1 Fibonacci Heaps

1. Describe the structure of a node in a Fibonacci heap and why each field is required. Hints:

   A fib-heap node has 8 fields.

   - the key of the node

     The metric which we are finding the minimum of – the "priority" of the value stored in the node.

   - the value of the node (or a pointer to the value)

     The value we wish to retrieve.

   - boolean to mark the node as loser or not

     To ensure fibonacci trees are sufficiently deep we must remove nodes from their trees and place them into the root list when two of their children are removed. After their first child is removed we mark them as losers.

   - pointer to the nodes parent

     When we call `decrease_key` (and indirectly, delete) we must access the parent in constant time to check whether the childs key is now less than the parents key. A pointer to the parent allows us to do this.

   - pointer to one child

     In `pop_min`, we promote all the minimum nodes children to the root. To do this we must iterate through the list of children. So we must be able to reach the children. Note that since all the children are linked to each other, we only need a pointer to *one* child.

     When merging trees in `cleanup` we insert the root with the larger key into the child list of the other root. Note again that this only needs a pointer to *one* child.

   - integer holding the degree of the node (the number of children that node has)

     In `cleanup` trees are merged with trees of the same degree. So we need to know the degree of nodes in constant time – otherwise `cleanup` would be $O((\lg n)^2)$ rather than $O(\lg n)$. So we store each nodes' degree.

   - pointer to left sibling

     This connects the roots (or children) into a linked list. Dependent on whether the node is in the root list or not, this either allows us to iterate through child lists or the root list.

   - pointer to right sibling

     This turns the linked list into a circular linked list allowing splicing from the linked list in constant time given a node. This is needed in `decrease_key` to remove the node we are passed and loser nodes.

2. Explain how to create an empty Fibonacci heap, and state the big-O cost.

   Create a new fibonacci heap object which has a null `min_node`, an empty root list and zero size. This has $\Theta(1)$ time complexity.

3. Explain how to add one new key/value pair into an existing Fibonacci heap, and state the big-O cost.

   Create a new node which holds the key-value pair with all other fields set to 0 or null as appropriate. Then insert it into the root list by changing four pointers – the new nodes left and right pointers and two adjacent nodes left and right pointers. This has $\Theta(1)$ time and space complexity.

4. Explain how to form the (destructive) union of two existing Fibonacci heaps, and state the big-O cost.

   Create a new fibonacci heap with `min_node` as the of the smaller of the two existing Fibonacci Heaps `min_node`. Then splice the linked lists of the two Fibonacci Heaps together. If you store the size of the Fibonacci Heap, (which makes the code faster and simpler but is not necessary) then set the new Fibonacci Heaps size equal to the two old Fibonacci Heap's size. Both these operations require constant time and space.

5. Explain how to identify the smallest item in a Fibonacci heap, and state the big-O cost.

   We keep a pointer to the `min_node` in the Fibonacci Heap. So identification of the smallest item in a Fibonacci Heap involves returning a pointer to the `min_node` – or the value of the `min_node` dependent on implementation.

   Removal of the `min_node` in the Fibonacci Heap is more complicated.

   - Find the `min_node`.

     This has $\Theta(1)$ space and time complexity.

   - Promote all of `min_node`'s children into the root list.

     To do this we splice the child list into the root list and change each parent pointer to point to null – or to keep the Fibonacci Heap in a consistent state throughout, insert each node one-by-one into the root list.

     Both have constant space complexity and time complexity $\Theta(d)$ where d is the degree of `min_node` and $d \in O(\lg n)$.

     So the space complexity is $\Theta(1)$ and the time complexity is $O(\lg n)$.

   - Clean up the root list.

     To do this we create an array of pointers of size $d_{\max}$ where $d_{\max}$ is the largest possible degree of a node in the Fibonacci Heap. $d_{\max} \in \Theta(\lg n)$. So the space complexity of this algorithm is $\Theta(\lg n)$.

     For each root in the root list, let d be the degree of the root. Check the d^{th} position in the array.

