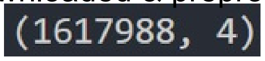
Data Download and Preprocessing

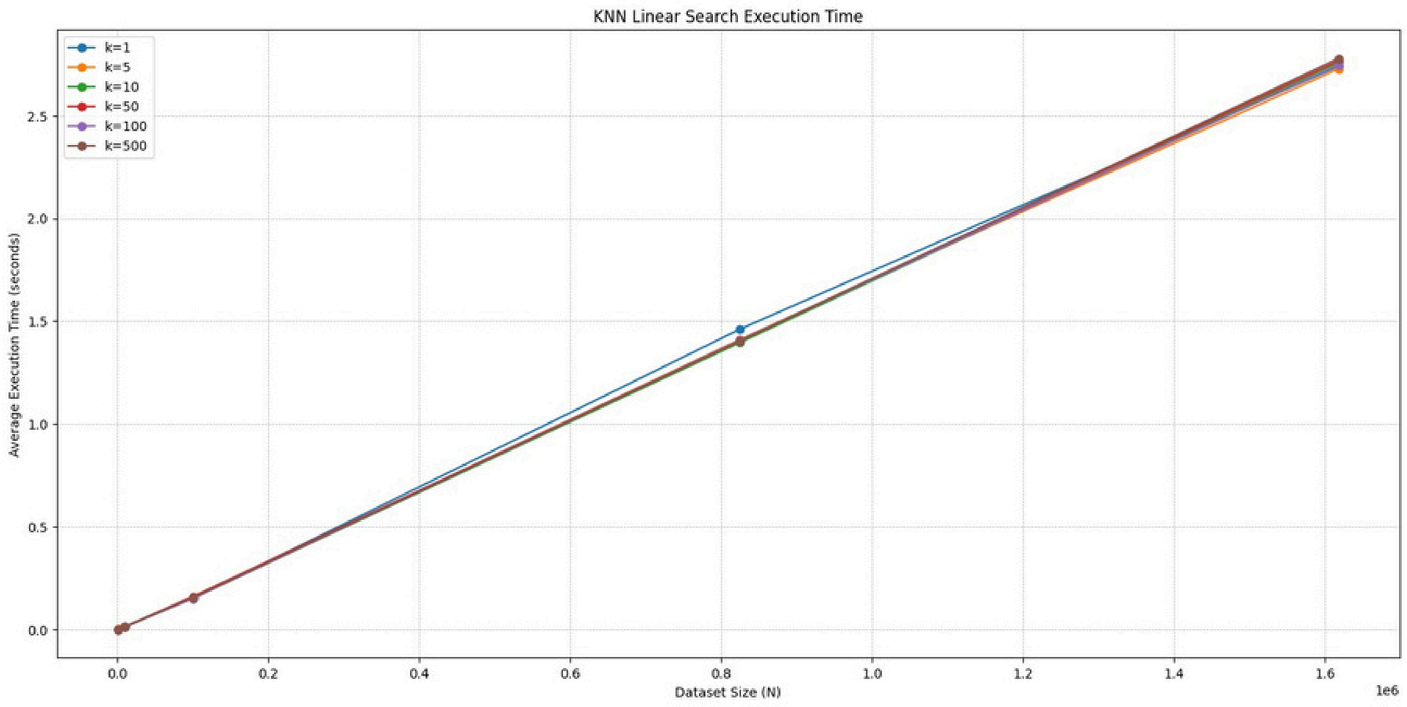
The data was downloaded & preprocessed according to the instructions. The shape of the data after preprocessing is



```
(1617988, 4)
```

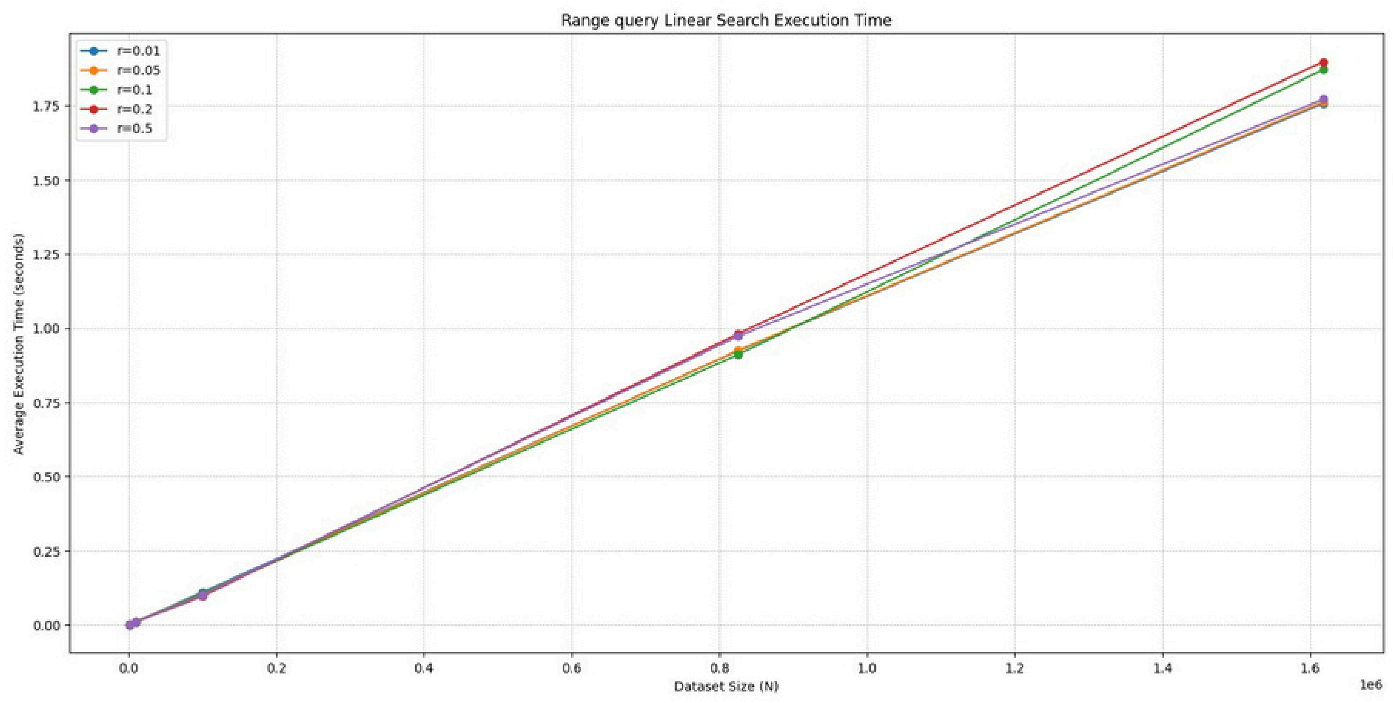
k-Nearest Neighbors (kNN) Query Using Linear Search

KNN using linear search was implemented by computing the Euclidean distance between the target POI and all other POIs. A max-heap of size k was used to efficiently retrieve the k-nearest neighbors. Since the most computationally expensive step is calculating distances for all N points, the value of k has minimal impact on execution time. However, as N increases, the execution time scales linearly, which is expected for an O(N) algorithm. The plot below confirms this trend, showing execution time for varying dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and values of k = [1, 5, 10, 50, 100, 500]. The nearly identical execution times for different k values further support that distance computations dominate the overall runtime.



Range Query Using Linear Search

Range query was implemented by linearly scanning through all POIs, computing the distance between each POI and the target, and selecting those within a radius r. Since every POI is checked regardless of r, the value of r has no significant impact on execution time. Similar to the KNN query, execution time increases linearly as N grows. This trend is evident in the plot below, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and radii R = [0.01, 0.05, 0.1, 0.2, 0.5].

