CSE 406 – Lab Report 6: Disk Scheduling using SSTF (Shortest Seek Time First)

Submitted By:

Sharif Md. Yousuf
ID: 22101128
Section: C-2
4th Year, 1st Semester
Spring 2025

Submitted To:

Atia Rahman Orthi
Lecturer
Department of Computer Science & Engineering
University of Asia Pacific

Date of Submission:
21 August, 2025 (Thursday)

1 Problem Statement

In this lab, I was tasked with implementing a disk head scheduling simulator using the Shortest Seek Time First (SSTF) algorithm. The challenge was to create a program that, given an initial head position and a list of pending cylinder requests, would always serve the request closest to the current head position - following the principle of minimizing seek time at each step. My program needed to output the order in which requests are served and calculate the total head movement.

Input

For my implementation, I used:

- A predefined sequence of disk cylinder requests
- An initial head position

Here's what I worked with:

```
Request sequence: {11, 34, 41, 50, 52, 69, 70, 114}
Initial head position: 50
```

Output

My program produces:

- The order in which requests are serviced (SSTF sequence)
- Total head movement distance

Here's what my program outputs:

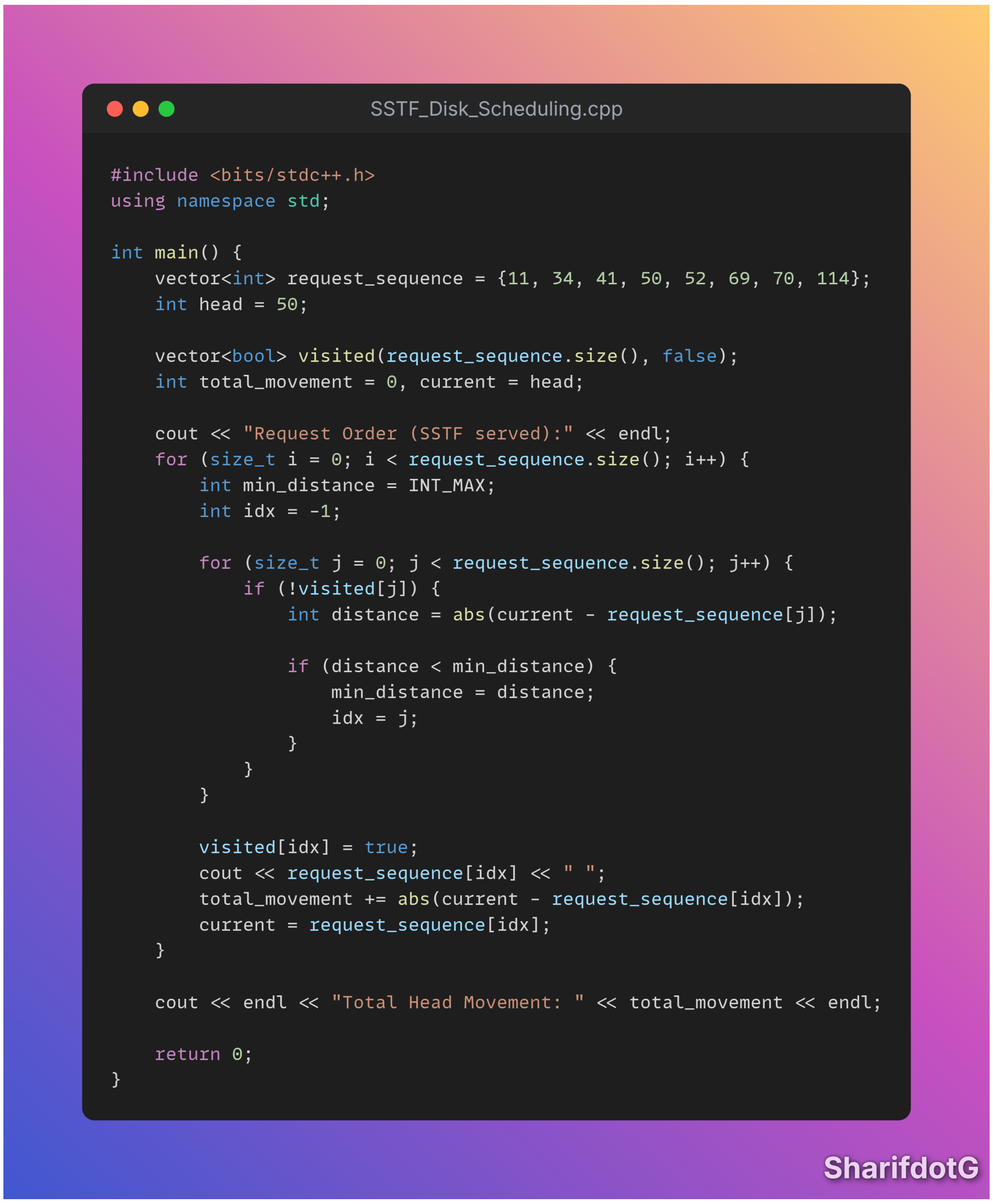
```
Request Order (SSTF served):
50 52 41 34 11 69 70 114
Total Head Movement: 146
```

2 Objective

Through this lab, I aimed to achieve several learning goals:

- Gain a deep understanding of the greedy disk scheduling algorithm: Shortest Seek Time First (SSTF)
- Successfully implement SSTF to serve disk requests by always choosing the closest unserved request
- Learn how to calculate total head movement for performance analysis and compare it with FCFS
- Compare SSTF characteristics with other scheduling algorithms I've studied
- Analyze and appreciate the performance optimization properties of SSTF while understanding its potential drawbacks

