PERFORMANCE ANALYSIS OF 'MAZER'

Samuel Dowling

School of Science
RMIT University, Australia

1 INTRODUCTION

This report presents the analysis of program entitled *'mazer'*, designed to generate, save and solve mazes based on user command line input. Generation of mazes takes place using either the Aldous Broder or Ellers Algorithm, which may then be saved to a binary or scalable vector graphic (SVG) file. These mazes can then be solved using the users choice of algorithms including breadth first search, depth first search and Dijkstra's algorithm, using either manhattan or euclidean distances as cost heuristics. This report will present a brief analysis of the development of this program, and then analyse the programs performance in the desired functionality.

2 DEVELOPMENT

In order to effectively analyse the structural changes made in the development of this application, a class diagram of the perceived structure was created at the beginning of the project. A class diagram was also created at the end of the project, and these are shown in Appendices A and B respectively.
As can be seen, the initial structure had the solving algorithms implemented as functions within the `Maze` class, and both the Aldous-Broder and Eller generation algorithms implemented within the `Generator` class. Whilst this is certainly a feasible way of implementing a solution, it included a large amount of code duplication, was not modular, and did not have a good implementation of object oriented principles such as polymorphism, abstraction, inheritance or encapsulation. This is mostly due to the fact that when development on this project began, the authors understanding of how to implement polymorphism or abstraction specifically, as it relates to the implementation of design patterns such as factory, prototype or strategy was relatively limited.
The final solution implements a number of patterns which were not foreseen as necessary, including both strategy and prototype design patterns, as well as abstraction of the classes `Generator` and `MazeSolver` to allow for better encapsulation of the subclasses `Aldous` and `Ellers`, and `Dijkstra`, `BFS` and `DFS` respectively. This abstraction and polymorphism allows for the implementation of a prototype in the form of the `GetBuilder` and `GetSolver` classes, and the implementation of a strategy in the main method, which creates only the relevant object necessary for generation and/or solving of a maze, as determined by command line input arguments.

3 PERFORMANCE

In analysing the performance of this program, each of the task completion times will be reviewed, including time taken to generate a maze, solve a maze and save the maze to a file.

Each maze generator was tested for each solving method over a range of dimensions: 10×10, 100×100, 250×250, 500×500, 750×750 and 1000×1000. The average times and standard deviations for these were taken, and are tabulated in Appendix C.
Figure 1 below shows the performance of the generation of a maze using both Aldous Broder and Ellers algorithms. Figure 2 shows the performance of save functionality in the program.

