I. Introduction

2048 is a single-player game where the player makes actions on a 4 x 4 grid. Initially, there is an empty 4 x 4 grid except two ‘2’ tiles placed on the board randomly. The player moves first, then the computer spawns a ‘2’ tile on an empty grid randomly. When it is the player’s turn, the player will make an action choosing one cardinal direction, ‘sliding’ the entire board in that direction. All non-empty tiles will move towards the edge of the chosen side if able. If two tiles with the same numerical value collide, they will merge and a tile twice the merged value will be spawned. The player wins if a tile with value 2048 is spawned, and loses if no more moves is available.

II. Task Definition

i. State-based Definition

Player input: actions (up, down, left and right) based on the current state
Computer input: adds a ‘2’ tile on an empty grid randomly based on the current state
Output: result state, reward (instant)
Terminal state (success): One ‘2048’ tile
Terminal state (failure): All adjacent tiles occupied and have different values, i.e. no more moves available

ii. Evaluation Metrics

The main goal of this project is to get a ‘2048’ tile as fast as possible. The success of the system is measured by the state variables, including the number of tiles occupied, tile values, number of moves taken, etc.

III. Infrastructure and Baseline

In this project, we have chosen an open-source simulator of 2048 written in C++ as our infrastructure (https://gist.github.com/chandruscm/2481133c6f110ced6dd7). The codes are modified to suit our needs.

Our baseline for this project is a random agent, where a random move of up/down/left/right is chosen at every action step. Comparisons with the baseline are made in the later parts.
IV. Approach

i. Reflex Agent (by Chun Wai LEE)

a. Agent definition

The reflex agent acts as a baseline to demonstrate how Expectimax search outperforms simple reflex agents. It performs maximization over all possible moves. Given the current state, a score is calculated for each action based on several heuristics. The action that leads to the highest score is chosen.

To be specific, for each direction (‘a’, ‘w’, ‘s’ and ‘d’), the program first simulates the state of the board by sliding to that way. It then gathers information about the state (e.g. values and positions of each tile of the board), which is used to compute some parameter values (e.g. number of empty tiles and possible merges). A final score is given to that direction by merging these values with different weights. The movement to the direction that will result in the best state is performed as the actual move. This is repeated until a terminating condition is met, i.e. no more moves available.

b. Heuristics & weightings

Three heuristics were chosen:
- Sum of the tile values
- Number of empty tiles
- Number of possible merges (i.e. orthogonally adjacent tiles with the same value)

Each parameter contributes to a sub-score and the final score is determined by multiplying each sub-score by a pre-defined weight and summing them up together. The heuristic weights were optimized by gradually adjusting each coefficient and selecting the combination that leads to the best results.

Eventually, we have selected the following settings:
- Sum of tile values: 300
- Number of empty tiles: 270
- Number of merges: 520

In other words, the heuristic score is computed by sum_of_tile_values * 300 + number_of_empty_tile * 270 + number_of_merges * 520.

c. Results

The result of running the reflex agent for 10,000 times are summarized as follows:

<table>
<thead>
<tr>
<th>Max tile value</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0</td>
<td>1,118</td>
<td>5,173</td>
<td>3,356</td>
<td>353</td>
</tr>
<tr>
<td>Percentage</td>
<td>0%</td>
<td>11.18%</td>
<td>51.73%</td>
<td>33.56%</td>
<td>3.53%</td>
</tr>
</tbody>
</table>
Maximum score: 26,544  
Minimum score: 1,584  
Average score: 7,273  
SD of score: 352

The result of running the random baseline for 10,000 times are also listed below for comparison:

<table>
<thead>
<tr>
<th>Max tile value</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>320</td>
<td>3,656</td>
<td>4,962</td>
<td>1,062</td>
<td>0</td>
</tr>
<tr>
<td>Percentage</td>
<td>3.20%</td>
<td>36.56%</td>
<td>49.62%</td>
<td>10.62%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Maximum score: 11,660  
Minimum score: 608  
Average score: 3,771  
SD of score: 407

Note: all the individual scores are computed by incrementing it by the score of the higher tier block when two lower tier blocks are merged.

d. Discussion

The reflex agent only considers the current state and does not predict the future as deep as Expectimax. Therefore, as expected, it does not perform well. It reaches 128 and 256 mostly, 512 rarely and never gets to 1024. This agent, however, runs rather fast. We were able to execute 10,000 iterations in about 10 seconds, meaning accuracy is compromised for better efficiency.

