Heaphull?

Andrea Mantler Jack Snoeyink
Department of Computer Science
University of North Carolina at Chapel Hill*
{mantler,snoeyink}@cs.unc.edu

Abstract

We present a planar convex hull algorithm, Heaphull, whose main data structure is a kinetic heap. We can limit the number of times a point needs to move up in the heap to establish that $O(n \lg n)$ heap operations are sufficient; the event queue for the kinetic structure brings the running time to $O(n \lg^2 n)$. We also report on experiments.

1 Introduction

Almost all major sorting algorithms have corresponding convex hull algorithms in the plane [4]: Quicksort and Mergesort correspond closely to Quickhull and divide-and-conquer. Even selection sort and insertion sort have analogues in gift-wrapping and incremental algorithms. One sorting algorithm that is noticeably absent from this list is Heapsort.

In the next section, we briefly review heaps and Heapsort, and describe the kinetic heap of Basch et al. [1]. We derive a first version of Heaphull from their kinetic heap in Section 3.1. We improve this algorithm by limiting the motion of points in the heap in Section 3.2. We have implemented both versions in Java, and report operation counts in Section 4.

2 Heapsort and Kinetic Heaps

A *heap* is a binary tree storing keys such that every node has the *heap property*: the key of any node is greater than or equal to the keys of its children. We use a heap that is compactly stored in an array in the standard manner: the children of a node at index i are at indices $2i$ and $2i + 1$. This ensures that the heap is a tree of depth $\lceil \lg n \rceil$.

Heapsort puts a list of numbers in an array, then establishes the heap property from the bottom up: whenever a parent's key is less than a child's, it swaps the parent's key with the larger key value of the two children and percolates the key down until the heap property is restored in the subtree. After heapifying, the largest key is at the top of the heap. Then Heapsort repeatedly removes the top key and places it in the sorted list, replaces the root key by a leaf's key, and percolates that key down until the heap property is restored.

Kinetic data structures, as defined by Basch, Guibas, and Hershberger [2], exploit the continuity or coherence of motion in geometric structures. Basch et al. [1] show how to maintain a *kinetic heap* when the key values are changing continuously according to known functions of time. Each heap edge joining a parent to child can be considered a *certificate* of the heap property that will expire at the next time that the child key grows larger than the parent key. As long as every edge has a valid certificate, the heap is valid. In the instant a certificate expires, the parent and child keys are equal, so they can be swapped in the heap without disturbing the validity of other certificates (although this will change the expiration times of the four certificates that involved the keys at these nodes). If expiration times are kept in an event queue, then discrete event simulation can maintain the heap with logarithmic overhead per swap in the heap.

3 Heaphull

The *convex hull* of a set of points in two dimensions is the convex polygon with smallest area that contains all of the points in the data set. The convex hull is determined by the data points that are extreme in some direction.

In Heaphull, we use a kinetic heap to find the extreme point in each direction as we rotate through the directions. These extreme points in the order they are found make up the convex hull. Our first version implements and analyzes this idea in a straightforward way using the kinetic heap.

*Both authors acknowledge support of NSF grant 9988742 for this research.

3.1 Heaphull without deletions

Like Heapsort, Heaphull starts off by heapifying the points based on an *up* direction. Figure 1 shows the *heap edges* as arrows from parent to child positions. The heap edges certify that the top of the heap is an extreme point, which can be added to the convex hull. Unlike Heapsort, we do not remove the element at the top of the heap once it is determined.

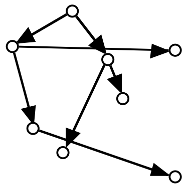

Figure 1: Heap in direction *up*

To find the next extreme point, we can rotate the *up* direction counter-clockwise. The heap property will be maintained until some heap edge becomes perpendicular to *up*. When this occurs, we can re-establish the heap property by swapping the parent and child keys (points) for this edge. We say that the child point has moved "up" and the parent point has moved "down."

This swap, which interchanges two points in a fixed heap, can be viewed from a geometric perspective as flipping a horizontal edge from right to left. Figure 2 illustrates that after a flip, the arrow goes from the point that has moved up to the point that has moved down. We can also see that a flip changes the slopes of up to four other edges incident on this parent or child.

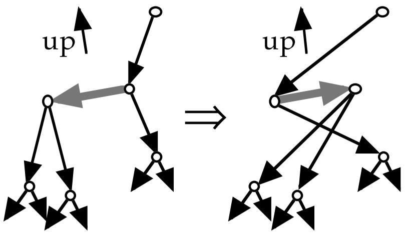

Figure 2: Swapping parent and child *flips* an edge; note that up to four other edges change slope, but remain directed downward.

Whenever a new point rises to the top of the heap, it is added to the convex hull. When the first point on the hull returns to the top, the hull is complete. Of course, we do not need to rotate the up direction continuously. The slopes of heap edges determine the directions where the heap changes, so we can identify these as the discrete events and update the kinetic heap at these events.

Theorem 1 *Heaphull without deletion can be implemented to run in $O(n \lg^3 n)$ time.*

Proof: We can implement the Heaphull algorithm using a kinetic heap consisting of a heap of points and an event queue for flips. The heap determines when points are added to the hull and can be maintained in constant time per edge flip, since only one pair of points needs to be swapped for each edge flip. The queue determines which edge flip is next. Several data structures (including heaps!) implement a priority queue with each operation running in $O(\lg n)$ time.

