Indian Statistical Institute
203 BT Road, Kolkata - 700108

On a Few Progressive Algorithms

Ankan Kumar Das
M.Tech. Computer Science
Roll: CS1815

Supervisor: Dr. Arijit Bishnu

A dissertation submitted in partial fulfilment of the requirement
for the degree of
Master of Technology in Computer Science

July 2020

CERTIFICATE

This is to certify that the dissertation titled **"On a Few Progressive Algorithms"** submitted by **Ankan Kumar Das** to Indian Statistical Institute, Kolkata in partial fulfilment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per regulation of this institute and, in my opinion, has reached the standard needed for submission.

Dr. Arijit Bishnu
Associate Professor,
Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor **Dr. Arijit Bishnu** for giving me an opportunity to work under him and providing me exciting problems to think upon. Throughout the development of this project he was always available for discussion and guidance. I would also like to thank **Prof. Subhas Chandra Nandy** for providing me useful suggestions and valuable feedback since mid-term evaluation. He never hesitated to discuss with me whenever I asked for. Special thanks to **Anup Bhattacharya** for helping me to understand many thing regarding research work and this project. Finally I would like to express my special gratitude to my first teacher my **parents**, who have always encouraged me in all aspect, for igniting the love for knowledge within me.

ABSTRACT

The progressive algorithms are algorithms that outputs intermediate solutions which approximate the complete solution to the given problem. The user can decide whether to continue the running of the algorithm based on the error of the partial solutions. In this dissertation, we have studied few problems from the perspective of progressive algorithm. We have proposed the following:

Huffman encoding: a progressive algorithm for finding optimal prefix encoding or huffman coding. We have proved that error of the partial solution in step r is bounded by $n/2^{r-2}$. Overall running time of the algorithm, we have shown, is $O(n \log n)$.

Convex hull in 2D: Next, we have moved towards geometric problems. We have presented a randomized progressive algorithm for finding convex hull of the points in $\mathbb{R}^2$. The algorithm runs in at most $\log n$ many rounds and expected running time of each round is $O(n)$.

Convex hull in 3D: We have also extended an existing progressive algorithm for finding convex hull of the points in $\mathbb{R}^2$ for the point set in $\mathbb{R}^3$. We have proposed a procedure to have an upper bound of $O(\log n)$ for the number of rounds of the algorithm for this problem. This work uses one observation whose proof eludes us but we have compelling experimental evidence for the observation.

Contents

Chapter 1

Introduction

The most natural approach when we design algorithms for a particular problem is to get necessary inputs at the very beginning and then to compute the complete solution without allowing for any interaction in between. Traditional algorithms are designed like that way and of course in many instances it is the desired approach. But consider the scenario when we have large data sets then computing the complete solution will take lot of time. That time will take anyway, but if on the way of computing the complete solution, the application can output partial and meaningful solutions then that will help user to decide whether it is necessary to continue the computation and perhaps in which way the computation should proceed. Such algorithms are called *progressive algorithms* [2]. We shall briefly discuss about progressive algorithm and formally present the framework in the next subsection (1.1). In this dissertation, we shall present and study progressive algorithms for the following problems: *huffman coding, computing convex hull in 2-dimension and 3-dimension.* For **huffman coding** our focus will be to get encoding of the symbols way before getting optimal encoding and as our algorithm progresses we shall argue that our encoding is getting better and better and ultimately when algorithm stops we should get the optimal encoding. For the **convex hull** we shall present randomized progressive algorithm when the points are in 2-dimension and a deterministic progressive algorithm when the points are in 3-dimension.

1.1 Progressive Algorithm

A progressive algorithm is an algorithm that works in *rounds*, reporting partial solutions with smaller error guarantees in each round, until in the last round a complete solution is reported. The error guarantees are specified by a *convergence function* $f_{conv} : \{1, \ldots, k\} \to \mathbb{R}_{\geq 0}$, where k is the number of rounds. More precisely, *a k-round progressive algorithm* with convergence function f_{conv} with respect to error function err is an algorithm that outputs, after the r-th round (for $1 \leq r \leq k$), a partial solution S_r with $err(S_r) \leq f_{conv}(r)$ [2]. It is preferable that each partial solution should be a

"refinement" of the previous one. It is to be noted that the convergence function f_{conv} is just a bound on the *err* function but it does not implicitly guarantee that a solution in the next round is better than the solution in the current round. Nevertheless, the basic idea behind progressive algorithm is that one should get better and better solution as the algorithm progresses.