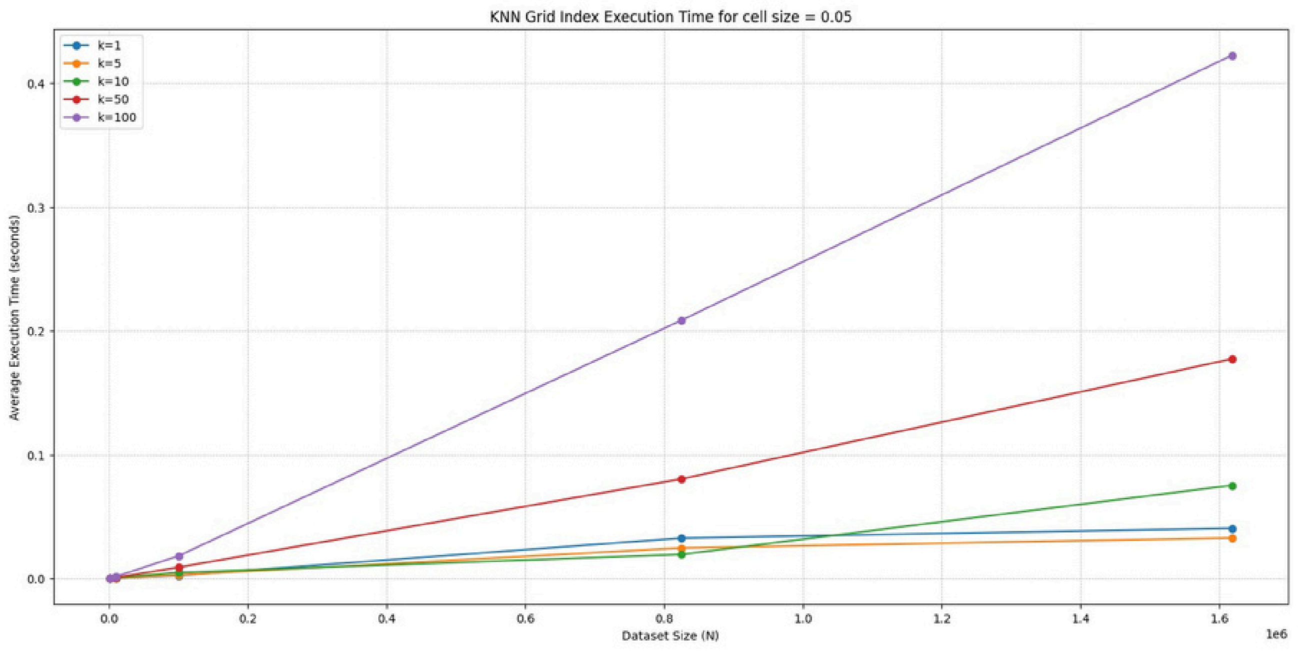


Grid Index

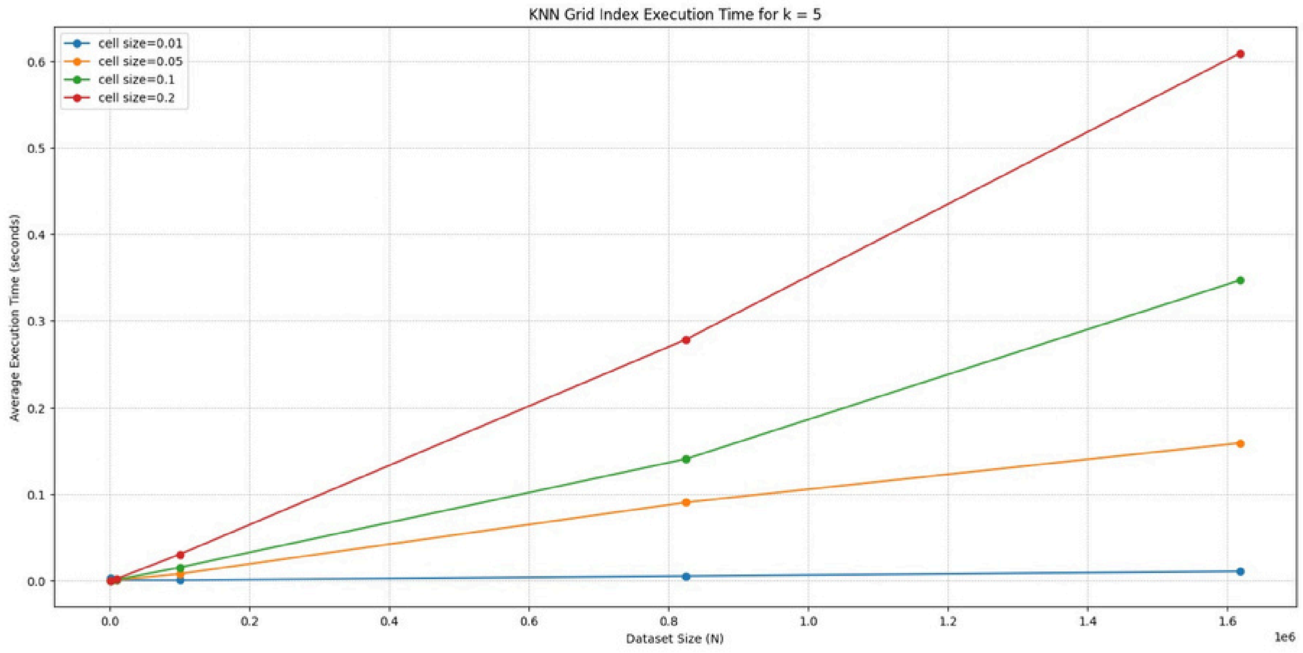
A grid index was constructed by partitioning the search space into uniform cells of size cell_size × cell_size, mapping each POI to its corresponding cell. This process was completed in linear time.

k-Nearest Neighbors (kNN) Query Using Grid Index

The process began by identifying the cell containing the target POI, then expanding outward in concentric squares until enough POIs were gathered to find the k nearest neighbors. A linear search was then performed on this reduced search space. Execution time increased slightly with larger values of k and also showed a slight increase as N grew. This trend is evident in the plot below, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and k values [1, 5, 10, 50, 100, 500].

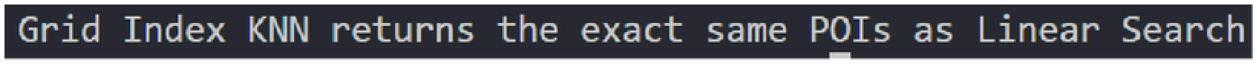


For k = 5, the execution time increases as cell_size increases. This happens because larger cell sizes include more POIs in the subset, increasing the number of distance computations. Additionally, as N increases, execution time also rises since the number of POIs per cell grows, leading to an even larger subset for searching. This trend is evident in the plot below, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988], k = 5, and cell_sizes = [0.01, 0.05, 0.1, 0.2].

