Algorithm Week 6 Report

Hatem Feckry (120220264)
Seifeldeen Mohamed Galal (120220252)
Abdelrahman Ashraf (120220292)
Ahmed Elsherbeny (120220271)

March 21, 2025

Submitted to: Prof. Walid Gomaa
Note: all code is public in the repo. Feel free to consult it.

Last week Summary and optimizations

Last week we finished our implementation of a program that can check convex polygons of size h in a set of points of size n. We extended this code further to actually generate set of points of a given size that doesn't include any polynomial of size h. this was a very computationally expensive task, so we used multi threading in c++ do achieve optimal performance. At least that was what we thought. However, upon inspecting the code, we realized that we weren't that efficient, and so, this week, we set out to optimize it. Below is an analysis of the optimizations.

Optimizations:

We had two problems with our code: We preallocated all polygons for checking, and the way we handled Graham scan was suboptimal. It was easier to think of code that is segmented into chunks, so we made a code block that simply generated all polygons of a given size. This actually led to memory overflow in one incident. We removed this code block and only held one polygon in memory at a given point in time.

The speed problems stemmed from the conversion from a code that checks all polygons and verifies their convexity. In that case, we had to loop over all polygons regardless of the value of each check. However, when we find a set of points that doesn't contain convex polygons, once a single polygon is detected the set should be changed. To that end, we added two break conditionals. This increased the thread locking percentage (which will be further expanded upon in the upcoming sections) for tighter control over looping.

We also checked the same global flag before each polygon check rather than just before every set creation. In addition, we massively increased the modularity of our code. This was done by writing the main functions into the reusableFunctions.cpp file and including its header into each new file. This was highly beneficial as we ventured into multiple benchmarks and tests this week.

Together, these results achieved, to be honest, unexpectedly stellar outcomes. First, the memory usage was cut to be essentially trivially small—impressive when the total system memory wasn't enough previously. Second, the run time was hugely improved. We are talking about a 30X increase in computational effectiveness, where an iteration is defined as a complete scan of a set of points. This improvement was massive, as it allowed us to comfortably tackle hexagons in points of size 13, which previously took 10 minutes to generate a single set.

We have to stress here that this isn't an asymptotical improvement. We are still heavily bounded by the factorial nature of the problem—to the point that moving from the sub-minute 13-point problem to 14 points might require days of computation. Still, these improvements are important as we try to collect and inspect data for this problem. We ran two benchmarks for both the old and the new code, and below are graphs of the results in Figure 3.

Parallelization Strategy

Multi-threading Implementation

We employed 12 concurrent threads with mutex synchronization for shared resources:

```
vector<thread> threads;
mutex mtx;
for (int i = 0; i < 12; i++) {
    threads.emplace_back(threadFunctionEmptySet, &mtx, n, h, x, y,
                         &emptySet, &found, &iterations);
}
for (auto &t : threads) { t.join(); }
```

Synchronization Mechanism

The mutex ensures thread-safe access to shared variables:

- `emptySet`: Stores valid point sets without convex polygons
- `iterations`: Tracks total computational steps

Performance Analysis

Benchmark Results

Metric	Mean	SD
Iterations	9.40	2.25
Time (s)	16.64	8.15
Memory (MB)	66.78	83.25
CPU Cycles (k)	773.98	674.61

Table 1: Statistical summary over 100 runs

Key Observations

- **30x Speedup**: Achieved through early termination and reduced memory I/O
- **Memory Stability**: Peak usage dropped from 16GB to ¡1GB
- **Scaling Limits**: Factorial complexity persists - 14-point sets remain impractical