1.1.1 The framework

We shall use the same framework as presented in [2]. Let D be the data set that forms the input to the problem. The set D defines a set $\mathcal{S}(D)$ of valid *partial solutions*, and we are given an *error function* $err : \mathcal{S}(D) \to \mathbb{R}_{\geq 0}$ that assigns every partial solution a non-negative value. A *complete solution* is partial solution S with $err(S) = 0$.

A progressive algorithm reports partial solutions with smaller error guarantees in each *round*, until a complete solution is reported. The guarantees are specified by a *convergence function* $f_{conv} : \{1, \ldots, k\} \to \mathbb{R}_{\geq 0}$, where k is the number of rounds and $0 = f_{conv}(k) \leq f_{conv}(k-1) \leq \cdots \leq f_{conv}(1)$. A *k-round progressive algorithm* with convergence function f_{conv} (with respect to error function *err*) is now defined as an algorithm that outputs after the r-th round, for $r = 1, 2, \ldots, k$, a partial solution S_r with $err(S_r) \leq f_{conv}(r)$. The convergence function specifies how rapidly the error decreases.

Another important metric in measuring the performance of any algorithm is the *running time*. Authors in the article [2] have defined two ways to measure the running time.

- $T_{max}(n)$, *the maximum time per round*. Let $T_r(n)$ be the worst case running time for round r. Then we have $T_{max}(n) := \max_r T_r(n)$. When algorithm takes little time every round, then user never has to wait too long for the next partial solution.

- $T^*_{max}(n)$, *maximum amortized time per round*. Let $T_{\leq r}(n)$ be the worst case total running time for rounds $1, \ldots, r$. Then we have $T^*_{max}(n) := \max_r T_{\leq r}(n)/r$. This measure gives the possibility to report initial partial solutions very quickly so that later rounds can take little more time.

In our analysis, we shall mainly use the first way i.e. $T_{max}(n)$ to measure running time of our algorithms. For randomized progressive algorithms we can use the following:

- $\mathbb{E}_{max}[T(n)]$, *the maximum expected time per round*. Let $\mathbb{E}[T_r(n)]$ be the expected running time for round r. Then $\mathbb{E}_{max}[T(n)] := \max_r \mathbb{E}[T_r(n)]$.

1.2 Problem Definition

Now we shall briefly discuss about the problems that are presented in this dissertation. For both the problems mentioned below the progressive algorithm presented in this thesis will follow the above framework.

1.2.1 Huffman Coding

Huffman coding is a lossless data compression algorithm. It is a particular type of optimal prefix code developed by David A. Huffman in 1952 [5].

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of n *symbol alphabets*. We are also given a set $W = \{w_1, w_2, \ldots, w_n\}$ of n positive *symbol weights* i.e., $w_i = weight(a_i)$ for $i \in \{1, 2, \ldots, n\}$. Usually symbol weights are proportional to the number of occurrences or probabilities of corresponding symbol alphabets. We need to output $C = \{c_1, c_2, \ldots, c_n\}$ set of n *binary codewords*, where c_i is codeword for a_i for $i \in \{1, 2, \ldots, n\}$. Let $ACL(C) = \sum_{i=1}^{n} w_i \times length(c_i)$ be the *average code length* for code C, then the condition to be satisfied is $ACL(C) \leq ACL(C')$ for any code C' of alphabet set A.

When the symbols are not presented in sorted order of their weights then the algorithm runs in $O(n \log n)$ time and at the end of computation it provides optimal binary codewords C. Our objective is to apply the progressive algorithm framework for this problem. While computing the optimal binary codewords, our algorithm will provide *meaningful binary codewords* $C_1, C_2, \ldots, C_k$ where C_r is the set of codewords after r-th round for $r = 1, 2, \ldots, k$. By *meaningful binary codewords*, we mean that the set of codewords should satisfy the following conditions.