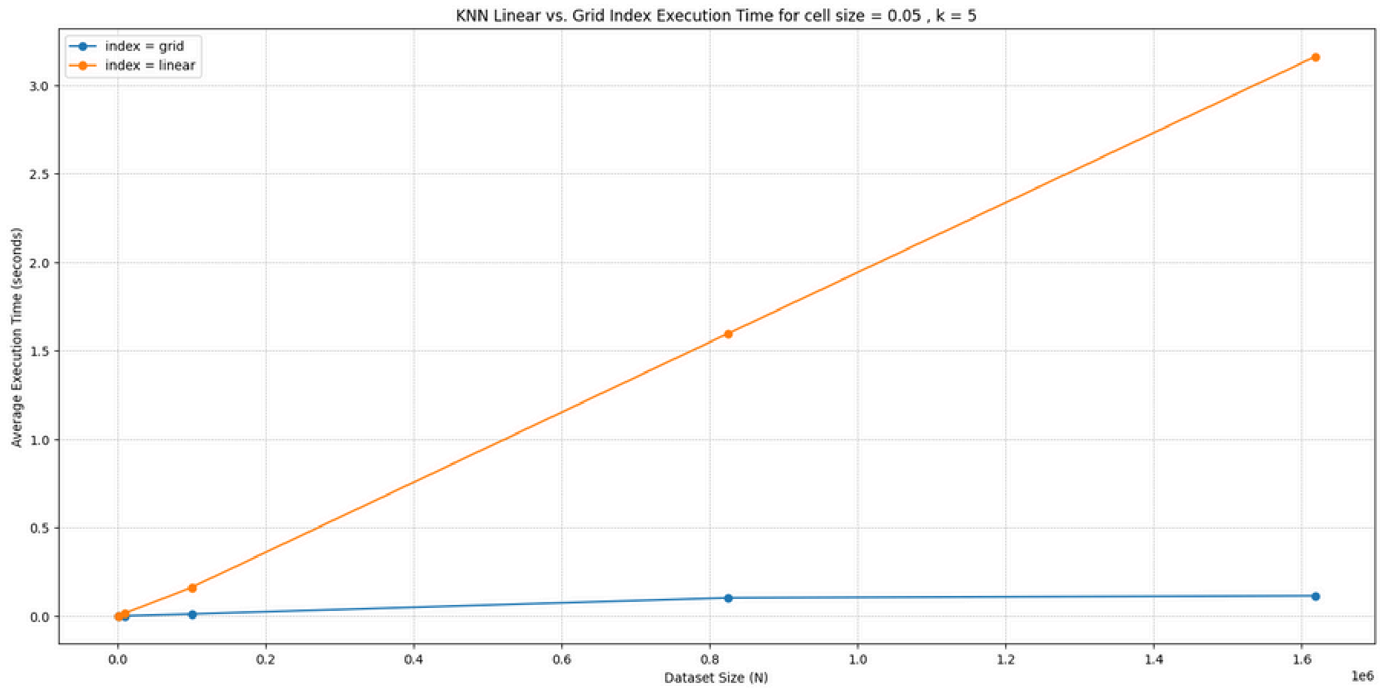


However, for k = 500, smaller cell sizes may result in longer execution times compared to larger cell sizes. This is because smaller cells may not initially contain enough POIs, requiring additional expansions to find k neighbors, whereas larger cells already have more POIs within fewer expansions.

When compared to brute force linear search, the grid index achieves significantly better performance in practice, even though both methods share the same worst-case runtime complexity. The grid index reduces the number of distance computations by limiting the search space, leading to much faster execution times. This improvement is evident in the plot, where execution time is measured for dataset sizes N = [1000, 10000, 100000, 825171, 1617988], k = 5, and cell_size = 0.05. The correctness of KNN using the grid index was verified by comparing the POI IDs and distances from the target obtained through both the grid index and linear search.

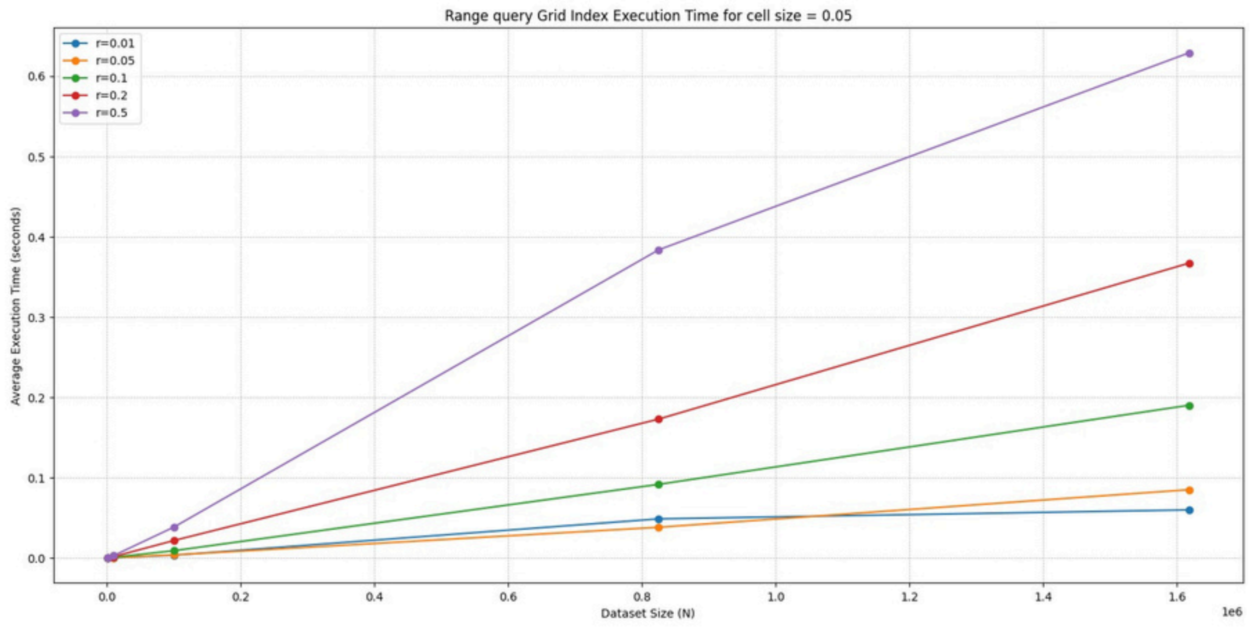


```
Grid Index KNN returns the exact same POIs as Linear Search
```

