Appendix C (code)

The code filenames are largely self-explanatory.

Heuristics

Heuristic.java : Interface for the heuristics

AdjacentMatches.java

AdjacentRuns.java

BoardScore.java

BoardScoreNormalised.java

LinearWeightedSum.java

Empties.java

Searchers

Searcher.java : abstract class for the Searchers

Jogger.java

Pruner.java

Strategy.java

StrategyBT.java

WideSearch.java

StrategyBTJogger.java

DLDFS.java

Miscellaneous

Board.java

Figure1.java

Figure2.java

Figure3.java

Figure4.java

FigureStrat.java

Tester.java

ThreesAI.java

Appendix H (heuristics)

Heuristics

BoardScore

The first and simplest heuristic is the straight raw score of the board. the aim of the game is to maximise this score, so that's what this heuristic aims to do. It is fast to calculate. It is difficult to use with the other heuristics though as the score can vary by large amounts between boards, and is exponentially related to depth in the search space, precluding comparison between depths. It uses a constant time lookup for the score for each cell, making the calculation very rapid.

BoardScoreNormalised

BoardScoreNormalised tries to fix the comparison problem of BoardScore by representing the score of a board at any given level as a double between 0 and 1. The score of a given board is divided by the potential maximum of that board.
This maximum is calculated as if all cells on the board that could be combined are combined. All boards at that level will have the same potential maximum score, so the performance of BoardScoreNormalised should be similar to that for BoardScore. In addition it can be used to compare between levels, allowing use in the priority queue.

Empties

Empties counts the number of empty squares on the board, on the basis that the more empty squares on the board, the longer you can keep playing and the better your final score. This count is converted to a double based on the formula:

$$\text{Empties} = \frac{\min(\text{count}, \text{k})}{\text{k}}$$

k is a constant, it sets the maximum number of empty cells that are valued. We experimented with different values of k and also using different operators for the conversion such as sqrt and polynomial, however linear worked well and was simple. The k value chosen was 7.

AdjacentMatches

This heuristic counts the number of adjacent numbers that can be combined, eg 3 and 3, or 2 and 1. The idea is that these numbers can soon be combined, so having them adjacent should be valued. This is a form of ‘lookahead’ heuristic, an attempt to minimise the horizon effect. The count output of the heuristic is transformed similarly to the Empties formula described above. Please note that this heuristic acting alone would likely not lead to good performance. The ranking of boards would prioritise keeping cells adjacent to combining them. However in conjunction with other heuristics it should improve performance.

AdjacentRuns

This heuristic is more complex: It counts ‘runs’ of values. A run could be for example 3 6 12 24. These runs are valued because if a 3 cell is combined with the 3 in the run, the resulting 6 can be combined with the 6 in the run, etc. Maintaining structures like this helps keep the board neat. The integer output of this heuristic is transformed as for the Empties.

LinearWeightedSum

A linear weighted sum was implemented to combine the other heuristics. This allows us to test different weightings between the heuristics.

Appendix S (searchers)

DLDFS

Depth Limited Depth First Search. Depth first searches are very memory efficient, only using at most O(b*D) memory, where b is the branching factor and D is the depth limit. Due to the no goal state nature of the board, a BFS or iterated deepening did not make sense, as there is no optimal answer to find.
The DLDFS searches in the graph using a stack to store seen nodes. It stores the best scoring (according to the heuristic) node at the depth limit, as well as the best scoring node with no children. This was done to ensure that if no node can be found at the depth limit, the game is going to inevitably end, and so we should choose the best terminal node seen. The DLDFS then returns the move from the root of the graph that the best node is descended from.

Completeness	The strategy always returns an edge, directed towards the best node at depth D if it exists.
Optimality	The strategy is guaranteed to find the best node according to the heuristic at depth D.
Time Complexity	O(nodes) ~ O(4^D + 4^D * heuristic)
Space Complexity	O(b*D)

Pruner

Pruner is a form of priority first search. There are two parameters, the depth limit D and the number of nodes to explore N. Pruner maintains a priority queue of nodes, ranked by the heuristic value of the board at that node. In this way the nodes with the highest heuristic score are explored first, and their children added. When nodes at depth D are found, their children are

not added, and the highest scoring node at depth D is maintained. After N nodes at depth D are found, the strategy returns the direction to the best node.
Without the N parameter, Pruner would retrieve the same terminal node as DLDFS. N exploits the idea that pruner will discover the best nodes at depth D first, and so does not need to explore them all.
Time complexity: The upper bound is exploring every node, as well as the cost of the priority queue, 4^D * lg(4^D) = D * (4)^D. The lower bound is exploring N nodes, with only N/4 parents, etc. This sums to 1.33N nodes, as well as the cost of the priority queue. These bounds are very wide but it is hard to be more specific with a general heuristic.
The space usage of the Pruner is not bounded, while space for our algorithm to use is. This is the disadvantage compared to DLDFS. Even though we can use D and N to tradeoff exploration and depth while maintaining speed, memory usage set a real limit to usage of pruner. We assumed 2Gb of memory, corresponding to a maximum safe depth of 11.62. We used D = 12.

Completeness	The strategy always returns an edge, directed towards the best node discovered at depth D if one exists.
Optimality	The strategy is not guaranteed to return the best node or even a good node, the performance depends on the board ranking from the heuristic and the properties of threes.
Time Complexity	O(N * lg N * heuristic) < Time < O(4^(D+1) * heuristic)
Space Complexity	O(4^D)

Jogger

This strategy was developed as a speedup. DLDFS and Pruner both return a single move to make after exploring their search tree, Jogger returns a list of moves (lenght L). In this way Jogger is run fewer times, allowing the speedup. The disadvantage however is that the horizon effect is exacerbated. A DLDFS search is used to find the best terminal node, however in Jogger this is implemented as a recursive dfs traversal instead of DLDFS’s stack based search. If jogger makes 4 moves at a time, a depth increase of 1 should be available with no time cost.

Completeness	The strategy always returns an edge, directed towards the best node at depth D if it exists.
Optimality	The strategy is guaranteed to find the best node according to the heuristic at depth D.
Time Complexity	O(nodes) ~ O(4^D + 4^D * heuristic)/L
Space Complexity	O(b*D)

Appendix Strategy

StrategyBT

This strategy uses an ‘easy’ and a ‘hard’ search. The idea is for the easy search to run very rapidly, allowing more time for the hard search. This amortises the too slow speed of the hard search over the entire set of moves, meaning the total move sequence runs in time. The swapping between the easy and hard searches is bluntly controlled; when the easy search fails by running into a dead end BT moves are backtracked. The hard searcher then takes over for 2*BT moves, before the easy searcher again takes command. This strategy uses the time and space analysis of whichever easy and hard searches are selected.

Completeness	The strategy always runs as well as the easy search
Optimality	The strategy will find at least as good a path as the easy search
Time Complexity	Easy * (Total moves - BT * backtracks) + Hard * (2*BT*backtracks)
Space Complexity	Maximum of Easy and Hard