A Comparative Analysis of 2048 Tree Search Algorithms

Henry Middleton (htm29) and Conrad Schmidt (cjs348)

Project Description

Our goal was to create an AI utilizing a minimax/expectimax algorithm to get the highest possible score in the game 2048. In this single-player game, tiles with value either 2 or 4 randomly spawn in, and the player can move either up, down, left, or right. When a player moves in a direction, all tiles shift as far as possible in that direction, and any two tiles of the same value that are then touching merge in that direction to create a new tile with a value equal to the sum of the old two tiles. The game is over when the board is clogged such that no further moves in any direction are possible.

Although there is open source code on GitHub that makes it easy to interface with the original game through a browser, we recreated 2048 in Python and then applied our algorithm to our own graphical interface. When we were researching this idea, we noticed that many people who have built AIs to play 2048 noted that running in a browser can significantly slow down the speed of the AI; thus, we decided to create our own implementation. Many of our speed improvements and heuristics were inspired by nneonneo's open-source 2048 AI.

Approach

Our approach was to implement various versions of tree search and heuristic search algorithms to create the best possible AI. We started with a simple minimax implementation as a base and from there looked to improve the efficiency and effectiveness of our AI through human heuristics. We quantitatively analyzed these metrics to compare our implementations and show our progress towards reaching a better AI overall. We also implemented expectimax and a random minimax where the AI plays as if their is an opponent however the opponent still places tiles randomly. As a reach goal for our project we considered implementing other algorithms and similarly working to improve their efficiency and effectiveness in order to provide a comparative analysis.

Base 2048 Implementation

We decided to implement 2048 ourselves, as opposed to using the open-source code for the most popular version of the game, in order to be able to optimize speed and flexibility. There are not many actions in this game – it's a simple loop of moving/merging tiles, spawning a new tile, and checking if the game is over. We considered representing board states with their own class, but decided against this approach as it would have been slow. In our initial iteration, boards were represented as two-dimensional lists, and operations on boards were done with for loops. Although this was a simple way to get started, it is relatively inefficient for a tree search algorithm to operate on, so we started brainstorming alternatives.

Our first attempt at speeding up board operations was to convert the state representation to a NumPy array. However, this turned out to be less efficient than the original, since most of the operations performed to evaluate and change board states were not easy or even impossible to convert to NumPy operations.

Next, we created a new implementation where a board state is represented as a 64-bit integer. Each of the 16 tiles on the board is represented as 4 bits, which allows for tiles with values up to 32768. Our approach never made it beyond the 8192 tile, so this representational limit was not a problem. All manipulations of the board state are done using bit operations on the integer representing the state.

