Ridge Shrubsall (21112211), Kimberley Siew (21125071) 30th May 2014

CITS3001 Algorithms, Agents and Artificial Intelligence

1 PROJECT SPECIFICATION

The aim of this project was to design and implement an AI player for the game *Threes!*. The game is setup on a 4 × 4 board, where each of the cells can contain either a number tile or an empty space. The subset of values that a number tile can have is of the form $\{1, 2\} \cup \{3 * 2^{k}\}$, where k is a real non-negative integer. The goal is to achieve the highest score possible by combining tiles together; higher individual tile values give a better overall score. The game is over when no more legal moves can be made. This state occurs once the sequence of input tiles has been exhausted, or if the board is full up and no more number tiles can be combined. For each turn, the player can make one of four possible moves shifting the board either Left, Right, Up or Down i.e. $\{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$. A pair of tiles can be combined if the sum of their individual values is a multiple of 3. After making a move, a new tile is added to the board at a deterministic position based on the already existing tiles.

2 PROJECT SOLUTION

The AI's implementation was given a heuristic function to determine the best course of moves in order to have a goal state with the highest possible score at the end of the game. In addition, look-ahead moves were given to the AI to enable it to examine states that are many moves in the future, much different to the actual game which only gives an indication of the next tile. The AI was tasked to output between 5 to 10 moves per second. The search function that we chose to implement was the iterative deepening depth-first search algorithm, and this was done using Java.

The moves were set out so that each sequential choice could be modelled in a tree, with each move representing an individual child node of the current board state. Each move consists of four possible choices: $\{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$. This was modelled by expanding each node in the tree into four child branches; each depth *d* of the tree then contains 4^{d} child nodes. At each node, a score was calculated using the total score gained by the tiles plus a heuristic score based on the overall 'quality' of the board (explained further later). The search algorithm's task was to traverse down the tree in order to find the greatest scoring sequence of moves.

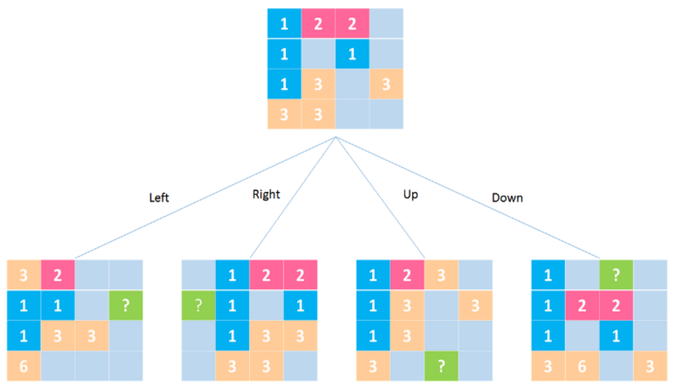

Figure 1: Search tree representing each board state at a node

Certain design choices were made in selecting what search algorithm and combination of heuristics should be used. These choices limited the AI's search capability to a realistic timeframe, in which the time and space constraints of the algorithm needed to be considered. The project specification required the AI to output around 5 to 10 moves per second, placing a high value on time efficiency. Other decisions that we had to consider were how to weigh up different combinations of heuristics against each other, e.g. do we value a high number of empty spaces more than the number of high-valued number tiles? These decisions affected how we rated the 'quality' of a board state.

3 HEURISTICS

In order to find a path to the best goal state in the tree, the choice of the next move at each node was assessed by how advantageous it would be to the final score and the longevity of the board. Future states that were considered to be beneficial were ones that lead to more open spaces on the board, the formation of combinable tile pairs and the prolonging of the end game state.

Empty spaces

Moves which increased the number of empty spaces on the board were highly valued. By increasing the number of empty spaces on the board, this would allow for more tiles to be able to move around and combine together and for further moves to be made, potentially resulting in a higher score.

Combinable tile pairs

Pairs of tiles next to each other which could be combined were preferred over other placements as they could lead a state providing both a higher overall score and opening a new space on the board. Combining tiles results in a higher individual tile value, which are also preferred as they lead to higher scores e.g. a 12 tile would be preferred over two 6 tiles. A 1 or 2 tile was given a score of 1, whereas tiles with a value of 3 or greater were given a score of the form $3^{\log_2\left(\frac{x}{3}\right)+1}$, e.g. a 12 tile would have a score of 27 whilst two 6's would each have a score of 9, adding to a total of 18.

