TIA Project Report
on

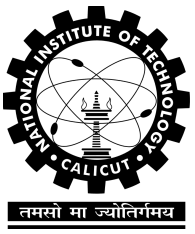
HEURISTIC BASED APPROACH FOR GRAHAM'S SCAN AND JARVIS'S MARCH ALGORITHM

Submitted in partial fulfillment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
B160345CS	AATHIL TA



Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India – 673 601

Monsoon Semester 2018

Department of Computer Science and Engineering

National Institute of Technology Calicut

Certificate

This is to certify that this is a bonafide record of the project presented by the students whose names are given below during Winter Semester 2018 in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Roll No	Names of Students
B160345CS	AATHIL TA

Dr.Subhasree M
(Course Coordinator)

Date:

Abstract

Virtual reality techniques have proved their importance in almost every field of knowledge, particularly in medical and architecture. Convex hull is an application of virtual reality which is used to draw the boundary of some object inside an image.This report dives in to an heuristic algorithm for convex hulls. The method is based on two already existing convex hull algorithms i.e. quick hull and grahams Scan algorithm. The proposed technique is an attempt to remove the deficiencies in the two above mentioned techniques of the convex hull.

Contents

Chapter 1

Objective

We have attempted to implement a heuristic for both graham's scan and jarvis's march.our aim is to neglect the points that come inside the convex to have better performance on existing nlogn algorithm

Chapter 2

Introduction

2.1 What is Convex hull?

In mathematics, the convex hull or convex envelope or convex closure of a set X of points in the Euclidean plane or in a Euclidean space (or, more generally, in an affine space over the reals) is the smallest convex set that contains X. For instance, when X is a bounded subset of the plane, the convex hull may be visualized as the shape enclosed by a rubber band stretched around X. Formally, the convex hull may be defined either as the intersection of all convex sets containing X, or as the set of all convex combinations of points in X. With the latter definition, convex hulls may be extended from Euclidean spaces to arbitrary real vector spaces,they may also be generalized further, to oriented matroids. Convexity: A set S is convex if x belong y and y belong S implies that the segment xy subset S . This can be taken as the primary definition of convexity. Note that this definition does not specify any particular dimensions for the points, whether is connected, bounded, unbounded, closed or open.

2.2 Importance of Convex hull

A few applications of convex hull are:

- Collision avoidance: If the convex hull of a car avoids collision with obstacles then so does the car. Since the computation of paths that avoid collision is much easier with a convex car, then it is often used to plan paths.
- Smallest box: The smallest area rectangle that encloses a polygon has at least one side flush with the convex hull of the polygon, and so the

hull is computed at the first step of minimum rectangle algorithms. Similarly, finding the smallest three-dimensional box surrounding an object depends on the 3D-convex hull.

- Shape analysis: Shapes may be classified for the purposes of matching by their "convex deficiency trees", structures that depend for their computation on convex hulls.

- convex relaxations: this is a way to find the 'closest' convex problem to a non-convex problem you are attempting to solve. While I could define this formally, I think a simple picture might be more interesting.

2.3 Convex hull Algorithms

In computational geometry, numerous algorithms are proposed for computing the convex hull of a finite set of points, with various computational complexities. Computing the convex hull means that a non-ambiguous and efficient representation of the required convex shape is constructed. The complexity of the corresponding algorithms is usually in terms of n the, estimated number of input points, and sometimes also in terms of h, the number of points on the convex hull.As of now we have plenty of algorithm,but in this report iam applying heuristic to two of convex hull algorithm,namely Graham's scan jarvis's march aka Gift wrapping Algorithms

- GRAHAM'S SCAN

- JARVIS MARCH

Chapter 3

Heuristic Approach

3.1 Implementation

The idea is to exclude as many points inside convex hull,because both algorithms ie, Grahams scan and jarvis march has running time directly propotional to number of points in plane.our aim to use preprocessing rules and reduce number of points without compromising the result.