Speed

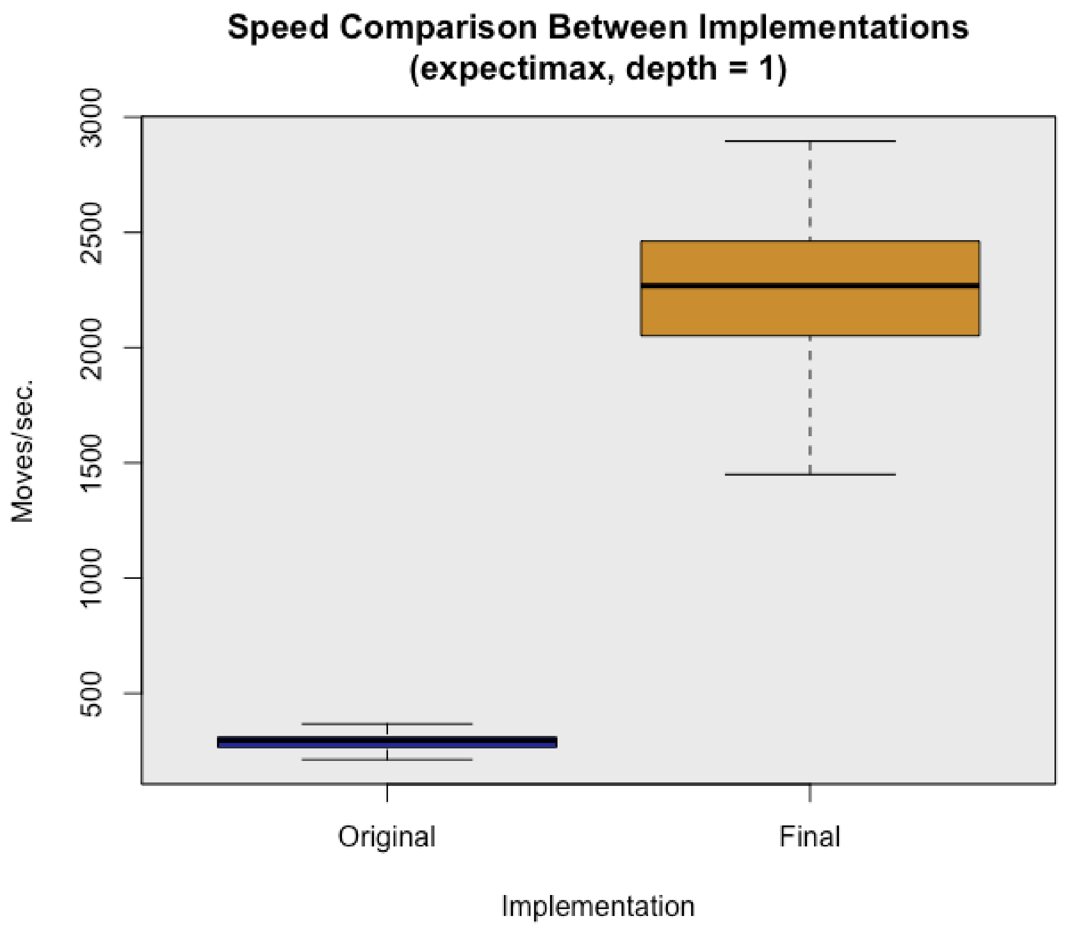
Our final implementation using integers to represent board states made the game slightly faster, but not by all that much. We found that for our tree search algorithms, every layer of depth added to the search was significant, so we wanted to optimize as much as we could.

The first step in our optimization process was to change all board moves and heuristic calculations to use a dictionary to look up the values instead of calculating them every time. There are 2^{64} possible boards under our implementation, so it wasn't feasible to have an entry for every board. Instead, the dictionaries were based on rows, of which there are only 2^{16} possibilities. It's intuitive why this would work for movement, since there are no diagonal moves, but not so much for heuristics. Although there are many different ways to go about evaluating the value of a board state, it's possible to do quite well with heuristics that only consider a single row or column at a time, which made caching heuristics viable as well. To get the heuristic value for an entire board, the heuristic values for each individual row and column within that board are summed together.

To verify that using an integer to represent boards was indeed faster, we conducted timing tests. On 100,000 randomly generated boards, our original implementation took 9.69 seconds total to calculate heuristic values for each board. The integer implementation took only 0.68 seconds to complete the exact same task.

To compare differences in move speed, we ran a series of trials using each implementation with random moves until the game was over. Since the check for the game being over is implemented by moving in every direction and checking for changes, the move speed is still the only variable here. To complete 10,000 trials, the original implementation took 148.9 seconds, but the newer implementation only took 24.9 seconds.

In addition to caching the values for moves and heuristics on individual rows, we also memoized our tree search functions. More detail of the search functions will follow in a later section, but this graph shows the effect of all speed boosts we added after our original implementation over 100 trials:



The reason for the higher relative variance in the final implementation is that the heuristics used were more successful. Expectimax runs much faster when the board is fuller, since it has to explore the possibility of spawns in every empty tile. If runs are more successful on average, there is a larger variance in how full the board is, leading to the larger variance in speed.

Heuristics

To begin our search for creating an AI to play 2048 we began by looking through the internet at work that had already been done on the topic to understand what issues we might face and possible tricks towards creating a better implementation ourselves. Through our research we found countless implementations of 2048 AI. Most of these however, were incomplete at best. Fortunately, by looking at these we were able to figure out some important characteristics that our implementation should contain, such as specific conditions to consider for our heuristics and game implementation strategies that would lead to more efficient execution and search times.

