Lab Manual

Course: Computer Algorithms Sessional
Department of CSE, AMUST

Task 1: Convex Hull using Brute Force Algorithm

1. Objective

To implement the Convex Hull problem using the brute force approach in C++. Students will learn to understand the geometric concept of convex hulls and how to identify extreme points among a set of points.

2. Theory

The convex hull of a set of points is the smallest convex polygon that contains all the points. In the brute force approach, we check every pair of points and determine whether all other points lie on one side of the line formed by the pair. If they do, that edge is part of the convex hull.

3. Algorithm Steps

1. Take a set of points in 2D space.
2. For every pair of points (p1, p2), form a line.
3. Check whether all other points lie on the same side of the line.
4. If so, (p1, p2) is part of the convex hull.
5. Repeat the process for all pairs of points.
6. Store and display the set of convex hull edges.

4. Sample Code in C++

```
#include <iostream>
#include <vector>
using namespace std;

struct Point {
   int x, y;
};
```

```
int direction(Point a, Point b, Point c) {
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

void convexHull(vector<Point>& points) {
    int n = points.size();
    cout << "Convex Hull Edges:\n";
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int pos = 0, neg = 0;
            for (int k = 0; k < n; k++) {
                if (k == i || k == j) continue;
                int d = direction(points[i], points[j], points[k]);
                if (d > 0) pos++;
                else if (d < 0) neg++;
            }
            if (pos == 0 || neg == 0)
                cout << "(" << points[i].x << "," << points[i].y << ") - ("
                     << points[j].x << "," << points[j].y << ")\n";
        }
    }
}

int main() {
    vector<Point> points = {{0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3}};
    convexHull(points);
    return 0;
}
```

5. Sample Input/Output

Input: A list of points: (0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3)

Output: Edges that form the convex hull:

```
(0,3) - (3,3)
(3,3) - (3,0)
(3,0) - (0,0)
(0,0) - (0,3)
```

6. Observation Table

Students should test different sets of input and record the convex hull results.

7. Viva Questions

- What is a convex hull?
- What is the time complexity of the brute force approach?
- Why do we check the direction of points?
- Can all points lie on the convex hull?

Task 2: Convex Hull using Graham's Scan Algorithm

1. Objective

To implement the Convex Hull problem using Graham's Scan algorithm in C++. This algorithm is efficient and makes use of sorting and orientation concepts to find the convex hull.

2. Theory

Graham's Scan is an efficient algorithm to compute the convex hull of a set of 2D points. It works in O(n log n) time and is based on sorting the points by polar angle and then processing them to build the hull.

3. Algorithm Steps

1. Find the point with the lowest y-coordinate (break ties using x-coordinate). This is the starting point (pivot).
2. Sort all the other points based on the polar angle with respect to the pivot.
3. Traverse the sorted points and maintain a stack to determine left turns (using orientation test).
4. If a right turn is detected, pop the top of the stack.
5. Repeat until all points are processed. The stack will contain the convex hull vertices.

4. Sample Code in C++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
using namespace std;

struct Point {
   int x, y;
};

Point p0;

// A utility function to find the orientation of ordered triplet (p, q, r)
// Returns 0 if colinear, 1 if clockwise, 2 if counterclockwise
int orientation(Point p, Point q, Point r) {
   int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
   if (val == 0) return 0;
   return (val > 0) ? 1 : 2;
}

// Distance squared
int distSq(Point p1, Point p2) {
   return (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y);
}

// Compare function for sorting by polar angle
bool compare(Point p1, Point p2) {
   int o = orientation(p0, p1, p2);
   if (o == 0)
      return distSq(p0, p1) < distSq(p0, p2);
   return (o == 2);
}

void grahamScan(vector<Point>& points) {
   int n = points.size();

   // Find the bottom-most point
   int ymin = points[0].y, minIndex = 0;
   for (int i = 1; i < n; i++) {
      if ((points[i].y < ymin) || (points[i].y == ymin && points[i].x < points[minIndex].x)) {
         ymin = points[i].y;
```