     If it is null then there is no root of size d in the array. In this case we should set the d^{th} position in the array to point to this root.

     Else we should merge this tree with the tree at position d and check the merged tree against position $d + 1$. We merge the two trees by comparing the roots and inserting tree with the larger root into the child list of the tree with the smaller root.

     This has space complexity $\Theta(\lg n)$ and ammortized time complexity of $\Theta(\lg n)$ – although a worst-case individual time complexity of $O(n)$.

   - Then parse through the new root list to find the `min_node`.

     This has space complexity $\Theta(1)$ and time complexity $\Theta(\lg n)$.

   So `pop_min` has space complexity $\Theta(\lg n)$ and ammortized time complexity $\Theta(\lg n)$.

I implemented a Fibonacci Heap in Java as part of the learning process for this supervision. However the code is (5.5 pages) long so I won't include it.

2 Convex Hulls

1. Describe how to determine whether or not a point is inside a simple plane closed polygon, paying proper attention to awkward cases.

2.1 Ray Tracing

If a point P is inside a closed polygon then any vector from that point to outside the polygon will intersect an odd number of lines.

Firstly, find a point Q which is outside the polygon. An easy way to do this is by considering the point $Q = \begin{pmatrix} x_{\max}.x + \delta \\ x_{\max}.y \end{pmatrix}$ where δ is an arbitrary positive constant.

For each line on the surface of the polygon, check whether the line PQ intersects it.

If the number of edges the PQ intersects is even then P is inside the polygon otherwise P is outside the polygon.

We can check whether $\overrightarrow{AB}$ intersects $\overrightarrow{PQ}$ with the following routine:

Check whether A and B are on opposite sides of the line $\overrightarrow{PQ}$ and whether P and Q are on opposite sides of the line $\overrightarrow{AB}$. If both are true then the lines intersect.

Given the line $\overrightarrow{AB}$ and points P, Q we can check whether A and B are on opposite sides of $\overrightarrow{PQ}$ by working out a line $\overrightarrow{\ell} \perp \overrightarrow{PQ}$. If $A \cdot \overrightarrow{\ell} - P \cdot \overrightarrow{\ell}$ has an opposite sign to $B \cdot \overrightarrow{\ell} - P \cdot \overrightarrow{\ell}$ then the points are on the opposite sides of the line.

We repeat this for the points P, Q and the line $\overrightarrow{AB}$.

2. Describe how, with luck, to exclude large numbers of points from the convex hull of a set of points in the plane, with due consideration of what can go wrong.

   The easiest way to "with luck" remove a large set of points is to get $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ and remove any points in the square formed by those four vertices.

   This works because all four of these vertices are on the convex hull and no point which is bounded by points on the convex hull can possibly be on the convex hull. So all points which are bounded by the square formed by these four points are known not to be on the convex hull. This optimisation is linear and will remove at least half the area of the graph from consideration on the convex hull. On "fairly" uniform graphs this will remove around half of the points in the graph.

   However this has no affect on the expected complexity. The following algorithm does:

Abstract

I extended this idea by reinventing (†) a vectorisable expected $O(n)$ (but $O(nh)$ worst-case) convex-hull algorithm and adding termination conditions if it starts performing worse than $O(n)$. After implementing this algorithm I found that it was able to speed up scipy.spatial.ConvexHull on graphs with a "low" proportion of points near the convex hull and made little difference on non-adversarial graphs.

On non-adversarial graphs the termination conditions are very unlikely to trigger; this means that on non-adversarial graphs the complexity of finding the convex hull is now $O(n + h \lg h)$ expected (although the worst-case is unchanged).

(†) At the time of writing I believed that the algorithm the filtering is based on was original but have since found out it already existed and is called "Quickhull". The adaptation to use it to filter points and everything written here is my own.

Outline

The summary of the filtering algorithm is as follows:

- Split the graph with a line from $x_{\min}$ to $x_{\max}$.
- Find the point with the largest perpendicular distance onto the line and remove all points bounded by it and the two end points. Recurse down each edge and repeat until there are no points in the edge or insufficient points are removed.