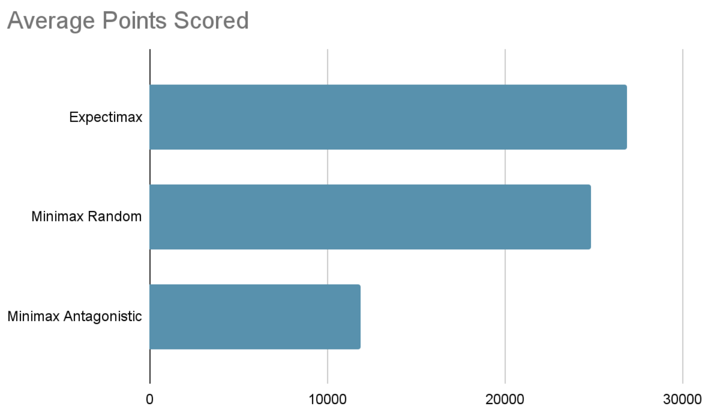
Initial Heuristics

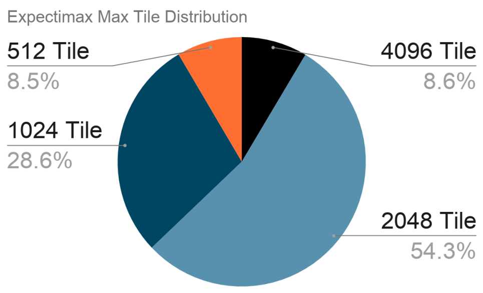
Corner gradient: Our initial heuristic was based on typical human strategies towards winning the game and the strategies that we used to play the game when it was released. A very typical strategy, and one that both of us knew to employ right away was keeping the biggest tile in a corner of the board (any corner) and building the next tiles to merge with it nearby. This usually meant that the board would have the highest value in a corner and to the immediate left or right the next highest tile waiting for merging. This is a natural and effective strategy towards reaching the 2048 tile as it allows for simple and safe merges to happen in either direction moving towards the corner and prevents small tiles from getting in between the larger ones and clogging up the board. We implemented this strategy using a corner gradient system in which a matrix of the same size as the board was used to scale each individual tile putting emphasis on the corners and the adjacent tiles. Here is an example of a top left corner gradient.

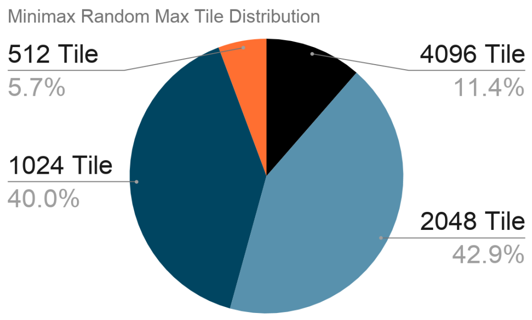
$$A = \begin{bmatrix} 3 & 2 & 1 & 0 \\ 2 & 1 & 0 & -1 \\ 1 & 0 & -1 & -2 \\ 0 & -1 & -2 & -3 \end{bmatrix}$$

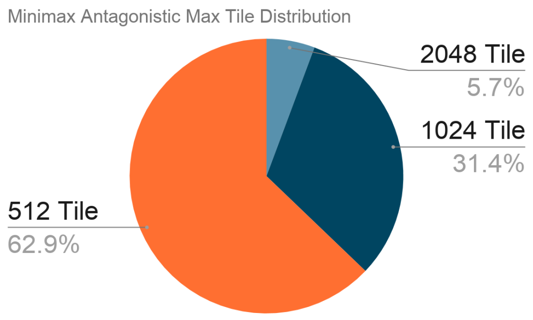
Empty tiles: Another clear strategy is to keep the amount of empty tiles as high as possible as this means that the player is keeping free space available for the tile spawns and ensuring that the game will not end from nowhere to spawn a tile. This idea, however, must be properly balanced as sometimes it is not logical to make as many merges as possible in order to properly line up larger merges later in the game.

Max tile: Our final initial strategy was simply to grant a scaling reward of points based on the max tile represented on the board. The way in which the game is regularly scored does not provide a good basis for a heuristic as it emphasizes points for merging over strategic play which can lead to the issue described above. Giving a reward based on the max tile represented gives the AI an incentive to strategically line up merges as it looks down the tree of possible moves to ensure that it can eventually merge to an even higher max tile.