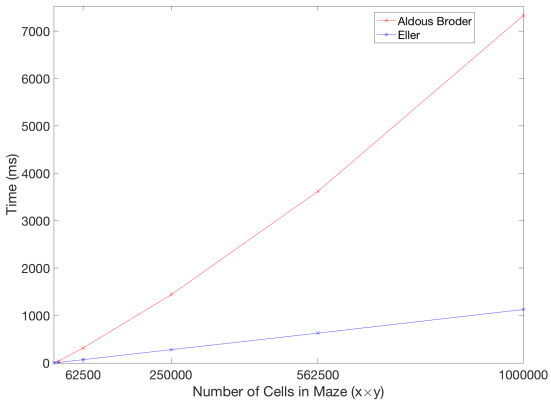

Figure 1: Maze generation performance using each algorithm

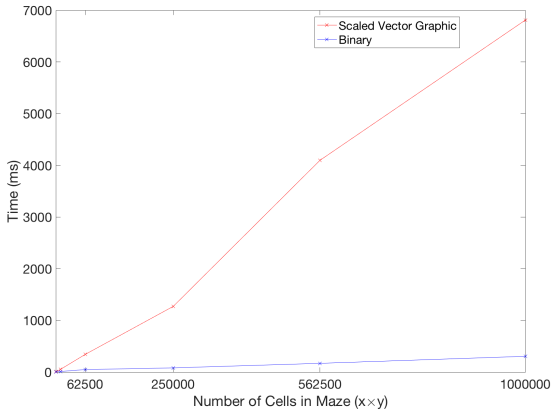

Figure 2: Maze save performance for binary and SVG file types

As can be seen in Figure 1, there appears to be a linear relationship between the number of cells in a maze and the time it takes to generate the maze. This is in keeping with expectation, as an increase in cells does not increase the relative workload significantly. It is also worth observing here that the generation time of a maze using Ellers algorithm is significantly faster than that of Aldous Broder. This is likely a property of the algorithm itself, as it ensures that all edges have been determined for each row before moving on to the next row. This is not the case with Aldous Broder due to its random behaviour, which may cause significant delays in progressing the state of the maze.
As depicted in Figure 2, saving a SVG is significantly more demanding than saving a binary file. This is due to the amount of information being written in each of these cases; the binary file contains the locations of each of the edges in binary form whereas, whilst this information is also contained in the SVG, also has to write the extraneous information required to make the SVG functional.

Figure 7 below shows the performance of each of the solving algorithms implemented.

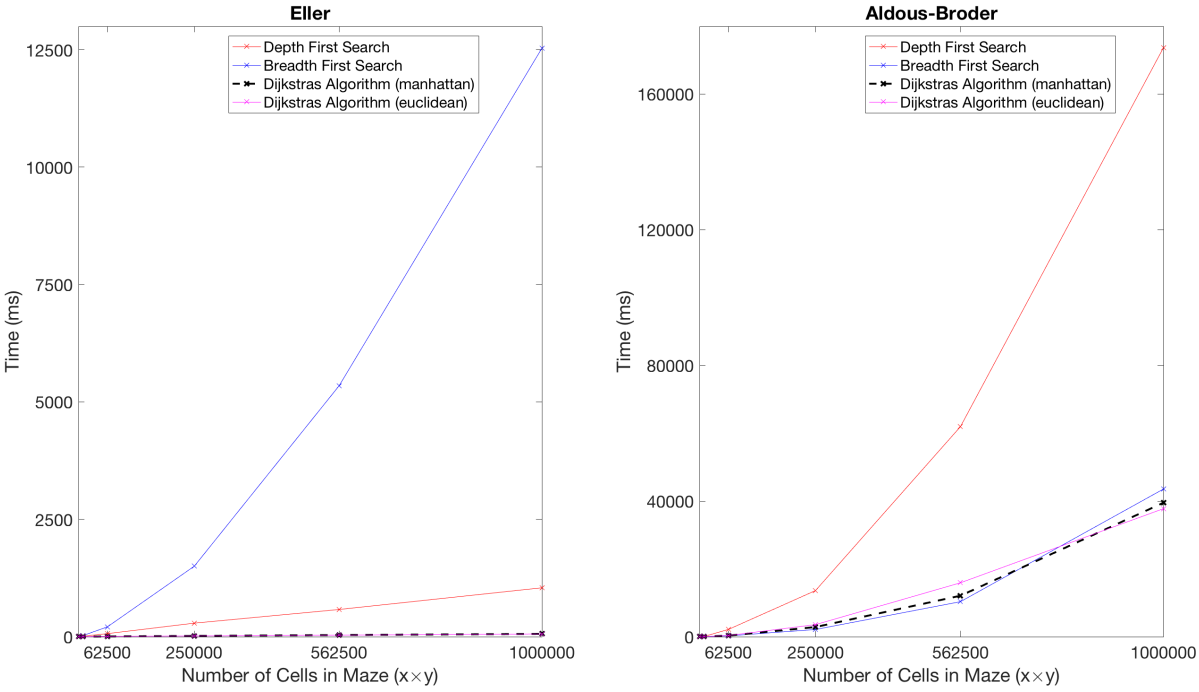

Figure 3: Maze solving performance using each algorithm

This shows that there is an exponential trend for all of the solving algorithms utilised. This is the case to a far lesser extent for Dijkstra's algorithm than it is for Depth First Search or Breadth First Search, however is not unexpected. As the number of cells in the maze increases, the complexity of the solution path also increases. In the case of depth first search specifically, unless it happens to find the solution path in the first number of attempts, the cost of a failed attempt has a larger impact as the path traversal is much longer. Similarly, Dijkstra's algorithm follows much the same protocol of following a path to it's terminus, however it utilises a heuristic to place a priority on the decisions to follow a branch rather than selecting the first alternative. This gives it a quasi-qualitative analytical behaviour, which has evidently lead to a significant improvement in solve time.

