Final report for Algorithms and Data Structures

FGFT2023 - Jo Colomban, Malmö

In this document I'll quickly go through the choices I made when implementing the required algorithms and data structures, with a brief section on performance at the end. Please look at the attached data.pdf file to see average tables, graphs and raw data from my tests.

Code walkthrough

The project has been structured in a very simple manner given the nature of the assignment. I tried to keep things clean, but I would have surely taken some different choices in a production environment, like not making Quicksort and Heapsort into classes and rather exposing their functionality through simple sequential code wrapped inside an appropriate namespace, or alternatively i'd have put all sorting algorithms in their own class and make them derive from a pure virtual Sorter class, so they could have been used polymorphically. That seemed excessive for the purposes of this assignment.

I will briefly go through each .h file in the following list:

- **Vertex.h** : simple template class to represent a vertex for my graph implementation. It uses std::vector to keep a track of pointers to the vertex's neighbors. It exposes a method to add edges to the vertex, which just encapsulates a push_back(newNeighbor) on the aforementioned vector. It has a custom equality operator to make comparing vertex
- **Graph.h**: straightforward template class for a graph, stores a pointer to the start vertex and the goal, and exposes a method to add a new vertex, and to get a vertex by ID. Given more time, I would have implemented a custom [] operator for neater abstraction. The graph class also contains the various traversal/pathfinding algorithms I implemented.
- **Quicksort.h**: class that exposes a .sort method, and contains the necessary helper methods for quicksort. I abstracted the pivot picking process in its own method to be able to easily try different ones for optimization purposes.
- **Heapsort.h**: same as above, but for heapsort
- **SimpleSorting**: a simple .h file that contains one function for sorting algorithm, wrapping them in a namespace for neater API access.
- **ArrayUtils**: utility functions for arrays wrapped in a namespace, they allow the user to create a random array given some parameters, to create an already sorted array, and to pretty-print an array given its length.

- **ParsingUtils:** contains all the (admittedly messy) functions necessary to read a graph from file and generate the corresponding instance of my Graph class, using the AssignmentNodes.txt file
- **ProfilingUtils**: contains a single function to pretty-print elapsed time, in a production environment i would have moved a lot of the performance testing here.

Performance

The attached .pdf file shows clearly that all the algorithms performed as expected, scaling according to their theoretical time complexity. The most interesting finding is the famous O(n2) worst case scenario for quicksort. The pivot choice you can see in the code is picking the last element as first pivot (Lomuto partition), which makes a fully sorted array the worst case scenario. I tried to experiment with picking the median of the first-middle-last element of the array (Sedgewick partition), which add some randomness to the pivot picking, severely mitigating the issues of trying to sort an already sorted array with Quicksort.