THE UNIVERSITY OF WESTERN AUSTRALIA

Achieving International Excellence

CITS3001 Project 1 Report

Nicolas La Verghetta 20762905

&

Andrew Edwards 20937641

Structure and design of our algorithm

Our algorithm design ended up being a variation of the A* informed search algorithm. We chose A* as we believed it to be the most appropriate algorithm for the task. A* came across as a highly efficient search algorithm in the lectures for this unit, due to it being both complete and optimal, whilst also maintaining impressive space and time figures.

In order to gain a better understanding of the dynamics of the game Threes, we begun by making our Swing GUI version of the game. Constructing this application provided us with a solid base to begin our project. Ideally we wanted our program to play using the GUI, however as time became a factor we decided not to pursue this.

Our program has two main packages, the GUI, which also handles the all input/output and the AI.

The GUI package contains the actual GUI Jpanel along with all methods involved in moving the board.

The AI package contains the files involved with the game playing. Because we were unable to make the AI play on the GUI we had to re-implement all of the board movements inside this package.

This package also contains our 3 search algorithms. The first search algorithm we wrote was an attempt at A*. Beginning with the root node, we take the nodes 4 possible next moves (children) and add them to a priority queue, for which we wrote our heuristic function as the comparator to sort the items, this orders the nodes by heuristic (H) score, from here we repeat the process until the game ends or we hit a predefined limit. We implemented this first algorithm in a very poor and messy fashion which resulted in a lot of bugs, we decided to re-implement it using a completely new approach later on.

Our second search algorithm was a recursive depth limited call of our first search function, we would call the search function and generate the next ‘n’ moves, after this we would take the first move made, and re generate everything again. Although this was sound in theory we were unable to make this function correctly.

Our final search was similar to our first however we focussed more on making the execution simple, this involved creating types for our board and nodes which now included various helper methods that simplify the search greatly.

We tested many heuristics in our algorithm but we did not have much luck finding any high performing heuristics. We constructed our heuristics through the guidance of the lecture notes on the matter, while also searching for ideas on the internet and came up with the following:

1. **Whitespace**: The amount of White space heuristic was the number of empty tiles a board has. This works well in theory but cannot be used alone. As it will preference boards with more whitespace over those with a higher scoreIt also had to be modified so that it still has an effect as the score and tiles become larger.
2. **Monotonocity**: We attempted to give higher scores to boards whose tiles were increasing from right to left and from bottom to top. Although this worked relatively well it is prone to losing games early.
3. **Smoothness**: This heuristic rewards boards that have a large number of combinable tiles adjacent to each other, like the others this does not perform well on its own as it will avoid combining tiles in order to receive a higher score.
4. **Minimising the number of 1's and 2s:** This was a simple heuristic we added that attempted to minimise the number of 1's and 2's on the board. 1 and 2 tiles only generate a score of 1 so combining these into 3s will result in a much better score. The heuristic score gets worse the more 1's and 2's there are on the board.
5. **Distance between biggest tiles:** In this heuristic we found the distance between the 2 biggest tiles on the board and gave a high score if they are close, or low if they are apart. However our distance calculation was not sound and thus resulted in a fairly counter productive heuristic
6. **Number of common tiles:** This heuristic returns the number of adjacent tiles that are greater than 2 (that is not a blank, a one or a two) and are equal, it is very similar to the smoothness heuristic
7. **Heuristic weight:** We made this function so that our other heuristics can continue to be relevant even as the game score increases.
8. **Count score:** We also tested our game using the current score as the only heuristic, although this will often produce a good score it is very 'short sighted'.

Interesting implementation details

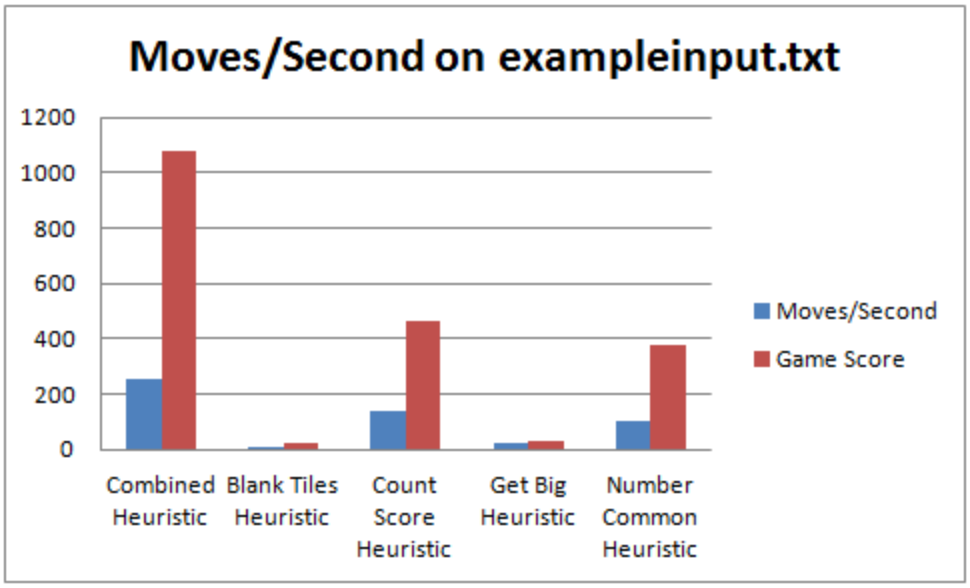
As previously mentioned our program contains a functioning clone of the Threes game, which can be played by invoking the Threes.jar file. The game was with Java's Swing library.

