Technical Summary: Efficient Computation of Crossing Components and Shortcut Hulls

Zohaib Aslam za08134 & M. Mansoor Alam ma08322

Based on work by Nikolas Alexander Schwarz and Sabine Storandt

1 Problem and Contribution

1.1 Problem

The paper addresses computational challenges in polygon simplification, specifically in the context of shortcut hulls. For a given polygon P with n vertices, a shortcut hull is a simplified polygon that fully contains the original polygon. The computation of shortcut hulls relies on identifying edge crossings among potential shortcuts and computing crossing components.

The main computational bottlenecks in the existing shortcut hull algorithm were:

1. Computing all edge crossings efficiently in the visibility graph
2. Determining crossing components from these crossings
3. Computing enclosing polygons (regions) for each crossing component

1.2 Contribution

The paper makes several significant algorithmic contributions:

1. An optimal output-sensitive algorithm for computing all edge crossings in $O(n + m + k)$ time, where n is the number of polygon vertices, m is the number of shortcuts, and k is the number of crossings.
2. An efficient algorithm for computing crossing components in $O(\min\{n + m + k, n^2\})$ time, which is significantly faster than the naive $O(n^4)$ approach, especially for inputs with many crossings.
3. A novel crossing component hierarchy data structure that efficiently encodes crossing components and allows for fast partitioning of the polygon.
4. A streamlined approach to compute shortcut hulls by using convex hull pockets instead of the previously required "sliced donut" construction.

These contributions reduce the overall computational complexity of shortcut hull computation from effectively $O(n^4)$ to $O(n^3)$.

2 Algorithmic Description

2.1 Edge Crossing Detection

The algorithm for detecting crossings in the visibility graph exploits the following observation:

Given two edges $e = (a, b)$ and $e' = (u, v)$ with $u < a$, they cross if and only if $a < v < b$.

Based on this observation, the algorithm:

1. Processes edges in topological order and maintains sorted adjacency lists

2. For each edge, efficiently identifies intersecting edges using a sweep line approach

3. Uses a novel matrix-based visualization to identify areas where crossings can occur

The adjacency lists are maintained as stacks, and the algorithm explores potential intersections by traversing rectangles in the visualization matrix column-wise from left to right and bottom to top within each column.

2.2 Crossing Component Computation

Rather than constructing the full intersection graph G_I (which could have $\Theta(n^4)$ edges), the algorithm constructs a smaller pseudo-intersection graph G_P with the same connected components. This approach leverages the following insight:

If edges e and e_2 cross, and e_1 satisfies certain position constraints relative to e and e_2, then e and e_1 must also cross.

This allows the algorithm to:

1. Connect edges in the same column of the visualization matrix that belong to the same crossing component

2. Report only the first edge when traversing rows that have been previously processed

3. Extract connected components efficiently from the resulting sparse graph

2.3 Crossing Component Hierarchy

To efficiently compute the regions (enclosing polygons) for each crossing component, the paper introduces a hierarchical data structure:

1. Each crossing component is represented as an interval (a, b) of the lowest and highest vertex indices it contains

2. These intervals form a hierarchy based on containment relationships

3. The hierarchy is constructed by sorting components first by upper bounds and then by lower bounds

4. This produces a tree structure similar to a B-tree where the pre-order traversal preserves the original ordering

For each component, the algorithm computes polylines between consecutive pairs of vertices, resolving conflicts with the polygon boundary by finding optimal convex chains.

2.4 Shortcut Hull Computation

The enhanced shortcut hull algorithm:

1. Computes the convex hull of the input polygon

2. Processes each pocket (region between the convex hull and original polygon) separately

3. Uses the crossing component hierarchy to guide the computation

4. Finds the cost-optimal shortcut hull based on the user-defined parameter λ

The parameter $\lambda \in [0, 1]$ balances the trade-off between perimeter length and area in the cost function: $c(Q) = \lambda \cdot \beta(Q) + (1 - \lambda) \cdot \alpha(Q)$, where $\beta(Q)$ is the perimeter and $\alpha(Q)$ is the area of the shortcut hull.

3 Comparison with Existing Approaches

- **Edge Crossing Detection:** The general Bentley-Ottmann algorithm detects crossings in $O(m \log m + k \log m)$ time, while the new algorithm achieves $O(n + m + k)$, which is optimal.
- **Crossing Component Computation:** The naive approach requires $O(n^4)$ time in the worst case. The new algorithm achieves $O(n^2)$ even when the number of crossings k is extremely large (potentially $\Theta(n^4)$).
- **Shortcut Hull Computation:** The original algorithm by Bonerath et al. assumed crossing components were given, effectively requiring $O(n^4)$ time with naive preprocessing. The improved algorithm reduces this to $O(n^3)$.
- **Polygon Decomposition:** The new approach using convex hull pockets simplifies implementation compared to the previously required "sliced donut" construction and enables parallel processing of subproblems.

4 Data Structures and Techniques

The algorithms employ several key data structures and techniques:

1. **Adjacency Lists:** Maintained as stacks with vertices sorted in descending order for efficient traversal
2. **Doubly-Linked Lists:** Simulated using two arrays to efficiently skip empty adjacency lists
3. **Pseudo-Intersection Graph:** A sparse representation of crossing relationships with size $O(n^2)$ instead of $\Theta(n^4)$
4. **Crossing Component Hierarchy:** A tree-like structure organizing crossing components by their vertex index ranges
5. **Sweep Line Algorithm:** For efficient detection of edge crossings
6. **Counting Sort:** Used for efficient ordering of vertices and edges
7. **Convex Chain Computation:** For finding optimal paths around crossing components
8. **Graph Search:** BFS or DFS to identify connected components in the pseudo-intersection graph

5 Implementation Outlook

Several technical challenges may arise when implementing these algorithms:

1. **Data Structure Complexity:** Data structures such as doubly-linked lists, sparse representation of pseudo-intersection graph, and a tree-like structure for crossing component hierarchy would need to be implemented. These complex data structures may lead to challenges in their correct implementation
2. **Memory Management:** For large polygons with many visibility edges, efficient memory management will be crucial to avoid excessive memory usage.