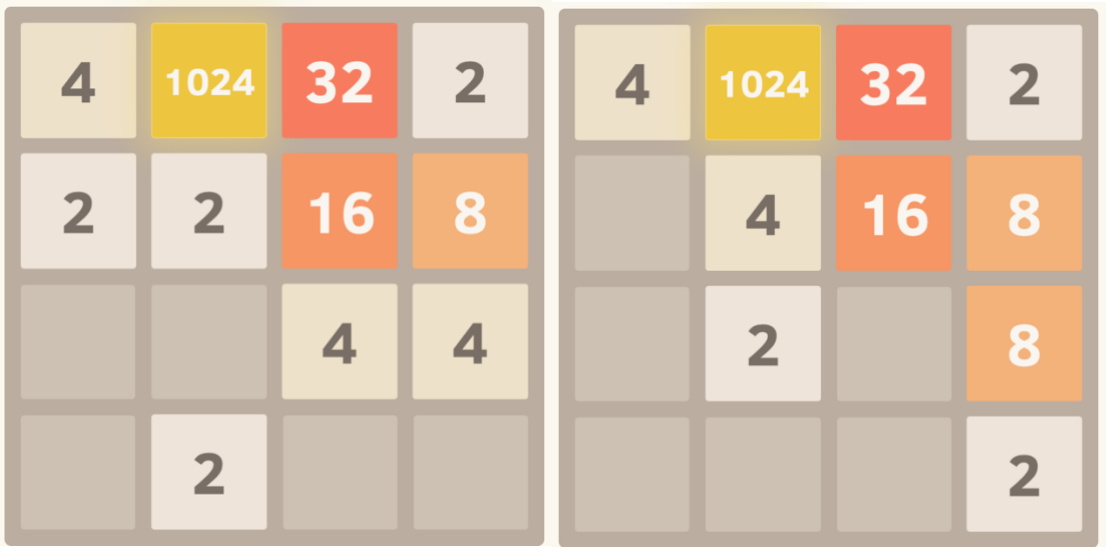

Figure 1, example of a move in 2048

In the example shown in figure 1, a move right is made, resulting in the 2s in the second row and the 4s in the third row combining to form into a 4 and 8 respectively. Also, the 2

in the bottom row shifts to the right side of the grid and a new tile of 2 spawns in the third row. By doing this move, the score counter would increase by 12.

The goal of this investigation is to find out how the score in 2048 can be maximized.

The math will first be done on a 2x2 version of 2048, where newly spawned tiles are all 2s to fully showcase the math and demonstrate the viability of the methods. Then the math can be expanded into the 4x4 version with some additional constraints. In the 2x2 version of 2048, the maximum value a tile can be is 16 as there are 4 spaces for tiles and $2^4 = 16$, so there is no space to fit in more tiles to create a 32 tile. For example, in Figure 2, an extra 2 tile would need to be in the game in order to have the tiles to create a 32 tile, but there simply is not enough space for this in the 2x2 grid. Because of this, getting the 16 tile will be the objective of the simplified version of the game.

Figure 2

By making the game 2x2 and limiting what the new tile can be, the computational complexity for calculating the optimal move is greatly decreased. As established earlier, the maximum tile in a 2x2 grid is 16, so there are 4 possible types of tiles in a 2x2 grid,

all shown in Figure 2. With 4 grid spaces total, that means there are $4^5 = 512$ possible grid configurations. For the worst-case scenario, each grid configuration has 4 valid moves and 3 possible spaces for a new tile to spawn, meaning each grid configuration potentially can lead to 12 other grid configurations. This means that the number of potential outcomes that needs to be calculated and taken into account for a 2x2 grid is $512 \text{ grids} \times 12 \text{ potential outcomes per grid} = 6144$. The real number is likely to actually be lower than 6144 as it was assumed that there were 4 valid moves and that there was only 1 tile in the grid (3 possible tile spawning locations), which are conditions that are not true most of the time. Having such a low number of potential outcomes means that a computer could easily calculate every possible grid configuration in milliseconds.

