Evaluation of Jarvis' March and Graham's Scan Algorithms over Convex Hull

Subject Code: COMP 20007_2024_SM1
The **University of Melbourne**, Computing and Software Systems
Date of submission:

Author:
Yueming (Miles) Li. Student Number: 1450710.

A) Abstract

This report presents an experimental analysis comparing the performance of Jarvis' March and Graham's Scan algorithms for computing convex hulls. Convex hulls are important in computational geometry and computing graphics (Fitzpatrick, 2024). Jarvis' March and Graham's Scan are two prominent algorithms for solving convex hull problems.

The implementation details of Jarvis' March and Graham's Scan are discussed, and their respective strategies for constructing convex hulls. Experimental results are provided for three distribution scenarios: points on a convex hull, randomly distributed points, and points forming a simple convex hull. Performance results such as total CPU runtime and operation count are examined by input sizes ranging from 1000 to 5000 points.

The performance evaluation shows the scalability and efficiency of both algorithms. Jarvis' March demonstrates simplicity and robustness across various scenarios, but with quadratic time complexity. In contrast, Graham's Scan exhibits consistent performance but suffers from quadratic time complexity due to its reliance on bubble sort for sorting points.

The discussion highlights the discrepancies between the two algorithms, considering implementation complexity, stability, and scalability. While Jarvis' March is suitable for smaller input sizes and diverse distribution scenarios, Graham's Scan offers predictability and efficiency for certain input distributions.

In conclusion, understanding the characteristics and performance of Jarvis' March and Graham's Scan is important for selecting the most suitable algorithm based on specific problem requirements. Further research could focus on optimizing implementation strategies and exploring alternative algorithms to enhance efficiency and scalability in practical applications.

Catalogue

1. Abstract ……p. 2

2. Introduction ……..p. 4

3. Implementation Details ……p. 5

4. Performance Evaluation ……p. 12

5. Conclusion ……..p. 17

6. Reference List ……p. 18

B) Introduction

Convex hulls are fundamental but important in computational geometry, which are being used in a wide range of applications, from computer graphics to geography analysis (Fitzpatrick, 2024). The problem of computing the convex hull of a set of points helps us to comprehend the algorithms and practice our abilities of algorithms implements significantly.

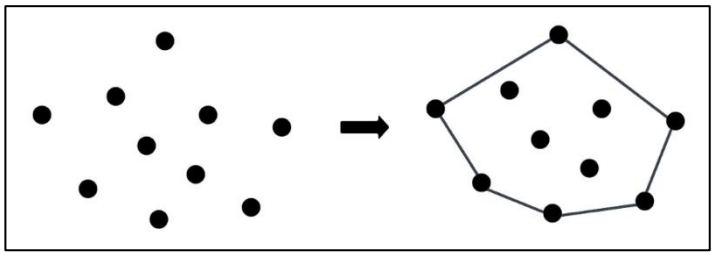


Figure 1. Source: Fitzpatrick, G. (2024). Convex

There are two important algorithms for computing convex hulls: Jarvis' March and Graham's Scam. They have been widely studied and implemented in computational geometry field (Fitzpatrick, 2024).

The purpose of this study is to conduct an experimental analysis to evaluate and compare the performance of Jarvis' March and Graham's Scan algorithms. By qualifying total basic operations over various input scales and configurations, I aim to gain insights into their efficiency and time complexities in solving the convex hull problems.

In this report, I provide a detailed analysis of the experimental results, discussing the implementation details of the algorithms, and the implications of my findings. By testing the performance of Jarvis' March and Graham's Scan under different scenarios, I seek and provide valuable discovery in to their strengths and flaws.

