Data structures project, Implementation document

Heikki Haapala and Aleksi Markkanen
Student numbers 014090190 and 013126382
10 pages

March 4, 2013

Contents

1 Structure of the Program

We divided the code into five different *Java* source packages: algorithms, comparators, datastructures, graphics and main.

Package *algorithms* contains an interface `Algorithm` which the algorithms will implement. Only one method is specified, namely `useAlgorithm`. All other methods are declared `private`.

Comparators contains only one class, `AngleComparator`. It implements a comparator for the `Point2D.Double` class, in which the points are sorted in ascending order by their polar angles. This method of sorting is used by the *Gift-wrapping algorithm*.

Package *datastructures* contains our implementation of the linked list. We implemented methods to add and remove points, to sort the list and to check wether a given object is in the list. The list also knows its length.

The *graphics* package contains code necessary to draw the results on the screen. In the *main* package we placed all other program logic, i.e. input parameters, reading and writing to and from files and so on.

2 Attained Computational Complexity

In the definitions document, we aimed for the best possible time complexities for our algorithms of choice. However, for simplicity, we decided to settle for $\mathcal{O}(n)$ space complexity. This way, we could generate a new linked list for the points of the convex hull. This also reduced the complexity of our algorithms, especially the Quickhull algorithm.

2.1 $\mathcal{O}$-analysis of the pseudocode

2.1.1 QuickHull

`QuickHull(`S`)`
Data: List S of points on a plane
Result: List H of points that form the convex hull of S

Find the points A and B that have the minimum and maximum values for x-coordinates, respectively. These points are bound to be a part of the convex hull.
Divide S into S_1 and S_2 so that points in S_1 and S_2 lie on the opposite sides of the line AB.
$H \leftarrow \{\}$.
$H = \text{FindHull}(S_1, A, B) \cup \text{FindHull}(S_2, B, A)$

Algorithm 1: Core method

`FindHull(`S, A, B`)`
Data: List S of points on a plane, Point A, Point B
Result: List H of points that form the convex hull of S and are on the right of the line AB

if S *is empty* **then**
  return A, B
end
Find $C = \text{argmax dist}(AB, C)$
Divide S into S_1 and S_2 so that points in S_1 lie on the right side of AC and points in S_2 lie on the right side of BC. The rest of the points can be discarded.
return FindHull$(S1, P, C) \cup$ FindHull$(S2, C, Q)$.

Algorithm 2: FindHull method

At first, it would seem that the time complexity of the recursive method is of order $\mathcal{O}(n^2)$. While this is true for some datasets, usually the recursive method discards many points with each iteration and thus brings the *average-case time complexity* down to $\mathcal{O}(n \log n)$.

2.1.2 Gift-wrapping

The following pseudocode specifies the Jarvis' march algorithm[2].

```
jarvis(S)
pointOnHull = leftmost point in S
i = 0
repeat
 P[i] = pointOnHull
 endpoint = S[0]         // initial endpoint for a candidate edge on the hull
  for j from 1 to |S|-1
   if (endpoint == pointOnHull) or (S[j] is left of line from P[i] to endpoint)
     endpoint = S[j]   // found greater left turn, update endpoint
  endfor
  i = i+1
  pointOnHull = endpoint
until endpoint == P[0]      // wrapped around to first hull point
```

The inner loop of the pseudocode is run for each input point. Thus, its time complexity is of order $\mathcal{O}(n)$. However, the outer loop is iterated over the hull points. If there are h hull points, the total time complexity is $\mathcal{O}(nh)$. This is a so-called *output-sensitive* algorithm.

2.1.3 Graham scan

Graham scan is given by the following pseudocode[3]:

We begin by defining an auxilliary function.

```
function ccw(p1, p2, p3):
    return (p2.x - p1.x)*(p3.y - p1.y) - (p2.y - p1.y)*(p3.x - p1.x)
```

Now we can write the graham scan in a simpler form.

```
let N           = number of points
let points[N+1] = the array of points
swap points[1] with the point with the lowest y-coordinate
sort points by polar angle with points[1]

# We want points[0] to be a sentinel point that will stop the loop.
let points[0] = points[N]
```

```
# M will denote the number of points on the convex hull.
let M = 1
for i = 2 to N:
    # Find next valid point on convex hull.
    while ccw(points[M-1], points[M], points[i]) <= 0:
          if M > 1:
                  M -= 1
          # All points are collinear
          else if i == N:
                  break
          else
                  i += 1

    # Update M and swap points[i] to the correct place.
    M += 1
    swap points[M] with points[i]
```

The actual algorithm has the time complexity $\mathcal{O}(n)$, but since it is necessary to sort the input first, it is dominated by the time complexity $\mathcal{O}(n \log n)$ of our *Mergesort* implementation.