This contrasts with the full 4x4 game, where assuming the game ends when there is a 131072 tile, the grid has 16 locations, there are 17 possible tile values for each grid location and the worst-case scenario is that there are 15 locations for the tile to spawn with two possible types of tiles spawning in each location, resulting in $16^{17} \times 15 \times 4 \times 2 \approx 5.90 \times 10^{22}$. Even the best-case scenario is too big. Since we know that the computer must at least evaluate every game state that can be reached, which is essentially every possible grid configuration, that means that at least $16^{17} = 2.95 \times 10^{20}$ operations must be done. Considering that the average computer can do roughly 10^8 calculations a second, it would be unfeasible to do any calculations on the entire 4x4 game or even only a small portion of the 4x4 game.

Markov Chains

Markov chains consist of a set of states S = {S_1, S_2, …, S_t}. The transitions between these states are dictated by probabilistic rules (Grinstead). The probability of these transitions happening, such as between states S_A and S_B is denoted as P_{AB}. All of these states and transitions must follow the Markov property, which is that the outcome and probability of future states is solely determined by the current state. In other words, the condition of past states does not affect the outcome or calculation of future states. The Markov property can be represented with mathematical notation shown below, where the probability of S_{t+1}happening given that the present state is S_t is the same as the probability of S_{t+1} happening, given all past states before S_t have happened.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, \dots, S_t)$$

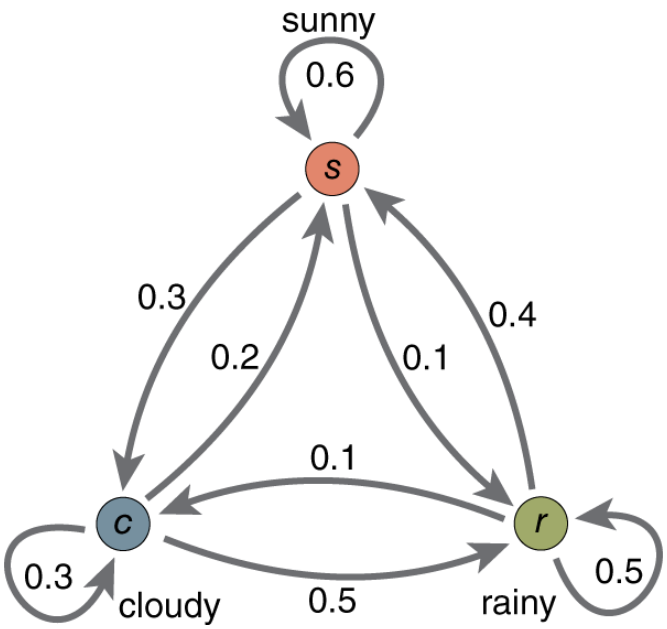

Figure 3, Markov chain represented visually (Bomfim)

Markov chains can also be shown in a visual way like in Figure 3, where there are multiple distinct states and arrows go out of each state showing the probability of going to the next state. For example, in Figure 3, if the current state is cloudy, the chance of the next state being cloudy is 0.3, the chance of the next state being sunny is 0.2 and the chance of the next state being rainy is 0.5. The Markov property is shown in Figure 3, as the probabilities of the next states occurring are solely based on the current state and are not affected by what the previous states were.

2048 follows the Markov property as when calculating what possible states can be reached from the current state, the way that the player has gotten to the current state is irrelevant. This means that Markov chains can be applied to 2048, the application of which is shown below on Figure 4. Note that for this Markov chain, it is assumed the probability of making each possible move is equal and the probabilities of transitioning to other states from a current state do not add up to one because this is only a section

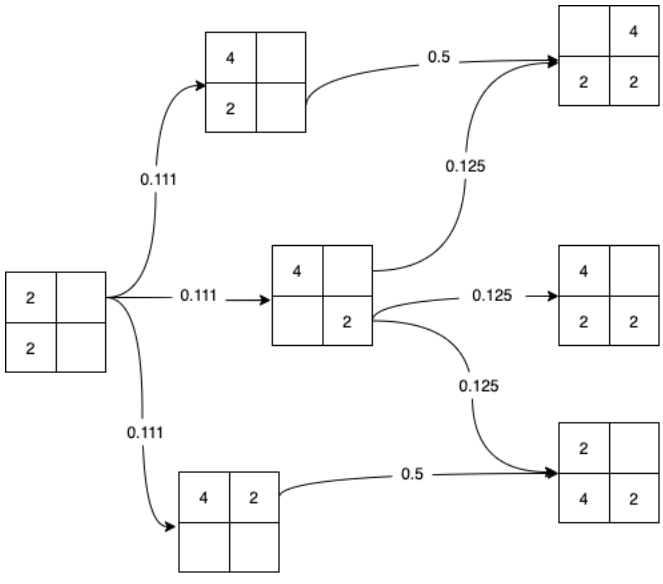

Figure 4, A section of a 2048 Markov chain

of a Markov chain. Additionally, the probabilities have been rounded to three decimal places for demonstration purposes. In reality, a computer can store up to 15 decimal points precisely (computers get inaccurate beyond 15 decimal places).