- None of the codewords C_i for $i = 1, 2, \ldots, k$ should be *ambiguous*. "The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of message is known" (Restriction mentioned in [5]). Say we have $A = \{a_1, a_2, a_3\}$ as 3 symbol alphabets and $C' = \{11, 10, 111\}$ be their codewords. Then we see that codeword of a_1 is a prefix of codeword of a_3. This is ambiguous as the encoder needs to encode additional information to distinguish those two symbols. Traditional huffman coding does not generate such codewords. All the intermediate codewords $C_1, C_2, \ldots, C_k$ should also follow the same rule.

- Codewords $C_1, C_2, \ldots C_k$ should be such that $err(C_i) \leq f_{conv}(i)$ for $i = 1, 2, \ldots, k$. The error function err will be precisely defined for this problem in the following chapter (2).

- At the end of the progressive algorithm, i.e., after the k-th round, the generated set of codewords C_k should be optimal in terms of average code length.

While this is good to generate meaningful codewords that gets better and better as the algorithm progresses, the user will not be happy if algorithm needs to compromise too much with the overall running time. In other words overall running time of this progressive algorithm should be $O(n \log n)$.

1.2.2 Convex Hull

In geometry, a subset of an affine space over reals is a convex set if given any two points in the set, it contains the whole line segment that joins them. Now given a point set P, *convex hull* of P is the smallest convex set that contains it. Similar to the previous problem, here also our objective again is to apply progressive algorithm framework to compute $CH(P)$, the convex hull of P. Let k be the number of rounds for the progressive algorithm then we define our partial solutions as $C_1, C_2, \ldots, C_k$, where C_r is the partial solution reported after the r-th round. Each of our partial solutions will be a convex polygon satisfying $C_r \subseteq CH(P)$. Convex polygons C_r are actually a convex hull of a subset of P carefully chosen by the algorithm. Based on the nature of the point set P, we have sub-categorized this into following.

Convex Hull in $\mathbb{R}^2$

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points where each of the point $p_i \in \mathbb{R}^2$ for $i = 1, 2, \ldots, n$. Alewijnse et al. [2] have developed a progressive algorithm for finding covex hull of P combining the ideas from construction of coresets and the *QuickHull* algorithm [3]. At each round r, their algorithm is selecting a subset $P_r \subset P$ such that all points in P_r are *extreme*. Their idea is to select the extreme points from P for a predetermined set of directions. In our case, a set of directions will be selected via a randomized algorithm. We shall discuss about error function, the algorithm and its analysis in chapter (3).

Convex Hull in $\mathbb{R}^3$

Our objective for this problem will be to extend the progressive algorithm for the point sets in $\mathbb{R}^3$. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points where each of the point $p_i \in \mathbb{R}^3$ for $i = 1, 2, \ldots, n$. We have imposed few restrictions on the point set P such as "no three of the points should belong to same straight line", "no four of the points should belong to same plane" (*general position assumptions*). At the r-th round, our solution C_r will be convex hull of the point set $P_r \subset P$. Since each of the points in P_r is extreme points in some direction, so the selected point set will be part of final solution. As a result, partial solutions will satisfy the following relation. $C_1 \subseteq C_2 \subseteq \cdots \subseteq C_k = CH(P)$. We shall define error function and present our algorithm and its analysis in chapter (4). During analysis we shall find convergence function $f_{conv}(r)$ that upper bounds the error of the partial solution C_r. This result is not fully provable because we could not formally prove an observation, but our observation is backed up by experimental results.

1.3 Related Work

The framework of the progressive algorithm was introduced by Alewijnse et al. in 2015 [2]. They studied problems in computational geometry like finding convex hull in a plane, computing k popular regions for a set of trajectories. They also observed a nice connection between approximation algorithm and progressive algorithm as later produces approximate solutions in each step. Amir Mesrikhani and Mohammad Farshi presented progressive algorithm for sorting in external memory model [8]. Progressive algorithms are also studied for finding euclidean minimum spanning tree [9] and in database also [6].

1.4 Organization of Thesis

We started with the definition of "Progressive Algorithm" followed by the framework. These helped us to describe the problems more specifically for our objectives for those problems in section (1.2).

In chapter (2) we study about our first problem i.e., Huffman Coding. We define an error function, develop an algorithm and find convergence function for this problem. In chapter (3), we turn our attention to convex hull. First we present the existing progressive algorithm, then propose our modification followed by its analysis. The ideas from this chapter help us to extend the algorithm (for the points in $\mathbb{R}^3$) and to bound the errors of the partial solutions in chapter (4).