Here is a more in-depth outline.

(a) Find the point with the largest x coordinate ($x_{\max}$) and the point with the smallest x coordinate ($x_{\min}$). If these points have the same x coordinate then the graph is degenerate and there is no convex hull.

(b) Draw a line L joining $x_{\max}$ and $x_{\min}$.

(c) Split the graph into those points above L and those points below L.

(d) For those points above the line L:

(e) Find the point P which has the largest projection onto the line perpendicular to L. P is on the convex hull. Remove all points bounded by P and L using the method described above.

(f) Split the points into those to the right of the convex-hull point D we have just found and those to the left of the convex-hull point G. We can combine this calculation with elimination of points bounded by P and L.

(g) If this call has not removed at least 25% of points then mark the next recursive call with kill=True. If this recursive call already had kill=True then do not recurse further. This ensures linear complexity.

(h) Repeat steps (e) - (i) for the points in D considering the line formed by P and the right point of L as L.

(i) Repeat steps (e) - (i) for the points in G considering the line formed by P and the left point of L as L.

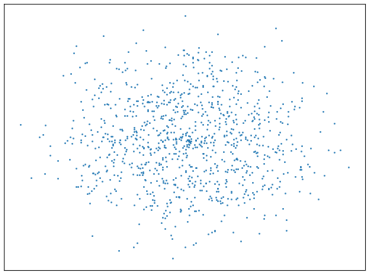
(j) Repeat steps (d) - (i) for those points below the line L between $x_{\min}$ and $x_{\max}$.

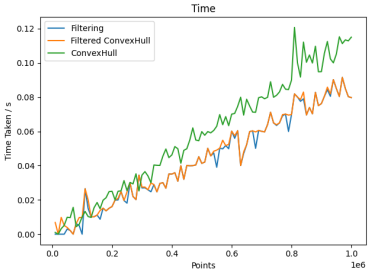
Here are several graphs illustrating three things:

- On graphs with a low proportion of points on the convex hull, this algorithm speeds up even highly optimised implementations.
- On non-adversarial graphs the filtering removes the overwhelming majority of points which are not on the convex hull. This is shown by how much quicker ConvexHull is after the filtering.
- The complexity is linear.

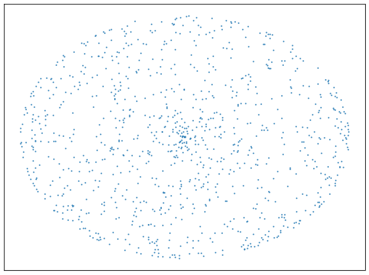
In the following graph "filtering" is the time the point filtering took, "Filtered ConvexHull" is the time taken to filter the points and then call scipy.spatial.ConvexHull and "ConvexHull" is the time taken to run scipy.spatial.ConvexHull without any filtering.

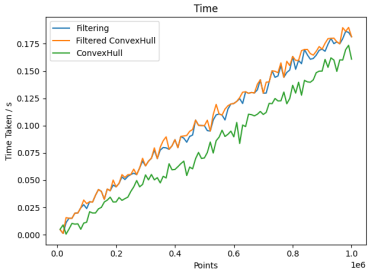
Typical graph with a low proportion of points on the convex hull (random angle with radius r normally distributed $r \sim N(1, 0.5^2)$):



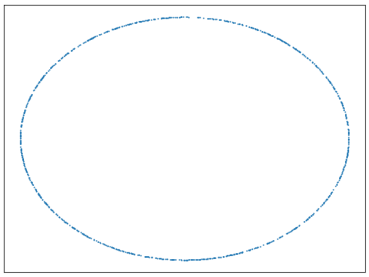


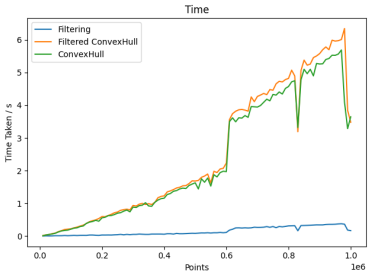
Random uniformly distributed within a circle (high proportion of points on the convex hull):





