Heavens' Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Algorithms Analysis & Design Sessional

Convex Hull, Matrix Multiplication

Submitted by

Kaif Ahmed Khan

Roll: 2103163

Submitted to

A. F. M. Minhazur Rahman
Assistant Professor

November 16, 2024

Contents

List of Listings

List of Figures

1 Convex Hull

1.1 Problem Statement

Implement and compare *Graham Scan* and *Quick Hull* algorithm based on various input size on randomly generated points. The comparison metric should be the execution time of each convex hull finding algorithm.

1.2 Code

Listing 1: *graham_scan.cpp*

```cpp
#include <algorithm>
#include <bits/stdc++.h>
#include <chrono>
#include <ctime>
#include <stack>
#include <utility>
#include <vector>
using namespace std;
using namespace std::chrono;

typedef pair<int, int> Point;
typedef vector<Point> vpii;
stack<Point> S;
vpii P; // points are stored as (y,x)
vpii sorted_points;
vpii hull_pts;

int get_y(const Point p) { return p.first; }
int get_x(const Point p) { return p.second; }
Point get_min(const vpii P) { return *min_element(P.begin(), P.end()); }

bool non_left_turn(const Point p1, const Point p2, Point p) {
  int cross_prod = (get_x(p2) - get_x(p1)) * (get_y(p) - get_y(p1)) -
                   (get_y(p2) - get_y(p1)) * (get_x(p) - get_x(p1));
  if (cross_prod > 0)
    return false;
  else if (cross_prod < 0)
    return true;
  return false;
}

Point next_to_top(stack<Point> &s) {
  Point top = s.top();
  s.pop();
  Point next_top = s.top();
  s.push(top);
  return next_top;
}

void read_points_from_file(const string file_name, vpii &points);
```

```cpp
void print_points(const vpii points, int n);

int main() {
  read_points_from_file("points.txt", P);
  Point p0 = get_min(P); // reference point
  auto order = [&](Point p1, Point p2) -> bool {
    int dx1 = get_x(p1) - get_x(p0), dy1 = get_y(p1) - get_y(p0);
    int dx2 = get_x(p2) - get_x(p0), dy2 = get_y(p2) - get_y(p0);

    int cross = dx1 * dy2 - dy1 * dx2;
    if (cross == 0)
      return (dx1 * dx1 + dy1 * dy1) < (dx2 * dx2 + dy2 * dy2);

    return cross > 0;
  };

  sorted_points = P;
  int m = sorted_points.size();
  // print_points(sorted_points, m);

  auto st = high_resolution_clock::now(); // clock start

  sort(sorted_points.begin(), sorted_points.end(), order);
  S.push(p0);
  S.push(sorted_points[1]);
  S.push(sorted_points[2]);

  for (int i = 3; i < m; i++) {
    Point p_i = sorted_points[i];
    while (S.size() > 1 && non_left_turn(next_to_top(S), S.top(), p_i)) {
      S.pop();
    }
    S.push(p_i);
  }

  while (!S.empty()) {
    Point p = S.top();
    hull_pts.push_back(p);
    S.pop();
  }
  auto et = high_resolution_clock::now(); // clock end
  double time_taken =
      chrono::duration_cast<chrono::nanoseconds>(et - st).count();
  time_taken *= 1e-6;
  cout << "Convex hull points using Graham Scan:" << endl;
  // print_points(hull_pts, hull_pts.size());
  cout << "Time taken for Graham Scan: " << time_taken << " ms" << endl;
  cout << "No. of Points: " << P.size() << endl;
}

void read_points_from_file(const string file_name, vpii &points) {
```

```
  ifstream inputFile(file_name);
  if (!inputFile.is_open()) {
    cerr << "Error opening the file!" << endl;
    exit(1);
  }
  string line;
  while (getline(inputFile, line)) {
    stringstream ss(line);
    int x, y;
    ss >> x >> y;
    points.push_back(make_pair(y, x));
  }
  inputFile.close();
}

void print_points(const vpii points, int n) {
  cout << "X = [ ";
  for (int i = 0; i < n; i++) {
    cout << get_x(points[i]) << ", ";
  }
  cout << get_x(points[0]) << " ]" << endl;
  cout << "Y = [ ";
  for (int i = 0; i < n; i++) {
    cout << get_y(points[i]) << ", ";
  }
  cout << get_y(points[0]) << " ]" << endl;
}
```