In our implementation of A* we chose to use a priority Queue for our open and closed lists, pairing this with our own comparator function we were easily able to choose the next best node from our open list.

Experimentation and theoretical analysis

We ran a series of trials of each of our heuristics on various input files collected from the project webpage along with some created by other students that were posted in the help3001 thread 'sample game high score thread' and ended up with this data:

The combined heuristic was a combination of monotonicity, smoothness the whitespace score along with the heuristic weight function.



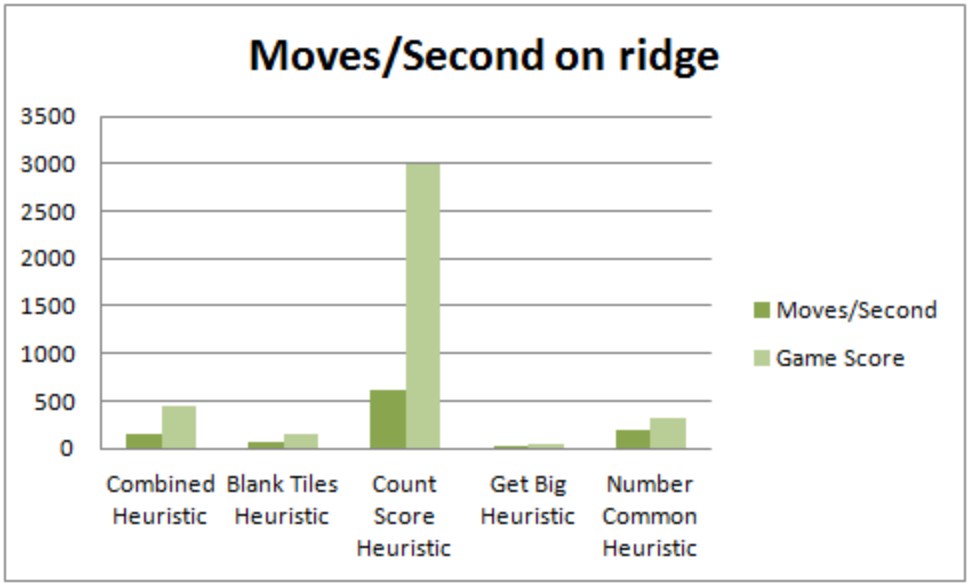
In order to maintain a common sample environment we capped the number of expanded nodes at 5000.

As we can see our combined heuristic provided the best performance of both the game score of 1080 and moves/second of 251.

Our blank tiles heuristic and the get big heuristic (the distance between the 2 biggest tiles) were the worst performers finishing the game very prematurely at 24 and 32 respectively and providing poor moves/second times of 10.2 and 25.1 respectively. For the blank tiles heuristic, this can be explained by any future move providing a lower number of blank tiles than the current best, thus causing the program to stop moving.

The Number Common heuristic along with the Count Score heuristic performed the best as individual heuristics. They provided scores of 464 and moves/second of 138 for Count Score and 380 and moves/second of 104.4 for Number Common.