Markov Reward Processes

The reward for reaching a game state is defined by: $R_s = E(R_{t+1}|S_t = s)$, where R_s is the expected reward of all the possible states that can be reached from state s that are immediately after state s (Jagtap). The expected reward is simply calculated by multiplying the probability of reaching a state by the reward of reaching that state. For 2048, some of the values that can be used as R_{t+1} to calculate reward are the value of the largest tile or the score counter. I decided to use the score counter because it has more possible values, thus being able to convey more information than using the largest tile. There should also be a large negative reward for losing the game in order for the Markov chain to also try to avoid losing, rather than just maximizing the score. This negative value is arbitrarily set, so I adjusted it in the process of calculating Markov chains in order to get better performance. I eventually settled on using the negative value of the score on the tile where the Markov reward process gives a game over. This was so that the "punishment" for getting a game over was smaller in the beginning of the game and bigger later in the game, which is a more proportionate response than having a flat value throughout the game.

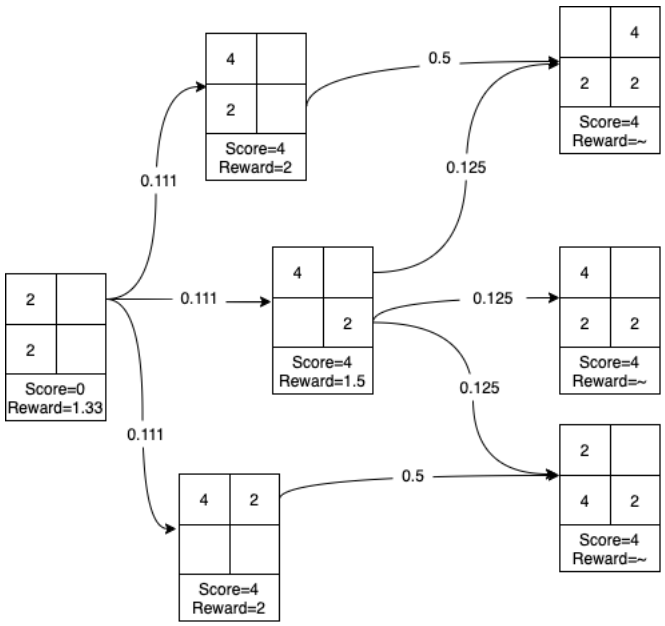

Figure 5, A section of a 2048 Markov chain with reward process

Sample calculations for Figure 5 (these calculations are done off only the game states shown in the diagram and the numbers have been rounded to three decimal places for demonstration purposes):

For the grid $\begin{array}{|c|c|}\hline 4 & 2 \\ \hline & \\ \hline \end{array}$: $\text{Reward} = 0.5 \times 4 = 2$

For the starting grid: $\text{Reward} = 0.111 \times 4 + 0.111 \times 4 + 0.111 \times 4 = 1.332 \approx 1.33$

Markov Decision Processes and Bellman Equations

Earlier, it was assumed that the probability of doing all possible moves were equal, which is essentially playing with random moves. However, in order to actually know which move to take, the calculations must take into account which moves are actually taken. This is why Markov decision processes are used. Markov decision processes are defined by the equation (Jagtap):

$$R_s^a = E(R_{t+1}|S_t = s, A_t = a)$$

This equation is similar to the equation for the Markov reward process, except it now takes into account the action that is taken and calculates the reward based on the action a being taken at state s. This calculation is shown in Figure 6 below, where at state s, a decision is made on which action a to take, based on the reward from doing that action. The reward is calculated using the nodes s' that you can reach by doing action a.