Double-weighted tile pairs

If a tile was located next to a tile with double its value, this could lead to a possible tile combination being formed later on. Thus, double-value pairings were also highly valued in addition to pairs that could be combined.

Trapped tiles

As well as adding bonus points for beneficial board states, states in which tiles could become 'trapped' were penalised. If a tile became trapped between either two high-valued tiles or a high-valued tile and an edge it would not be able to combine with any neighbouring tiles, resulting in a major complication. This would act as a wall between a high-valued tile and other potentially combinable tiles, limiting the number of later possible moves. To stray away from these states, the total score of the state after accounting for the individual tiles was deducted by 5 points for each trapped tile.

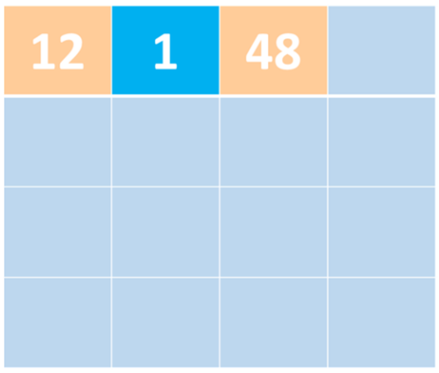

Figure 2: Example of a 1 tile being 'trapped' between two higher-value tiles

Other heuristic strategies were considered which examined the values and positions of all of the tiles on the board. One such strategy that was believed to result in a higher overall score was to move the higher valued tile(s) to one side of the board and then collect the lesser valued tiles in other places. This would then allow for the lesser-valued tiles to be combined and positioned in ascending order, causing a chain reaction in which a sequence of tiles can be combined one after the other resulting in a new high-value tile. Problems involved in using this strategy include generating high-valued tiles nearer the lower-valued ones, potentially creating a 'trapped' tile placement. This heuristic was initially included in the AI, but proved to be less effective than previously thought as the final scores resulting from this chain reaction strategy did not have any significant improvement.

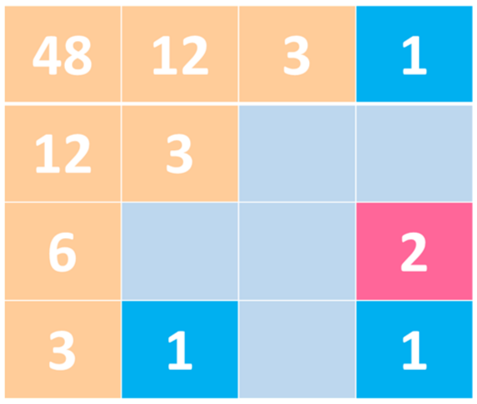

Figure 3: Example of cornering the highest valued tiles

4 STRUCTURE AND DESIGN OF ALGORITHM

Depth-first search (DFS) traverses a tree down to its leaf nodes by exploring each branch of the tree as far as possible. This is done by starting at the root node and then expanding each successive child node, travelling down the tree. If a leaf node is reached, it traverses back up the tree to its parent node and then searches its next child. This process continues through all of the branches in the tree until every node in the tree has been examined. The time complexity of doing a depth-first search is $O(b^m)$, as it follows each path down to the bottom of the tree. Its main advantage over a breadth-first search is that it has a lesser space complexity, equal to $O(bm)$. This is due to the fact that a DFS only needs to store the nodes in its current path rather than all of the nodes at a particular depth of the search space. The algorithm also only needs to keep track of the best path found; suboptimal paths in the tree can be disregarded. This method (on its own) is an incomplete search algorithm, and thus it was not chosen as an ideal strategy. Threes has a near-infinite search space only limited by the number of available tiles, and as a consequence it is infeasible to perform a search for all of the goal states due to the high branching factor. Using an unbounded depth-first search to look through the tree would result in a massive exponential growth in the number of nodes, and thus it is not a practical search algorithm to use.