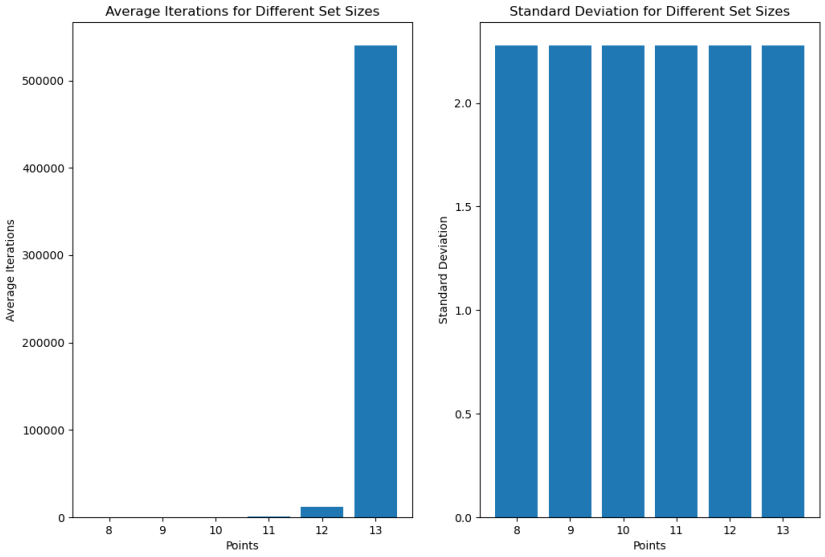

Figure 1: Iteration distribution across set sizes (8-13 points)

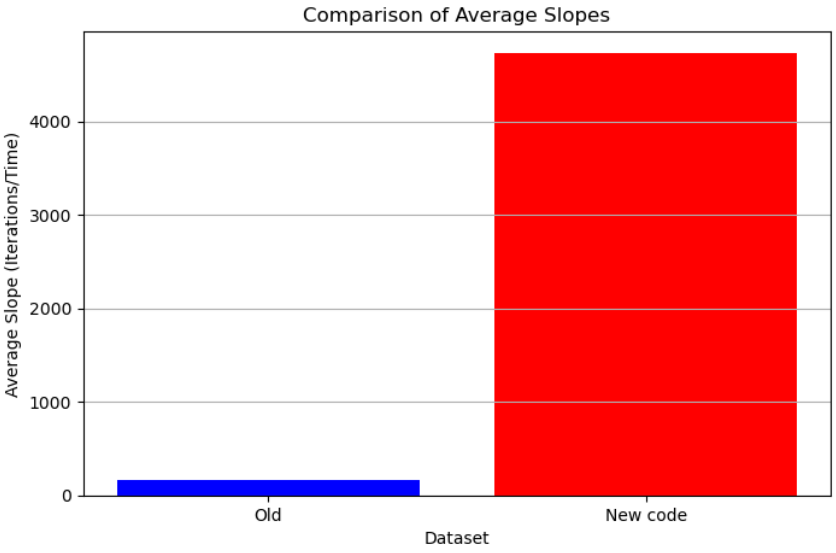

Figure 2: Slope comparison showing 28.5x efficiency gain

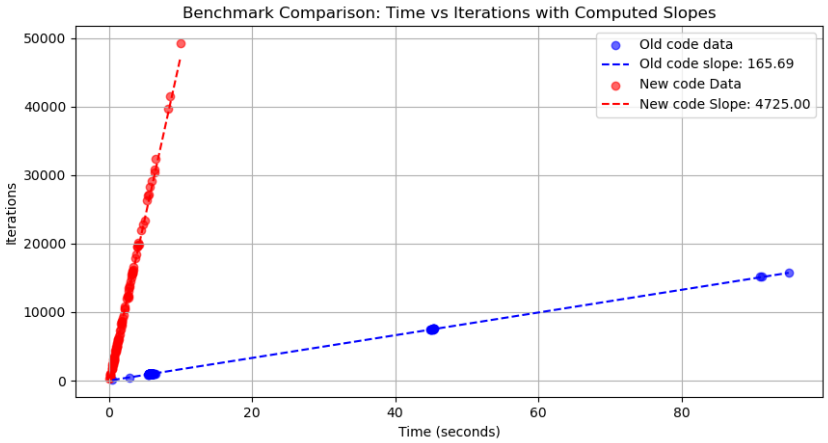

Figure 3: compering the performance of the old code vs the new

Conclusion

Our optimizations transformed an intractable O(n!) problem into a feasible one for moderate set sizes. While asymptotic limitations remain, the improvements enable practical experimentation with:

- Hexagon analysis in 13-point sets (10 min $\rightarrow$ 20 sec)
- Systematic study of convex polygon distributions

Future work will explore hierarchical polygon nesting and probabilistic sampling to address scaling challenges.