3 Comparing the Different Algorithms

We compared the performance of our algorithms using two test cases. First, we had a hardest case dataset for which all of the input points were part of the convex hull. This was achieved by taking evenly spaced numbers using the *Octave* command `linspace` and applying sin and cosin functions to them. This produced a set of points that lie on the unit circle in the plane.

Other test case consisted of generating random points from a Gaussian distribution. This in our mind represents a ”average case” since Gaussian distributions are quite prevalent in natural sciences.

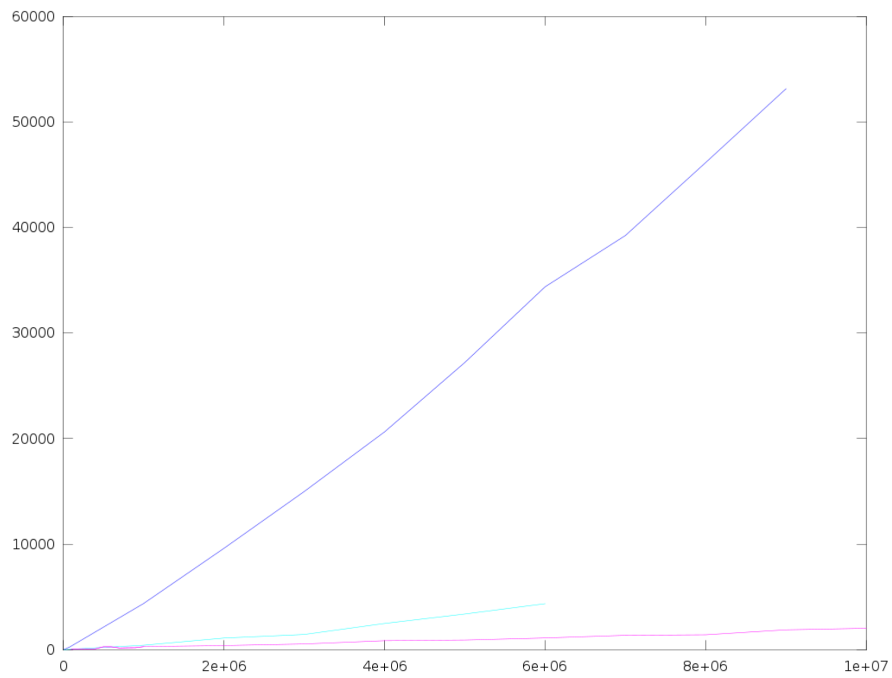

Figure 1: Average case performance

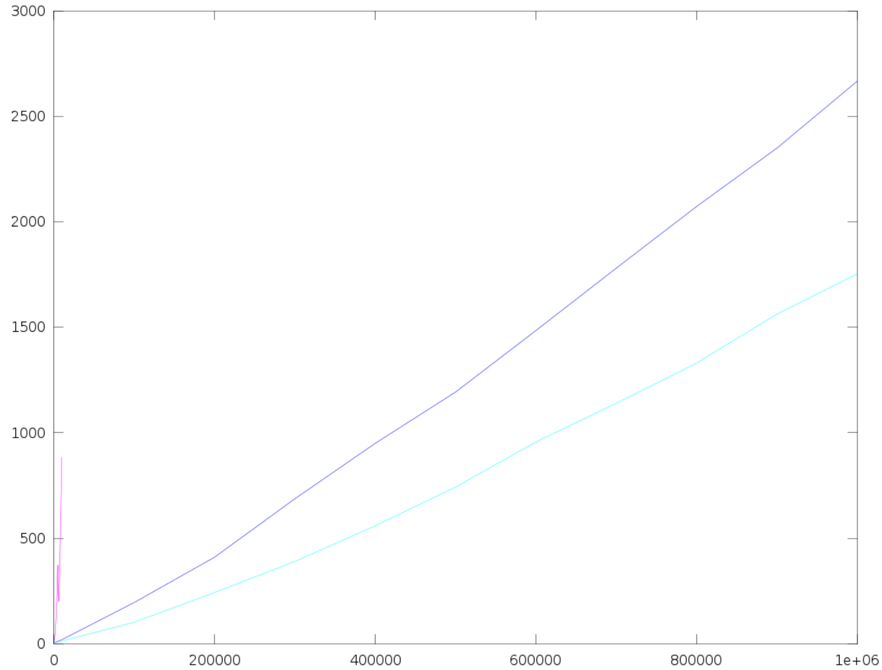

Figure 2: Worst case performance

In the figures, the Quickhull algorithm is plotted in cyan, gift-wrapping in magenta and Graham scan in blue.

X-axis is the amount of points and y-axis is the time in milliseconds. Quickhull could not be performed for large datasets because of the Java stack size.

References

[1] Convex hull algorithms,
Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Convex_hull_algorithms

[2] Gift Wrapping Algorithm,
Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Gift_wrapping_algorithm

[3] Graham Scan,
Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Graham_scan