Top left - Average points scored for each algorithm from 50 runs

Top right - Max tile distribution for Expectimax from 50 runs depth 3

Bottom left - Max tile distribution for Minimax with random tile placement from 50 runs depth 5

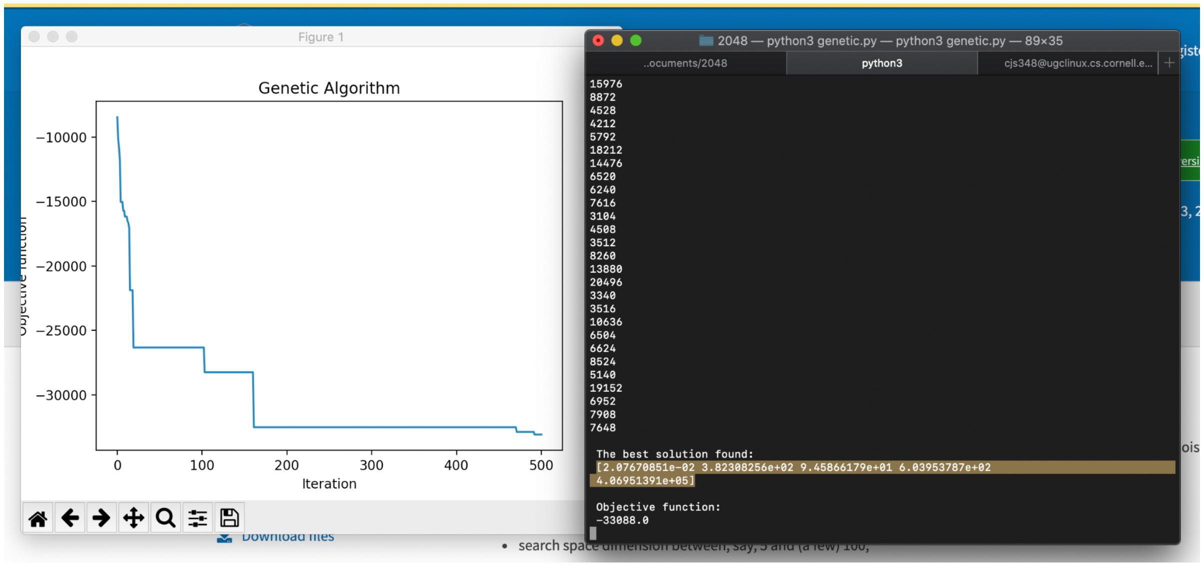
Bottom right - Max tile distribution for Minimax with antagonistic placement from 50 runs depth 5

Attempt at learning

The first step towards utilizing a tree search algorithm for an AI is defining what the heuristics will be, however the next step is to balance the weights of these heuristics to optimize the scores that the AI will achieve. In order to reach high scores the heuristics must be weighted in order to balance their importance among each other and oftentimes must be scale in value and importance for higher tiles and certain board configurations. Initially we were unaware how to properly train heuristic weights and decided to hand tune them to the best of our ability. This worked decently well as our initial heuristics were simple enough to view the effects that they had on the play style of the AI as it made its moves and we could fine tune values following each run.

We knew that in order to reach the maximum potential that our heuristics could achieve we would have to go beyond hand tuning and utilize an algorithm to train each weight. For this we chose to go with a genetic algorithm. We used the python library created by Ryan Mohammed Solgi called geneticalgorithm. This is a lightweight python library for using standard and elitist genetic algorithms. This package solves continuous, combinatorial and mixed optimization problems with continuous, discrete, and mixed variables.

We thought the genetic algorithm would be perfect for our 2048 AI as it was simply a combination of a handful of weights to manipulate the heuristic function and we hoped that we would be able to get our implementation to a speed at which we could run high depth tree searches with relative ease. Unfortunately due to a variety of factors, many of which are described above in the timing section, the feasibility of using a genetic learning algorithm to compute the optimum weights for the heuristics was extremely low. The high population size and number of iterations in order to have a successful result from the genetic algorithm prevented us from running it on anything other than depth one and two for the different tree search approaches. Below is an example of a GA run with a population size of 50, a max iteration size of 500, a mutation probability of 0.1, and a crossover probability of 0.5.

