Checkpoint 3 Progress Report

Efficient Computation of Crossing Components and Shortcut Hulls

Zohaib Aslam (za08134), M. Mansoor Alam (ma08322)

1 Implementation Summary

We have successfully implemented all components of the polygon simplification algorithm described in the research paper. Our implementation includes:

- **Polygon and Edge Representation:** Data structures for simple polygons as ordered sets of points, with edges stored in topological order.
- **Edge Crossing Computation:** Adjacency lists $N^+(v)$ and $N^-(v)$ for each vertex v, sorted in descending order.
- **Crossing Component Computation:** Implementation of the pseudo-intersection graph G_P approach instead of the full intersection graph G_I, with BFS for connected components.
- **Shortcut Hull Processing:** Convex hull computation and pocket segmentation for separate processing.
- **Optimal Shortcut Hull Computation:** Dynamic programming algorithm using the cost function $c(Q) = \lambda \cdot \beta(Q) + (1 - \lambda) \cdot \alpha(Q)$ where $\beta(Q)$ is perimeter, $\alpha(Q)$ is area, and λ is a user parameter.

2 Correctness Testing

Our testing combines unit tests, integration tests, and visual validation:

2.1 Unit Testing

Each algorithm component was tested individually with specific test cases:

- **Edge Crossing Detection:** Geometric verification with cases like squares with crossing diagonals, complex polygons with known crossings, and edge cases (collinear edges, shared vertices and near-parallel edges).
- **Adjacency List Construction:** Validation against reference structures, verifying correct sorting & empty lists.
- **Component Detection:** BFS traversal tested against expected connected components.

2.2 Integration Testing

We tested how components work together:

- Verified pseudo-intersection graph against full intersection graph on small test cases
- Tested shortcut hull computations with varying λ values (0.0, 0.25, 0.5, 0.75, 1.0)
- End-to-end tests on polygons with known expected simplifications

2.3 Visual Validation

We implemented visualization tools:

- Original polygons with crossing edges highlighted
- Color-coded crossing components
- Side-by-side comparisons of original polygons and shortcut hulls
- Lambda-effect visualizations

2.4 Sample Test Cases - Here are two specific cases

1. **Square with Crossing Diagonals:**
    - Input: Square with vertices at (0,0), (4,0), (4,4), (0,4)
    - Shortcuts: (1,3), (2,4)
    - Result: Algorithm correctly identified one crossing
2. **Star-shaped Polygon:**
    - Input: 10-vertex star polygon
    - Result: Algorithm correctly identified 3 components

3 Complexity and Runtime Analysis

3.1 Theoretical Analysis

The algorithm achieves improved performance across its core components. For **Edge Crossing Detection**, the time complexity is $\mathcal{O}(n+m+k)$, where n is the number of vertices, m the number of potential shortcut edges, and k the number of edge crossings. This efficiency is enabled by a column-wise sweep algorithm and adjacency list structures, avoiding the naive $\mathcal{O}(m^2)$ approach. **Component Computation** runs in $\mathcal{O}(\min(n+m+k, n^2))$ by leveraging a pseudo-intersection graph and performing BFS, which is significantly better than the naive $\mathcal{O}(n^4)$ method. Finally, the **Shortcut Hull Dynamic Programming** component operates in $\mathcal{O}(h\chi^3 + \chi n)$, where h is the number of crossing components and χ is the maximum size of any component. By focusing on component hierarchies instead of the full polygon, the algorithm reduces the search space and optimizes pathfinding.

3.2 Empirical Analysis

Our benchmarking methodology involved:

- Polygons with controlled complexity (100 to 10,000 vertices)
- Multiple runs per size to account for variance
- Measurement of wall-clock time and CPU cycles

Benchmark results confirm sub-quadratic scaling:

The runtime grows smoothly and remains practical even for large inputs (under 4 seconds for 10,000 vertices), validating our implementation approach.

Polygon Size	Runtime (s)	Shortcut Hull Size
100	0.027	20
200	0.068	28
400	0.162	41
800	0.425	58
1000	0.601	64
2000	1.362	90
4000	2.457	128
10000	3.874	203

4 Baseline and Comparative Evaluation

We compared against three baseline approaches:

1. **Naive Edge Crossing Detection:**
   - For $n = 1000$, our optimized algorithm: 0.601s
   - Naive implementation: 7.823s (13x slower)
2. **Greedy Shortcut Selection:**
   - Our DP-based approach produces solutions with 18% lower total cost on average
   - For complex pocket structures, improvement increases to 25-30%
3. **Douglas-Peucker Algorithm:**
   - Our algorithm better preserves topological properties
   - Our approach balances perimeter and area optimization through λ
   - Douglas-Peucker is faster but produces less optimal results on our cost function

5 Challenges and Solutions

- **Algorithm Complexity:** The algorithm has multiple complex steps and data structures.
  - *Solution:* Used a modular approach, breaking implementation into small, testable chunks.
- **Inconsistency in Shortcut Hull Results:** Some polygons yield incorrect results.
  - *Solution:* Currently analyzing code for bugs, comparing with the paper's implementation.

6 Enhancements

- **Testing on Different Dataset:** Using custom dataset with polygons of up to 10,000 vertices for comprehensive testing.
- **Visualization Tools:** Developed additional tools:
  - Lambda-effect visualizer
  - Component coloring
  - Side-by-side comparison of original polygon, convex hull, and shortcut hull
- **Pocket Detection Optimization:** Implemented:
  - More Robust equality checking for floating-point coordinates
  - Optimized search for hull vertices & Early termination for trivial pockets