3.2 Preprocessing method

The heuristic is based on the fact that the final convex polygon will contain the convex polygon formed by lowest and highest x and y coordinates.due to this feature of hull we can neglect points inside smaller polygon for hull calculation thus improving our running time

3.3 Improvement over existing algorithm

We can improve this algorithm by adding 4 more point and making it polygon with 8 sides. It is based on the efficient convex hull algorithm by Selim Akl and G. T. Toussaint, 1978. The idea is to quickly exclude many points that would not be part of the convex hull anyway. This method is based on the following idea. Find the two points with the lowest and highest x-coordinates, and the two points with the lowest and highest y-coordinates. (Each of these operations takes O(n).) These four points form a convex quadrilateral, and all points that lie in this quadrilateral (except for the four initially chosen vertices) are not part of the convex hull. Finding all of these points that lie in this quadrilateral is also O(n), and thus, the entire operation is O(n).

Optionally, the points with smallest and largest sums of x- and y-coordinates as well as those with smallest and largest differences of x- and y-coordinates can also be added to the quadrilateral, thus forming an irregular convex octagon, whose insides can be safely discarded. If the points are random variables, then for a narrow but commonly encountered class of probability density functions, this throw-away pre-processing step will make a convex hull algorithm run in linear expected time, even if the worst-case complexity of the convex hull algorithm is quadratic in

3.4 Source Code: GRAHAM'S SCAN

```
import sys
import numpy as np
import matplotlib.pyplot as plt

# Function to know if we have a CCW turn
def RightTurn(p1, p2, p3):
        if (p3[1]-p1[1])*(p2[0]-p1[0]) >= (p2[1]-p1[1])*(p3
            [0]-p1[0]):
                return False
        return True

# Main algorithm:
def GrahamScan(P):
        print ("before sort")
        for i in range(7):
           print (P[i])
        P.sort()
        print ("after sort")                    # Sort the
            set of points
        for i in range(7):
           print (P[i])
        print ("l upper")
        L_upper = [P[0], P[1]]           # Initialize upper
            part
        # Compute the upper part of the hull
        for i in range(2,len(P)):
                L_upper.append(P[i])
                #print (L_upper[0])
                while len(L_upper) > 2 and not RightTurn(
                    L_upper[-1],L_upper[-2],L_upper[-3]):
                        del L_upper[-2]
        L_lower = [P[-1], P[-2]]         # Initialize the
            lower part
        # Compute the lower part of the hull
        for i in range(len(P)-3,-1,-1):
```

```
                L_lower.append(P[i])
                while len(L_lower) > 2 and not RightTurn(
                    L_lower[-1],L_lower[-2],L_lower[-3]):
                        del L_lower[-2]
        del L_lower[0]

        del L_lower[-1]
        L = L_upper + L_lower           # Build the full
            hull
        return np.array(L)


def main():
        try:
                N = int(sys.argv[1])
        except:
                N = int(input("Introduce N: "))

        # By default we build a random set of N points with
            coordinates in [0,300)x[0,300):
        P = [(np.random.randint(0,100),np.random.randint
            (0,100)) for i in range(N)]
        L = GrahamScan(P)
        P = np.array(P)
        plt.figure()
        plt.plot(L[:,0],L[:,1], 'b-', picker=5)
        plt.plot([L[-1,0],L[0,0]],[L[-1,1],L[0,1]], 'b-',
            picker=5)
        plt.plot(P[:,0],P[:,1],".r")
        plt.axis('on')
        plt.show()
#       for j in range(N):
#         print (P[j])
        x_min_x=float('inf')
        x_max_x=float('-inf')
        y_min_y=float('inf')
        y_max_y=float('-inf')
        for q in range(N):
          if P[q][0]>x_max_x:
             x_max_x=P[q][0]
             x_max_y=P[q][1]
          if P[q][0]<x_min_x:
             x_min_x=P[q][0]
             x_min_y=P[q][1]
          if P[q][1]>y_max_y:
             y_max_y=P[q][1]
             y_max_x=P[q][0]
          if P[q][1]<y_min_y:
             y_min_y=P[q][1]
```

```
            y_min_x=P[q][0]

        print (x_min_x,x_min_y)
        print (x_max_x,x_max_y)
        print (y_min_x,y_min_y)
        print (y_max_x,y_max_y)
        j=0
        #preprocessing rules
        for i in range(N):
                if (P[i][0]-x_min_x)*(x_min_y-y_min_y) > (P
                    [i][1]-x_min_y)*(x_min_x-y_min_x) and
                    (P[i][0]-y_min_x)*(y_min_y-x_max_y) > (
                    P[i][1]-y_min_y)*(y_min_x-x_max_x) and
                    (P[i][0]-x_max_x)*(x_max_y-y_max_y) > (
                    P[i][1]-x_max_y)*(x_max_x-y_max_x) and
                    (P[i][0]-y_max_x)*(y_max_y-x_min_y) >(P
                    [i][1]-y_max_y)*(y_max_x-x_min_x) :
                     print ("qw")
                else:
                     j=j+1

        A=[(0,0) for i in range(j)]
        j=0
        for i in range(N):
                if (P[i][0]-x_min_x)*(x_min_y-y_min_y) > (P
                    [i][1]-x_min_y)*(x_min_x-y_min_x) and
                    (P[i][0]-y_min_x)*(y_min_y-x_max_y) > (
                    P[i][1]-y_min_y)*(y_min_x-x_max_x) and
                    (P[i][0]-x_max_x)*(x_max_y-y_max_y) > (
                    P[i][1]-x_max_y)*(x_max_x-y_max_x) and(
                    P[i][0]-y_max_x)*(y_max_y-x_min_y) > (P
                    [i][1]-y_max_y)*(y_max_x-x_min_x) :
                     print (P[i])
                else:
                     A[j]=P[i][0],P[i][1]
                     j=j+1
        # Plot the computed Convex Hull:
        print("Number of pints pre processed:")
        print (N-j)
#       for q in range (j):
#             print (A[q])
        np.array(A)
        X=GrahamScan(A)
        A = np.array(A)
        #for q in range (j):
         #     print (A[q])
        plt.figure()
        plt.plot(X[:,0],X[:,1], 'b-', picker=5)
```

```
        plt.plot([X[-1,0],X[0,0]],[X[-1,1],X[0,1]], 'b-',
            picker=5)
        plt.plot(A[:,0],A[:,1],".r")
        plt.axis('on')
        plt.show()

if __name__ == '__main__':
        main()
```