It is interesting to note that Breadth-First search has such a large variation in the maximum time recorded. As can be seen, for the largest maze size using the Aldous-Broder algorithm, Breadth-First search took approximately 40 seconds to complete. This is in contrast to Ellers' algorithm in which it took only 12.5 seconds. This is somewhat ambiguous when comparing the results in Figure 7, as for the case of the Ellers' maze, Breadth First Search takes significantly longer to solve the maze in comparison to the other algorithms. This disparity between the two Breadth First Search results is curious, as both Ellers and Aldous-Broder algorithms are supposed to generate a perfect maze; a maze with only one solution.

The expectation for this would be that these two completion times should be very similar due to the fact that Breadth First Search searches each hierarchical level indiscriminately before searching the cells' neighbours. Further analysis of the maze generation algorithms has shown that the implementation of Aldous-Broder does generate a perfect maze, however Ellers' algorithm does not. This explains the large variance in solution generation times for each algorithm; the path to the goal is discovered much earlier as there are multiple paths, and also explains why

Depth First search is faster than Breadth First Search for the maze dimensions analysed.
In terms of comparing the utility of the three algorithms, it seems that Dijkstra is the most effective at solving the maze due to the fact that it can make a decision based on additional information. Depth First and Breadth First searchers are brute force methods, however they still have their utility. It seems as though for mazes with multiple solution paths, Depth First will be more effective than Breadth First at finding a solution in the shortest amount of time; however, this path will not be the shortest. Due to the way the algorithm analyses each hierarchy indiscriminately before moving down, Breadth First Search is guaranteed to find the shortest path, even if it takes longer to find it.
For mazes with only one solution, however, Breadth First Search is clearly the preference. The length of the path is not a factor as there is only one solution, however as presented by Figure 7, it is much more effective at finding the solution efficiently.
This analysis has thus far shown that the majority of time consumption in executing this program is dominated by the solving algorithms. As discussed previously, Ellers does not produce a perfect maze, and so for realistic insight, solving performance will only be analysed in the context of the Aldous-Broder algorithm. Callgrind output from the valgrind command line tool was analysed to determine what specifically was causing the large solve times. Figure 4 below shows the percentage time spent in particular calls during the programs execution:

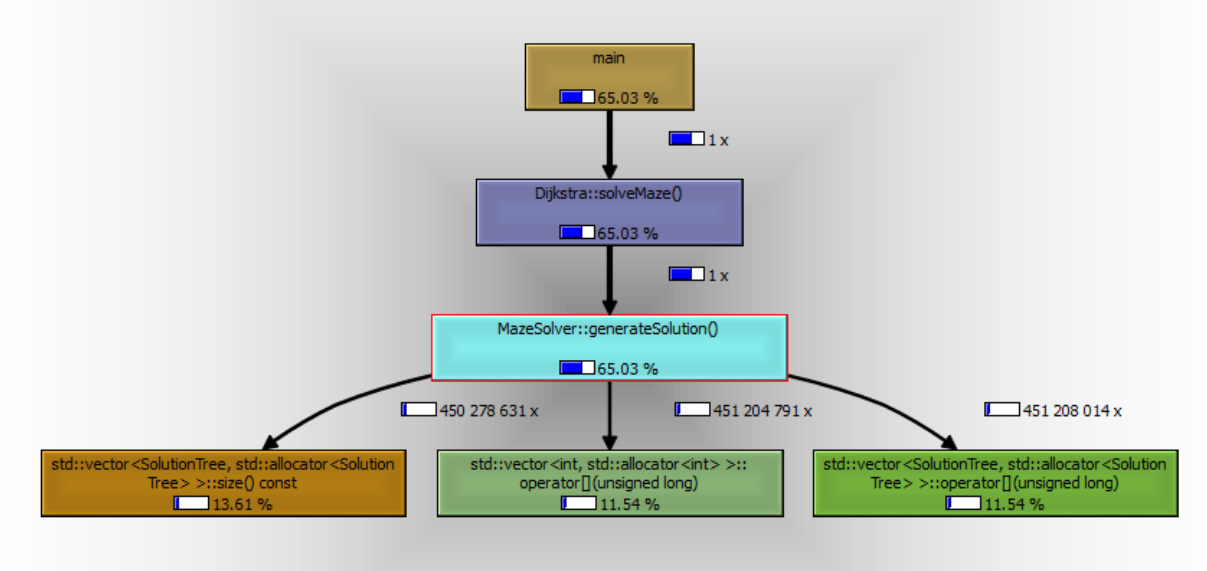

Figure 4: Percentage of time in function calls for an Aldous-Broder maze being solved using Dijkstra's algorithm and a euclidean cost heuristic

