BLG336E -Analysis of Algorithms- HW2

Emir Kaan Erdogan

April 26, 2023

1 Convex Hull

1.1 Introduction

In my report, I try to show my work in the perspective of the following instructions:

- Write your pseudo-code for each part. Try to keep your pseudo-code human-readable. Code snippets will not be graded.
- Show the time complexity of your algorithms on the pseudo-code.
- Why should you use the divide and conquer approach for this problem? How does this affects the complexity of the Convex-Hull problem? (5 points)
- How does increasing the number of the cities affects the memory complexity of the algorithms? (15 points)
- Show the space complexity of your algorithms on the pseudo-code.
- Show run-time of all cases in a graph with number of nodes in x-axis and run-time in y-axis.

1.2 Abstract

To calculate the convex hull of a set of point, we can use several methods. Within the scope of the assignment, I implemented divide and conquer approach. To give information about the methodology , suppose we first calculate the convex hull of the left hull and the right half hull, then we can achieve the combined convex hull by merging these hulls together. First we find upper and lower tangents, then merge counterclockwise. For this purpose, I find the rightmost point of left hull and leftmost point of the right hull and create a line using these 2 points. Then, I control the next and previous nodes conditions such as whether it is above or below the line. In the next step, I traverse the hulls in direction to the upper nodes to find the upper tangent. If the target tangent is the lower tangent, then one should traverse to the nodes which are detected to be below the line. Once we find upper and lower tangents, we merge the points in counterclockwise order.

To implement the project, I have 2 functions, a divider and a combiner function. I assist the functions with several functions. While solving the problem, we can be confronted with several problems such as follows:
1) Arranging the points counterclockwise. To do that I utilized the cross product features in

linear algebra. To arrange ccw order, I first calculate the cross product of two vectors formed by this points. The resulting value cp is the angle between vectors. if the value of cp>0 then the points are counterclockwise, otherwise they are clockwise. Therefore, if the cp>0, I can return the points as they are, else we have to modify the vector by swapping operations.
2) Detecting whether the point is on,above or below the line.
First I create the line using simple line equations=mx+b. First I calculate the slope of the line which corresponds to m. Then , I find b and add the coefficients up to a vector to store the equation. At the end, within the get position function, I calculate the corresponding Y value for the current X point and extract the Y value of the point from the calculated Y. If the result<0, then the point should be above the line.
Now, I want to explain the main methods briefly. In the divider method, I pursue the simple divide approach, and send the counterclockwise arranged points to the merger function. Within the merger function, I iterate over the hulls checking the conditions of the next and previous functions. Having founded the upper and lower tangent of the set of points, I merge these points in counterclockwise order. In the following I want to give a visual about the first case in order to create a more clear image in reader's mind. First, see the distribution of the set of the points in the first case:

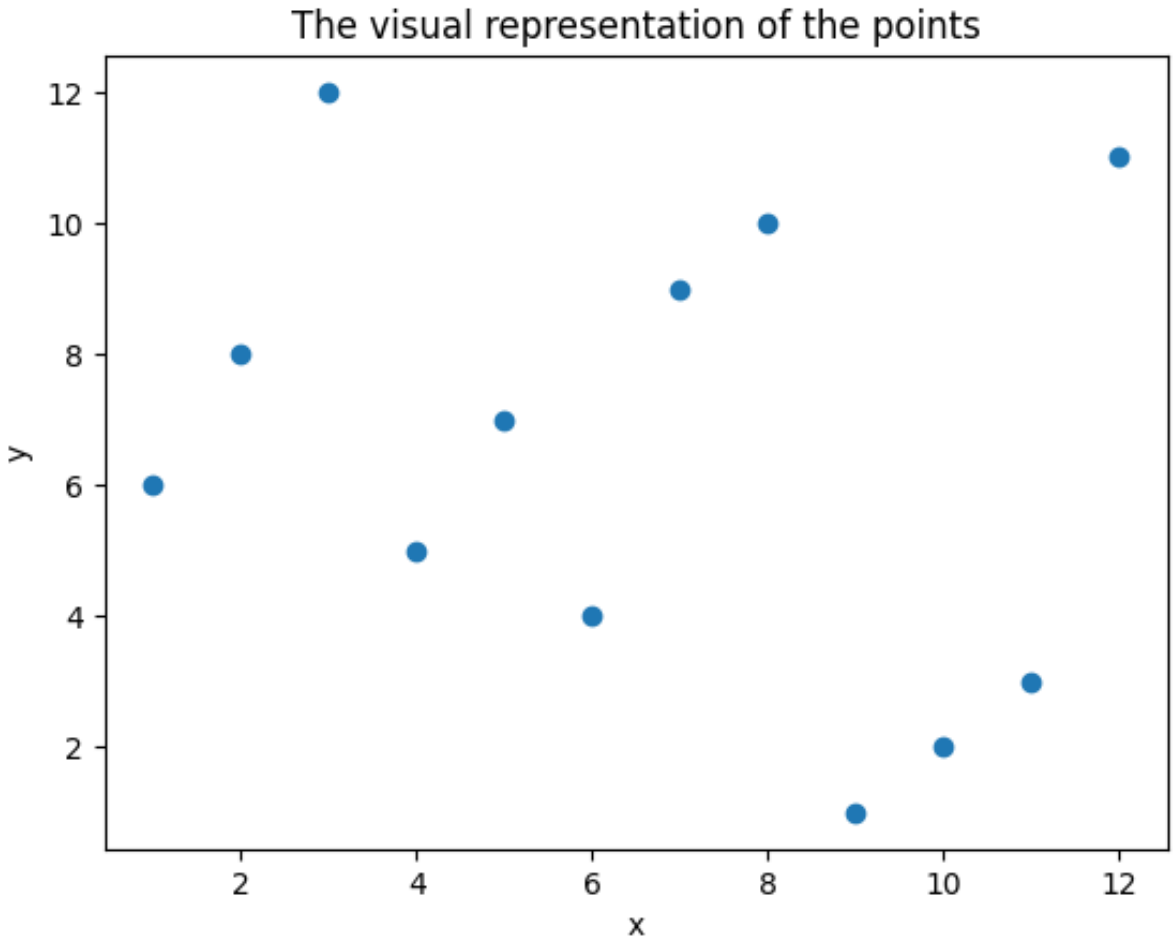

Figure 1: Points

Now, one can see what the program obtain as a result.

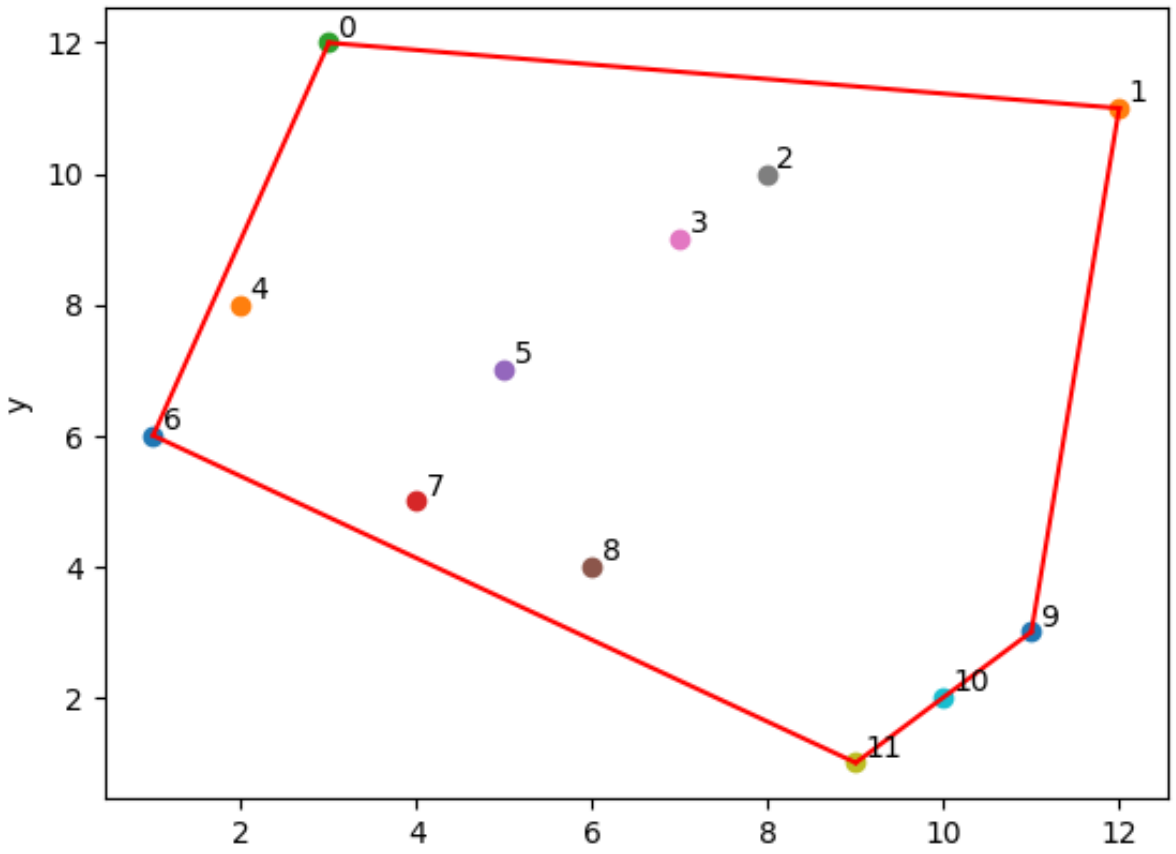

Figure 2: Convex Hull

A) The code can be compiled and run with the following instructions !
Program Compile Command:g++ -std=c++11 -Wall -Werror convex.cpp -o convex
./convex <input.txt>
In the following the pseudocode will be given and the time complexity of the algorithms is to analyze.