3.5 Source Code: JARVIS MARCH

```
import sys
import numpy as np
import matplotlib.pyplot as plt

# Function to know if we have a CCW turn
def CCW(p1, p2, p3):
        if (p3[1]-p1[1])*(p2[0]-p1[0]) >= (p2[1]-p1[1])*(p3
            [0]-p1[0]):
                return True
        return False

# Main function:
def GiftWrapping(S):
        n = len(S)
        P = [None] * n
        l = np.where(S[:,0] == np.min(S[:,0]))
        print (S[l[0][0]])
        pointOnHull = S[l[0][0]]
        i = 0
        while True:
                P[i] = pointOnHull
                endpoint = S[0]
                for j in range(1,n):
                        if (endpoint[0] == pointOnHull[0]
                            and endpoint[1] == pointOnHull
                            [1]) or not CCW(S[j],P[i],
                            endpoint):
                                endpoint = S[j]
                i = i + 1
                pointOnHull = endpoint
                if endpoint[0] == P[0][0] and endpoint[1]
                    == P[0][1]:
                        break
        P=list(filter(None.__ne__, P))
        return np.array(P)

def main():
```

```
        try:
                N = int(sys.argv[1])
        except:
                N = int(input("Introduce N: "))

        # By default we build a random set of N points with
            coordinates in [0,300)x[0,300):
        P = np.array([(np.random.randint(0,300),np.random.
            randint(0,300)) for i in range(N)])
        #L = GiftWrapping(P)
                ##################################################################

        for j in range(N):
          print (P[j])
        x_min_x=float('inf')
        x_max_x=float('-inf')
        y_min_y=float('inf')
        y_max_y=float('-inf')
        for q in range(N):
          if P[q][0]>x_max_x:
             x_max_x=P[q][0]
             x_max_y=P[q][1]
          if P[q][0]<x_min_x:
             x_min_x=P[q][0]
             x_min_y=P[q][1]
          if P[q][1]>y_max_y:
             y_max_y=P[q][1]
             y_max_x=P[q][0]
          if P[q][1]<y_min_y:
             y_min_y=P[q][1]
             y_min_x=P[q][0]

        print (x_min_x,x_min_y)
        print (x_max_x,x_max_y)
        print (y_min_x,y_min_y)
        print (y_max_x,y_max_y)
        j=0
        for i in range(N):
                if (P[i][0]-x_min_x)*(x_min_y-y_min_y) > (P
                    [i][1]-x_min_y)*(x_min_x-y_min_x) and
                    (P[i][0]-y_min_x)*(y_min_y-x_max_y) > (
                    P[i][1]-y_min_y)*(y_min_x-x_max_x) and
                    (P[i][0]-x_max_x)*(x_max_y-y_max_y) > (
                    P[i][1]-x_max_y)*(x_max_x-y_max_x) and
                    (P[i][0]-y_max_x)*(y_max_y-x_min_y) >(P
                    [i][1]-y_max_y)*(y_max_x-x_min_x) :
                     #print (P[i])
                     print ("EXCLUDING  point")
                else:
```