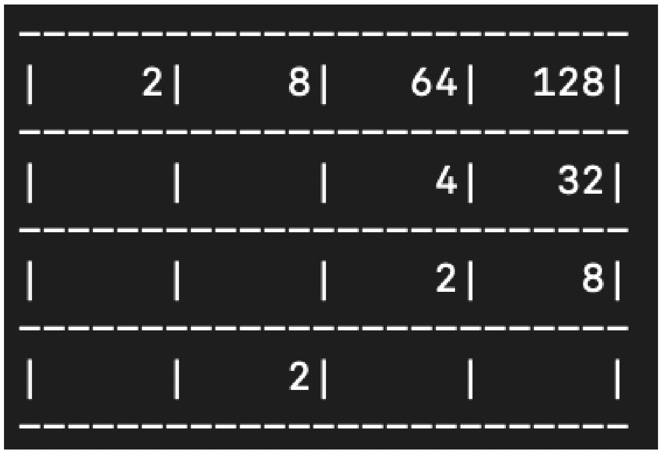


```
15976
8872
4528
4212
5792
18212
14476
6520
6240
7616
3104
4508
3512
8260
13880
20496
3340
3516
10636
6504
6624
8524
5140
19152
6952
7908
7648

 The best solution found:
 [2.07670851e-02 3.82308256e+02 9.45866179e+01 6.03953787e+02
 4.06951391e+05]

 Objective function:
 -33088.0
```

Better Heuristics

As our research continued into heuristics and algorithm improvements we came across a recurring theme in the best 2048 tree search AI implementations. They used two metrics, monotonicity and smoothness, emphasizing their importance above the rest. These two heuristics naturally manifest themselves in how humans play the game but can be difficult to understand and put into words.

Monotonicity: Monotonicity is typically used to refer to mathematical functions. A function that is perfectly monotonic is either completely non-increasing or completely non decreasing. In terms of 2048 this is an important characteristic for rows to have as if a row is either completely non-increasing or completely non decreasing it ensures that future merges are not being blocked by smaller or larger valued tiles between them. This is actually something that happens naturally when using the corner gradient but only in the rows and columns building into that corner. By ensuring that rows and columns are punished more being non monotone the AI is able to achieve the same effect the corner gradient has on all rows and columns. Below is an example of some perfectly monotone rows and columns.

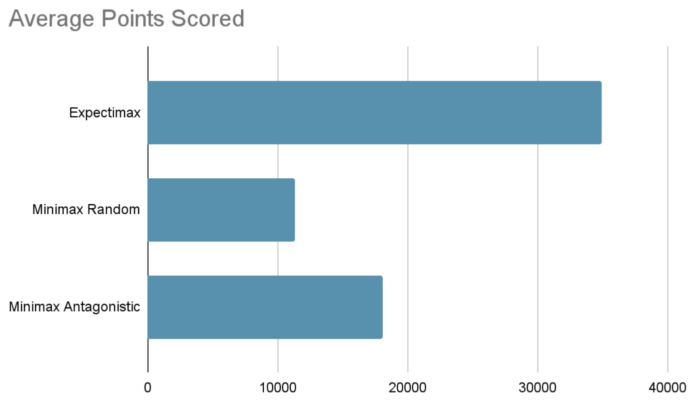


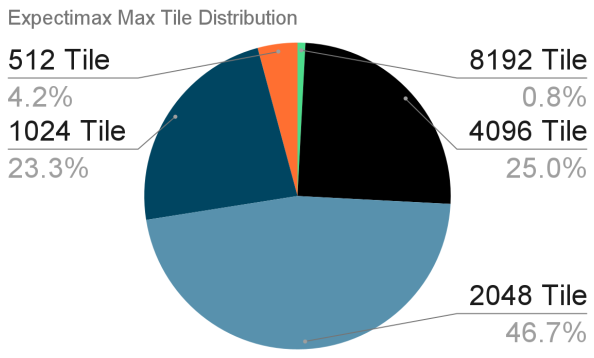
```
-------------------------
|    2|    8|   64|  128|
-------------------------
|     |     |    4|   32|
-------------------------
|     |     |    2|    8|
-------------------------
|     |    2|     |     |
-------------------------
```