Here is a degenerate case all the points are on the surface of the circle and so all points are on the convex hull:

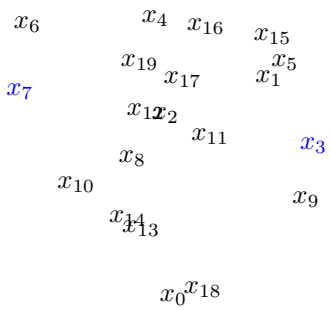




Worked Example

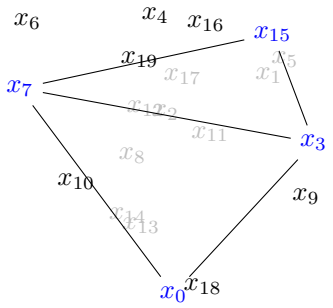
After finding a node is on the convex hull, I will colour it blue. If I find that a node cannot possibly be on the convex hull, I will colour it light gray. I will also perform all the executions at the i^{th} level at once for demonstration purposes – in reality these are not done in parallel.

Start with the graph as below.

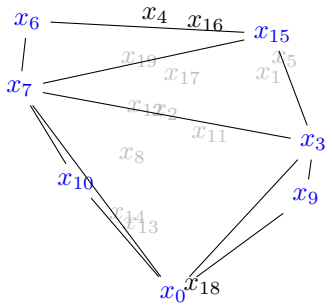


Firstly we join the rightmost and leftmost nodes together. These nodes are guaranteed to be on the convex hull. We should project all the points onto the vector perpendicular to this line and form a triangle from the point which is the greatest distance from this line in each partition.

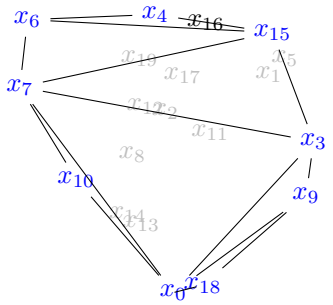
We then note that all the points enclosed by either triangle must not be on the convex hull. So we can remove them.



Next recurse down the new edges formed.



The edges $x_0 \to x_9$, $x_3 \to x_9$ and $x_6 \to x_7$ do not have ≥ 2 points and so we should not recurse down them.



Finally we recurse into the side containing x_{16} and the algorithm terminates. In this specific case (as in most cases), the early-termination condition is not triggered.

Python Implementation

Here is a vectorised python implementation of this algorithm:

```
import numpy as np


def filter_points(points):
```

```
def remove_points(start, end, xpoints, ypoints, perpendicular, kill=False
    ):
    argm = np.argmax(xpoints * perpendicular[0] + ypoints * perpendicular
        [1])
    maxpoint = (xpoints[argm], ypoints[argm])
    leftperp = (start[1] - maxpoint[1], maxpoint[0] - start[0])
    rightperp = (maxpoint[1] - end[1], end[0] - maxpoint[0])
    vleft = np.dot(leftperp, maxpoint)
    vright = np.dot(rightperp, maxpoint)
    left = xpoints * leftperp[0] + ypoints * leftperp[1] > vleft
    left[argm] = False
    right = xpoints * rightperp[0] + ypoints * rightperp[1] > vright
    right[argm] = False
    xleft = xpoints[left]
    xright = xpoints[right]
    lsize = xleft.size
    rsize = xright.size
    if 4 * (lsize + rsize) > 3 * xpoints.size:
        if kill:
            left[argm] = True
            return left | right
        kill = True
    if lsize:
        left[left] = remove_points(start, maxpoint, xleft, ypoints[left],
             leftperp, kill)
    if rsize:
        right[right] = remove_points(maxpoint, end, xright, ypoints[right
            ], rightperp, kill)
    left[argm] = True
    return left | right

xs = points[:, 0]
ys = points[:, 1]
argxmax = np.argmax(xs)
argxmin = np.argmin(xs)
xmax = np.array([xs[argxmax], ys[argxmax]])
xmin = np.array([xs[argxmin], ys[argxmin]])
partition = np.array([xmin[1] - xmax[1], xmax[0] - xmin[0]])
line = np.dot(partition, xmax)
above = xs * partition[0] + ys * partition[1] > line
below = xs * partition[0] + ys * partition[1] < line
if np.any(above):
    above[above] = remove_points(xmin, xmax, xs[above], ys[above],
        partition)
if np.any(below):
    below[below] = remove_points(xmax, xmin, xs[below], ys[below], -
        partition)
above[argxmin] = True
above[argxmax] = True
return points[above | below]
```