```
            minIndex = i;
        }
    }
    swap(points[0], points[minIndex]);
    p0 = points[0];

    sort(points.begin() + 1, points.end(), compare);

    stack<Point> hull;
    hull.push(points[0]);
    hull.push(points[1]);
    hull.push(points[2]);

    for (int i = 3; i < n; i++) {
        while (hull.size() > 1) {
            Point top = hull.top();
            hull.pop();
            Point nextToTop = hull.top();
            if (orientation(nextToTop, top, points[i]) != 2)
                continue;
            else {
                hull.push(top);
                break;
            }
        }
        hull.push(points[i]);
    }

    cout << "Convex Hull Points (Graham's Scan):\n";
    while (!hull.empty()) {
        Point p = hull.top();
        cout << "(" << p.x << ", " << p.y << ")\n";
        hull.pop();
    }
}

int main() {
    vector<Point> points = {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    grahamScan(points);
    return 0;
}
```

5. Sample Input/Output

Input: Points = {(0,3), (1,1), (2,2), (4,4), (0,0), (1,2), (3,1), (3,3)}

Output: Points that make up the convex hull, printed in counterclockwise order.

6. Observation Table

Record the convex hull result for different sets of points.

7. Viva Questions

- What is the significance of the pivot point in Graham's Scan?
- Why do we sort points based on polar angle?
- What is the time complexity of Graham's Scan?
- How do we check for a left or right turn?

Task 3: Convex Hull using QuickHull Algorithm

1. Objective

To implement the Convex Hull problem using the QuickHull algorithm in C++. QuickHull is a divide-and-conquer algorithm useful for understanding recursive problem-solving in computational geometry.

2. Theory

QuickHull is a computational geometry algorithm to find the convex hull of a set of points.
It operates similarly to QuickSort by recursively finding the outermost points and eliminating inner points that cannot be part of the convex hull.
Time complexity is expected O(n log n), though it may degrade to $O(n^2)$ in the worst case.

3. Algorithm Steps

1. Find the points with the minimum and maximum x-coordinates; these form the initial segment.
2. Divide the remaining points into two groups: left and right of this segment.
3. For each group, find the point farthest from the segment.
4. Form a triangle using this farthest point and the original segment.
5. Recursively find the farthest point from the triangle sides.

6. When no more points are found outside the triangle, the hull is complete.

4. Sample Code in C++

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

struct Point {
    int x, y;
};

// Function to find distance of point C from line AB
int distance(Point A, Point B, Point C) {
    return abs((C.y - A.y) * B.x - (C.x - A.x) * B.y + C.x * A.y - C.y * A.x);
}

// Determine the side of point P with respect to line AB
int findSide(Point A, Point B, Point P) {
    int val = (P.y - A.y) * (B.x - A.x) - (B.y - A.y) * (P.x - A.x);
    if (val > 0) return 1;
    if (val < 0) return -1;
    return 0;
}

// Recursive QuickHull function
void quickHull(vector<Point>& points, Point A, Point B, int side, vector<Point>& hull) {
    int index = -1;
    int max_dist = 0;

    for (int i = 0; i < points.size(); i++) {
        int temp = distance(A, B, points[i]);
        if (findSide(A, B, points[i]) == side && temp > max_dist) {
            index = i;
            max_dist = temp;
        }
    }

    if (index == -1) {
        hull.push_back(A);
        hull.push_back(B);
        return;
    }

    quickHull(points, points[index], A, -findSide(points[index], A, B), hull);
    quickHull(points, points[index], B, -findSide(points[index], B, A), hull);
}

// Driver function
void findConvexHull(vector<Point>& points) {
    if (points.size() < 3) {
        cout << "Convex hull not possible\\n";
        return;
    }

    int min_x = 0, max_x = 0;
    for (int i = 1; i < points.size(); i++) {
        if (points[i].x < points[min_x].x) min_x = i;
        if (points[i].x > points[max_x].x) max_x = i;
    }

    vector<Point> hull;
    quickHull(points, points[min_x], points[max_x], 1, hull);
    quickHull(points, points[min_x], points[max_x], -1, hull);

    cout << "Convex Hull Points (QuickHull):\\n";
    for (auto& p : hull) {
        cout << "(" << p.x << ", " << p.y << ")\\n";
    }
}
```

```
int main() {
   vector<Point> points = {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};
   findConvexHull(points);
   return 0;
}
```

5. Sample Input/Output

Input: Points = {(0,3), (1,1), (2,2), (4,4), (0,0), (1,2), (3,1), (3,3)}

Output: Points that make up the convex hull, printed in counterclockwise order.

6. Observation Table

Record the convex hull result for different sets of points.

7. Viva Questions

- What is the QuickHull algorithm?
- How does it differ from Graham's Scan?
- What is the time complexity of QuickHull in average and worst case?
- What is the geometric meaning of "farthest from a line"?
- How is recursion applied in QuickHull?