Chapter 2

Huffman Coding

The problem and our objective are already described in the previous chapter (check section 1.2.1). Now we describe the error function we have considered for this problem and then proceed to the progressive algorithm and its analysis.

2.1 Error Function

Let $C' = \{c'_1, c'_2, \ldots, c'_n\}$ be any intermediate set of binary codewords generated by the algorithm and $C = \{c_1, c_2, \ldots, c_n\}$ be an optimal set of binary codewords then the error function we consider is

$$err_{C'} = ACL(C') - ACL(C) \tag{2.1}$$

where $ACL(C)$ is the *average code length* for set of codewords C and is defined as follows

$$ACL(C) = \sum_{\substack{c_i \in C \\ w_i \in W}} w_i \times length(c_i) \tag{2.2}$$

W is the set of weights associated with set of symbol alphabets for an instance of this particular problem.

Optimal set of codewords C may not be unique but for any given problem instance, average code length $ACL(C)$ is unique by the condition of optimality. So we are effectively subtracting constant value for each of the intermediate solutions. At the end of the algorithm we must get a set of codewords C'' such that $err_{C''} = 0$. Then by equation 2.1 we get $ACL(C'') = ACL(C)$, which is the desired output.

2.2 Cooking an Algorithm

When the symbols are not presented in sorted order of their weights, traditional algorithm maintains a *min heap*. It extracts two minimum weighted nodes from the heap,

creates a new node and insert that back to the heap. It continues till we have only one node in the heap [5]. As we need codewords for all the alphabets, it is possible only when there is single node left in the heap. So, we could not bring this to the progressive algorithm framework.

Later in the year 1976, J. van Leeuwnen presented a linear time algorithm when inputs are presented in sorted order of their weights [10]. This is useful in our case as it can generate set of codewords for all the alphabets in $O(n)$ time. Only thing it needs is weights should be in sorted order. Authors in the article [2] presented a progressive algorithm for sorting. So we combine these two ideas to present a progressive algorithm with respect to the error function (2.1) for this problem.

Input: Alphabet set A, Weight set W, Two indices $first$, $last$.
Output: Rooted binary tree whose leaf contains alphabets $A[first]$ to $A[last]$. Root node also stores sum of their weights.

1 **Function** `GroupAlphabets`$(A, W, first, last)$**:**
2 $root \leftarrow newNode()$; `// generates tree with two empty leaf nodes`
3 $temp \leftarrow root$;
4 **for** $(it \leftarrow first \text{ to } last - 2)$ **do**
5 $temp.left \leftarrow A[it]$;
6 $temp.right \leftarrow newNode()$;
7 $temp \leftarrow temp.right$;
8 $temp.left \leftarrow A[last - 1]$;
9 $temp.right \leftarrow A[last]$;
10 $root.weight \leftarrow \sum_{i=first}^{last} w_i$;
11 **return** $root$;

Remark. *(GroupAlphabets 1) Above procedure creates a binary tree for a set of alphabets. It is obvious that each call to the procedure will take $O(last - first)$ time.*

This procedure will be useful to group unsorted alphabet symbols. Once we present our main algorithm, the importance will be clearer. Before that, we present the procedure based on [10] to generate Huffman code for sorted input weights.

Input: *inputQueue*, a queue containing nodes in non-decreasing order of their weights.
Output: *Rooted binary coding tree* which we can use to generate set of binary prefix code for alphabets.

```
1  Function EfficientHuffmanCoding(inputQueue):
2      Q_1 ← inputQueue ;
3      Q_2 ← emptyQueue() ;
4      while (Q_1.size() ≥ 2 or Q_2.size() ≥ 2) do
           /* popMin() method compares front of the two queues removes and
              returns the node which is minimum                          */
5          first ← popMin(Q_1, Q_2) ;                  // 1st minimum node
6          second ← popMin(Q_1, Q_2) ;                 // 2nd minimum node
7          temp ← newNode() ;
8          temp.left ← first ;
9          temp.right ← second ;
10         temp.weight ← first.weight + second.weight ;
11         Q_2.pushBack(temp) ;     // Inserts the node to the back of the queue
12     return Q_2.front() ;
```

