Dominant Check Report

Introduction

This report is intended to test various implementations and data structures to enhance the performance of the dominant check in the multi-resource constraint path-finding problem. This problem can be broken down into two stages: firstly, exploring the path on the map, and secondly, comparing each path to identify which one is completely dominant over the other using the dominant check. By doing this, we can reduce the number of paths to explore, and consequently, significantly decrease the run time. One of the primary challenges in solving the problem quickly lies in minimizing the complexity of the dominant check, as this is one of the most substantial factors contributing to the problem-solving time.

# Experiment

Here, I have tested several methods to optimize the complexity of dominant checking: forming sorted arrays, unsorted arrays, and linked lists with a path structure of:

**struct** path {

**double** cost;

**int** load;

**int** time;

**bool** visited[100];

}

## Unsorted Array (Naive) Approach:

The concept behind this approach is to individually compare each path that reaches the destination against all other paths. This process is divided into two main stages. Firstly, we loop through the entire set of paths to determine if any existing path dominates the current path. If it does, the new node will not be inserted into the vector of paths. If not, we proceed to the next step. To optimize the process and avoid unnecessary checks, we split it into two turns. In the second turn, we evaluate whether the newly inserted path dominates any of the current paths. If it does, we store the positions of the dominated paths in a 'removed' vector. After completing the dominant check for the current path, we remove the paths that were marked for removal. This removal process needs to occur after each check to reduce the size of the vector and improve efficiency.

## Sorted Array Approach:

The principle driving this strategy aims to minimize the forward and backward checks of dominance by arranging the path based on a specific constraint. For this experiment, we chose 'time' as the constraint, sorting paths from low to high based on this factor. This sorting allows us to examine if the newly inserted path is dominated by any existing paths with a lesser time constraint.

This process is divided into two stages. The second stage only proceeds when no existing path dominates the newly inserted node. We then inspect if this new node dominates any of the already-inserted nodes. This method enables us to conduct the check for 'n' number of paths at most for any new insertion compared to the '2n' required when the paths are unsorted. This difference significantly reduces the checking process.

However, this implementation does have a downside. The insertion of a new node into a sorted vector can be time-consuming. This process requires 'log(n) + n' time, with 'log(n)' accounting for the binary search to find an appropriate position and 'n' for shifting the vector to create space for insertion. Although we have lessened the number of checks, we inevitably add more time when inserting a new path into the vector.

Linked List Approach:  
This approach operates under the assumption that accessing a randomly ID-addressed location, storing the path information, is just as efficient as accessing data in a vector. With this assumption, the implementation of the linked list aims to improve the naïve implementation by reducing the insertion and deletion time of a newly inserted element in the vector. More precisely, it seeks to decrease the time required for deletion, which involves shifting an element for each dominated path. This adjustment can significantly reduce deletion time, not only because of the minimized element shifting but also because we can't alter the vector while executing a loop.

In the worst-case scenario, where a new element strictly dominates all the existing paths, the total time to delete all elements using the naïve approach could reach O(n^2). This is because we have to delete each element individually, and each deletion requires O(n) execution time. However, with the linked list approach, we can eliminate all elements at once, requiring at most n time for the entire vector, thereby significantly improving the efficiency of the deletion process.

# Result

A graph with different colored bars

Description automatically generated

The findings suggest that the naïve approach still dominates over the other methods, with the linked-list approach performing comparably, albeit slightly less effectively. This indicates that it is more time-efficient to go through each element and check for dominance rather than inserting an element at a sorted position to reduce comparison time. The results also confirm that accessing a random ID is less efficient than looping through the entire vector to locate the desired path.

# Conclusion

The experiment has provided valuable insights into different approaches to the dominant check in the multi-resource constraint path-finding problem. While the naive approach proved most effective overall, other methods also revealed their strengths. Specifically, sorting resources by a given constraint, as in the sorted array approach, significantly reduces the number of checks required, while the node structure used in the linked list approach enables quicker insertions and deletions.

While the experiment didn't lead to a revolutionary shift in dominant check approaches, it has pointed the way to potentially more efficient solutions. By building on these strong points in future research, we can explore new approaches, such as integrating these elements in a tree structure. These efforts will continue to contribute to the ongoing quest for optimal solutions to complex path-finding problems.