Running our heuristics on another input file called ‘ridge’, we get the following graph:



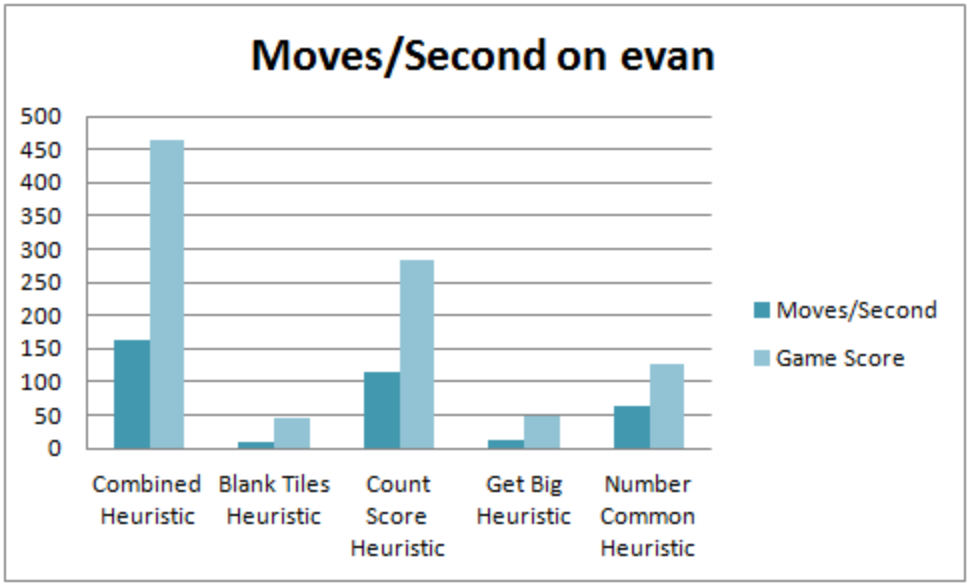
The file ridge contains predominantly small tiles.

Again, each trial was limited to 5000 nodes.

Here we can see that Count Score and Number Common are again the best of the individual heuristics. In this case, Count Score managed to outperform our combined heuristic function by a fair margin (score of 2986 vs. 453 and moves/second of 614.9 vs 148.43).
This was due to our weighting function not being dynamic enough to handle this sequence. This meant that our combined heuristic score was poor and counter productive.

Get Big and Blank Tiles heuristics again performed poorly.

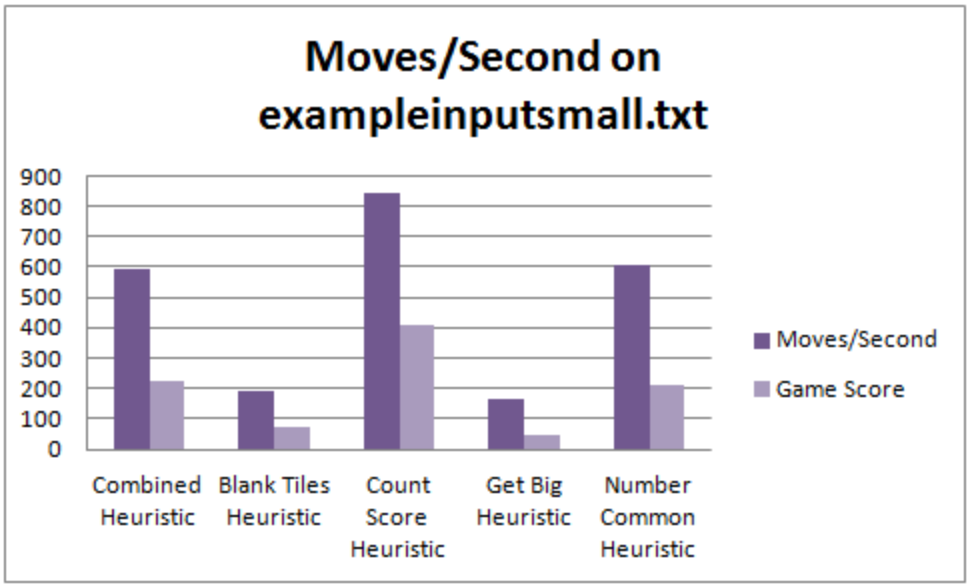
We then ran our heuristics on another input file called ‘evan’.



Again each trial was limited to expanding 5000 nodes.

Here, the combined heuristic managed to outperform all individual heuristics. Again, the same pattern emerges of Count Score and Number Common being the best individual heuristics while Blank Tiles and Get Big performed poorly.

Finally, we ran our heuristics on an input file called ‘exampleinputsmall.txt’.



This had trials expanding 129 nodes. (this was the limit of new tiles)

This chart shows another previously seen pattern when examining the effect of heuristics on ‘ridge’. The Count Score and Number Common both are on par or better than the combined heuristic.

Conclusions of Experiments and Project

We determined that our heuristic function was what let us down most in this project. We had great difficulty extending our pool of heuristics any further than what we submitted, and so as a result, our results are quite underwhelming.

In addition to the low quality heuristics we used, our weightings for each of them was also incorrect, and it was common for us to receive exceptionally high scores for some inputs while receiving exceedingly low scores for another.