Remark. *(EfficientHuffmanCoding 2) At each iteration of the while loop, we are removing two nodes and inserting one node again, effectively the total number of nodes are reduced by one. Therefore, this procedure runs in linear time with respect to the inputQueue size.*

There is a natural bijection between *binary prefix codes* and *rooted binary coding trees*, in which any node is either leaf or it has two children. So we have omitted the procedure to generate binary prefix code from a coding tree. If we get a coding tree we can simply replace left link with 0 and right link with 1 to get binary prefix code for the alphabets. One can use any other approaches but as long as there is a bijection, things will go through. Since all the ingredients are ready, now we cook our final recipe (ProgressiveHuffmanCoding 3).

Algorithm 1: ProgressiveHuffmanCoding

Input: Alphabet set A and Weight set W.

Output: After each round algorithm outputs a *rooted binary coding tree* which specifies set of binary prefix codes for alphabets

```
1  Function ProgressiveHuffmanCoding(A, W):
2      n ← A.size() ;
3      Q ← emptyQueue() ;
4      k ← ⌈log2(n)⌉ ;                                   // Number of rounds
5      pivotPos ← 2^k ;                          // lowest power of 2 not less than n
6      for (r ← 1 to k) do
7          start ← 1 ;
8          end ← min(n, start + 2 × pivotPos − 1) ;
9          while (start + pivotPos − 1 <= n) do
               /* partition() method will work on the segments A[start...end] and
                  W[start...end], in this segment pivotPos-th largest weight will
                  be placed in proper position s.t.  following conditions hold
                  W[i] ≥ W[start + pivotPos − 1] for i = {start, ..., start + pivotPos − 1},
                  W[j] ≤ W[start + pivotPos − 1] for j = {start + pivotPos, ..., end}    */
10             partition(A, W, start, end, pivotPos) ;
11             start ← end + 1 ;
12             end ← min(n, start + 2 × pivotPos − 1) ;
13         if (pivotPos > 2) then
14             start ← 1 ;
15             end ← min(n, start + pivotPos − 1) ;
16             while (start ≤ n) do
17                 tree ← GroupAlphabets(A, W, start, end) ;
18                 Q.pushFront(tree) ;    // Insert node to the front of the queue
19                 start ← end + 1 ;
20                 end ← min(n, start + pivotPos − 1) ;
           /* when pivotPos = 2, the alphabets become sorted in non increasing
              order of their weights                                              */
21         else
22             for (i ← 1 to n) do
                   /* create a new leaf node with symbol and weight               */
23                 node ← leafNode(a_i, w_i) ;
24                 Q.pushFront(node) ;
25         C_i ← EfficientHuffmanCoding(Q) ;
26         print C_i ;                    // tree to generate codewords for r-th round
27         pivotPos ← pivotPos/2 ;
```

Remark. *(ProgressiveHuffmanCoding 3) Step* 4 *ensures that the for loop at step* 6 *runs* $O(\log n)$ *many times. Since selection can be performed in linear time [4], so our partition method (which will implicitly call selection method) at step* 10 *can be done in linear time too. Other procedures inside for loop (step* 6*) are also linear. Therefore, for each round* r*, we are spending constant time per alphabet, i.e. each round takes* $O(n)$ *time. Hence, worst case overall time complexity for this progressive algorithm is* $O(n \log n)$*.*

2.3 Understanding the Algorithm

Now we shall try to understand how the algorithm just described (ProgressiveHuffmanCoding 3) generates a series of coding trees after each round. We describe a small problem instance and run our algorithm on that instance for better understanding. Let $A = \{a_1, a_2, \ldots, a_{12}\}$ and $W = \{0.02, 0.004, 0.11, 0.05, 0.019, 0.018, 0.042, 0.015, 0.007, 0.025, 0.48, 0.21\}$. Let us see how it works.