As can be seen, 65% of total operation time is spent solving the maze, specifically, generating the solution. This function is utilised by all solving algorithms, and iterates over a vector of visited cells to construct a new vector containing the final solution. It does this by taking the x and y index of the parent cell of the current cell, starting with the goal, and following this path until it reaches the origin. This appears to be highly inefficient, as the visited cells vector must be iterated over for as many cells are in the solution, which in this case has resulted in approximately 450×10^6 for each of the `std::vector` calls shown in Figure 4. The total number of `std::vector` function calls made in the entire program for each of these approximates to 455×10^6, and so it can be seen that this is an excessively large proportion. This highlights

an obvious area for optimisation. The initial implementation of this function involved `struct SolutionTree` incorporating a pointer to the previous cell, however storing this within a `std::vector` data structure proved problematic as it resized and destroyed the location the pointers were pointing to. In hindsight, this implementation may still be feasible if the memory for the `std::vector` holding the cells was pre-allocated. The number of cells explored, for example, cannot exceed the total size of the maze if each cell is only visited once, so this would be an appropriate solution, albeit one that may allocate more memory than is required. In the case of using a Dijkstra solution calculated using a euclidean cost heuristic for a 500×500 maze, a sample indicates that this solution path size may be in the order of 3000 elements. When compared the the roughly 450×10^6 calls to `std::vector` currently made, it can be seen that this would reduce the amount of calls required to generate the solution by approximately 99.9993%. This is a common issue for all solving algorithms, and would improve the effectiveness of the program as a whole.
In addition to this, Depth First Search has other bottlenecks, as shown in Figure 5 below:

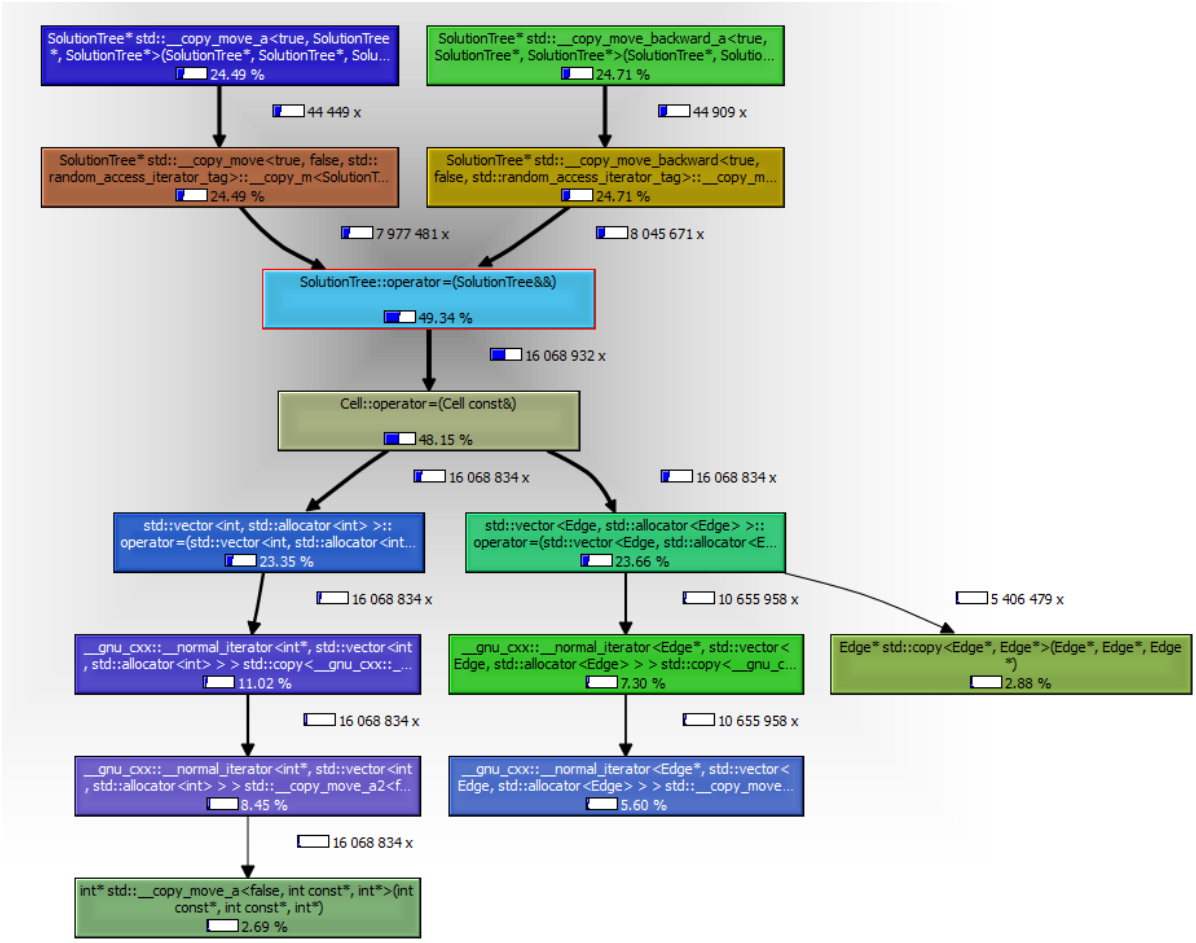

Figure 5: Percentage of time in function calls for an Aldous-Broder maze being solved using Depth First Search

As can be seen, 49% of program operation time is spent moving `SolutionTree` objects, which corresponds to approximately 16×10^6 function calls. This occurs as part of the allocation process to deal with the ”current” cell, as the assignment is *pass-by-value* instead

of *pass-by-reference*, and uses the '=' operator. This would be relatively straight forward to rectify, and could be addressed by creating a constructor to take the reference of the object, allowing `SolutionTree currentNode` to be instantiated with a reference of the desired cell, instead of copying the desired cells value.

4 CONCLUSION AND RECOMMENDATIONS

To conclude, this report has addressed the changes to the application over the development cycle in terms of structure, conducted a performance analysis, and found that the biggest bottleneck affecting the performance of the application in its current state is the `generateSolution()` function in the `MazeSolver` class. This bottle neck is due to the use of `std::vector`, specifically the fact that it traverses every visited cell to find the cell previous to the current, which is highly inefficient. To rectify this, it is the recommendation of this report that a pointer to each previous cell be stored in the current cell `SolutionTree` struct, making adjustments to ensure that the `std::vector` container does not resize and put these pointers out of scope. Additionally, changes should be made to the Depth First Search algorithm to ensure that object allocations are *passed-by-reference* to reduce the amount of time it takes to copy data entirely.