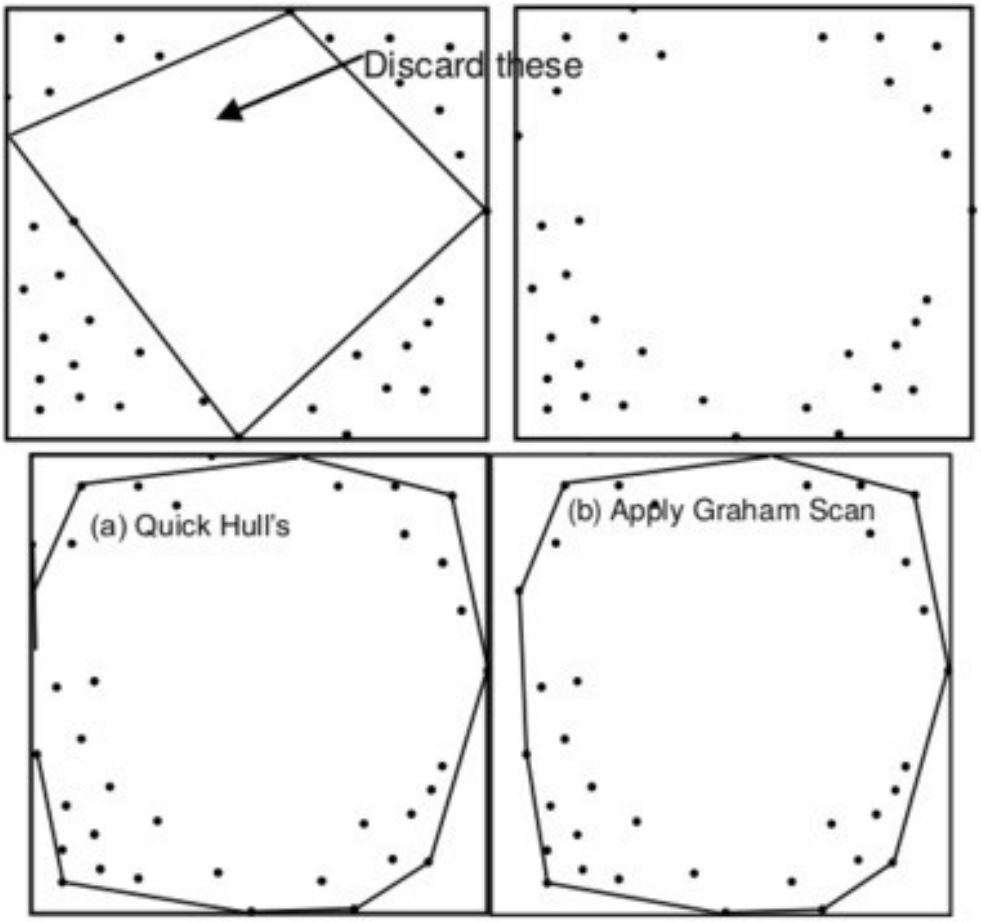
```
                        j=j+1

        A=[(0,0) for i in range(j)]
        j=0
        for i in range(N):
                if (P[i][0]-x_min_x)*(x_min_y-y_min_y) > (P
                   [i][1]-x_min_y)*(x_min_x-y_min_x) and
                   (P[i][0]-y_min_x)*(y_min_y-x_max_y) > (
                   P[i][1]-y_min_y)*(y_min_x-x_max_x) and
                   (P[i][0]-x_max_x)*(x_max_y-y_max_y) > (
                   P[i][1]-x_max_y)*(x_max_x-y_max_x) and(
                   P[i][0]-y_max_x)*(y_max_y-x_min_y) > (P
                   [i][1]-y_max_y)*(y_max_x-x_min_x) :
                    print (P[i])
                   # print ("qw")
                else:
                     A[j]=P[i][0],P[i][1]
                     j=j+1

        # Plot the computed Convex Hull:
        print("Number of points pre processed:")
        print (N-j)
#       for q in range (j):
#             print (A[q])
        A=np.array(A)
        X=GiftWrapping(A)
        A = np.array(A)
        #for q in range (j):
         #     print (A[q])
        plt.figure()
        plt.plot(X[:,0],X[:,1], 'b-', picker=5)
        plt.plot([X[-1,0],X[0,0]],[X[-1,1],X[0,1]], 'b-',
           picker=5)
        plt.plot(A[:,0],A[:,1],".r")
        plt.axis('on')
        plt.show()


if __name__ == '__main__':
  main()
```