Listing 2: *quick_hull.cpp*

```
#include <algorithm>
#include <bits/stdc++.h>
#include <chrono>
#include <utility>
#include <vector>
using namespace std;
using namespace std::chrono;

typedef pair<int, int> Point;
typedef vector<Point> vpii;

vpii hull_points;

bool order(Point p1, Point p2) {
  Point ref = hull_points[0];
  int dx1 = p1.first - ref.first, dy1 = p1.second - ref.second;
  int dx2 = p2.first - ref.first, dy2 = p2.second - ref.second;

  int cross = dx1 * dy2 - dy1 * dx2;

  if (cross == 0)
    return (dx1 * dx1 + dy1 * dy1) < (dx2 * dx2 + dy2 * dy2);

```

```
  return cross > 0;
}

int get_side(Point p1, Point p2, Point p) {
  int cross_prod = (p2.first - p1.first) * (p.second - p1.second) -
                   (p2.second - p1.second) * (p.first - p1.first);
  if (cross_prod > 0)
    return 1;
  else if (cross_prod < 0)
    return -1;
  return 0;
}

Point get_min(const vpii pts) { return *min_element(pts.begin(),
↪  pts.end()); }
Point get_max(const vpii pts) { return *max_element(pts.begin(),
↪  pts.end()); }

int dist(Point p1, Point p2, Point p) {
  return abs(((p2.first - p1.first) * (p1.second - p.second)) -
             ((p1.first - p.first) * (p2.second - p1.second)));
}

void quick_hull(const vpii pts, int n, Point p1, Point p2, int side) {
  // find point with max dist
  int max_dist = 0;
  int max_index = -1;
  for (int i = 0; i < n; i++) {
    int temp_dist = dist(p1, p2, pts[i]);
    if (get_side(p1, p2, pts[i]) == side && temp_dist > max_dist) {
      max_index = i;
      max_dist = temp_dist;
    }
  }
  // push 2 extreme points
  if (max_index == -1) {
    hull_points.push_back(p1);
    hull_points.push_back(p2);
    return;
  }

  quick_hull(pts, n, pts[max_index], p1, -get_side(pts[max_index], p1, p2));
  quick_hull(pts, n, pts[max_index], p2, -get_side(pts[max_index], p2, p1));
}

void read_points_from_file(const string file_name, vpii &points);
void print_points(vpii points, int n);

int main() {
  vpii P;
  read_points_from_file("points.txt", P);
```

```cpp
  int n = P.size();
  // print_points(P, n);

  auto st = high_resolution_clock::now(); // clock start

  Point min = get_min(P);
  Point max = get_max(P);

  quick_hull(P, n, min, max, 1);
  quick_hull(P, n, min, max, -1);

  auto et = high_resolution_clock::now(); // clock end
  double time_taken =
      chrono::duration_cast<chrono::nanoseconds>(et - st).count();
  time_taken *= 1e-6;

  sort(hull_points.begin(), hull_points.end(), order);
  hull_points.erase(unique(hull_points.begin(), hull_points.end()),
                    hull_points.end());

  cout << "Convex hull points using Quick Hull:" << endl;
  // print_points(hull_points, hull_points.size());
  cout << "Time taken for Quick hull: " << time_taken << " ms" << endl;
  cout << "No. of Points: " << n << endl;
}

void read_points_from_file(const string file_name, vpii &points) {
  ifstream inputFile(file_name);
  if (!inputFile.is_open()) {
    cerr << "Error opening the file!" << endl;
    exit(1);
  }
  string line;
  while (getline(inputFile, line)) {
    stringstream ss(line);
    int x, y;
    ss >> x >> y;
    points.push_back(make_pair(x, y));
  }
  inputFile.close();
}

void print_points(const vpii points, int n) {
  cout << "X = [ ";
  for (int i = 0; i < n; i++) {
    cout << points[i].first << ", ";
  }
  cout << points[0].first << " ]" << endl;
  cout << "Y = [ ";
  for (int i = 0; i < n; i++) {
    cout << points[i].second << ", ";
```

```
    }
    cout << points[0].second << " ]" << endl;
}
```