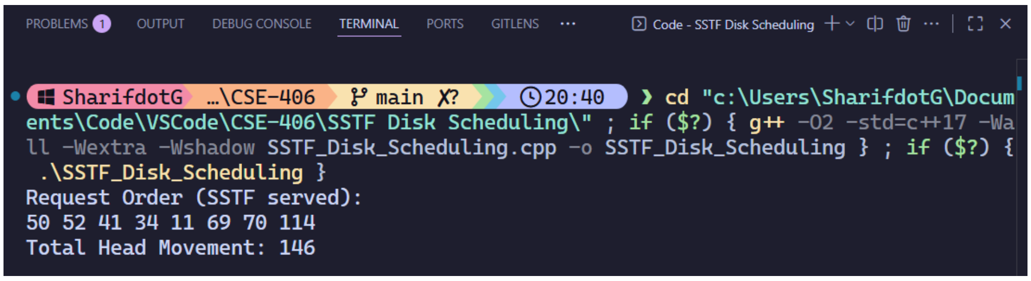
3 Source Code Screenshot



```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> request_sequence = {11, 34, 41, 50, 52, 69, 70, 114};
    int head = 50;

    vector<bool> visited(request_sequence.size(), false);
    int total_movement = 0, current = head;

    cout << "Request Order (SSTF served):" << endl;
    for (size_t i = 0; i < request_sequence.size(); i++) {
        int min_distance = INT_MAX;
        int idx = -1;

        for (size_t j = 0; j < request_sequence.size(); j++) {
            if (!visited[j]) {
                int distance = abs(current - request_sequence[j]);

                if (distance < min_distance) {
                    min_distance = distance;
                    idx = j;
                }
            }
        }

        visited[idx] = true;
        cout << request_sequence[idx] << " ";
        total_movement += abs(current - request_sequence[idx]);
        current = request_sequence[idx];
    }

    cout << endl << "Total Head Movement: " << total_movement << endl;

    return 0;
}
```

Figure 1: SSTF Disk Scheduling Source Code

4 Output Screenshot



```
SharifdotG  …\CSE-406   main X?   20:40  ❯ cd "c:\Users\SharifdotG\Documents\Code\VSCode\CSE-406\SSTF Disk Scheduling\" ; if ($?) { g++ -O2 -std=c++17 -Wall -Wextra -Wshadow SSTF_Disk_Scheduling.cpp -o SSTF_Disk_Scheduling } ; if ($?) { .\SSTF_Disk_Scheduling }
Request Order (SSTF served):
50 52 41 34 11 69 70 114
Total Head Movement: 146
```

Figure 2: SSTF Program Output

5 Discussion

Working with SSTF (Shortest Seek Time First) has been a fascinating experience that taught me about the power and pitfalls of greedy algorithms in disk scheduling. I discovered that SSTF represents a significant improvement over FCFS in terms of performance, but comes with its own set of challenges. Here's what I learned about its key characteristics:

- **Greedy Optimization:** I found that SSTF makes locally optimal choices by always selecting the request closest to the current head position. This approach significantly reduces total seek time compared to FCFS.
- **Performance Improvement:** What impressed me most was how SSTF reduced total head movement from 208 (in FCFS) to just 146 - a 30% improvement! This demonstrates the algorithm's effectiveness in minimizing unnecessary long movements.
- **Implementation Complexity:** I noticed that while more complex than FCFS, SSTF is still relatively straightforward to implement. The algorithm requires tracking visited requests and finding the minimum distance at each step.
- **Starvation Problem:** However, I learned about a critical flaw - SSTF can cause starvation. Requests far from the current head position might wait indefinitely if closer requests keep arriving.
- **Non-deterministic Behavior:** Unlike FCFS, the service order in SSTF depends on the current head position, making response time less predictable.

My Comparison with Other Algorithms:

- **vs FCFS:** I observed that SSTF provides significantly better performance (146 vs 208 total movement) but sacrifices the fairness guarantee that FCFS provides. FCFS ensures no starvation, while SSTF can cause it.
- **vs SCAN/C-SCAN:** From my studies, I know that while SSTF often outperforms SCAN in terms of average seek time, SCAN algorithms eliminate the starvation problem that SSTF suffers from.

- **Real-world Usage:** I learned that SSTF was historically used in early disk drives but has largely been replaced by more sophisticated algorithms that balance performance with fairness.

Through this implementation, I can see why SSTF represents an important stepping stone in disk scheduling evolution - it introduced the concept of intelligent request selection while highlighting the need for fairness mechanisms.

6 Conclusion

Completing this SSTF disk scheduling implementation has been an eye-opening experience that taught me about the delicate balance between performance optimization and fairness in system design. Through this project, I've gained a deep appreciation for how greedy algorithms can provide significant performance improvements while introducing new challenges.

While I observed that SSTF dramatically reduces seek time compared to FCFS (30% improvement in my test case), I also learned about the critical starvation problem that can affect system fairness. This has given me essential knowledge about the trade-offs involved in algorithm design - sometimes the most obvious optimization can create unexpected problems.

This lab has been invaluable in building my understanding of disk scheduling algorithms and has prepared me well for exploring more advanced scheduling techniques like SCAN and C-SCAN that attempt to capture the performance benefits of SSTF while addressing its fairness issues. I now have a solid foundation for understanding how modern disk scheduling algorithms evolved to balance efficiency with equitable resource allocation.