When $r = 1$

At the beginning of this iteration

A	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
W	0.02	0.004	0.11	0.05	0.019	0.018	0.042	0.015	0.007	0.025	0.48	0.21

$pivotPos = 16$

(No partition for this round)

GroupAlphabets will return the tree $T_{1,1}$

EfficientHuffmanCoding will also return the same tree i.e. $C_1 = T_{1,1}$

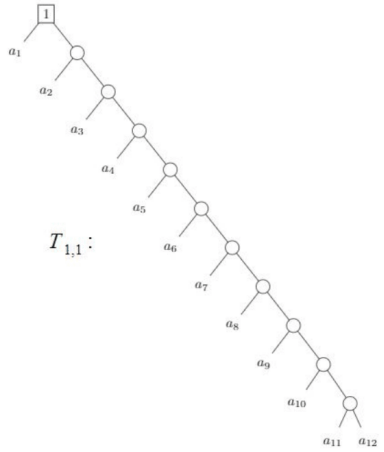


Figure 2.1: *ProgressiveHuffmanCoding when round* $r = 1$*. Average Code Length* $ACL(C_1) = 9.078$

When $r = 2$

At the beginning of this iteration

A	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
W	0.02	0.004	0.11	0.05	0.019	0.018	0.042	0.015	0.007	0.025	0.48	0.21

$pivotPos = 8$

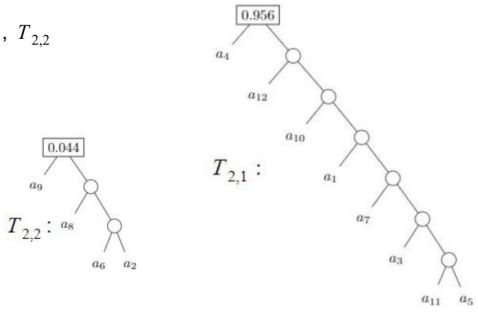
After partition (current pivots are highlighted in blue)

A	a_4	a_{12}	a_{10}	a_1	a_7	a_3	a_{11}	a_5	a_9	a_8	a_6	a_2
W	0.05	0.21	0.025	0.02	0.042	0.11	0.48	0.019	0.007	0.015	0.018	0.004

GroupAlphabets will return the trees $T_{2,1}$, $T_{2,2}$

Initial Q in non decreasing order

Node Name	$T_{2,2}$	$T_{2,1}$
Node Weights	0.044	0.956



EfficientHuffmanCoding will return the tree C_2

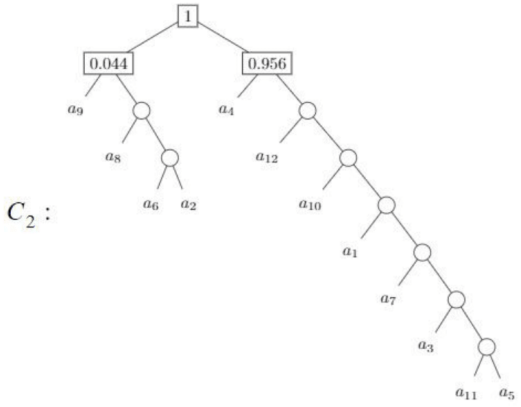


Figure 2.2: *ProgressiveHuffmanCoding when round* $r = 2$. *Average Code Length* $ACL(C_2) = 6.091$

When $r = 3$

At the beginning of this iteration (previous pivots are highlighted in grey)

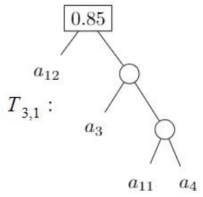
A	a_4	a_{12}	a_{10}	a_1	a_7	a_3	a_{11}	a_5	a_9	a_8	a_6	a_2
W	0.05	0.21	0.025	0.02	0.042	0.11	0.48	0.019	0.007	0.015	0.018	0.004

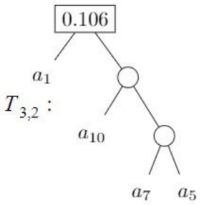
$pivotPos = 4$

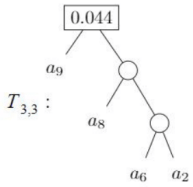
After partition (current pivots are highlighted in blue)

A	a_{12}	a_3	a_{11}	a_4	a_1	a_{10}	a_7	a_5	a_9	a_8	a_6	a_2
W	0.21	0.11	0.48	0.05	0.02	0.025	0.042	0.019	0.007	0.015	0.018	0.004