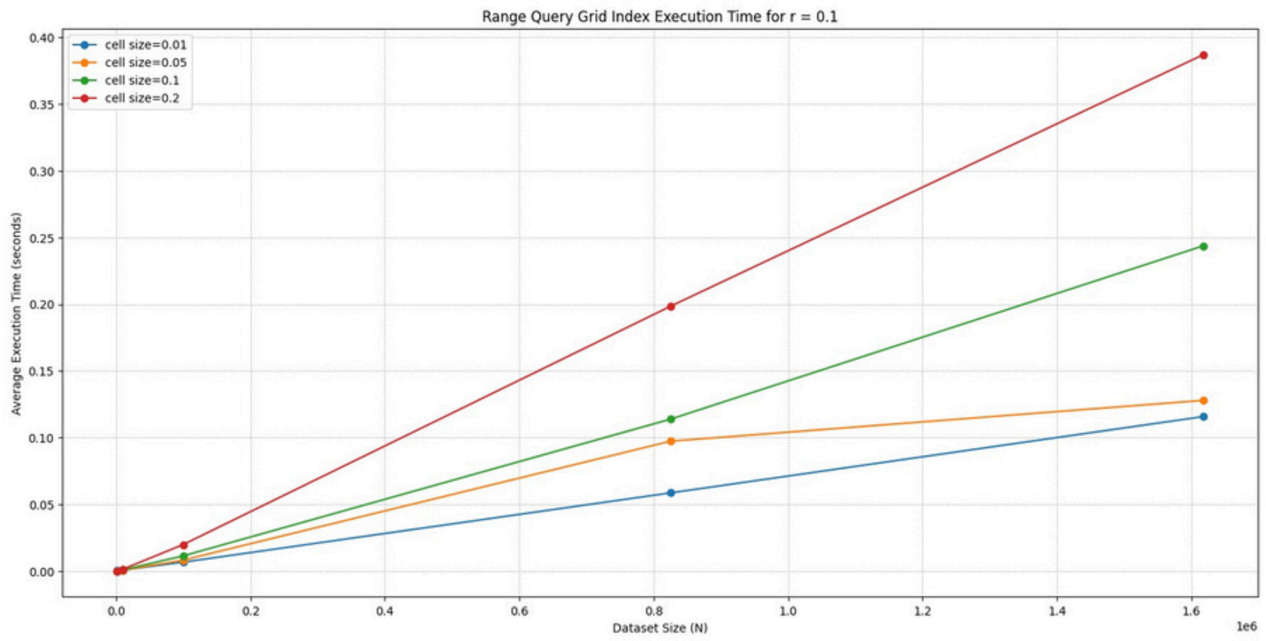


Range Query Using Grid Index

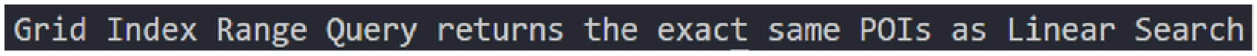
The range query was performed by first identifying the cell containing the target POI and then collecting all POIs within the square bounding the circle defined by the target POI as the center and the given radius. A linear search was then conducted on this subset to determine which POIs fell within the specified radius. As the radius increases, more POIs are included in the subset, leading to an increase in execution time. Additionally, as N increases, the density of POIs in each cell also increases, further expanding the subset and increasing execution time. This trend is evident in the plot below, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and R = [0.01, 0.05, 0.1, 0.2, 0.5].



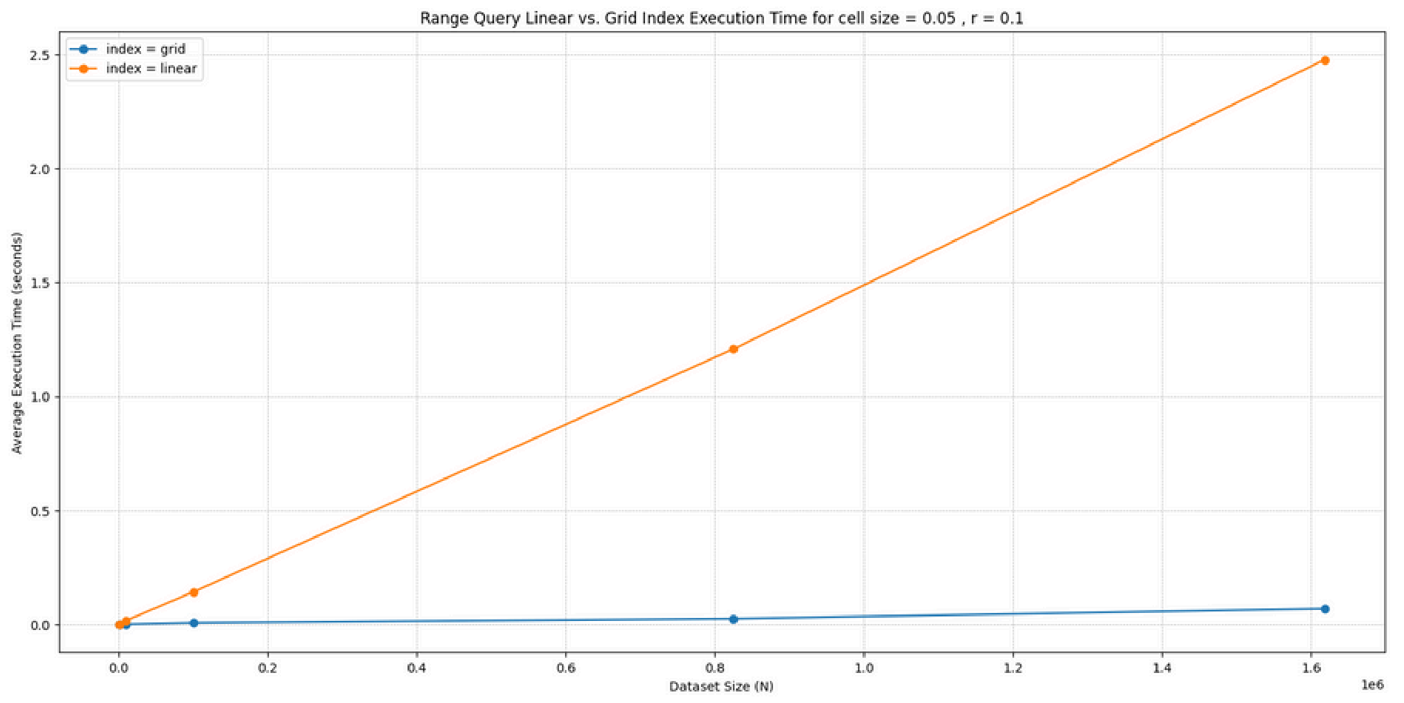
For r = 0.1, the execution time increases as cell_size increases. This happens because larger cell sizes include more POIs in the subset, increasing the number of distance computations. Additionally, as N increases, execution time also rises since the number of POIs per cell grows, leading to an even larger subset for searching. This trend is evident in the plot below, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988], r = 0.1 and cell_sizes = [0.01, 0.05, 0.1, 0.2].



When compared to brute force linear search, the grid index achieves significantly better performance in practice, even though both methods share the same worst-case runtime complexity. The grid index reduces the number of distance computations by limiting the search space, leading to much faster execution times. This improvement is evident in the plot, where execution time is measured for dataset sizes N = [1000, 10000, 100000, 825171, 1617988], r = 0.1, and cell_size = 0.05. The correctness of Range Query using the grid index was verified by comparing the POI IDs and distances from the target obtained through both the grid index and linear search.