There is room for improvement for this type of agent. Firstly, the choice of heuristics could be further enhanced to favor certain states that are more likely to achieve the goal. For example, states with larger values on the edges of the board are favored to push tiles with huge values to the side and avoid them from blocking the small values to merge. Moreover, the proportion of heuristic weights could be improved by adopting different meta optimization algorithm. In this project, only a limited choice of heuristics was employed since having more parameters complicates the optimization of heuristic weights as more weighting combinations will have to be tested.

ii. Depth-limited Expectimax (by Ming Hong LUI)

a. Expectimax definition

For this search method, the game is broken down into a pseudo-two player game, with the player agent taking one of four actions ‘w’, ‘s’, ‘a’ or ‘d’, which pushes the grid towards the chosen cardinal direction, and a computer ‘adversary’, who randomly spawns a new ‘2’ tile where available. Given these behaviors, we can see that Expectimax search can comfortably describe the state transitions, as the
goal is for the player to obtain the highest score possible. Based on this, two node classes were defined, *maxNode* and *expNode*, each storing the grid state, the action leading to the state, the parents and children nodes, and the score. Respective functions, *maxAgent* and *expAgent*, were taken at the corresponding nodes to return a score value which propagates back to the parent.

b. **Reward definition**

The scores were calculated in two steps. If the node is a leaf node at the deepest level, it returns an expected value as the score, based on the composition of the state (see below). This is added to the instant rewards which is the merging of tiles following the actions taken. One nuance in this method is that since the *expNode*’s action is simply to spawn a 2, all the instant reward for *expNode* is neglected as they cancel out due to being a constant, so the heuristic only takes merging into account to simplify the computation process. At the terminal state, the reward is simply defined as the sum of values across all tiles, which is also proportional to the number of moves made before the terminal is reached.

c. **Expected value definition**

Much similar to the heuristics for single player agents, the expected value at the deepest leaves returns a sum of multiple intuitive parameters, including 1) adjacency, which refers to whether there are ‘pairs’ on the board ready to be combined in the next move, which is also value dependent; 2) difference, which refers to the numerical difference between adjacent occupied tiles, since we know that it is never ideal to have an ‘1024’ next to a ‘2’ as it clogs up future potential combinations; 3) emptiness, which is a measure of unoccupied grids in the game. This value is proportional to the number of empty grids, which fits its early-game importance, but diminishes in effect late-game; 4) tile similarity, which gives a comparison on how many tiles are not the same before the move. This is a crude estimate for stability of the board, as one of the optimal ‘human’ strategies is to stick to one corner as much as possible, since leaving the corner will expose the largest value to many chances of ‘2’ spawning adjacent, which greatly disfavors tile merging.

The four parameters are varied with a few selected weights, and based on the test performances, the final weight distribution are shown here:

\[
\text{\{Adjacency, Difference, Emptiness, Similarity\}} = \{2, -1, 100, -100\}
\]

d. **Results**

Due to limited computational power, the maximum traversable depth for this simulation is 3 (3 maxAgents and 3 expAgents), but the package is easily scalable
with no notable memory leak issues. Each depth from 1 to 3 have been run 50 times, and the score variances are shown below:

<table>
<thead>
<tr>
<th></th>
<th>Mean score</th>
<th>SD of score</th>
<th>% reaching 2048</th>
<th>% reaching 1024</th>
<th>% reaching 512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random agent</td>
<td>278.5</td>
<td>87.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(baseline, 500 trials)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth: 1</td>
<td>332.4</td>
<td>113.3</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Depth: 2</td>
<td>829.0</td>
<td>284.1</td>
<td>0</td>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>Depth: 3</td>
<td>1017.7</td>
<td>331.7</td>
<td>0</td>
<td>24</td>
<td>52</td>
</tr>
</tbody>
</table>

From the above result we can see that increasing search depth has a positive effect on both the score and the best achievable state. While none of the three levels has successfully reached 2048, our proposed ‘win’ state, we can still claim that Expectimax could be a reliable method in solving this game, should its improvement in behavior does not fall off at later depths.

An example of console output at the end of the entire simulation (left) and individual runs (right).

**e. Discussion**

From the results, we can see that expectimax has a rather high fluctuation in performances. This is understandable given the many different possible states that are branching from each node, and the path taken will diverge quickly.

Expectimax has demonstrated to be a capable tool to solve 2048, but its efficiency is still to be discussed. Notably during the simulations, its speed decreased drastically at higher depths. In depth 4, it took around 10 seconds to finish making one move. Therefore, it is worth looking into the memory and space complexities.