1.3 Pseudocodes

Pseudo-Code for the Divide Function:
vector<City*> separator(vector<City*> city_vector)
if (city_vector.size() <= 3)
return ccw_arranger(city_vector);
int mid = city_vector.size() / 2;
// Split the input vector into two halves, left and right. vector<City*> left;
vector<City*> right;
for (int i = 0; i < mid; i++)
left.push_back(city_vector[i]);
for (int i = mid; i < city_vector.size(); i++)
right.push_back(city_vector[i]);
left = separator(left);
right = separator(right);
return updated_combiner(left, right);

Pseudo-Code for the Merge function

```
vector <City*> updated_combiner (vector <City*> left_hull, vector<City*>right_hull)
left_most = find_left_most() ;
right_most= find_right_most() ;
int size_left= left_hull.size();
int size_right= right_hull.size();
vector <City*> combined_hull ;
vector <double>upper_tangent_line ;
// find upper tangent
bool upper_tangent_found = false;
while(!upper_tangent_found)
upper_tangent_found= true;
upper_tangent= find_upper_tangent()
next_position= get_position(upper_tangent,right_hull,index+1);
prev_position=get_position(upper_tangent,right_hull,index-1)
while(! (position_next<=0position_prev<=0) )
// update right tangent index;
right_tangent_index= (right_tangent_index -1+size_right)
// end loop
// find new line with updated points
upper_tangent_line=create_line()
next_position= get_position();
prev_position= get_position
while(! (next_position<=0 prev-position <=0 ))
// find new left index_position index, update line, check new next and prev points
left_tangent_index= (left_tangent_index+1 ) mod left hull size
upper_tangent_line=create_line()
position_left_hull_next = get_position()
position_left_hull_prev= get_position()
// update loop condition
upper_tangent_found=false;

// find lower tangent the same manner
// combine left and right hull counterclockwise using tangent indices
int index= upper_tangent_index[0];
combined_hull.push_back(left_hull[index]);
while(index!=lower_tangent_index[0])
index= (index+1) mod left size
combined_hull.push_back(left_hull[index]);
index= lower_tangent_index[1];
combined_hull.push_back(right_hull[index mod right_size]
while(index!=upper_tangent_index[1])
index= (index+1) mod right size
combined_hull.push_back(right_hull[index]);
return combined_hull ;
```

Time Complexity:
While implementing the divide and conquer approach we have basically 4 steps.

- sort the points by x once for all
- Divide the set of points into two halves recursively until the base case is met.
- Compute CH(LEFT) , CH(RIGHT)
- Combine the two convex hulls together to obtain the convex hull of the entire set of points.

To sort the cities once for all, I used the sort() function in the algorithm library of C++. It uses the introsort algorithm, which is a combination of quicksort, heapsort, and insertion sort. The time complexity of sort() is typically $O(n \log n)$. Then, the dividing step requires O(n) complexity, whereas we merge the points in O(log(n)) . Overall , we have the time complexity: **O(n*log(n))** !

1.4 Questions

- **Q1:** Why should you use the divide and conquer approach for this problem? How does this affects the complexity of the Convex-Hull problem? (5 points)
- **Q2:** How does increasing the number of the cities affects the memory complexity of the algorithms? (15 points)

1.4.1 Q1:

From a brute-force perspective, one can implement the Convex Hull in O(pow(N,3)).
However, we want to achieve a better algorithmic complexity. Here, the divide and conquer method is a very powerful tool since it breaks down the problem into smaller sub-problems, solves them recursively, and then combines the solutions to obtain the final result. In our problem, while computing the Convex-Hull, the divide and conquer approach implies dividing the set of points into two subsets of roughly equal size, computing the Convex-Hull of the left and right hulls recursively, and then combining these two to obtain the Convex-Hull of the original set of points. This brings a lower algorithmic complexity of O(N*log(n)).

1.4.2 Q2:

The time complexity of the Convex-Hull algorithm depends on the number of cities in the input. In general, as the number of points increases, the time complexity of the algorithm increases as well, since it is directly proportioned to the N*log(n). For instance, when we double the size of the input, we can expect a larger running time which can be calculated as it follows: (2n*log(2n)) / (n/log(n)). This means that if the number of points in the input is doubled, the running time of the algorithm will increase by a **constant factor.**

1.5 References

- Lecture slides and Recitations
- MIT Open Course: "https://www.youtube.com/watch?v=EzeYI7p9MjUt=2807s"