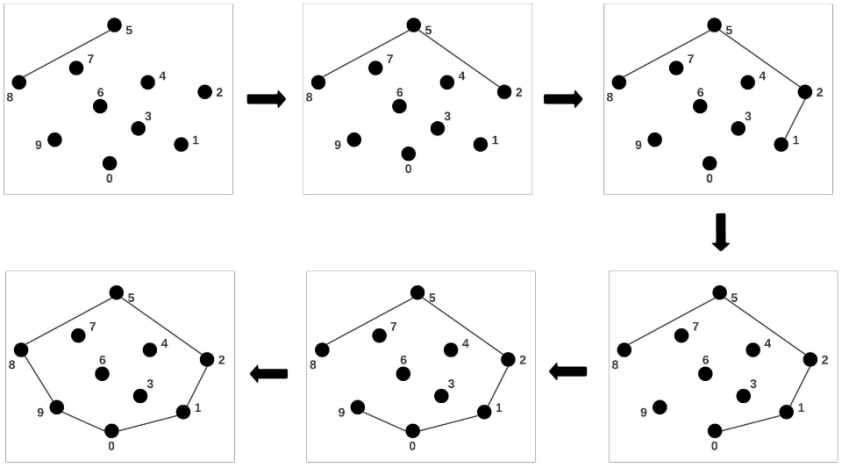
C) Implementation Details

Jarvis' March:

To implement Jarvis' March algorithm, I found the leftmost point first. Then iterate all the remaining points, select the most anticlockwise point relative to the current point until go back to the starting point (Fitzpatrick, 2024).

```
JarvisMarch(points):
  // Ensure there are at least 3 points
  if length(points) < 3:
    return empty convex hull

  // Initialize an empty list to store convex hull points
  convexHull <- empty list

  // Find the leftmost point (pivot) among the given points
  leftmost <- leftmost_point(points)

  // Start from the leftmost point
  current <- leftmost
  repeat:
    // Add current point to the convex hull
    add current to convexHull

    // Find the next point 'nextPoint' such that it forms a counterclockwise turn
    // with the current point and any other point in the set
    nextPoint <- points[0]
    for each point in points:
      if nextPoint = current or orientation(nextPoint, current, point) = counterclockwise:
        nextPoint <- point

    // Set 'nextPoint' as the current point for the next iteration
    current <- nextPoint

  // Repeat until we return to the starting point (leftmost)
  until current = leftmost

  // Return the list of points in the convex hull
  return convexHull
```

Pseudocode and diagrammatise (modified by Fitzpatrick, 2024):



Code implementation:

To find the most left point: iterate though all the points and find the point with the smallest x-coordinate value, if there are more than one points on the most left side, find the most bottom one.

```
// Find the leftmost point and put it in the hull
int mostLeftIdx = 0;
for (int i = 1; i < p->numPoints; i++)
{
    // More left than marked point
    if (p->pointsX[i] < p->pointsX[mostLeftIdx])
    {
        mostLeftIdx = i;
    }
    // Horizonally same, find most bottom point
    else if (p->pointsX[i] == p->pointsX[mostLeftIdx] &&
             p->pointsY[i] < p->pointsY[mostLeftIdx])
    {
        mostLeftIdx = i;
    }
}
```

Iterate though all points, and find the most anticlockwise one, store them into a double linked list then iterate again.

```
// Draw an intact convex hull by anticlockwise direction
int currentIdx = mostLeftIdx;
while (TRUE)
{
    insertTail(hull, p->pointsX[currentIdx], p->pointsY[currentIdx]);

    int nextPointIdx = 0;

    for (int i = 0; i < p->numPoints; i++)
    {
        if (nextPointIdx == currentIdx ||
            orientation(p, nextPointIdx, currentIdx, i) ==
                COUNTERCLOCKWISE)
        {
            nextPointIdx = i;
        }
    }
    currentIdx = nextPointIdx;

    if (currentIdx == mostLeftIdx)
        break;
}
```

To compare the position of two points, I used the function orientationResult written by Sutherland and modified by Fitzpatrick (2024), which returns 0 if 2 points are collinear, 1 if clockwise, and 2 if anticlockwise:

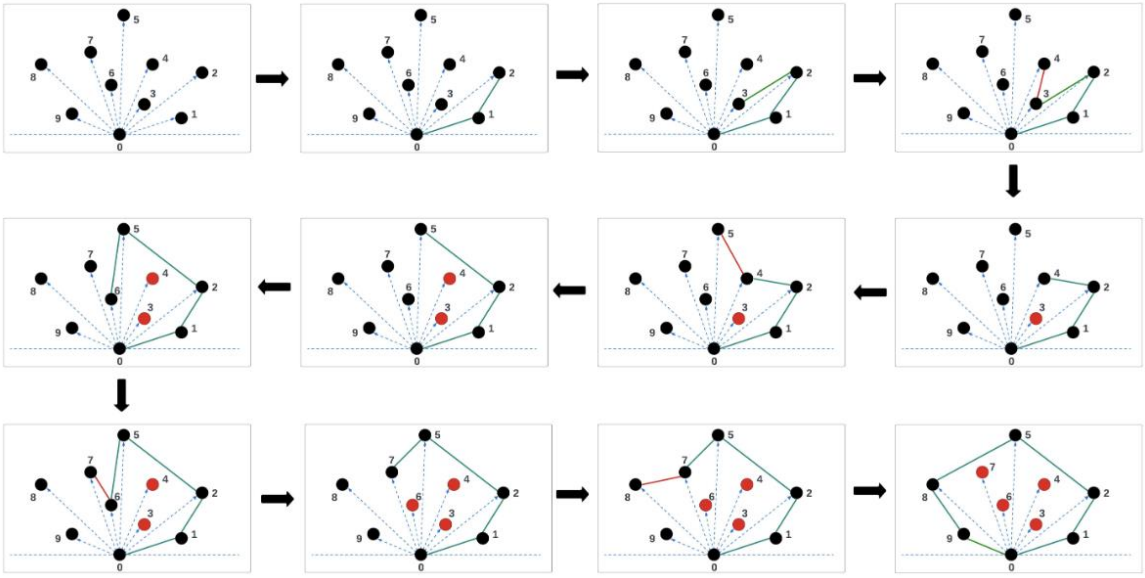
```c
enum orientationResult orientation(struct problem *p, int idxFirst,
                                   int idxMiddle, int idxFinal)
{
    assert(idxFirst >= 0 && idxFirst < p->numPoints);
    assert(idxMiddle >= 0 && idxMiddle < p->numPoints);
    assert(idxFinal >= 0 && idxFinal < p->numPoints);

    /* Use cross-product to calculate turn direction. */
    long double p0x = p->pointsX[idxFirst];
    long double p0y = p->pointsY[idxFirst];

    long double p1x = p->pointsX[idxMiddle];
    long double p1y = p->pointsY[idxMiddle];

    long double p2x = p->pointsX[idxFinal];
    long double p2y = p->pointsY[idxFinal];

    /* Cross product of vectors P1P0 & P1P2 */
    long double crossProduct = (p0x - p1x) * (p2y - p1y) - (p0y - p1y) * (p2x - p1x);

    if (crossProduct == 0)
    {
        if (idxFirst == idxMiddle)
        {
            /* Special case where we are only looking for positive slope of P1P2. */
            if (p2x == p1x)
            {
                /* Special case: dx = 0, vertical */
                if (p2y < p1y)
                {
                    /* Directly upwards */
                    return COUNTERCLOCKWISE;
                }
                else if (p2y == p1y)
                {
                    /* Same point. */
                    return COLLINEAR;
                }
                else
                {
                    return CLOCKWISE;
                }
            }
            long double m = (p2y - p1y) / (p2x - p1x);
            if (m >= 0)
            {
                return COUNTERCLOCKWISE;
            }
            else
            {
                return CLOCKWISE;
            }
        }
        return COLLINEAR;
    }
    else if (crossProduct > 0)
    {
        return CLOCKWISE;
    }
    else
    {
        return COUNTERCLOCKWISE;
    }
}
```

Graham's Scan:

To implement Graham's Scan algorithm, it is needed to find the bottom-most point first. Then sort all other points by the angle between the bottom-most point, pushing and popping them from the stack, and store the points into a list to ensure it is maintained a convex hull (Fitzpatrick, 2024).