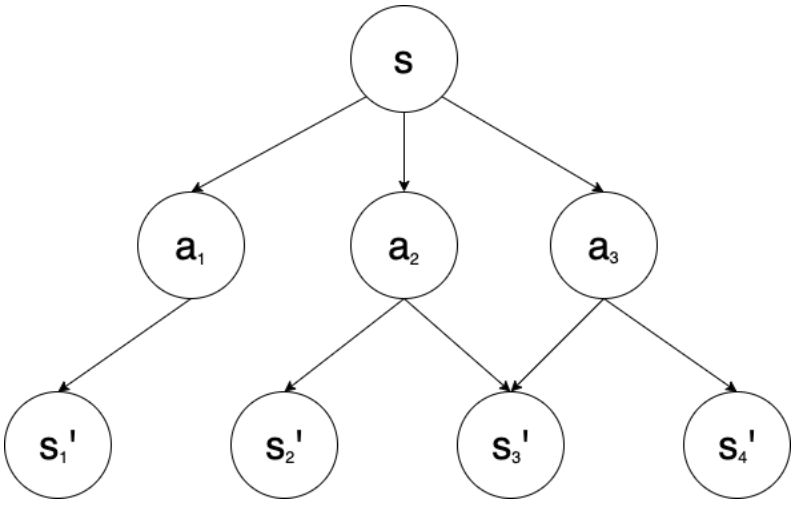

Figure 6, Markov decision process

Markov decision processes also incorporate the idea of return, denoted as G_t, so that states not immediately after the current state are also taken into account. Return is first

calculated using the expected return of all possible states that can eventually be reached from the current state as opposed to the reward of just the possible next states and follows the equation (Torres):

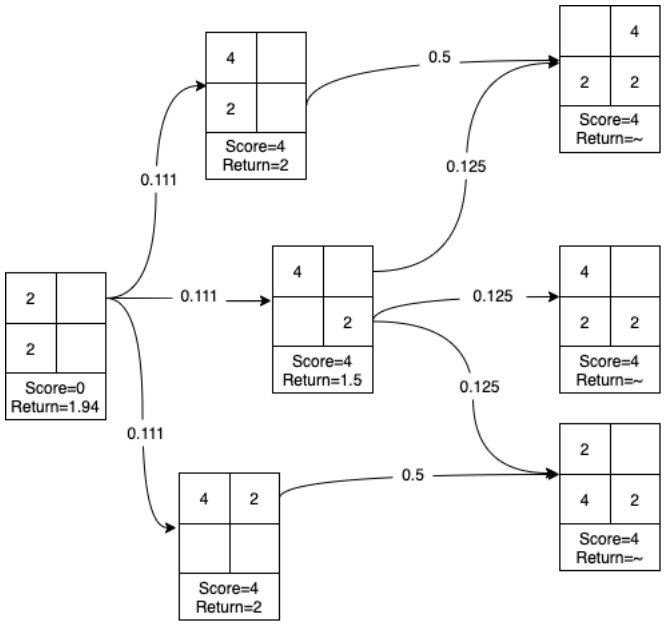

Figure 7, A section of a 2048 Markov chain with undiscounted return

$$G_t = E(R_{t+1}, R_{t+2}, R_{t+3}, \dots | S_t = s)$$

The difference between reward and return can be seen in figure 5 and figure 7, where the starting tile originally had a reward of 1.332, but now also includes the states more than one turn away, resulting in a return of 1.94.

$$\text{Return} = 1.332 + (0.111 \times 0.5 \times 4) + (0.111 \times 0.125 \times 4) + (0.111 \times 0.125 \times 4)$$

$$+ (0.111 \times 0.125 \times 4) + (0.111 \times 0.5 \times 4) = 1.9425 \approx 1.94$$