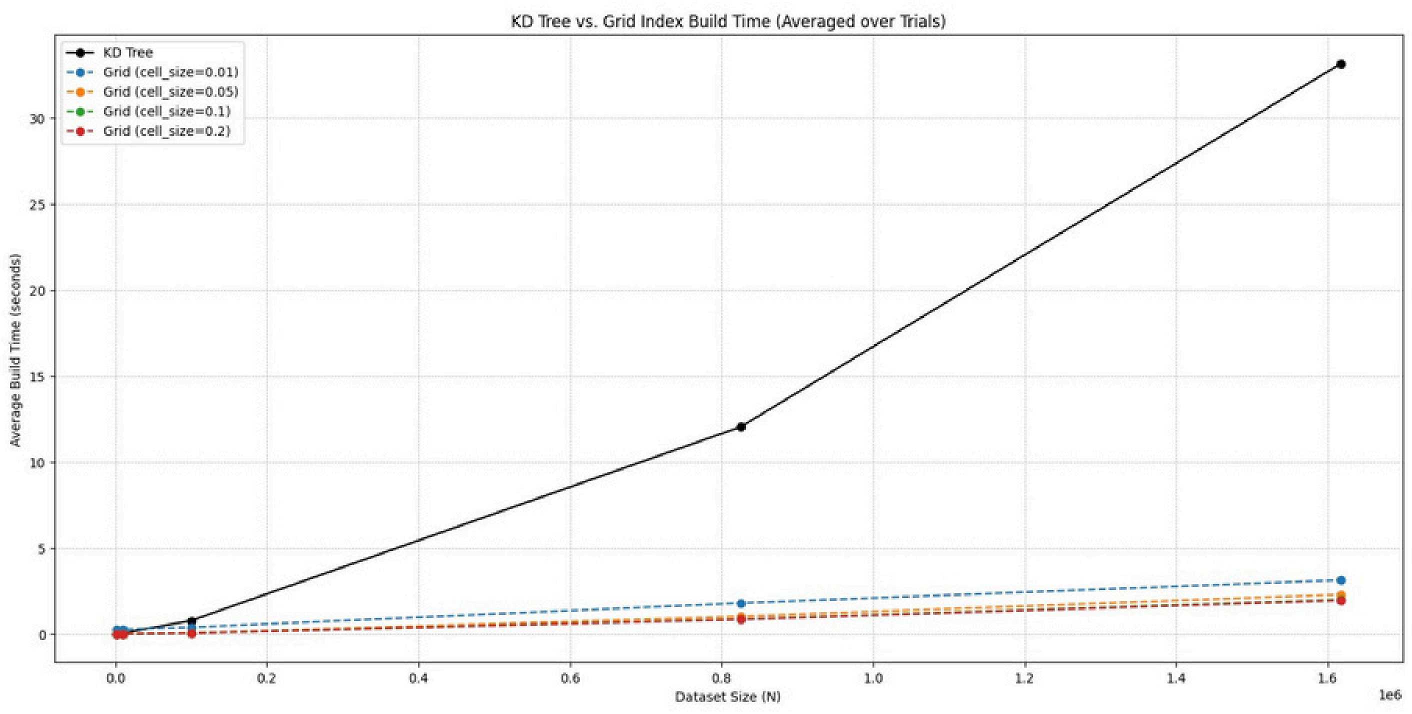


```
Grid Index Range Query returns the exact same POIs as Linear Search
```

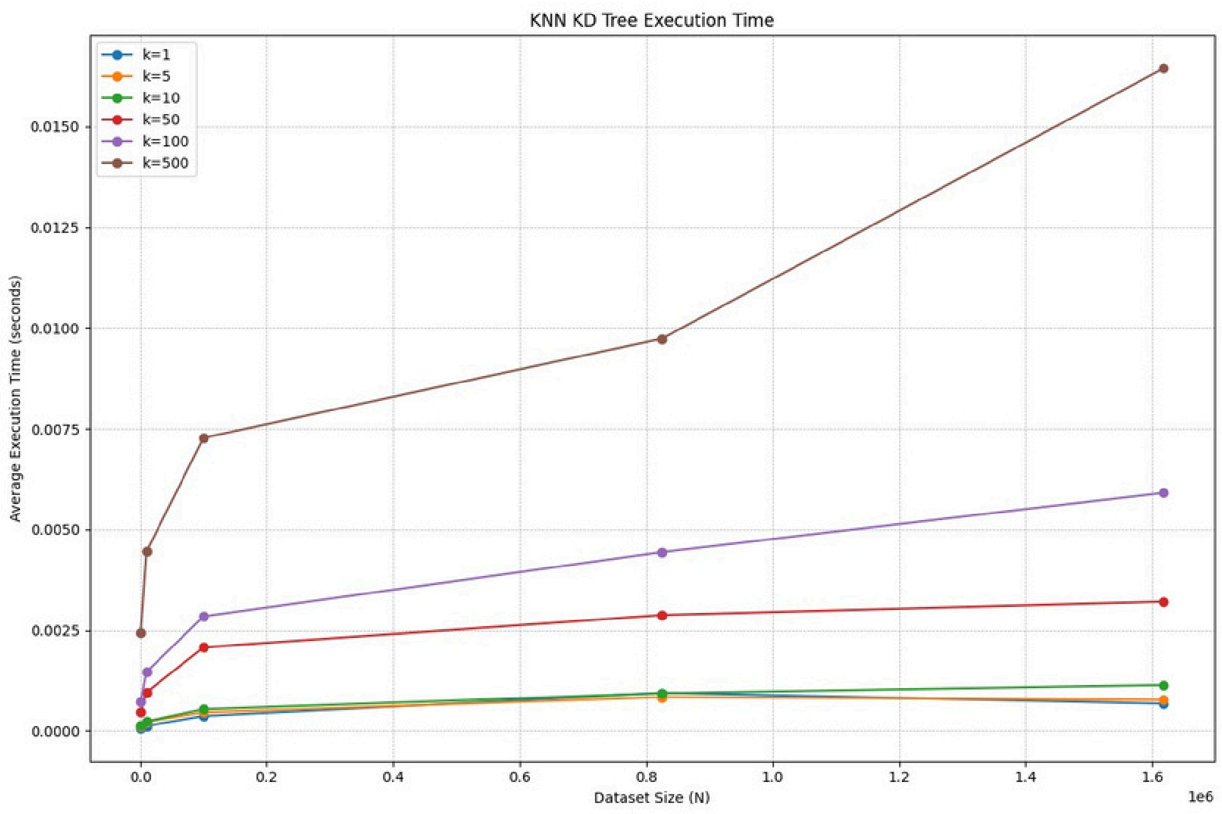


KD-Tree Index

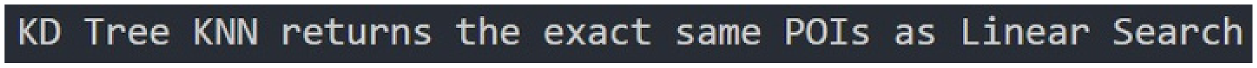
A KD-Tree was built by recursively splitting the data at the median, alternating between latitude and longitude at each level. This process has a time complexity of O(n log n), primarily due to the need for sorting to determine the median at each step. The following plot compares the build times of the KD-Tree and the grid index for different cell sizes [0.01, 0.05, 0.1, 0.2]. The results show that the KD-Tree takes significantly longer to build compared to the grid index, as expected due to the sorting overhead required for median selection in each recursive step. When comparing cell sizes, smaller cell sizes take slightly longer to build, but the difference is minor.



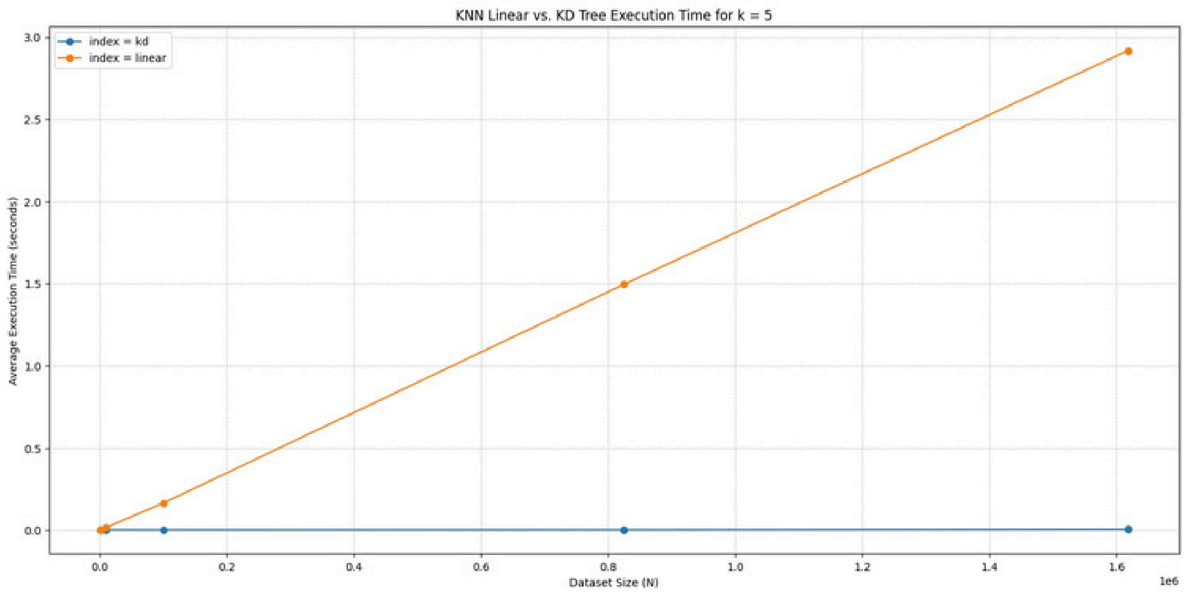
k-Nearest Neighbors (kNN) Query Using KD-Tree Index
In a KD tree, KNN works by first traversing down to the leaf node where the target POIwould be, then backtracking to check if closer points exist in other branches. Execution time increased significantly as k increased, especially for larger values of k. For small k, execution time remained low and stable, but as k grew, maintaining a larger heap of nearest neighbors introduced more overhead. Unlike linear search, where k has minimal impact, in a KD tree, larger k causes more backtracking and distance comparisons, leading to a noticeable increase in execution time. As n increased, execution time also increased slightly, but for very large n, execution time shot up. This trend is reflected in the plot, where execution times are recorded for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and K = [1, 5, 10, 50, 100, 500].



