ADS II Profiling Report

Explanation of Underlying Algorithms and Data Structures

My first choice of data structure was the hash map to hold the data from *NameList.csv*. I chose hash map because its main advantage is speed, which is the primary goal of the task. Hash map has its speed advantage in this task because of the average time complexity of the operations used (search and insert) is O(1), with the worst case being O(n). This delivers faster performance than a TreeMap, for example, because a TreeMap focuses on memory consumption (keeping entries sorted, unlike a HashMap) at the expense of speed. This means a TreeMap, with its average search time complexity of O(log n), would be less suited than a HashMap for this project.

Text

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

The speed of my HashMap and its operations can be seen in the above evidence, where the addData method uses 0.08% of CPU time and is only sampled twice during runtime. Given that we know AddData is called 100 times (for each entry in *NameList*.csv), this suggests that the method runs quickly enough that it is only sampled twice out of a possible 100, unlike other, less speedy methods like *SocialNetwork.Load().* The profiling evidence tells us that this function does not run for a very long time, again suggesting that *AddData* is a speedy function and a significant improvement on the linked-list approach I had used previously.

My second choice of data structure was the adjacency matrix that underpinned my graph for *ADS2Graph.* I built this adjacency matrix using two-dimensional arrays, and chose it over an adjacency list for one key reason: testing whether two vertices are adjacent to each other is constant in an adjacency matrix (O(1)), and slower in an adjacency list. As this operation is very frequently carried out in my project, particularly for the pathfinding for ‘friends you may know’, I decided to choose adjacency matrix. However, adjacency matrix isn’t the best for every situation: listing neighbours is constant with adjacency list, so depending on the approach taken to obtain all connected nodes it is possible for adjacency lists to deliver similar or better performance. In future, I would pursue a hybrid approach to maximise speed efficiency. However, for this project I decided to solely pursue adjacency matrices for their speed advantages (as I didn’t need the significant space efficiency improvements of the adjacency list).

Text

Description automatically generated

As the profling evidence shows, the *isAdjacent* method is indeed quick using an adjacency matrix, delivering a similar level of performance to AddData. The two operations should have the same time complexity, and the evidence suggests that in my project they do. This would seem to suggest that we have indeed implemented our adjacency matrix correctly, as it delivers the expected performance for this operation. Had I used an adjacency list, I would have expected this operation to be slower, with more samples and higher percentages.

My first choice for the searching algorithm for my ADS2List data was a mixed approach based on a linear search, using my hash functions and open addressing (to avoid collisions) when appropriate. This provided good (O(n)) but not excellent time complexity and could be optimised (detailed in optimisation report). Profiling evidence supports this, showing a sample rate of 1 for an operation only performed once (as opposed to the 100 times *AddData* was used). Extrapolating this data suggests that were this operation carried out 100 times instead, we would see 50 samples – comfortably higher than that of *AddData* and indicative of the linear time complexity O(n) that we would expect. As such, although this is significantly less of a hotspot than other areas (such as the path-finding), the crucial nature of search and its repeated use means that this is still a hotspot that could benefit from further optimisation, such as switching to the faster binary search.

A picture containing diagram

Description automatically generated

For my shortest path algorithm, I decided to use a slightly modified version of Dijkstra’s algorithm to calculate the distance between the current node and all non-direct linked nodes. I chose Dijkstra’s for this project after considering the requirements and potential issues, noting that Dijkstra’s known problem with negative weights would not be an issue (as no negative weights are used in the project). Furthermore, Dijkstra’s algorithm in this context has a logarithmic time complexity *O(V+E logV),* where *V* is the number of vertices and *E* is the total number of edges. This means that it will scale well with a larger graph, and also that it will deliver a reasonably fast lookup (although as with all things the time taken will still increase as the graph grows in size). Looking at my code, there are three key methods involved – *isAdjacent, FindShortestPath* and *FindNewCurrent*. Below is the profiling evidence for *FindShortestPath,* the function containing my shortest-path algorithm. The times outputted are in nanoseconds.

A picture containing text

Description automatically generated

A picture containing text

Description automatically generated

As can be seen from the profiling evidence, *FindShortestPath* runs for longer, taking more samples, than the functions I have previously outlined. It should be noted that *FindShortestPath* visits every node, resulting in a lot of use, however it appears clear from the evidence that it does indeed take longer than the other functions. This again tallies with its expected performance (as it has logarithmic time complexity). Other algorithms such as A\* are available (and could give a certain performance improvement by making use of heuristics) but are considerably more difficult to implement. Given the single aim of this project is speed, and that this is a particular hotspot for my developed algorithm, I would in future attempt to do two things in pursuit of execution speed. Firstly, I would make enhancements to the original Dijkstra-based algorithm, verifying that it has a PriorityQueue or priority-queue like structure to ensure that every node is dequeued and processed exactly once for maximum speed (avoiding re-computation). Secondly, I would conduct a feasibility enquiry into the use of A\* without Java collection classes, and, if it was possible, would consider the necessary changes for A\* to work to its maximum speed (including the necessary heuristics). By doing this (if possible), I would deliver better performance by using heuristics – the quality of said heuristics would dictate how much of a performance increase was received.

Finally, the sorting algorithm used to find the top 10 friends in the *GetRecommended* is also a potential hotspot. From the profiling evidence it is clear that this algorithm runs for a long time, with 20 samples – it also takes up a greater percentage of its parent than even *FindShortestPath*. This is because the sorting algorithm used is essentially linear and therefore doesn’t deliver the performance advances of one with a smaller time complexity, while also not scaling particularly well. To improve execution speed a different, faster sorting algorithm should be used – an example is QuickSort, as its logarithmic complexity in the average case would scale better and run faster. The primary pursuit of this task is execution speed, so any increased memory usage as a result would not be an issue.