The return in then adjusted using the discount factor, which is denoted $\gamma, \gamma \in [0,1]$ in order to adjust for short term and long term gain. The return of layers further away is multiplied by the discount factor more times, resulting in them becoming smaller and being prioritized less. A lower discount factor promotes short term gain, while a larger discount factor promotes long term gain. The full equation for return is shown below.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The actual decision making is then also taken into account using the value function, denoted $v_\pi(s)$. The value function is simply the expected return of a state given that a policy of π is taken. For 2048 the policy that should be followed is choosing the action that results in the greatest return. Value is defined by the equation (Torres):

$$v_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s]$$

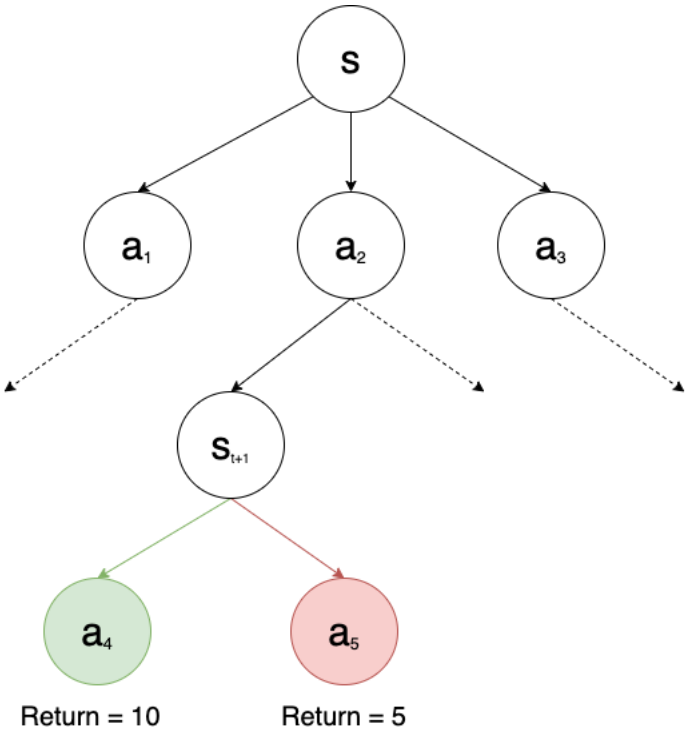

Figure 8, Diagram of Q function

Finally, as outlined earlier, in Markov decision processes we must calculate which action to take at the current state, so the calculations must be based on the return of taking a particular action a at the current state, which is why the Q function is used. The Q is just the value of taken action a at the current state s and essentially evaluates how "good" taking action a at state s would be. The equation is shown below (Torres).

$$Q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a]$$

Figure 8 helps demonstrate how the Q function and by extension, how the value function works. When the return of s_{t+1} is being calculated, the return of the red node is ignored, because according to the policy π, the Markov chain should choose the action that has the largest return, which is the green node. Since the green node has the largest return, all other nodes connecting to s_{t+1}can be ignored. The reason that this is done is that the Markov chain does not and should not need to take into account the actions it will not take or states it will not go to. In simpler terms, if the Markov chain is at state s_{t+1}, it will always choose to do the action in the green node, so the red node's return is not relevant to the value of state s_{t+1}.

Calculating $Q_\pi(s,a)$ can require a huge amount of computational power since for every single state that is reached, all successive states have to be calculated and taken into account, as shown by $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. The amount of computational power can be greatly lowered by using the Bellman equation. The Bellman equation decomposes the value function into two separate parts, the reward of the next state given that action a is taken and the value of the next state (Jagtap). This allows the value function and Q function to be calculated recursively, meaning that the value of all states only needs to be calculated once, rather than repeatedly calculated every time a new state is reached. The proof for the Bellman equation is shown below.

$$v_\pi(s) = E_\pi[G_t \mid S_t = s]$$

$$v_\pi(s) = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s]$$

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s]$$

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots)|\, S_t = s]$$

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma G_{t+1}|\, S_t = s]$$

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(s_{t+1})|\, S_t = s]$$

$$\therefore Q_\pi(s, a) = E_\pi[R_{t+1} + \gamma Q_\pi(s_{t+1})|\, S_t = s,\ A_t = a]$$

Applying the Bellman equation to 2048 (2x2)

The Bellman equations were first applied to the simplified 2x2 version of 2048, previously outlined in the introduction. This was in order to test the basic concept and ability of Bellman equations. I used the programming language C++ to code a simulated version of the game and to calculate the Q function of every game state that the game would reach. The code for calculating the Q function with Bellman equations can be found in Appendix A. The Q function was calculated with a discount factor of 0.9 and losing game states had a penalty equal to the score of the game state.