Treating both maxAgent and expAgent into one depth level, the maximum branching factor for this search is 60 (4 cardinal directions for maxAgent and 15 possible empty spaces for expAgent). However, realistically speaking, the actual branching factor is much lower, and has a tendency to decrease later into the
game since, the grid will be filled with more clutter (lower value tiles before merging with high value tiles), together with more invalid moves in maxAgent. While it is hard to provide a reasonable approximate due to the highly random process, we can say that the upper bound for space complexity is $O(b^d \cdot \text{node size})$. Further pruning, after eliminating invalid moves, is also likely inefficient given the random nature of expAgent.

In terms of realistic improvement in efficiency, the data structure should be one of the key aspects. Currently `std::vector<int>[16]` is adopted for each grid state storage, which is fairly large in size and computationally expensive. To improve, a custom notation can be considered, as each tile value can be simplified into its log₂ digits, where 2048 becomes a mere 11. With a dedicated class definition, one may even consider implementing bitwise manipulation to improve the computational speed. However, these improvements are beyond our capability for now for this one-month project.

### iii. Markov Decision Process (by Adrien PROST)

In this discussion we will see if 2048 can be modelled as an MDP and if it can be solved efficiently using either value iteration or Q-value reinforcement learning.

An intuitive way of modelling the game as an MDP would be:

$S =$ set of all possible states of the game until a 2048 tile is created: set of combination of all different tiles that are either blank or have values under 2048

$= \{\text{empty}, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$

For a 4x4 game board this gives us $|S| = 11^4$ different states (assuming all can be reached which is probably not the case)

$A = \{A, S, W, D\}$ which describes the directions we choose to tilt the board, depending on the state, some of the actions are "illegal" because they would not change the position of the tiles currently on the board.

$T(s,a,s')$ : for each legal action a, there are a certain number of states s’ that can be reached (equal to the number of empty tiles), with probability uniformly distributed (i.e. the board tilts and a new 2 tile spawns in any empty tile)

$R(s,a,s')$ : this depends on what kind of heuristics we want to set for a state. In this project we have chosen to give states values based on some heuristics, instead of using a reward function.

**VALUE ITERATION**

Let’s first see how we would model an agent that could solve the MDP using value iteration.
The idea would be to fix a $V_0$ value to all states (computable on the go).

For example:

$$V_0(s) = a*(\text{# of free tiles on the board}) + b*(\text{number of tiles that can be merged}) + c*(\text{difference between adjacent tiles (more is bad)}) - \text{ when } s \text{ is neither a losing or win state}$$

Otherwise:

$$V_0(s) = W \text{ when } s \text{ is a win state and } V_0(s) = L \text{ when } s \text{ is a loose state}$$

Now let's assume we have found convenient/coherent weights $a,b,c,W,L$ for our heuristic.

The idea would be then to apply value iteration on all states where

$$V(k+1)(s) = \max(a) \sum(T(s,a,s') V(k)(s'))$$

The idea would be to iterate over $k$ until the values converge.

However, this poses a problem since it requires us to store the $V(k)(s)$ values and $|S|$ has an upper bound of $11^{16}$ which is not feasible.

There are probably ways of optimizing the storing of values but this is out of the scope of the course.

**REINFORCEMENT LEARNING**

Now let's see how we could solve the MDP using reinforcement learning.

The idea would be to let the agent play the game (using random moves or with a fixed policy) and at every step we take we compute the q-value of that step in the following way:

- All the q values of the states are initialized at 0
- We use a reward function
  $$R(a,s,s') = a*(\text{number of tiles merged}) + b*(\text{difference between adjacent tiles in state } s') + c*(\text{# of possible merges in the resulting state})$$
  (where $s$ is the initial state, and $s'$ the resulting state with the newly spawned random '2' tile)
- To compute the new q value every time we take a step, we use a learning rate $A$ (that can vary over time), and compute the new q-value the following way:
  $$Q(s,a) = (1-A)*Q(s,a) + A*(R(a,s,s')+ \arg\max_Q(Q(s',a)))$$

After some learning using random moves, we can continue learning using with a probability $P$ a policy that uses the action which has the highest Q-value.

Again the same problem in the value iteration process arises, storing all of the $Q(s,a)$ would take too much memory and hence some other storing methods should be thought to solve this problem using MDP's.
V. Conclusion

Based on the above results, we can conclude that 2048 could be learned in multiple ways, from the simplest reflex agents to reinforcement learning. Simple algorithms may provide less satisfactory results, based on the terminal state reached, but is quick and time-efficient. Expectimax search was an improvement given the outcomes, yet high space and time complexity at larger depth, which supposedly further improves the result, may deter it from being a feasible agent. Therefore, given the discussion, it may be wise to turn towards feature-based learning methods, to design more efficient and generalized learning methods for better performance.