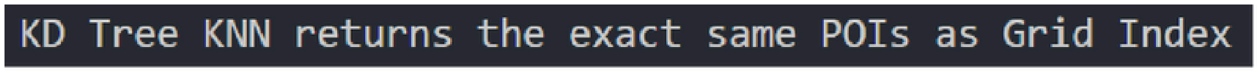
Compared to linear search, the KD-Tree is much faster, especially for large datasets. The correctness of the implementation was verified by comparing the output POI IDs and their distances. This improvement is evident in the plot, where execution time is measured for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and k = 5.



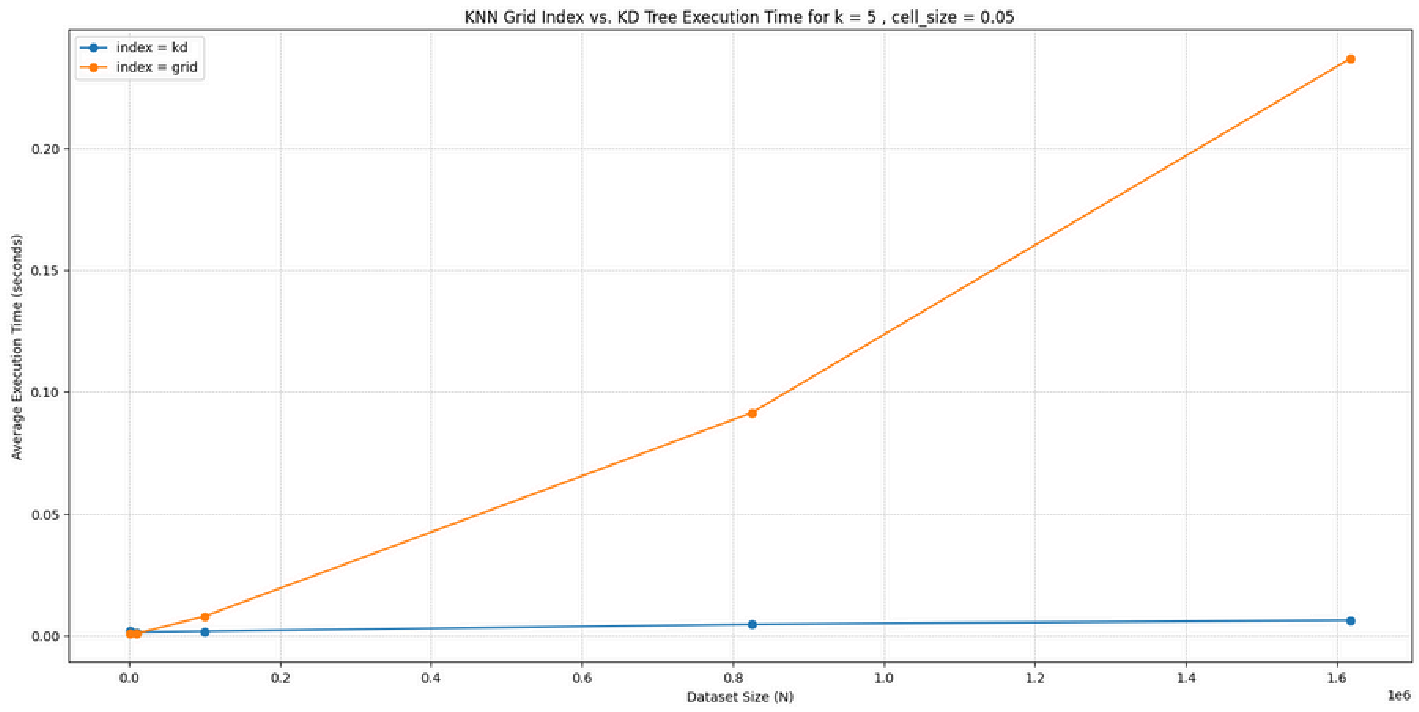
```
KD Tree KNN returns the exact same POIs as Linear Search
```



Compared to the grid index, the KD-Tree is also faster, especially for larger datasets. The correctness of the implementation was verified by comparing the output POI IDs and their distances. This improvement is evident in the plot, where execution time is measured for dataset sizes N = [1000, 10000, 100000, 825171, 1617988], k = 5, and cell_size = 0.05.

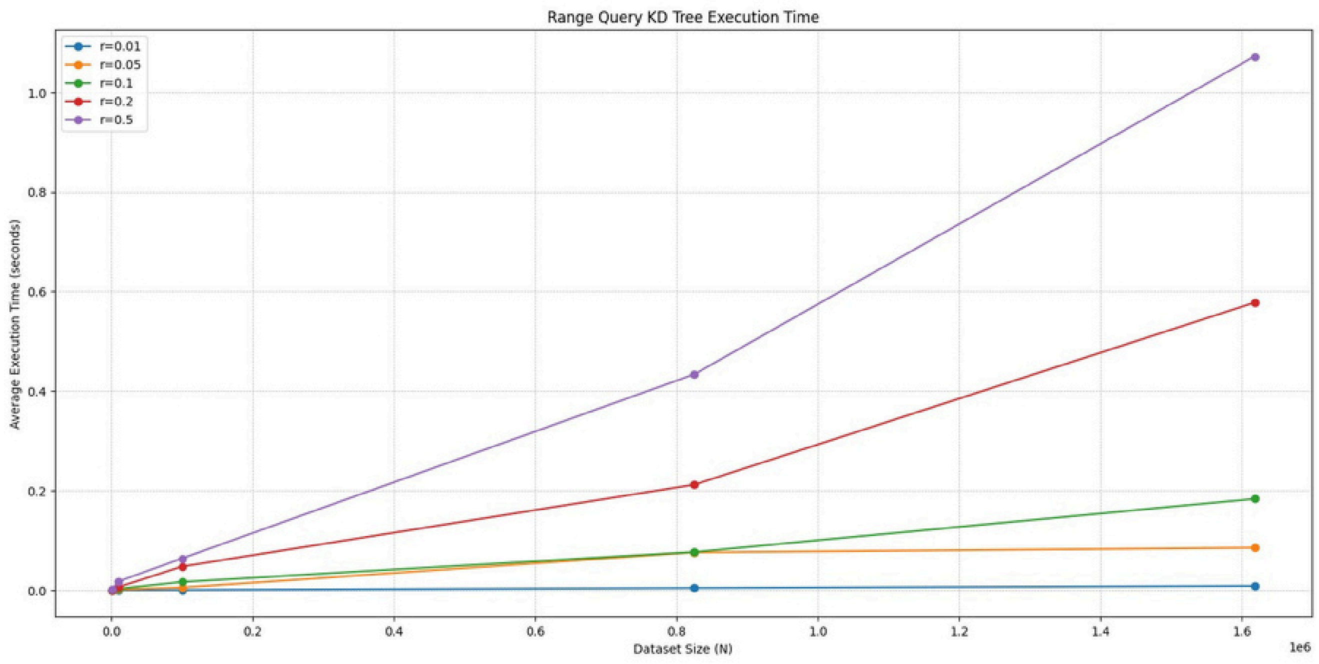


```
KD Tree KNN returns the exact same POIs as Grid Index
```

