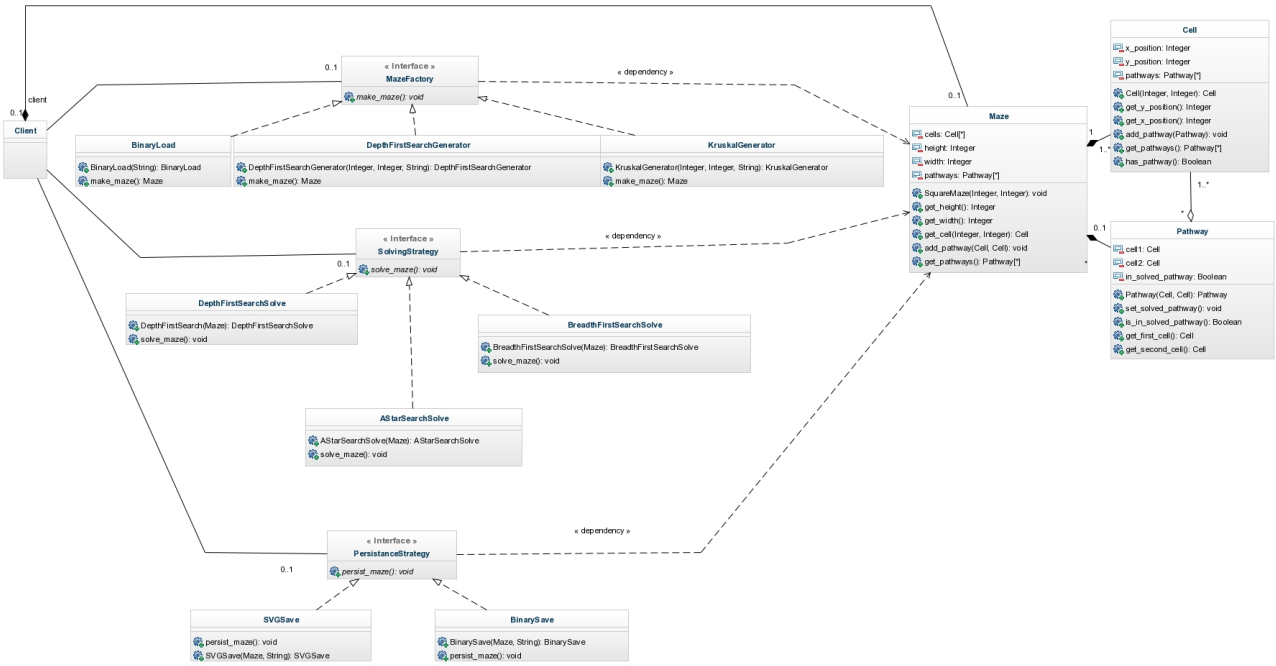
Maze generator report

Programming in C++ RMIT semester 2 2014

Design Changes



Original design (See attached design.jpg for larger picture)

For the most part, the project design has suffered little re-factorisation. The factory/strategy interfaces have been changed into abstract classes and some functionality of the solving strategies have been conformed into the parent. The client is also lacking its functionality in the initial design.

Benchmarking

Besides creating, saving and solving mazes, the program outputs the time taken for each function. By using this information, we can compare the performance of different algorithms. To get a reasonable approximation, multiple tests can be run for each.
Included is a bash script (benchmark.sh) that allows you to automate tests. I did not provide a quick way to output clean data, but compilation is easily done.
I ran the set of tests on RMIT's core teaching servers and received the data in benchmark_results.txt. Afterward I cleaned up this data by using a spreadsheet and with this I was able to easily calculate the averages and the standard deviations of tests (benchmark.ods). All the tests were for a maze 2000 x 2000 with seed 0.

Josh Nithsdale (s3346764@student.rmit.edu.au)

Generation strategies

Two different algorithms have been implemented: Depth first search with recursive backtracking and Kruskal. Please note that the depth first search may be referenced as recursive.

The data indicated that for this maze the average time to generate in recursive was less than half of the average time for the Kruskal generation. There are a number reasons why this happens; firstly the disjoint set initialises a subset for each cell. Secondly, the Kruskal strategy creates a vector of all possible pathways (~ number of cells * 4); a lot of which wont be used and then finally it shuffles this large vector which adds even more overhead.

We can also see that saving a maze as an SVG can be almost 30 times slower than saving as a binary on average. A reasonable explanation for this is that an SVG has a tremendous amount of overhead (shapes, colour, size, ascii etc) as compared to a binary only persisting the core data structures for the cells and pathways.

Solving strategies

I had implemented three different solving strategies: Depth first search, breadth first search and A* search.

We can see that for both types of generated mazes, the depth first search solve is the best strategy. Breath first search is slower because all cells are considered before we find a pathway (the exit cell is always the last node to be queued). A* is built around a heuristic that a straight diagonal path towards the exit node is the most appropriate path; but since building a maze like this would not be challenging and is thus not build with our generators; this assumption seems underdeveloped. Furthermore the A* contains more overhead due to the repeating calculations of known pathway statistics and heuristic estimates.
One interesting thing to note however is that the A* strategy had a considerable performance boost when executed on a Kruskal maze. The reason for this is probably the decrease in maze difficulty where the heuristic can have a better impact.

Initially I expected the A* search to be of best performance since it uses a heuristic, but was surprised when I started conducting experiments.

My implementation of the A* search looks for the best path (shortest) in a maze, but both maze types that are generated can have only one path. This makes this part of the algorithm redundant and its removal would definitely improve the performance.
The implemented kruskal factory attempts to look for disjoint sets of cells for all possible pathways, this can be improved by stopping when there are no disjoint sets remaining.

Josh Nithsdale (s3346764@student.rmit.edu.au)