Listing 3: *Makefile*

```
CC=g++

convex: points graham quick

points: generate_points.cpp
        $(CC) $^ -o points.out
        ./points.out
graham:         graham_scan.cpp
        $(CC) $^ -o graham_scan.out
        ./graham_scan.out

quick: quickhull.cpp
        $(CC) $^ -o quickhull.out
        ./quickhull.out

clean:
        rm *.out
```

1.3 Output

```
cse-22/algorithm-lab on  master [!?] took 44s
❯ make graham
g++ graham_scan.cpp -o graham_scan.out
./graham_scan.out
Convex hull points using Graham Scan:
Time taken for Graham Scan: 2.73556 ms
No. of Points: 5000

cse-22/algorithm-lab on  master [!?]
❯ make quick
g++ quickhull.cpp -o quickhull.out
./quickhull.out
Convex hull points using Quick Hull:
Time taken for Quick hull: 1.20198 ms
No. of Points: 5000
```

(a) *Execution time for n=5000*

```
cse-22/algorithm-lab on  master [!?] took 4s
❯ make graham
g++ graham_scan.cpp -o graham_scan.out
./graham_scan.out
Convex hull points using Graham Scan:
Time taken for Graham Scan: 5.5254 ms
No. of Points: 10000

cse-22/algorithm-lab on  master [!?]
❯ make quick
g++ quickhull.cpp -o quickhull.out
./quickhull.out
Convex hull points using Quick Hull:
Time taken for Quick hull: 2.73906 ms
No. of Points: 10000
```

(b) *Execution time for n=10000*

```
cse-22/algorithm-lab on  master [!?] took 5s
❯ make graham
g++ graham_scan.cpp -o graham_scan.out
./graham_scan.out
Convex hull points using Graham Scan:
Time taken for Graham Scan: 29.7845 ms
No. of Points: 50000

cse-22/algorithm-lab on  master [!?]
❯ make quick
g++ quickhull.cpp -o quickhull.out
./quickhull.out
Convex hull points using Quick Hull:
Time taken for Quick hull: 8.96497 ms
No. of Points: 50000
```

(c) *Execution time for n=50000*

```
cse-22/algorithm-lab on  master [!?] took 5s
❯ make quick
g++ quickhull.cpp -o quickhull.out
./quickhull.out
Convex hull points using Quick Hull:
Time taken for Quick hull: 17.9454 ms
No. of Points: 100000

cse-22/algorithm-lab on  master [!?]
❯ make graham
g++ graham_scan.cpp -o graham_scan.out
./graham_scan.out
Convex hull points using Graham Scan:
Time taken for Graham Scan: 61.4065 ms
```

(d) *Execution time for n=100000*

Figure 1: *Convex hull algorithm execution time*

Summary of Execution time

Table 1: *Comparison table for Graham Scan & Quick hull algorithm*

	Execution time (ms)	
Input Size	Graham Scan	Quick Hull
1000	0.460258	0.413883
5000	2.73556	1.20198
10,000	5.5254	2.73906
50,000	29.7845	8.96497
1,00,000	61.4065	17.9454

1.4 Analysis & Discussion

Graham Scan is faster than quick hull for very number of points. But with increasing number of points quick hull is significantly efficient than graham scan. This is because Graham scan needs to sort the points before doing its calculation. And this is more costly. In case of quick hull, there is no need to sort the points, rather it can find the points by divide and conquer approach, which significantly decreases the execution time for quick hull.

2 Matrix Multiplication

2.1 Problem Statement

Implement and compare *Graham Scan* and *Quick Hull* algorithm based on various input size on randomly generated points. The comparison metric should be the execution time of each convex hull finding algorithm.