GroupAlphabets will return the trees $T_{3,1}$, $T_{3,2}$, $T_{3,3}$







Initial Q in non decreasing order

Node Name	$T_{3,3}$	$T_{3,2}$	$T_{3,1}$
Node Weights	0.044	0.106	0.85

EfficientHuffmanCoding will return the tree C_3

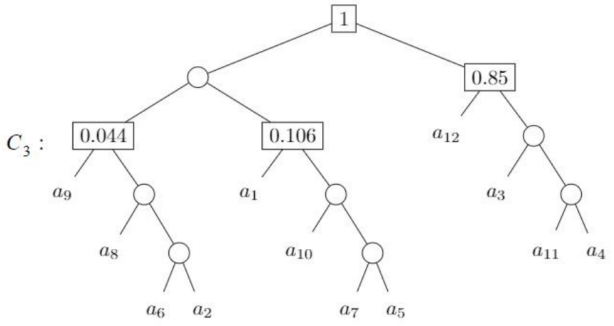


Figure 2.3: *ProgressiveHuffmanCoding when round* $r = 3$. *Average Code Length* $ACL(C_3) = 3.526$

When $r = 4$

At the beginning of this iteration (previous pivots are highlighted in grey)

A	a_{12}	a_3	a_{11}	a_4	a_1	a_{10}	a_7	a_5	a_9	a_8	a_6	a_2
W	0.21	0.11	0.48	0.05	0.02	0.025	0.042	0.019	0.007	0.015	0.018	0.004

$pivotPos = 2$

After partition (current pivots are highlighted in blue)

A	a_{11}	a_{12}	a_3	a_4	a_7	a_{10}	a_1	a_5	a_6	a_8	a_9	a_2
W	0.48	0.21	0.11	0.05	0.042	0.025	0.02	0.019	0.018	0.015	0.007	0.004

Since $pivotPos = 2$, we do not require to call $GroupAlphabets$ (notice that W array is in non increasing order). Just reverse the array to form initial queue Q and send it to the procedure $EfficientHuffmanCoding$.

$EfficientHuffmanCoding$ will return the following tree C_4

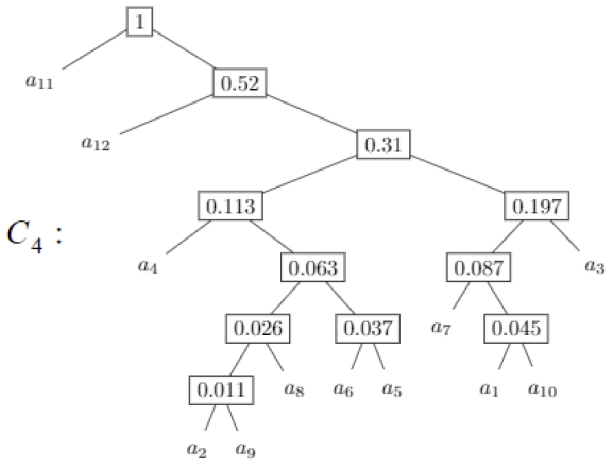


Figure 2.4: *ProgressiveHuffmanCoding when round* $r = 4$. *Average Code Length* $ACL(C_4) = 2.409$

2.4 Convergence Function

Lemma 2.4.1. *Let C_r be the partial solution after round r generated by the algorithm 3. Then $err_{C_r} \leq n/2^{r-2}$.*

Proof. As we want a bound of the partial solutions with respect to the error function defined in the section 2.1, we observe the following points.

- When we are at the r-th round, partitioning of the set will ensure that the rank of the alphabet with largest weight will be at most $2n/2^{r-1}$.
- At r-th round root to leaf maximum length of the trees generated by the procedure *GroupAlphabets* is bounded by *pivotPos* which is again bounded by $2n/2^{r-1}$.
- Initial queue length (size of Q) at round r is at most 2^r. We feed Q to the procedure $EfficientHuffmanCoding$.
- Let l_i^r be the root to leaf length for alphabet a_i at the coding tree C_r. It has two parts namely $l_{i_g}^r$, root to leaf node for alphabet a_i after *GroupAlphabets* at r-th round and $l_{i_e}^r$, extra length added for a_i when we call $EfficientHuffmanCoding$ for r-th round. $l_i^r = l_{i_g}^r + l_{i_e}^r$.