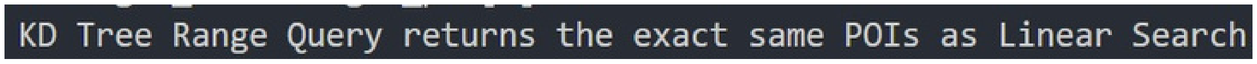


Range Query Using KD-Tree Index

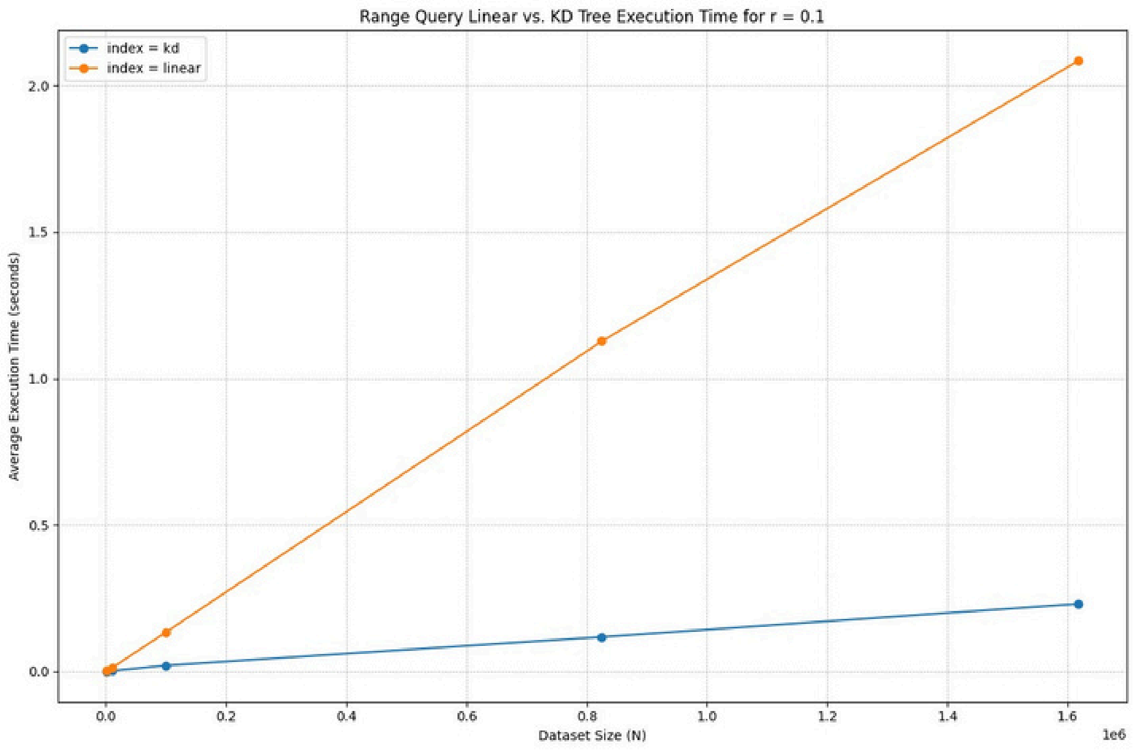
In a KD tree, range queries work by first traversing down to the region where the target POI is located, then checking neighboring nodes if they could contain points within the given radius. Execution time increased as the radius r grew, as more points had to be considered and more branches needed to be explored. As n increased, execution time also increased due to a higher density of points in the tree. This trend is reflected in the plot, where execution times are recorded for dataset sizes n = [1000, 10000, 100000, 825171, 1617988] and r = [0.01, 0.05, 0.1, 0.2, 0.5].



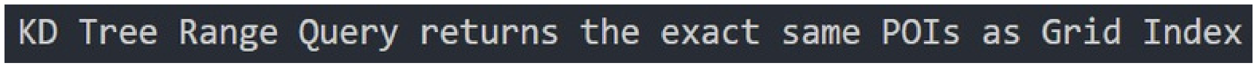
Compared to linear search, the KD-Tree is much faster, especially for large datasets. The correctness of the implementation was verified by comparing the output POI IDs and their distances. This improvement is evident in the plot, where execution time is measured for dataset sizes N = [1000, 10000, 100000, 825171, 1617988] and r = 0.1.



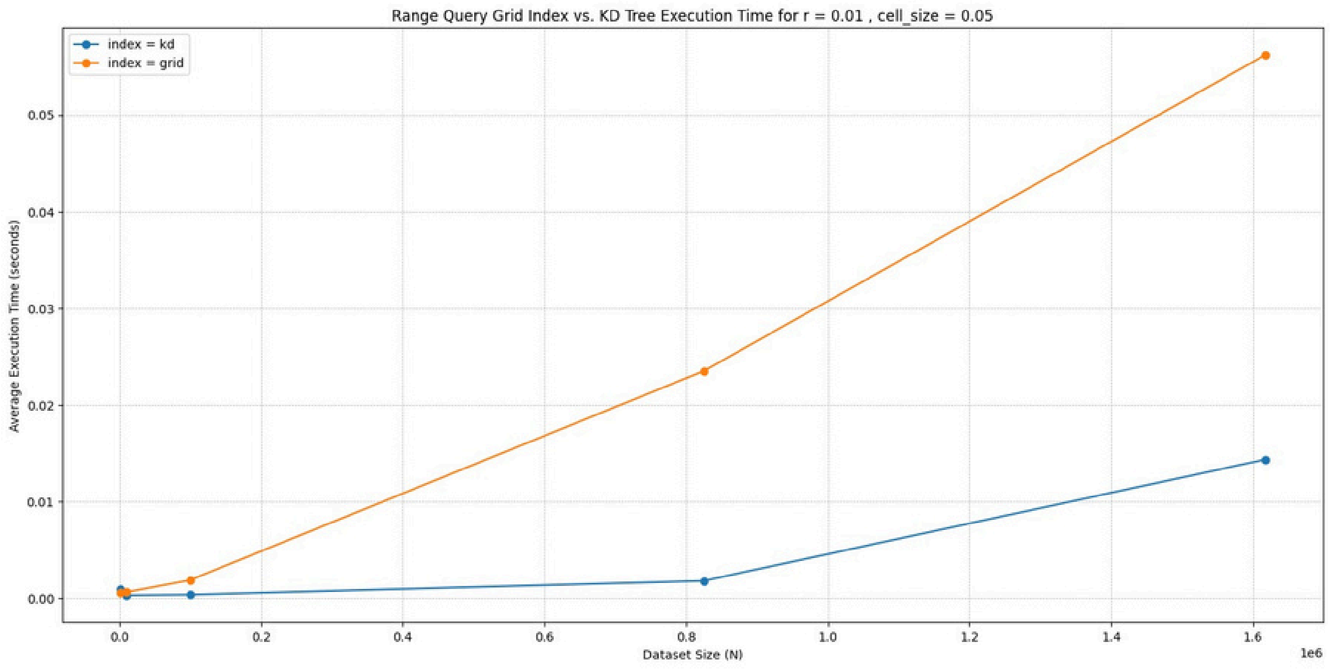
```
KD Tree Range Query returns the exact same POIs as Linear Search
```



