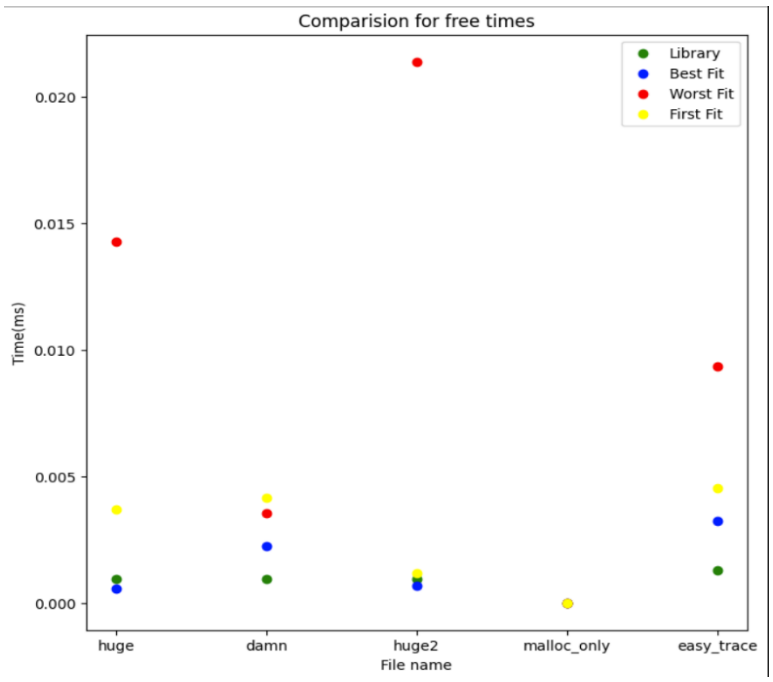
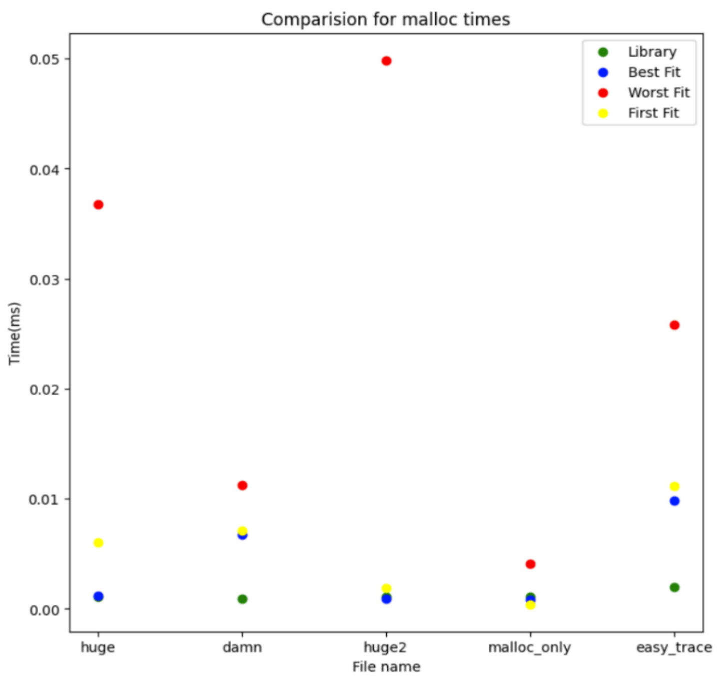
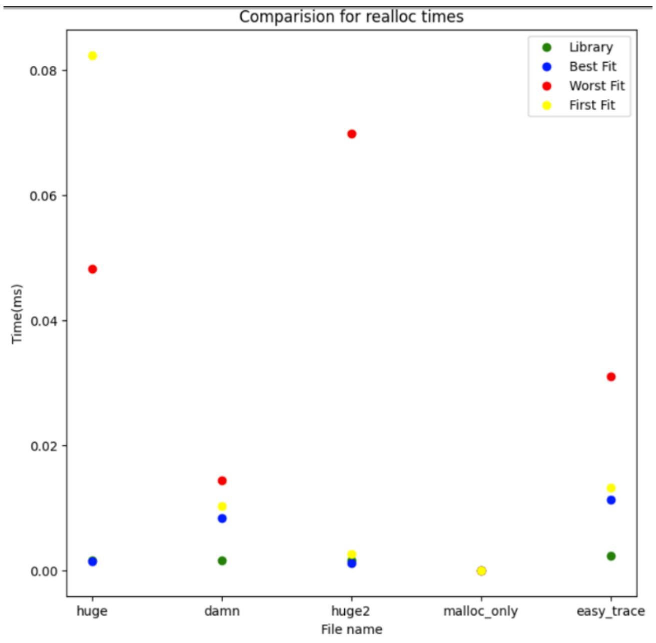




