SPP Assignment 1

Author: Rutvij Menavlikar (**2019111032**)

Sequential Matrix Multiplication

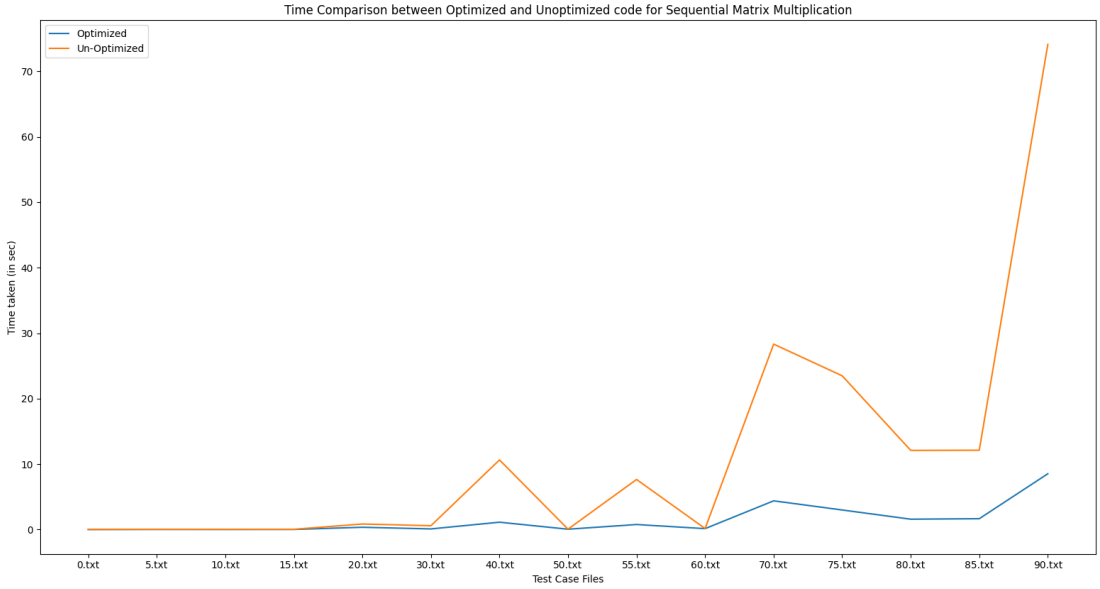
Naive Approach

- The most common approach is to take input of matrices sequentially and multiply the next matrix with the product of the previous matrices.
- And while multiplying matrices of dimensions $n \times m$ and $m \times k$, loop over n, k and m and add product of values, that is loop over every cell of the new matrix and calculate it's value.

My Approach

- I first take input of all matrices.
- Then, by using the dimensions of the matrix, I find the order to multiply the matrices which minimises the number of operations performed, and multiply them using a recursive function.
- While mutiplying two matrices, I use the tiled matrix multiplication algorithm with tile size of 64×64
    - In this algorithm, for a tile size s, we usually access the matrices in step-by-step. And in each step we only consider rows of size less than or equal to s elements.
- I also flipped orders of loop such that, while multiplying matrices of dimensions $n \times m$ and $m \times k$, loop over n, m and k in this order so as to obtain continuous blocks of memory and avoid cache misses.
- Another optimization I performed was to unroll the innermost loop of matrix multiplication. That is, in one iteration of the innermost loop I perform 16 operations and in the loop make jumps of 16 positions.
- Some more general optimizations are:
    - Using `register int` for counters of loops.
    - Dynamic allocation of memory to store matrices and arrays.
    - Avoiding dereferencing pointers inside loops as much as possible.
    - Avoiding calculations inside loops as much as possible.
    - Using pre-increments in `for` loops instead of post-increments.
    - Using `restrict` keyword while passing pointers to functions.

Comparison of Optimized code to Unoptimized code



Floyd Warshall Algorithm

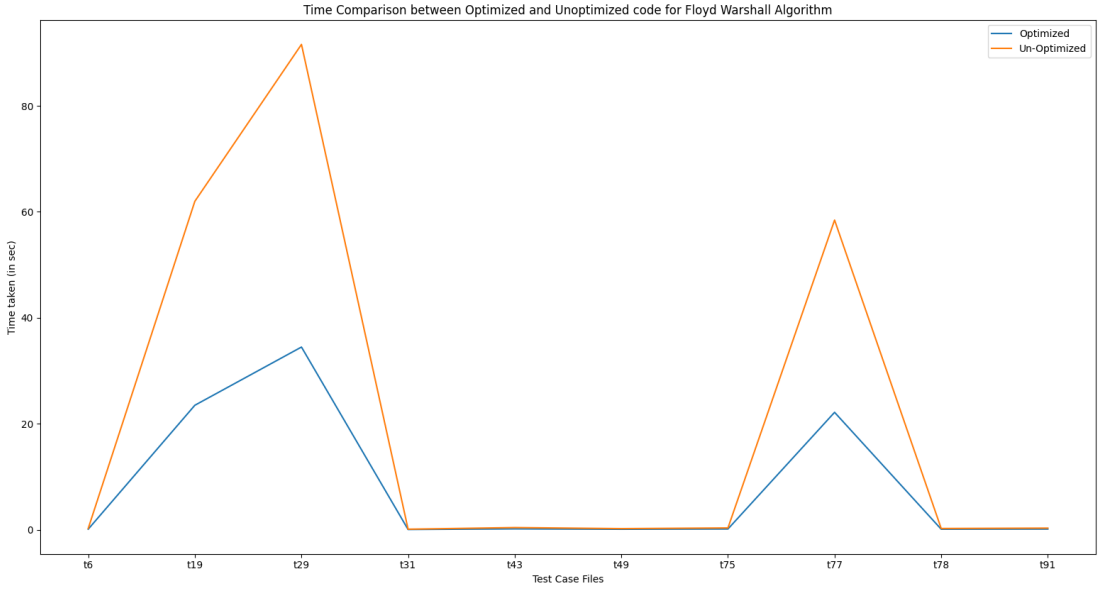
Naive Approach

- The most common approach is to take input of edges, prepare adjacency matrix and iterate over *intermediate nodes*, *beginning nodes* and *destination nodes*.
- This algorithm goes uses the entire $O(v^3)$.

My Approach

- I first take input of all edges and set my adjacency matrix.
  My assumptions/method of creating adjacency matrix is:
  - Initially set the adjacency matrix such that,
    When $i \neq j$, `adjm[i][j]=INF` (INF is a very large value)
    When $i=j$, `adjm[i][j]=0`
  - For an edge taken as input, set the adjacency matrix value to the minimum of the existing value at that value and the weight of the edge given as input.
- Then I run the algorithm regularly, with just one change.
  If the adjacency matrix value of the *beginning node* and the *intermediate node* is `INF` , then I skip the innermost loop iteration for that value of *beginning node* and *intermediate node*.
- Some more general optimizations are:
  - Using `register int` for counters of loops.
  - Dynamic allocation of memory to store matrices.
  - Avoiding dereferencing pointers inside loops as much as possible.
  - Avoiding calculations inside loops as much as possible.
  - Using pre-increments in `for` loops instead of post-increments.
  - Using `restrict` keyword while passing pointers to functions.

Comparison of Optimized code to Unoptimized code


Time Comparison between Optimized and Unoptimized code for Floyd Warshall Algorithm
Optimized
Un-Optimized
Time taken (in sec)
0
20
40
60
80
t6
t19
t29
t31
t43
t49
t75
t77
t78
t91
Test Case Files