Pseudocode and diagrammatise (modified by Fitzpatrick, 2024):

```
GrahamScan(points):
  // Ensure there are at least 3 points
  if length(points) < 3:
    return empty convex hull

  // Find the point with the lowest y-coordinate
  lowest = point with lowest y-coordinate in points
  // Sort the points based on their polar angles with respect to the lowest point
  sort points by polar angle with respect to lowest

  // Initialize an empty stack to store convex hull points
  stack = empty stack

  // Push the first three points to the stack
  push points[0] to stack
  push points[1] to stack
  push points[2] to stack

  // Iterate over the remaining points
  for i = 3 to length(points):
    // While the current point and the two points below the top of the stack
    // make a non-left turn, pop the top of the stack
    while orientation(second_top(stack), top(stack), points[i]) != counterclockwise:
      pop top of stack

    // Push the current point to the stack
    push points[i] to stack

  // The stack now contains the convex hull points
        return stack
```



Code implementation:

Firstly, find the most bottom point, if there are more than one most bottom points, get the most left one:

```
// Find the lowest point
int lowestIdx = 0;
for (int i = 0; i < p->numPoints; i++)
{
    if (p->pointsY[i] < p->pointsY[lowestIdx])
        lowestIdx = i;
    else if (p->pointsY[i] == p->pointsY[lowestIdx] &&
             p->pointsX[i] < p->pointsX[lowestIdx])
        lowestIdx = i;
}
```

Secondly, sort all the points by the angle between the bottom-most point and other points. I implemented the sorting procedure with bubble sort algorithm and store all sorted indices of points into an integer array. For the sorting part, I compared the angles by calling the orientationResult function. If it returns COUNTERCLOCKWISE, it shows the angle of point 1 is bigger than point 2, then swap the indices of these two points in the indices array. If those 2 points are COLLINEAR, compare the distance of those 2 points between the most bottom point. If distance of point 2 is larger than point 1, swap them.

```
/* Sort the points by angle */
// Use bubble sort, sort all the indices accoding to the angles
sortedIndices[0] = lowestIdx;
for (int i = 1; i < p->numPoints; i++)
{
    if (i == lowestIdx)
        continue;
    sortedIndices[i] = i;
}
bubbleSort(sortedIndices, p->numPoints, p);
sortedIndices = realloc(sortedIndices, (p->numPoints + 1) * sizeof(int));
sortedIndices[p->numPoints] = sortedIndices[0];
```

```c
// Use bubble sort to sort the points by angles
void bubbleSort(int indices[], int n, struct problem *p)
{
    int tempIdx;
    for (int i = 1; i < n - 1; i++)
    {
        for (int j = 1; j < n - i - 1; j++)
        {
            // If angle of point j+1 is bigger than point j, swap
            if (orientation(p, indices[j], indices[0], indices[j + 1]) ==
                COUNTERCLOCKWISE)
            {
                tempIdx = indices[j];
                indices[j] = indices[j + 1];
                indices[j + 1] = tempIdx;
            }
            // If angles are same, compare the distance
            else if (orientation(p, indices[j], indices[0], indices[j + 1]) ==
                     COLLINEAR)
            {
                double distance1 = sqrt(pow(p->pointsY[indices[j]] -
                                            p->pointsY[indices[0]],
                                        2) +
                                    pow(p->pointsX[indices[j]] -
                                            p->pointsX[indices[0]],
                                        2));
                double distance2 = sqrt(pow(p->pointsY[indices[j + 1]] -
                                            p->pointsY[indices[0]],
                                        2) +
                                    pow(p->pointsX[indices[j + 1]] -
                                            p->pointsX[indices[0]],
                                        2));
                if (distance2 < distance1)
                {
                    tempIdx = indices[j];
                    indices[j] = indices[j + 1];
                    indices[j + 1] = tempIdx;
                }
            }
        }
    }
}
```