Initially, a depth-limited search (DLS) was used as the search algorithm. This algorithm, as the name suggests, searches a tree in exactly the same way as a depth-first search but it has a fixed depth limit instead of exploring to the very bottom of the tree. Thus a DLS still has the optimality of a DFS whilst also being a complete algorithm as it manages to avoid the pitfall of searching to near-infinite depths. To search the tree, the algorithm traverses down to the specified depth limit. Upon reaching a goal state or the depth cut-off, the node's heuristic score is evaluated and the path to the best node is stored. The AI then makes a move towards the direction of the best node. In choosing the best depth limit, the time and space complexity had to be considered. Having a low depth limit would cause the AI to make very suboptimal move choices, but the depth limit cannot be too high either as the search needs to be performed within a reasonable timeframe in order to meet the requirement of 5 to 10 moves per second. Thus initially, a depth limit of 6 was decided upon as it was the highest value that performed well and met the required moves per second. Using a depth-limited search gave reasonably good scores, but with further tests we wanted to see if this could be improved.

The final search algorithm used was an iterative deepening depth-first search (IDDFS), where multiple depth-limited searches are performed increasing the lookahead depth each time up to a maximum limit. The algorithm initially searches with a lookahead of 1 throughout the whole tree, then it goes back to the initial board state and does a second search with a lookahead of 2, and so on up to a maximum of 6. The upper bound on the time complexity is then $\mathrm{O}(\mathrm{b}^{\mathrm{d}})$ where *d* is the maximum lookahead. This algorithm also meets the required moves per second as the total moves and time taken are now amortized across all searches.

Using an IDDFS, more goal states are able to be found than just doing a single depth-limited search. Instead of finding a single optimal goal state with a lookahead of 6, the algorithm finds the optimal goal states for all lookaheads between 1 and 6 and then selects the best one. This balances out the cases where because of the imposed depth limit the AI will make moves that are initially good but then lead to an early end game (more on this below).

After optimizing the final search algorithm and adjusting the heuristics used, it was found that the maximum lookahead could be increased from 6 to 8 whilst still running in time. On a 20,000 tile input file, the IDDFS algorithm now runs with an average of 12 moves/second. The DLS search in the last iteration with a lookahead of 8 also meets the minimum requirement of 5 moves/second.

Quiescence

Just using a heuristic score to find the best goal states through analysing the number of empty spaces, combinable tile pairs and trapped tiles is not enough to achieve a high score – the longevity of the board also need to be considered. While we want to find the highest scoring goal state, moves that result in a large score increase may actually lead to a suboptimal goal later on. Using a quiescence search, the depth-limited search function can be extended to search deeper on nodes that appear to be 'noisy', i.e. they may be close to an end game state. An iterative deepening approach was chosen over a quiescence search as it seemed to offer better scores when varying the number of tiles in the input.

Backtracking

A third option for trying to avoid reaching a suboptimal goal would be to go back up the tree upon reaching a goal and look for another goal in a different move branch for some maximum number of extra goals to examine. This method was never implemented as it was thought to be too difficult, and it is unknown whether this would offer any significant improvement without drastically slowing down the search runtime.

5 NOTES ON IMPLEMENTATION

Multithreading

To try and increase the overall number of moves per second, we experimented with a multithreaded search algorithm. Before, the IDDFS algorithm could only do one search iteration at a time but using multithreading the IDDFS algorithm can perform multiple search iterations simultaneously through utilising a multicore processor. This approach results in a faster search runtime, but on large input files it only saved a few minutes on average, so it was left out in the end as it did not provide a significant enough benefit to warrant overcomplicating the code and it was not thoroughly tested.

Storing tiles in memory

Initially the AI would keep track of the tiles that it had used by copying a list of the remaining tiles in memory, however after doing some testing with larger input files it was found that it would be much more efficient for all of the board states to reference the same list and just keep track of how many tiles had been used so far in each state instead. This optimisation allowed the maximum lookahead to be increased whilst achieving a similar performance to before.

6 EXPERIMENTAL AND THEORETICAL ANALYSIS

In testing our search algorithms, we ran our AI on 50 randomly generated boards consisting of 20,000 tiles. The algorithms that we compared were the IDDFS with all of the heuristics included, the IDDFS with just the trapped tile penalty and a single DLS with a lookahead of 8.

The best performing algorithm was the IDDFS with all heuristics, with an average score of 476,000 and an average runtime of 11 minutes at 12 moves per second.