While coding the game, I also developed an alternative method of calculating the score. Since the score is calculated based on combining tiles, the score can be calculated off of only the current game state by just seeing what tiles there are. For example, a 4 tile is the result of combining two 2 tiles, so would contribute to the score by 4. An 8 tile is the result of combining two 4 tiles. The two 4 tiles would have added to the score by $2 \times 4 = 8$ and the act of combining them would add 8 to the score, so each 8 tile has $8 + 8 = 16$ score attributed to it. This property is important because it means that previous game states would not need to be known in order to calculate the score, which is something that is important for Markov chains. Additionally, it should be noted that this method of calculating score only works when all newly spawned tiles are 2s.

This relationship can be described using the recursive formula $score_n = score_{n-1} * 2 + 2^{n+1},\ n \geq 1,\ score_0 = 0$, resulting in the table:

n	0	1	2	3	4	5	6	7	8	9	10
tile	2	4	8	16	32	64	128	256	512	1024	2048
score	0	4	16	48	128	320	768	1792	4096	9216	20480

This method of calculating score was later also applied to the 4x4 version of the game.

Using the Bellman equations to find which move should be taken resulted in the program reaching the tile of 16 every time and also reaching the maximum score of 68. As mentioned earlier in the introduction, the maximum score can be achieved when there is a 16, 8, 4 and 2 tile inside the grid. Although this may seem like an impressive feat, this is actually quite easy due to how low the level of randomness is in the 2x2 version of 2048, as there are at worst only three locations a new tile can spawn, and the new tile will always be a 2. It is also quite easy for an unskilled human player to also reach the maximum score every time in this version of the game.

Applying the Bellman equation to 2048 (4x4)

The code was then expanded to use it on a version of the 4x4 2048 where only 2s spawn. I decided to do this in order to make the computation less complex as allowing 4s to spawn would double the amount of game states that need to be evaluated. In addition, this would also allow the method to calculate score that is shown in the previous section to be used.

While previously in the 2x2 version, the entire Markov chain was calculated, in this version the Markov chain was only calculated to a depth of 5, or else it would take too long for the program to run. The code for the 4x4 version was the same as the 2x2, but with a small modification to limit the depth. The discount factor used was 0.95, in order to prioritize long term thinking.

Unfortunately, the performance of the Bellman equations was not as good as the 2x2 version for the 4x4 version of the game. Using the moves suggested by the Q function resulted in a mean score of 3217 and a median score of 3454, as shown below.

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
3328	2232	4320	3336	3420	3620	3488	3740	1148	3540
Median = 3454									
Mean = 3217									

The primary reason for this is likely that the Bellman equations work only to maximize the score, while ignoring the location of the tiles. An example that shows why the location of tiles is important is that a 256 tile right next to a 128 tile is far more valuable

than a 256 tile and 128 tile on opposite sides of the grid and separated by other tiles. The Bellman equations are unable to see that one of these configurations is better than the other since the score from these two tiles is the same no matter where there are. This problem might be smaller if the entire Markov chain could be calculated, as the Q function could then “see” and take into account the negative effects of separating two large tiles, which occur further into the future. Since the Q function can only search 5 turns into the future, it cannot “see” the negative consequences of separating two large tiles, as losing the game due to the separation of the tiles might occur hundreds of turns in the future.

One possible way to amend this issue is by getting the Q function to prioritize keeping the bigger tiles in one of the corners, which is a commonly known strategy in 2048 (Johnston). I chose to have the top left corner prioritized. This can be achieved by increasing the value of tiles that are closer to the top left corner when calculating score of a game state. However, this does not improve scores significantly (results shown below) and also deviates from the intended goal of maximizing score.

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
2092	3460	2492	5844	6384	2848	6424	3248	10328	5708
Median = 4584									
Mean = 4883									

The Monte Carlo Method

The Monte Carlo method is algorithm that uses randomness to calculate an approximation of something. The Monte Carlo method is done by repeating a problem with randomness in it over and over again and each time recording down the final result. Because of the randomness, every repetition of the problem can result in a different output. Due to the law of large numbers, the probability distribution recorded from the Monte Carlo method will eventually approach the actual probability distribution of the problem as the number of trials increases (Romualdo).