Finally, iterate through the sorted points, and use a stack structure (Modified by Fitzpatrick, 2024) to store points which are on the convex hull: push the points in sorted order into the stack, check if it is anticlockwise relative to last point, if not, pop it and iterate to next point. At the last, the points remained in the stack are all points on the convex hull. For the sorted indices array, I allocated one more position for the start point at the end, to ensure the convex hull is closed in the end.

```
    // Push first 3 points into the stack
    struct stack *points = createStack(sortedIndices[0]);
    points = push(points, sortedIndices[1]);
    points = push(points, sortedIndices[2]);

    // Iterate over the points left
    for (int i = 3; i < p->numPoints + 1; i++)
    {
        while (
            orientation(p, getSecondTop(points), getTop(points), sortedIndices[i]) !=
            COUNTERCLOCKWISE)
        {
            points = pop(points);
        }
        points = push(points, sortedIndices[i]);
    }
    // Pop the origin point
    points = pop(points);

    // Put points in stack to a list
    while (points != NULL)
    {
        int idx = getTop(points);
        points = pop(points);
        insertHead(hull, p->pointsX[idx], p->pointsY[idx]);
    }

    // The end
    s->convexHull = hull;
    return s;
```

D) Performance Evaluation

There are three kinds of distributions are tested to evaluate the performance of both Jarvis' March and Graham's Scan algorithms:

1. All the points are on a convex hull (circle below).
2. All the points are distributed randomly (random below).
3. All the points are involved into a simple convex hull (simple convex hull below).

To test each distribution, I generated 1000, 3000, and 5000 points for each scenario, and measured the operation counts and total CPU runtime.

Jarvis' March:

Jarvis' March			
Count of Points	Distribution	Total CPU Runtime (sec)	Operation Count (in average)
1000	circle (worst case)	0.029	1000000
3000		0.157	9000000
5000		0.399	25000000
1000	random (general case)	0.005	17000
3000		0.009	84000
5000		0.011	120000
1000	simple convex hull (best case)	0.005	4000
3000		0.008	12000
5000		0.01	20000

Performance Evaluation:

For the basic operations of Jarvis' March algorithm, each iteration involves cross product and comparison between angles between the current point and all other points to determine next point in the convex hull. With the larger input size, the number of comparisons between angles increases significantly according to the total running.

Circle Scenario (Worst Case):

- As the number of points increases from 1000 to 5000, the total CPU runtime and operation count increase significantly.
- In this case, Jarvis' March requires a large number of iterations to construct the convex hull due to the number of all the points.
- The operation count grows roughly proportionally to square of the number of points, indicating a quadratic time complexity.
- Despite the increasing runtime, the algorithm's scalability is limited by the inherent complexity of the worst-case distribution.

Random Scenario (General Case):

- In the general case where points are distributed randomly, Jarvis' March performs better than circle scenario.
- The operation count and CPU runtime increase with the number of points, but at a slower rate.
- With random distribution leading to more efficient execution compared to highly structured distributions like a circle.
- However, even in the general case, Jarvis' March demonstrates quadratic time complexity, as evident from the increasing operation count with larger input sizes.

Simple Convex Hull Scenario (Best Case):

- The operation count and CPU runtime remain relatively low and consistent across different input sizes, indicating that the algorithm can quickly identify the convex hull without extensive iterations.
- Since the points are already in a convex hull, Jarvis' March requires minimal processing to determine the hull's boundary, resulting in linear or near-linear time complexity.
- This scenario highlights Jarvis' March's effectiveness in cases where the convex hull is readily identifiable.

Analysis:

- The algorithm's performance is affected by the distribution of points, but also affected by the input size of points. The CPU runtime and operation count are proportional to the input sizes.
- For the worst case (circle), the time complexity of Jarvis' March is $\Theta(n^2)$.
- For the general case (random), the time complexity of Jarvis' March is O(n^2) and $\Omega(n)$.