Chapter 4

Conclusion

Convex hull is a very important term in the field of computational geometry. It is helpful where we need to model the objects with some non deterministic shapes. It has its applications in pattern recognition, image processing and GIS. In this paper a new technique to construct convex hull is presented. The technique can also be referred as the improvement of the graham scan algorithm. The graham scan algorithm performs faster for smaller number of input points. To reduce the input point set the quick hulls initial step is applied to the points due to which unnecessary calculations are eliminated and ultimately the task is performed faster and we can have the results faster than before. This is also proved by the results that the proposed technique is faster than any of two algorithms. As discussed earlier the main problem with the graham scan is that it has to make calculations on every single point in the set. Main idea behind this approach is to reduce the number of calculations, as more points are discarded in the initial step. Using quick hull, the points in the input set can be reduced.

PICTORIAL REPRESENTATION



COMPARISON OLD AND NEW

	Points	Graham Scan (sec)	Quick Hull (sec)	New technique (sec)
1	5000	3.73	1.26	0.86
2	10000	7.35	2.64	1.63
3	15000	11.96	4.68	2.65
4	20000	19.23	6.95	3.96
5	25000	23.6	7.99	5.86

The above results can be easily understood by the help of following graph (see figure 4).

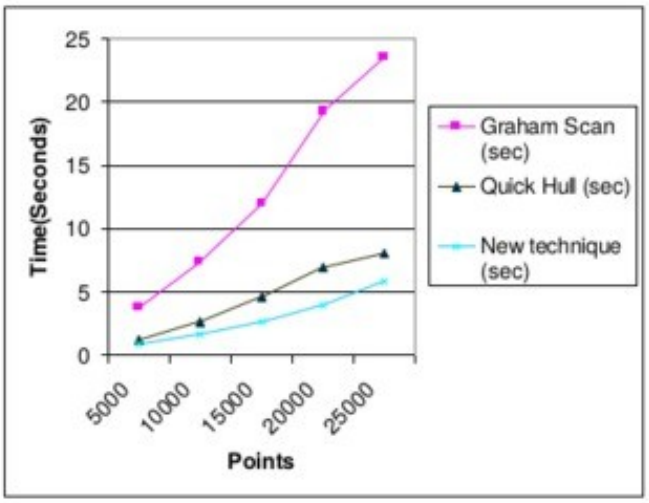


Figure 4: Graphical performance Comparison among the existing and proposed techniques

References

[1] roman10
Herve Bronnimann, John Iacono, 2002. Convex Hull Algorithms. Latin American Theoretical Informatics citeseer.ist.psu.edu/612454. Html.

[2] Mucke, E, 2009. Computing Prescriptions: Quickhull: Computing Convex Hulls Quickly. IEEE Journal on Computing in Science

[3] Preparata, Shamos, Computational Geometry, Chapter "Convex Hulls: Basic Algorithms"

[4] convex hull algorithms wikipedia