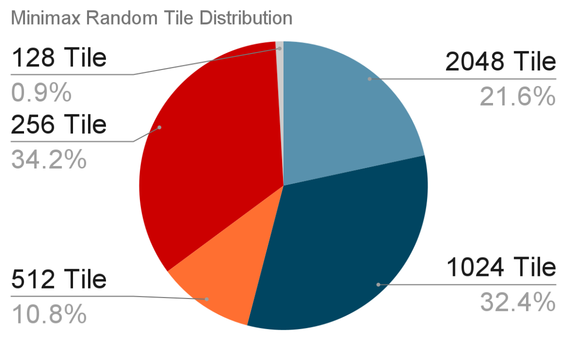
Smoothness: Smoothness is also typically used to refer to functions, however our interpretation is much more simple for 2048. Essentially this heuristic is used to counteract the negative effects of monotonicity. Rows and columns may be perfectly monotonic, but if the tiles that make them up are vastly different in value then it would be nearly impossible to expect to merge them relatively quickly or nicely without cluttering the board. A smoothness penalty punishes rows that contain tiles of great difference in value at a scale with their distance from each other. Thus, if properly balanced with monotonicity, these new heuristics will ensure that rows and columns are ready for merging and their tiles are close enough in value to merge soon.

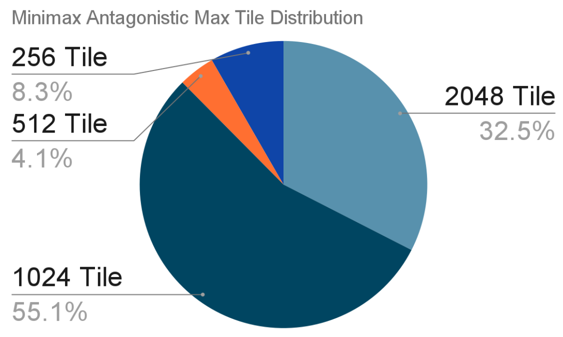
Final Heuristics

Due to our inability to successfully train our heuristic values in a timely manner and our desire to see our AI operating at peak performance we looked to nneonneo's open-source 2048 AI for heuristics and weights. Much of what we had already implemented existed in this AI including monotonicity, smoothness, and empty tiles. However, this AI also had a heuristic called a sum penalty and a weight that affected monotonicity called monotonicity power. The monotonicity power weight is necessary because tiles that are farther apart in value will be harder to merge. As such, a pattern of 2-16-2 is much less likely to cause a loss than a pattern of 256-4096-256, even though the difference in the log values of 2 and 16 and the log values of 256 and 4096 is equal. We don't fully understand the purpose of the sum penalty, but we think it exists to prioritize merging smaller tiles over large tiles. If there's an even distribution of tile values, it's easier for the algorithm to make clever moves that clean up messy board states, so this penalty could be beneficial since due to the fact that tiles are constantly being spawned, bigger tiles will always be generated eventually. In addition to these two weights, this AI also contained a weight for a loss penalty to determine how much to penalize states where the game is over. Our original implementation didn't consider this as a variable weight, and just used a fixed number that we didn't train.









Top Left - Average points scored for each algorithm from 120 runs

Top right - Max tile distribution for Expectimax from 120 runs depth 3

Bottom left - Max tile distribution for Minimax with random tile placement from 120 runs depth 5

Bottom right - Max tile distribution for Minimax with antagonistic placement from 120 runs depth 5

Expectimax vs minimax

Overall, expectimax outperformed minimax significantly, even with a much lower search depth. This was our initial expectation, and makes sense since tile spawning is random, not selected.

The difference in performance between minimax with antagonistic and random spawning was striking. With our original heuristics, random spawning led to higher scores, but with our final heuristics, antagonistic spawning led to higher scores. We aren't sure what the reason for this is. It could make sense that random spawning would be better, since the tiles aren't being placed in the worst possible position. It could also make sense that antagonistic spawning would be better, since moving in a direction that assumes the worst possible spawn allows for spawns in spots that are nearly as bad, but unanticipated by the minimax algorithm.

Although pruning was very useful for speeding up our minimax runs, we were unfortunately not able to prune our expectimax trees. nneonneo's implementation used a clever form of expectimax pruning where if enough 4 tiles spawned in a branch, then exploration of that branch stopped. This works because for each spawn there is only a

10% chance that the spawned tile will be a 4, so it’s not worthwhile to explore these branches. While pruning in this manner is extremely effective, since it can eliminate over half of the branches of a node, it is only feasible at higher depths, since two or three fours spawning in a row is common enough with a large enough effect on success that such an event must be considered. Due to the relative slowness of our implementation, we weren’t able to reach depths where this pruning was reasonable.