How the Monte Carlo method works can be more clearly illustrated with a simple game. In this game a die is rolled once, and the player is rewarded with the value of the dice roll. The expected return for playing this game can be directly calculated as $\frac{1}{6} \times 1 + \frac{1}{6} \times 2 + \frac{1}{6} \times 3 + \frac{1}{6} \times 4 + \frac{1}{6} \times 5 + \frac{1}{6} \times 6 = 3.5$. However, the expected return can also be approximated by repeatedly simulating the game. While initially, the mean expected return calculated from playing the game might be quite far from 3.5, after playing enough games the mean return will eventually approach 3.5. This approach is quite useful when the exact properties of the randomness is unknown or when a problem with randomness is too computationally complex, which is the case for 2048.

The Monte Carlo method can be applied to 2048 by repeatedly running simulations of playing the game with random moves, and then seeing which initial move results in the

highest average return. This average return of a move is an approximation of the real return. The discounted return of each simulation can be calculated using the formula:

$$\text{Return} = \text{Score}_{t+1} + \gamma\text{Score}_{t+2} + \gamma^2\text{Score}_{t+3} + \cdots$$

The mean of the discounted returns of all the simulations can then be calculated in order to get an approximation of the real discounted return of taking a move.

The biggest benefit of this method is that it allows the calculations to take into account moves further into the future, which as mentioned earlier, is a major problem with calculating the full Markov chain up to a certain depth. However, this comes at the cost of lower accuracy, as many branches of the Markov chain are left unexplored and uncalculated.

In my implementation of the Monte Carlo method, which can be found in Appendix B, I decided to simulate the game 200 times at a depth of 100 moves, because this resulted in a good balance between accuracy (which requires more search width) and long-term evaluation (which requires more depth), while saving processing power. Another reason I decided to only search to a depth of 100 moves because at this point the discount will mean that new moves have a minimal impact on the score. For example, with a discount of 0.95, which was what I used, the 100th move will be multiplied by just $0.95^{100} \approx 0.00592$.

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
16416	7204	8500	16080	14672	15316	16444	16100	20816	14668
Median = 15698									
Mean = 14622									

The vastly improved performance of the Monte Carlo method compared to previous methods adds credence to the theory that poor performance when searching the full Markov chain to a depth of 5 was due to the fact that it was unable to “see” the long-term negative effects of separating two big tiles. Just by watching the Monte Carlo method’s choices of playing the game progress, it can be seen that it is much more likely to keep bigger tiles together, but it is still not completely immune to this issue. The problem can likely be eliminated by going even deeper allowing each simulation of the game to play until there is either a loss or the largest tile is reached. However, this would come at the cost of significantly higher computational complexity.

Conclusion

In conclusion, by using Markov chains, Markov reward processes and Markov decision processes, the optimal move at any given game state can be figured out with the Q function. However, when applied to the actual game, calculating the entire Markov chain and all game possibilities is unfeasible due to computational constraints. In order to decrease the computational complexity of the task, many methods are used, such as the use of the Bellman equation or the Monte Carlo method, whose results are shown below. The results that I, an unskilled human player, get when playing the game are also shown below. For perspective the highest score I was able to find for a human in 2048 was 630304 (Xiao). Keep in mind that such high score requires luck and multiple retries and do not represent the average performance that a human may achieve.

	Bellman Equation	Bellman Equation with modified score	Monte Carlo Method	Unskilled human
Mean score	3454	4584	15698	5234
Median score	3217	4883	14622	4912

Works Cited

Blackburn. “Reinforcement Learning : Markov-Decision Process (Part 1).” *Medium*, Towards Data Science, 18 July 2019, towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da

Bomfim, Diney. “This is a virtual representation of a Markov Chain”. Medium, 19 July 2018, medium.com/@dineybomfim/practical-swift-markov-model-get-the-best-out-of-ml-8ec355d3c1ce

Chandruscm. “A Simple Implementation of the Popular 2048 Puzzle in C++ . Visit Http://Chandruscm.wordpress.com/2014/10/25/2048-In-c-c/ for Code Explanation.” *Gist*, 27 June 2020, gist.github.com/chandruscm/2481133c6f110ced6dd7.

Grinstead, Charles M., and J. Laurie Snell. Markov Chains. 28 Nov. 2020, https://stats.libretexts.org/@go/page/3171.

Jagtap Rohan. “Understanding Markov Decision Process (MDP) - towards Data Science.” *Medium*, Towards Data Science, 27 Sept. 2020, towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150.