The second-best algorithm was the IDDFS with only the trapped tile penalty, with a lower average score of 415,600 but a faster average runtime of 9 minutes at 13 moves per second.

The single DLS algorithm came in last, with the lowest average score of 372,200 and an average runtime of 7 minutes at 5 moves per second. Note that the moves per second is significantly lower than the other two algorithms because this is only a single iteration with a lookahead of 8.

Algorithm statistics

Iterative deepening depth-first search (all heuristics):

	Score	Total moves	Time taken	Moves per sec
Min	68,890	4,098	263.9 s	8
Mean	476,000	8,047	684.4 s	11.7
Max	797,300	10,980	1022.0 s	16

Iterative deepening depth-first search (trapped penalty only):

	Score	Total moves	Time taken	Moves per sec
Min	68,890	3,557	229.2 s	9
Mean	415,600	6,962	547.5 s	12.6
Max	789,700	9,959	810.3 s	18

Depth-limited search (with 8-ply lookahead):

	Score	Total moves	Time taken	Moves per sec
Min	68,890	783	166.1 s	4
Mean	372,200	2,313	425.8 s	4.9
Max	789,700	4,056	664.7 s	6

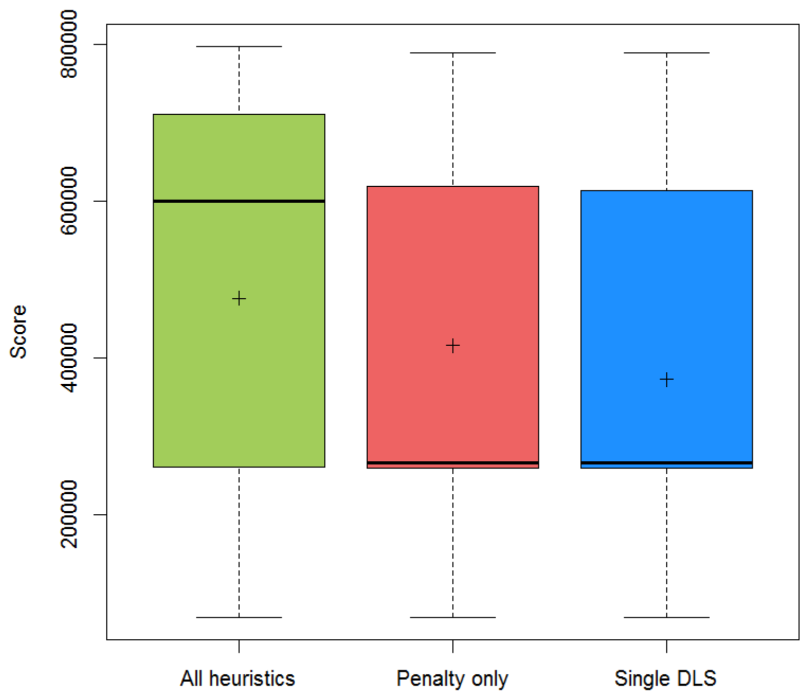

Figure 4: Comparison of scores for each search algorithm

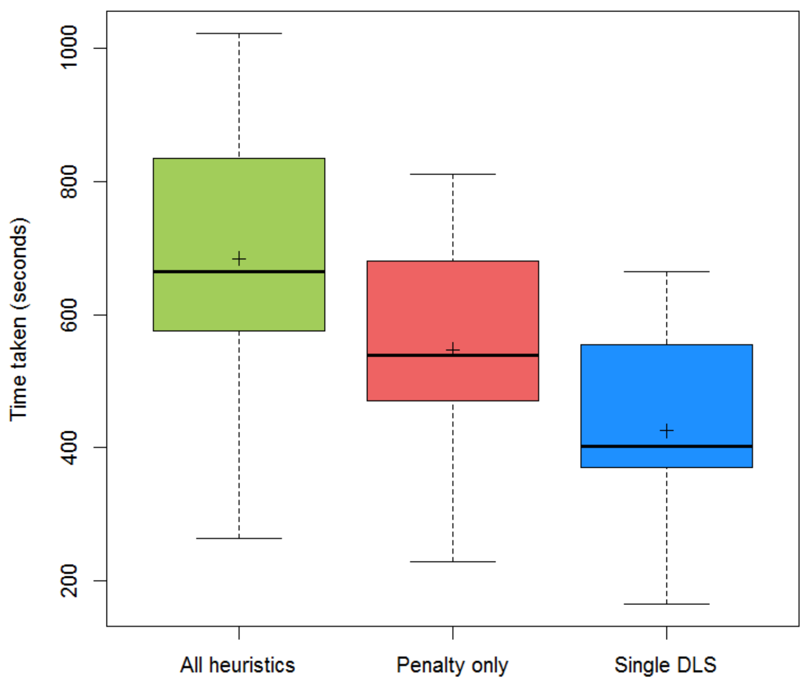

Figure 5: Comparison of runtimes for each search algorithm

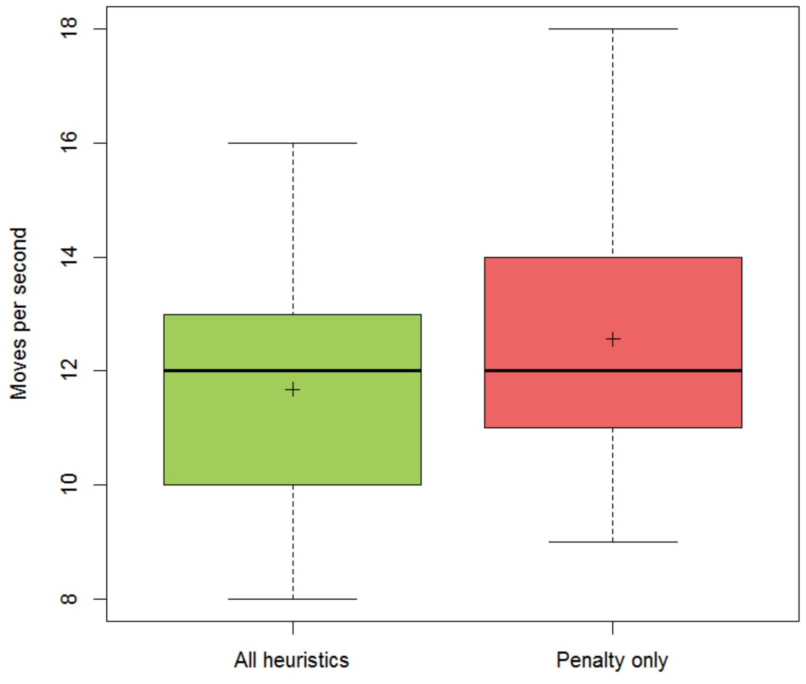

Figure 6: Comparison of moves/second for the IDDFS algorithm

Heuristics on small input files