- For the best case (simple hull), the time complexity of Jarvis' March is $\Theta(n)$.

Graham's Scan:

Graham's Scan			
Count of Points	Distribution	Total CPU Runtime (s)	Operation Count (in average)
1000	circle (worst case)	0.025	497503
3000		0.123	4492503
5000		0.302	12487503
1000	random (general case)	0.024	497503
3000		0.13	4492503
5000		0.292	12487503
1000	simple convex hull with points in it (best case)	0.025	497503
3000		0.125	4492503
5000		0.294	12487503

Performance Evaluation:

- According to the data I got above, it reflected that regardless of the input distributions, Graham's Scan algorithm's performance remains consistent in terms of both runtime and operation count.
- The total CPU runtime and operation count are proportional to the number of points in the graph.
- Graham's Scan exhibits a time complexity of O(n^2) due to the use of bubble sort for sorting the points. The total CPU runtime and operation count demonstrate a quadratic relationship with the number of points across all distribution conditions.
- The stability of performance across different distributions suggests that Graham's Scan is not significantly affected by the arrangement of points but rather by the total number of points.

Analysis:

- The consistent performance of Graham's Scan across different distribution conditions highlights its robustness and predictability.

- The Algorithm's simplicity and ease of implementation make it suitable for practical application where input sizes are small or moderate.
- Since it always need to sort points first in Graham's Scan, and I used bubble sort to sort points, the time complexity of my Graham's Scan implementation should be Ω(n^2) = O(n^2). Thus, the actual time complexity of my Graham's Scan implementation is $\Theta(n^2)$.
- For the large input sizes, the quadratic time complexity may become a limiting. The ideal condition is implementing Graham's Scan with more efficient sorting algorithms such as Merge sort, Quick sort, or Heap sort. It will make the time complexity of Graham's Scan Θ(n*logn).

E) Discussion

While both Jarvis' March and Graham's Scan exhibit quadratic time complexity, they differ in their implementation details and computational strategies.

Jarvis' March is characterised by its easy implementation, making it suitable for small to moderate input sizes and cases where the convex hull's structure is not well-defined.

Graham's Scan, despite its quadratic time complexity, offers a more efficient approach for certain input distributions, especially when points are already sorted and arranged in a convex hull. However, it may suffer from poor performance for large input seizes due to the utilisation of bubble sort. To improve this, it should be implemented with better sorting algorithms which have time complexity of O(n*logn).

F) Conclusion

In conclusion, this report conducted an experimental analysis to evaluate and compare the performance of two fundamental algorithms for computing convex hulls: Jarvis' March and Graham's Scan. Through quantifying the total basic operations over various inputs and configurations, I aimed to get insights into their efficiency and time complexities in solving convex hull problems.

Jarvis' March demonstrates simplicity and robustness, making it suitable for various input distributions and smaller input sizes. Despite its quadratic time complexity, it performs adequately well across different scenarios, particularly in cases where the convex hull's structure is not well-defined.

On the other hand, Graham's Scan exhibits consistent performance across different distribution conditions, highlighting its stability and predictability. While its quadratic time complexity may pose limitations for large input sizes, its efficiency can be improved by implementing more efficient sorting algorithms.

Overall, understanding the strengths and weaknesses of each algorithm is essential for selecting the most appropriate approach based on the specific requirements and constraints of the problem at hand. Further research could focus on optimizing the implementation of both algorithms and exploring alternative strategies to enhance their efficiency and scalability in practical applications.

G) Reference List

Fitzpatrick, G. (2024). *Convex Hull*. University of Melbourne. https://edstem.org/au/courses/15984/lessons/50785/slides/344316

Sutherland, W., & Fitzpatrick, G. (2024). *Convex Hull Algorithm Implementation Structure*. [convexHull.c, convexHull.h, linkedList.c, linkedList.h, problem.c, problem.h, stack.c, stack.h, problem1a.c, problem1b.c]. University of Melbourne.