A INITIAL MAZER CLASS DIAGRAM

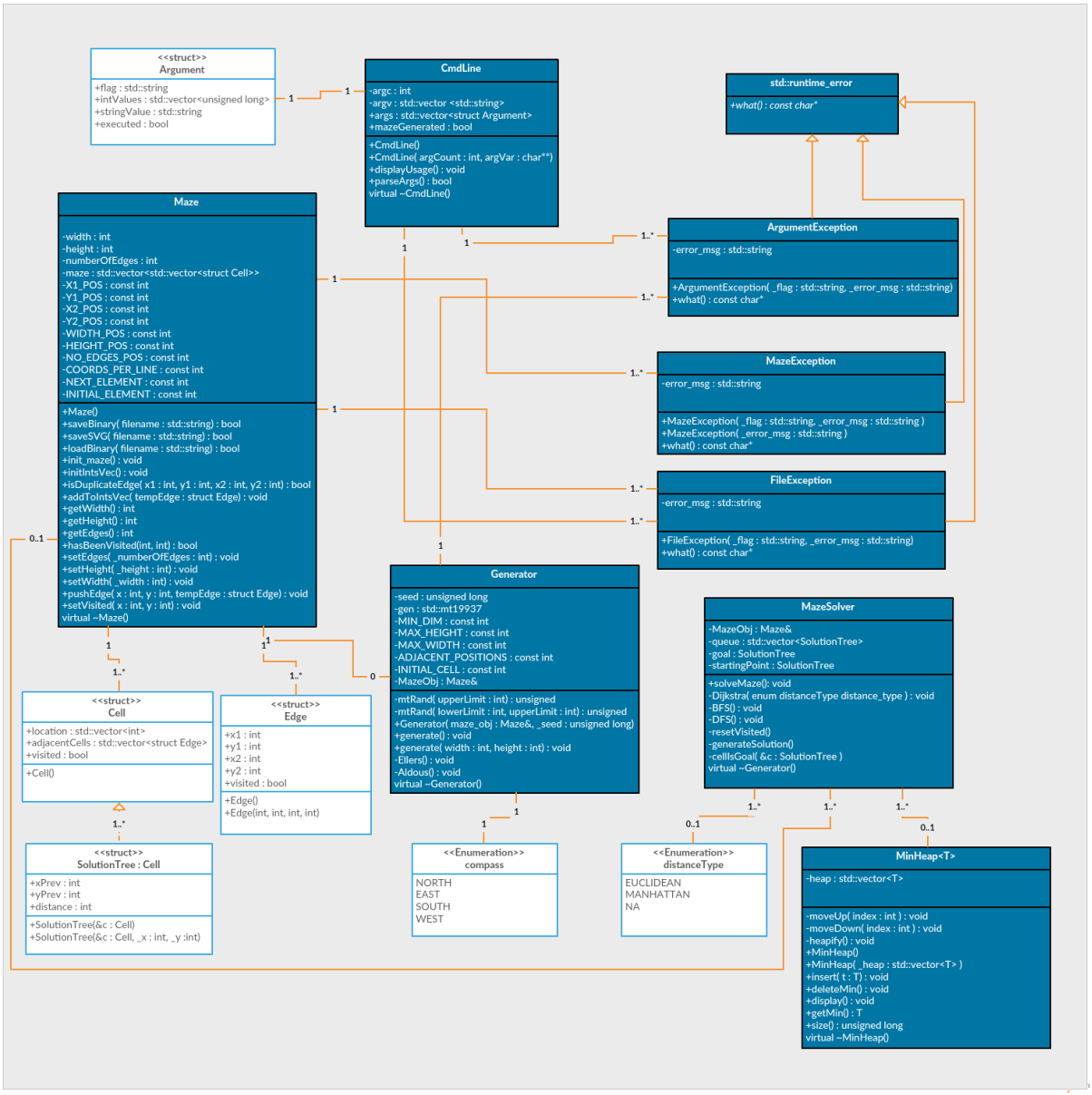

Figure 6: Maze solving performance using each algorithm

B FINAL MAZER CLASS DIAGRAM

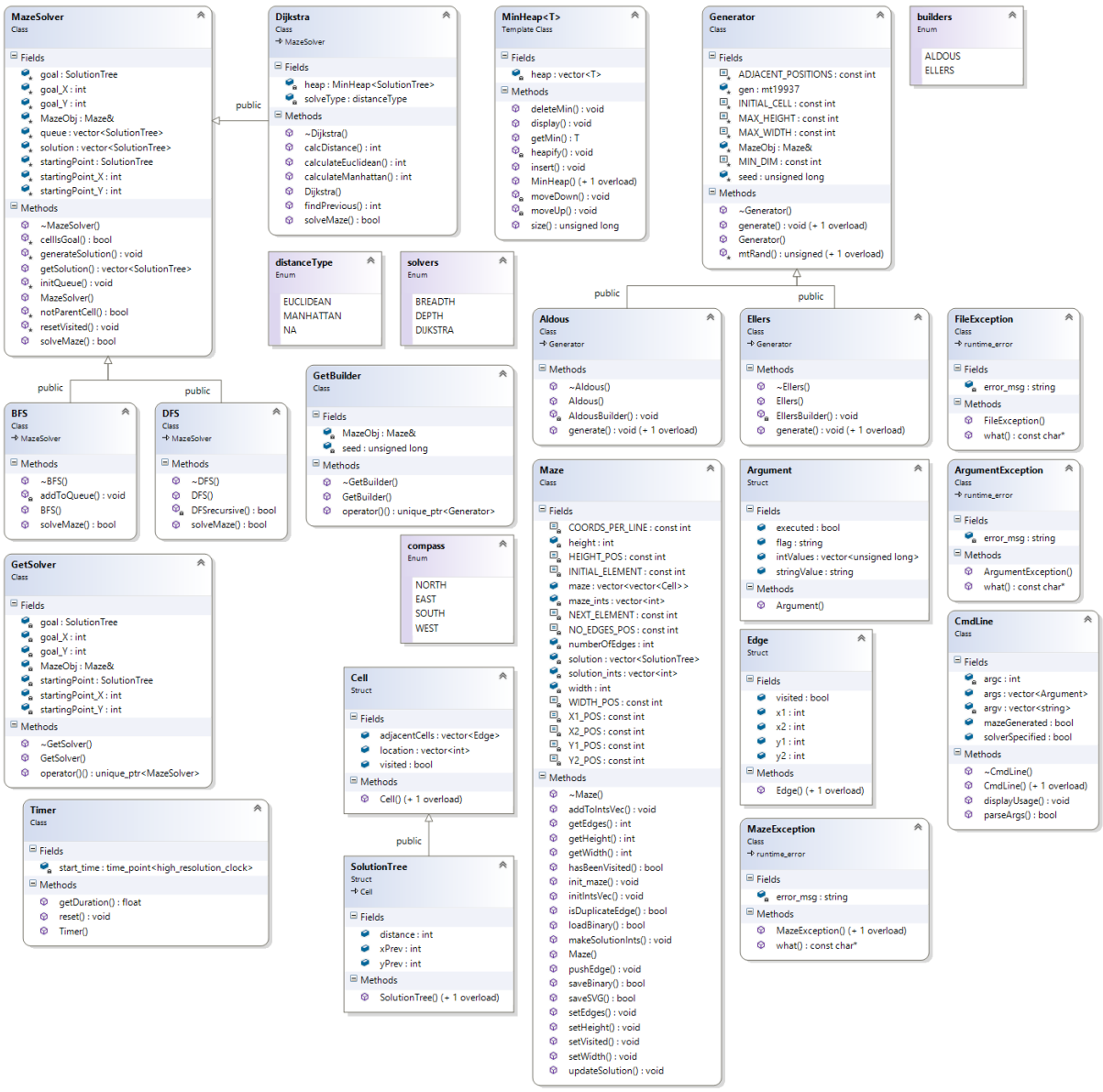

Figure 7: Maze solving performance using each algorithm

C RESULTS

Table 1: Generation and Save time results (ms) and standard deviation (σ)

	Generation Time (ms)				Save Time (ms)			
Maze Size	Aldous	σ	Ellers	σ	Binary	σ	SVG	σ
10×10	0.24	0.04	0.17	0.01	5.72	0.67	6.48	1.46
100×100	38.1	6.26	11.3	0.17	11.91	29.5	53.5	29.6
250×250	317	56.2	69.9	1.11	48.9	110	342	134
500×500	1441	276	281	1.65	81.6	46.1	1271	127
750×750	3623	452	629	0.78	169	55.0	4097	503
1000×1000	7333	1060	1129	21.3	306	225	6810	503

Table 2: Solve time results (ms) and standard deviation (σ)

		10×10	100×100	250×250	500×500	750×750	1000×1000
Ellers	BFS	0.20	25.5	211	1503	5345	12532
	σ	0.02	0.61	3.66	29.2	50.6	240
	DFS	0.19	11.5	65.7	289	580	1042
	σ	0.04	0.92	3.73	51.0	22.4	87.8
	Dijkstra (m)	0.15	1.84	5.84	17.6	35.9	60.5
	σ	0.02	0.13	0.42	1.76	1.67	5.32
	Dijkstra (e)	0.15	1.98	5.81	17.09	33.9	59.0
	σ	0.02	0.48	0.18	1.04	2.49	3.27
Aldous	BFS	0.22	30.5	261	2126	10335	43538
	σ	0.03	4.91	171	725	4865	21999
	DFS	0.35	96.0	2160	13608	61942	173663
	σ	0.05	24.2	1185	6002	21753	80390
	Dijkstra (m)	0.29	34.6	337	2852	12089	39503
	σ	0.15	9.29	212	1287	8281	15130
	Dijkstra (e)	0.24	33.0	364	3561	15904	37755
	σ	0.14	16.0	92.6	2179	6134	21051