To complete the analysis, we need to count the number of edge flips. This is a difficult problem for a kinetic heap, as Basch et al. [1] observed. They used a clever potential function to show that at most $O(n \lg^2 n)$ swaps occur in an application that constructs the lower envelope of an arrangement of lines. Sweeping these lines gives the dual of the problem of maintaining an extreme point for a rotating direction, as long as we split the range of directions into two ranges of less than 180° by omitting some line that is not parallel to any line defined by two data points. (We prefer not to add this "non-Heapsort" step directly to the algorithm, but are glad to use it in the analysis.) In each range, any edge is flipped at most once. Thus, we can apply their analysis directly to the dual problem. $\square$

3.2 Heaphull with deletion

One difficulty in analyzing a kinetic heap is that points need not move monotically in the heap; they can move up, then down, then up again. In the Heaphull application, if a point moves down after moving up, then either it has already appeared on the hull, or it never will.

Lemma 2 *If a point that has moved up in the heap moves down, then it cannot appear again on the convex hull.*

Proof: From the perspective of a point p, an edge $\overline{pq}$ can correspond to one move up and one move down, each 180° apart. If p is on the convex hull, then the circular sequence of events from all possible edges has a run of "ups" followed by a run of "downs", and p is extreme between the last "up" and the first "down." The subset of edges/events used in the kinetic heap will therefore have the same sequence. $\square$

Thus, we may modify Heaphull to mark each point that goes up, and if a marked point will go down we instead delete it from the heap by recursively promoting the higher child, down to a leaf.

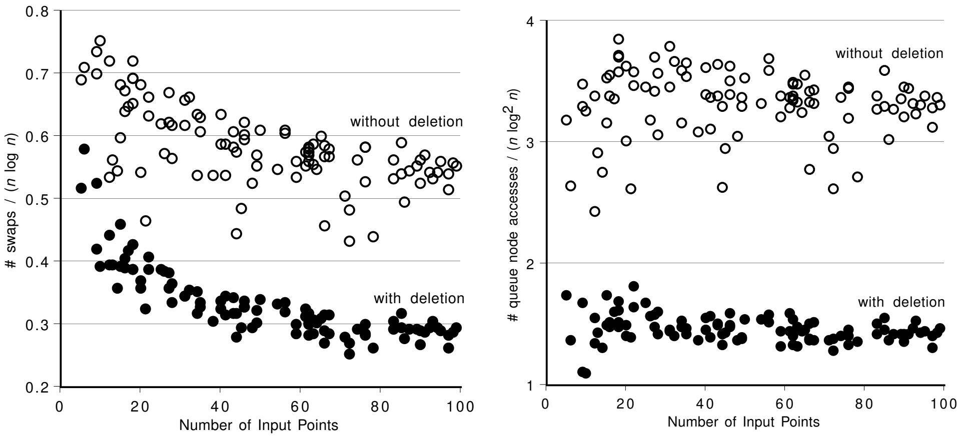

Figure 3: Graphs of the number of swaps and priority queue accesses for Heaphull with and without deletions

Theorem 3 *Heaphull with deletion can be implemented to run in $O(n \lg^2 n)$ time.*

Proof: We can charge each edge flip to the key that is moving up in the heap. Deletion only moves the remaining keys up faster, so each key can move up at most $\lg n$ times. When a key is deleted, it can pay $O(\lg n)$ for the heap updates that it causes. The total number of heap updates is therefore $O(n \lg n)$. For each, $O(\lg n)$ operations are used to update the priority queue. Thus, the total running time of the algorithm in the worst case is $O(n \lg^2 n)$. □

4 Experiments and Observations

We implemented this algorithm using a heap in an array and a standard red-black tree implementation of a priority queue [3]. We measured the number of heap swaps and priority queue node accesses.

We report on a small set of experiments with points generated uniformly at random from the unit square. Figure 3 plots, for 100 sets with sizes chosen at random from $[3, 100]$, the ratios of number of swaps divided by $n \lg n$ and the priority queue accesses divided by $n \lg^2 n$. Initial heap construction is not counted, since it is the same for the algorithm with and without deletion and takes $O(n)$ time.

From the worst-case analysis, one might expect logarithmic growth in the plots without deletion. Instead, it appears that that bound is pessimistic, at least for randomly generated point sets, and that deletion makes its improvement to the constant factor.

Figure 4 demonstrates that one should not expect *output sensitive* dependence on the number of hull edges h. The number of edges flipped is $O(n \lg n)$ and not $O(n \lg h)$, since every point will rise to at most one step away from the top of the heap.

5 Conclusion

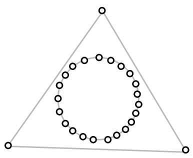

Figure 4:

Our Heaphull algorithm has its similarities with Heapsort, but also has one significant difference. As part of the kinetic heap, it uses a priority queue to keep track of events. Maintaining this structure adds programming complexity, as well as an extra logarithmic factor. Thus, the question remains, is there Heaphull algorithm that is $O(n \lg n)$?

Acknowledgements

We thank Leo Guibas for discussions on kinetic data structures in general, and the kinetic heap in particular.

References

[1] J. Basch, L. J. Guibas, and G. Ramkumar. Sweeping lines and line segments with a heap. In *Proc. 13th Annu. ACM SCG*, pages 469–472, 1997.

[2] J. Basch, Leonidas J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM SODA*, pages 747–756, 1997.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[4] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 3rd edition, October 1990.