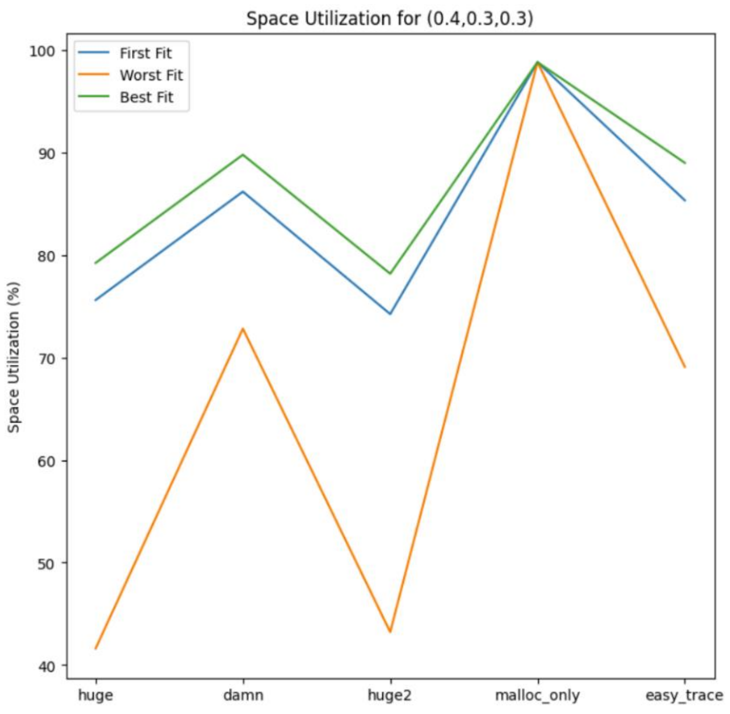


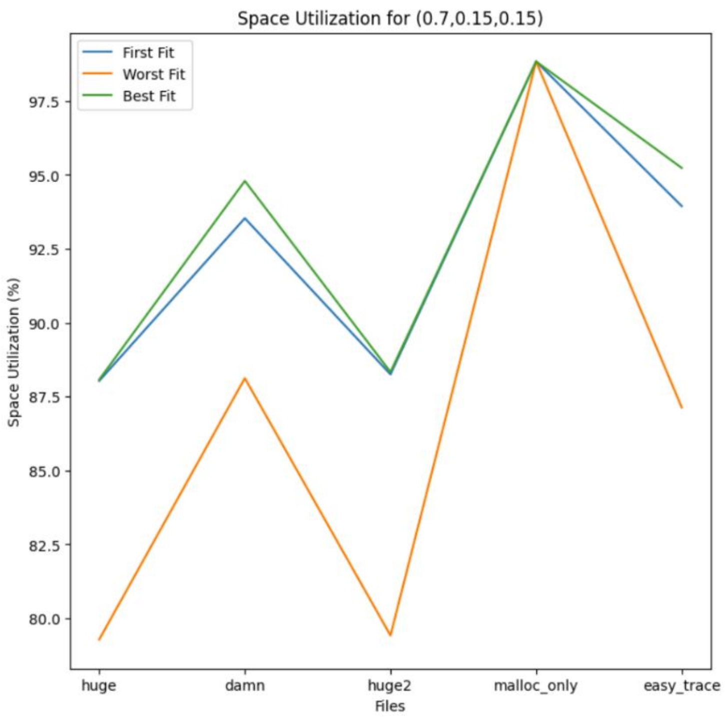
*library is glibc memory allocator

**default test case workloads

The graphs above compare the average malloc, free, and reallocation times of the four algorithms (best fit, worst fit, first fit and the original glibc library). The malloc times are higher for the worst-case algorithm. This is because the algorithm traverses the whole list before finding a free space, and many large free spaces are left after allocation (since this algorithm finds the largest free block), leading to a greater traversal time through pointers. In short, more external fragmentation leads to a greater traversal time as the number of free list nodes increases.

In most cases, the default library algorithm performs better than all other three algorithms. Among the other three algorithms, best fit has a lower time for the operations in the same trace files. This is even though best fit traverses the whole free list before finding the block with the minimum size. Best fit reduces external fragmentation and has a lower free list traversal time.





The trend of the space utilization percentage can be seen above. The trend is almost the same for both workloads. I set the workloads of all trace files (except malloc_only) the same (0.4,0.30.3 and 0.7,0.15,0.15), respectively. The worst fit algorithm performs the worst in space utilization. This is because the algorithm finds the largest free blocks of memory to accommodate a specific allocation size. This results in a larger chunk of the total heap to be left free. Best fit yields the best utilization percentage for both workloads. It finds the smallest free block that can accommodate a specific allocation size, leading to less wastage of memory. First fit performs much better than worst fit, but it is slightly worse than best fit.

Space utilization increases in damn files for both workloads and for all types of algorithms as the file has a greater number of allocations and allocates, frees and reallocates a larger number of bytes compared to other files. The easy file has fewer operations, but the memory size being requested, freed or reallocated is small. Its utilization is almost the same as damn. Huge and Huge2 have a larger number of operations, but they request, free or reallocate smaller memory size, which leads to a lesser utilization percentage. The trend between the number of operations and the requested memory size is now apparent. (A larger number of operations and larger memory request, free or reallocate sizes or a smaller number of operations and smaller memory request, free or reallocate sizes, both yield higher utilization percentage).

Overall, the utilization percentage increases as the number of malloc increases and the number of free and reallocations decreases.

The drawback of my implementation is that it takes longer time for allocation, freeing and reallocation as I initialize a small heap (1024 bytes) and extend it by the same number of bytes. This leads to more extension calls, increasing the time for the first, worst, and best-fit algorithms. If I increase the initialization and heap extension sizes, the time reduces but the space utilization percentage also reduces. There is a tradeoff between time and space utilization.

If performance is interpreted to be a larger space utilization, I would decrease the initial heap size and the size by which heap is extended even more. However, if performance is interpreted to be a reduced time, I would increase both the initial heap size and the size by which it gets extended.