Without the heuristics being included in the IDDFS algorithm, it was found that the score on the first example input file had a better score than with the heuristics included. The example input files only contained less than a hundred tiles whereas the larger files that we tested with contained tens of thousands of tiles. This difference in scores can be attributed to a flaw in not merging high-value tiles before running out of possible moves. However, this only seemed to happen with some of the small input files as including the heuristics was much more optimal with the larger input files.

`Small input file without heuristics`	`772 points, 12 moves/sec`
	
	`96  1  0  2`
	` 0 12  0  0`
	` 3  2  2  0`
	` 0  0  6  0`

`Small input file with heuristics`	`530 points, 11 moves/sec`
	
	`48 48  3  2`
	` 3  0 12  0`
	` 2  0  0  0`
	` 0  6  0  0`

Table 1: Comparing a small input file with and without heuristic bonuses/penalties

7 CONCLUSION

Through testing each of our different search algorithms, the iterative deepening depth-first search with the heuristics included was found to be the most optimal algorithm compared to a single depth-limited search. The IDDFS algorithm had a longer average runtime but it resulted in the highest overall scores.

8 REFERENCES

[1] Walt Destler (2014) *Walt Writes Games.* Retrieved from: <http://blog.waltdestler.com/2014/04/threesus.html>

[2] Walt Destler (2014) *An A.I. program that knows how to play Threes!* Retrieved from: <https://github.com/waltdestler/Threesus>

[3] Rian Hunter (2014) *A Simple AI for the iPhone Game "Threes".* Retrieved from: <https://github.com/rianhunter/threes-solver>