Johnston, Stephen. “2048 Game Strategy - How to Always Win at 2048.” *Gameskinny.com*, GameSkinny, 26 Feb. 2020, www.gameskinny.com/lnagr/2048-game-strategy-how-to-always-win-at-2048.

Romualdo, Gabriel. “Gabriel Romualdo.” *Gabriel Romualdo (Xtrp) – Full Stack Web-Developer*, 2020, xtrp.io/blog/2020/09/12/using-the-monte-carlo-tree-search-algorithm-in-an-ai-to-beat-2048-and-other-games/.

Torres, Jordi. “The Bellman Equation - towards Data Science.” *Medium*, Towards Data Science, 11 June 2020, towardsdatascience.com/the-bellman-equation-59258a0d3fa7.

Xiao, Robert. “2048 AI - 32768 Tile Achieved, Score 630304.” YouTube, 22 July 2014, www.youtube.com/watch?v=JQut67u8LIg&ab_channel=RobertXiao. Accessed 29 Sept. 2021.

Appendix A

```
map<vector<int>, double> rewards;
int optimalDirection;
char moves[4] = {'w', 'a', 's', 'd'};
double discount = 0.9;
double search(vector<int> current, int d){

    double maxval = -9999999;
    int deadcheck = 0;
    // For all four possible move directions
    for (int i = 0; i < 4; i++){
        vector<int> newgrid;
        // update grid according to move selected by loop
        newgrid = movecycle(moves[i], current);
        // if the grid actually changes and the move is thus valid
        if (newgrid != current){
            double rsum = 0;
            int emptyspaces = 0;
            // if this game state has not been evaluated yet
            if (rewards.count(newgrid) == 0){
                // calculate reward from all gamestates reachable from current gamestate
                for (int i = 0; i < newgrid.size(); i++){
                    if (newgrid[i] == 0){
                        newgrid[i] = 2;
                        rsum += discount * search(newgrid, d + 1);
                        newgrid[i] = 0;
                        emptyspaces++;
                    }
                }
                // calculate reward for current gamestate
                if (emptyspaces == 0){
                    rewards[newgrid] = findscore(newgrid);
                }else{
                    rewards[newgrid] = (rsum + findscore(newgrid))/emptyspaces;
                }
            }
            if (d == 0){
                cout << i << ' ' << rewards[newgrid] << endl;
                optimalDirection = i;
            }
            maxval = max(maxval, rewards[newgrid]);
        }else{
            deadcheck++;
        }
    }
```

```
    //check if gameover
    if (deadcheck == 4){
        int x = 200;
        for (int i = 0; i < current.size(); i++){
            if (current[i] == 16){
                x = 0;
            }
        }
        //penalize with negative reward
        return findscore(current) - x;
    }
    return maxval;
}
```

Appendix B

```
map<vector<int>, double> rewards;
int optimalDirection;
double discount = 0.95;
bool unviable;
double search(vector<int> current, char move){
    // initiate search based on initial move
    vector<int> temp = current;
    current = movecyclefast(move, current);
    // if the move is not possible, return a reward of 0
    if (temp == current){
        unviable = true;
        return 0;
    }else{
        int y = rand() % 16, attempts = 0;
        while (current[y] != 0){
            y = rand() % 16;
            attempts++;
        }
        current[y] = 2;
    }

    double average_r = 0;
    // for two hundred search width
    for (int i = 0; i < 200; i++){
        double score = 0, extended_discount = 1;
        // for next one hundred moves (search depth)
        for (int j = 0; j < 100; j++){
            // pick a random move and execute it
            int x = rand() % 4;
            vector<int> previous = current;
            current = movecyclefast(moves[x], current);

            //generate new tile
            if (previous != current){
                int y = rand() % 16, attempts = 0;
                while (current[y] != 0){
                    y = rand() % 16;
                    attempts++;
                }
                current[y] = 2;
            }

            // if game over, penalize with negative reward and end current attempt
            if (checkstate(current)){
                score += -findscore(current) * extended_discount;
                break;
            // else, game is not over, add discounted reward
            }else{
                score += findscore(current) * extended_discount;
                extended_discount = extended_discount * discount;
            }
        }
```

```
        // return game state back to original state before the one hundred random moves
        current = movecyclefast(move, temp);
        // update average reward
        average_r += score/600;
    }

    return average_r;
}
```