# CITS3001 Project 1 Report

## Josh La Verghetta & Andrew Edwards

### Due 30/05/2014 4pm

Our program makes use of the common A-star informed search algorithm with a few changes, and multiple heuristic functions to determine the best move to make. We started building our program by designing a playable GUI version of the “Threes” game laying the groundwork for building a search algorithm on an already working implementation of the games rules.

## Structure and design of our algorithm

We decided to take the idea of the open and closed lists and a tree of nodes used in the A-star algorithm and give it a twist by, instead of a tree, using a priority queue holding a series of “nodes”. This allowed us to choose the best next node according to our heuristics by simply removing the head of the open priority queue, since it is already sorted for us.

* Way too tired for words so im doing dot points
* We ended up using A\* except there is no goal state because there is no goal
* We started by making a Swing GUI of the program, this helpd us understand
* 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 i/o and the AI
* The Gui package contains the gui all the methods involved in moving the board etc
* The AI package contains the files involved with the game playing. Because we were unable to get the gui running with the AI we had to re-implement all of the board movements inside this package.
* This package also contains our 2 search algorithms, the first was a very basic uniform search(I think you can call it that) beginning with the root note we take the nodes 4 possible next moves (children) and add them to a priority queue, we also wrote a custom Pqueue that orders the nodes by H score, from here we repeat the process until the game ends.
* We implemented this first in a very messy fashion (I don’t even know) which resulted in a lot of bugs so we re-implemented it again.
* Our second search algorithm was a recursice 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 idea works 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 with any

* Whitespace
* Works well in theory but cannot be used alone, it also has to be modified so that it still has an effect as the score and tiles become larger
* increasing order
* we gave a higher score to boards whos tiles where increasing from right to left or from bottom to top, although this worked relatively well it is prone to losing games earl
* diff between adjacent tiles
* we tried to make the difference between 2 adjacent tiles as small as possible so that there is a greater chance of combination
* minimize the about of 1’s and 2’s this was a fairly simple one that deducts from the hscore if there is a lot of 1’s and 2’s

## Interesting implementation details

Using a priority queue instead of a tree enabled us to have on the go sorting along with an easy way to choose the next best “node” to expand by simply removing the head of the queue.

* We use both now.. sorta

## Experimentation and theoretical analysis