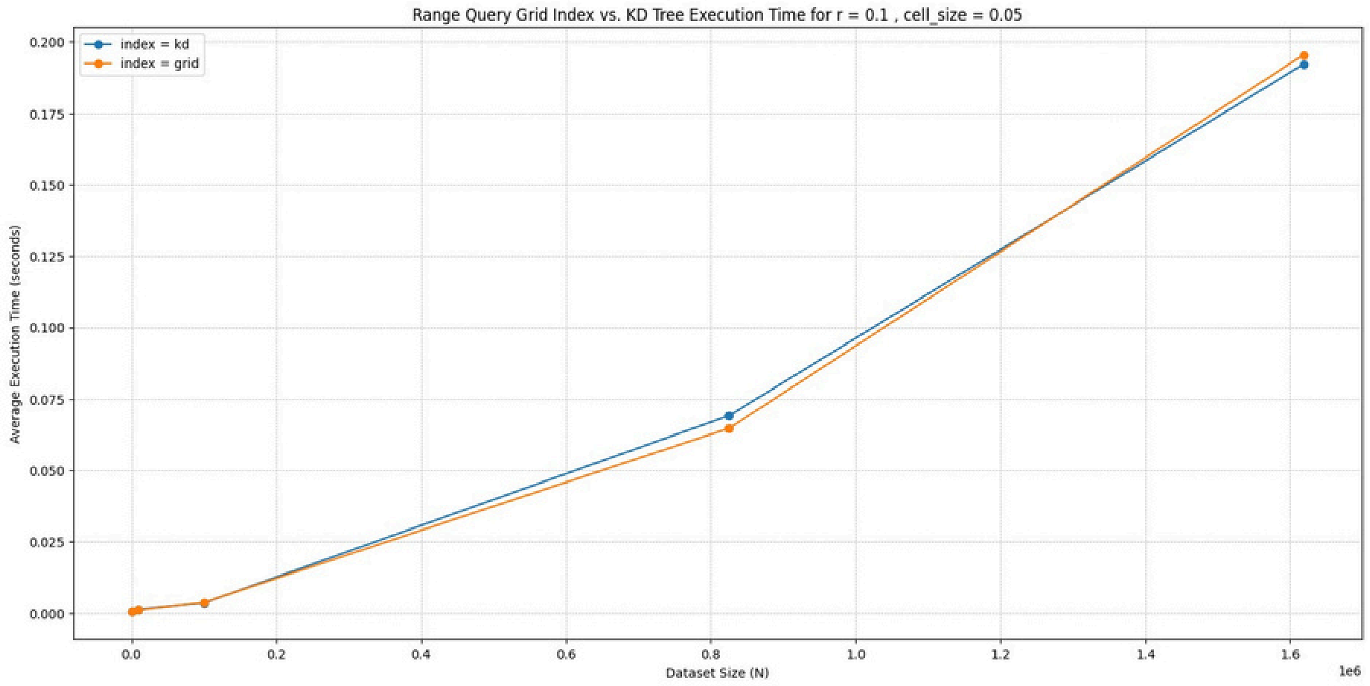
Compared to Grid Index, KD Tree range query is much faster for smaller r. This happens because a smaller r limits backtracking in the KD Tree, allowing it to prune large sections of the tree efficiently, reducing the number of distance computations.



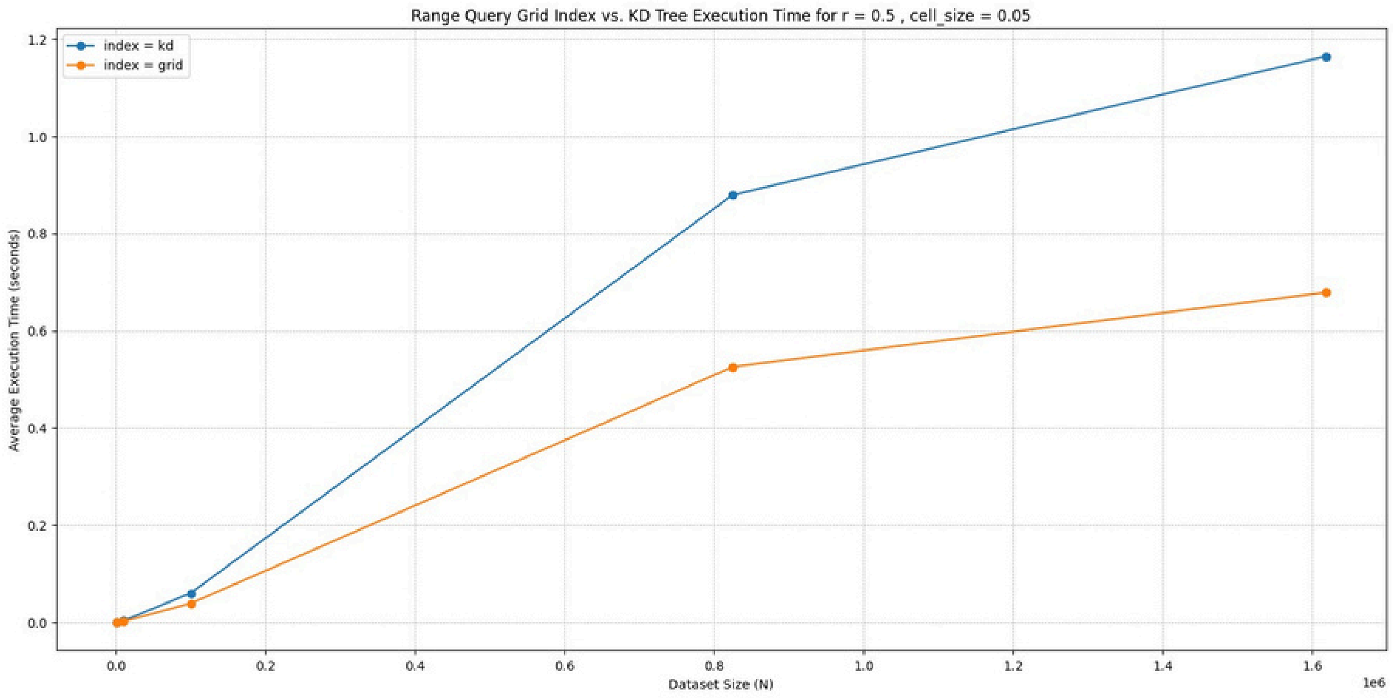
```
KD Tree Range Query returns the exact same POIs as Grid Index
```



For r = 0.1, the execution time for KD Tree and Grid Index is almost the same.



For larger r, the execution time for KD Tree is longer than that of Grid Index. This happens because when r is large, the Grid Index quickly retrieves points from the relevant cells, while the KD Tree still has to traverse the tree structure, which introduces additional overhead.



Overall, KD Tree is more efficient for KNN with an average runtime of 0.005 seconds. For range queries, efficiency depends on r. For smaller r, KD Tree is faster due to efficient pruning, but for larger r, Grid Index can be more efficient as it retrieves points directly from relevant cells without excessive backtracking.