Proof of Complexity

I will rigorously prove the complexity of my extension of the algorithm and explain (without proof) the expected complexity of the algorithm it is based on.

In the first iteration, we pass through the list of points once. This means that the complexity of filter is $\Omega(n)$.

Each iteration of the algorithm makes a constant number of passes through the list of points. So each recursive call is linear time.

The algorithm will terminate by the "kill" mechanic if two calls fail to remove at least 25% of the points in the graph. So we can have two n passes through the list and the rest will form the recurrence relation:

$$T(n) = T\left(\frac{3n}{4}\right) + O(n)$$

This forms the geometric sequence:

$$\sum_{k=0}^{n} n\frac{3}{4}^{k} = \frac{n}{1-\frac{3}{4}} = 4n$$

If we add in the original $2n$, then we have no more than $6n$ which is $O(n)$. So the algorithm is $O(n)$. Since we have now bounded the time complexity from above and from below, we have a tight bound of $\Theta(n)$. So the complexity is $\Theta(n)$. Which is also supported by the graphs.

The expected complexity of Quickhull is $O(n)$.

At each iteration Quickhull gets a tighter bound on the area which contains points. Then disregards all points in a section of it. The proportion of the area which points could lie in that we eliminate is dependant on the distribution and number of points. On random distributions and non-adversarial distributions we are expected to remove a lot of points at each iteration. For example on uniform graphs if we are passed n points, after one recursion we are expected to have $\frac{1}{n+1}\left(n^{\frac{3}{2}} - \frac{n^2}{n+1} - 1\right) \approx \sqrt{n}$ points remaining.

3. Describe how to compute economically the convex hull of the points that are left after the measures you described above.

   I will describe Graham's Scan. Here are the reasons why I chose Graham's Scan over Jarvis March:

   - If the filtering algorithm is effective then the number of points passed to the convex hull algorithm is $\approx h$. In this case Graham's Scan has a complexity of $\Theta(h \lg h)$ while Jarvis March has a complexity of $\Theta(h^2)$.
   - There are two reasons the filtering algorithm could be inneffective:

   Case 1: $h \approx n$ and so there are not many points to remove. In this case Graham's Scan is $\Theta(n \lg n)$ and Jarvis March is $\Theta(n^2)$. So Graham's Scan is asymtotically better.

   Case 2: There is a small proportion of points on the convex hull and a very large proportion of points near the convex hull. While possible it is unlikely this to be the case in a large graph such that $h \in o(\lg n)$ – which would be required for Jarvis March to beat Graham's Scan.

   Graham's Scan is as follows:

   - Find coordinates inside the convex hull. The easiest way to do this is to find the points with the largest x coordinates and smallest x coordinates and take the mean of these coordinates.
   - Sort all points by their angle in polar coordinates.
   - Pass through the list of vertices anticlockwise. Any points which have outer angles $\leq 180°$ are not on the convex hull.
   - After passing through the list, continue checking the first vertices knowing their predecessor. Any vertex can only be added to the convex hull once and removed from the convex hull once. So the asymtotic cost of this search is $O(n)$. So Graham's Scan is dominated by the cost of sorting the vertices.

   A more optimal implementation would change the base case of the filtering algorithm to call Graham's Scan rather than terminate.