So we have,

$$
\begin{aligned}
err_{C_r} &= ACL(C_r) - ACL(C) && [C \text{ be an optimal coding tree}] \\
&= (\sum_{i=1}^{n} w_i \times l_i^r) - ACL(C) \\
&= (\sum_{i=1}^{n} w_i \times (l_{i_g}^r + l_{i_e}^r)) - ACL(C) \\
&= \sum_{i=1}^{n} w_i \times l_{i_g}^r + (\sum_{i=1}^{n} w_i \times l_{i_e}^r - ACL(C)) \\
&\leq \sum_{i=1}^{n} w_i \times l_{i_g}^r && [\text{As, } \sum_{i=1}^{n} w_i \times l_{i_e}^r \leq ACL(C), \\
& && \text{intuitive since we have } min(2^r, n) \\
& && \text{many nodes in sorted order}] \\
&\leq \sum_{i=1}^{n} w_i \times (2n/2^{r-1}) && [\text{As, } l_{i_g}^r \leq 2n/2^{r-1}] \\
&= n/2^{r-2} && [\text{As, } \sum_{i=1}^{n} w_i = 1]
\end{aligned}
$$

□

Lemma 2.4.2. *The algorithm will output an optimal coding tree at final round i.e.* $err_{C_k} = 0$.

Proof. This is due to the fact that alphabets will be sorted according to their weights at final round (Alewijnse et al. [2]) and once they are sorted Leeuwen's algoritm [10] ensures that we get an optimal coding tree. □

Theorem 2.4.1. *Given a set of* n *alphabets and their corresponding weights, there is a progressive algorithm for finding an optimal coding tree with convergence function* $f_{conv}(r) = n/2^{r-2}$ *for* $r = 1, 2, \ldots, k-1$ *and* $f_{conv}(k) = 0$ *with respect to the error function defined in section 2.1.*

Proof. The theorem directly follows from Lemma 2.4.1 and Lemma 2.4.2. □

Theorem 2.4.2. *Overall running time of the algorithm (ProgressiveHuffmanCoding 3) given a set of* n *alphabets and their corresponding weights is* $O(n \log n)$.

Proof. Step 4 of the algorithm (ProgressiveHuffmanCoding 3) ensures that there are $O(\log n)$ many rounds and as discussed earlier each round takes $O(n)$ time to compute and output a partial solution. Hence the claim holds. □

Chapter 3

Convex Hull in $\mathbb{R}^2$

We have described the problem in section 1.2.2. In this chapter first we define an error function then present the existing progressive algorithm, our modification and its analysis. In order to extend the algorithm to higher dimensions, we have slightly modified the notations as used in the article [2].

3.1 Error Function

Given $P = \{p_1, p_2, \ldots, p_n\}$, a set of n points in $\mathbb{R}^2$, and a partial solution (also a convex polygon) $C \subseteq CH(P)$, the error function which Alewijnse et al. [2] have considered is

$$err_C = \max_{\vec{v}} \left\{ 1 - \frac{width_{\vec{v}}(C)}{width_{\vec{v}}(CH(P))} \right\} \tag{3.1}$$

$\vec{v}$ is any vector of the form $x\hat{i} + y\hat{j}$. Let the angle between $\vec{v}$ and the positive x-axis be θ. Then $width_{\vec{v}}(C)$ is the distance between two tangent lines of C making an angle $\theta + \pi/2$ with the positive x-axis.

In the Figure 3.1, $P = \{p_1, p_2, \ldots, p_{10}\}$. Polygon C whose edges are colored red is a partial solution. $\vec{v}$ is any arbitrary vector. l_1, l_2 are the tangent lines of C and l_1, l_3 are tangent lines of $CH(P)$ perpendicular to $\vec{v}$. Then $width_{\vec{v}}(C)$ is the distance between l_1, l_2 and $width_{\vec{v}}(CH(P)$ is the distance between l_1, l_3.

Since partial solution C is a convex hull of a subset of the original points, so it will never be the case that $width_{\vec{v}}(C) > width_{\vec{v}}(CH(P))$. Also note that according to the framework of progressive algorithm at final round, err_C should be 0 which is only possible when convex polygon $C = CH(P)$ otherwise if, $C \subset CH(P)$ then there will always exist a vector $\vec{v}$ such that $width_{\vec{v}}(C) < width_{\vec{v}}(CH(P))$.

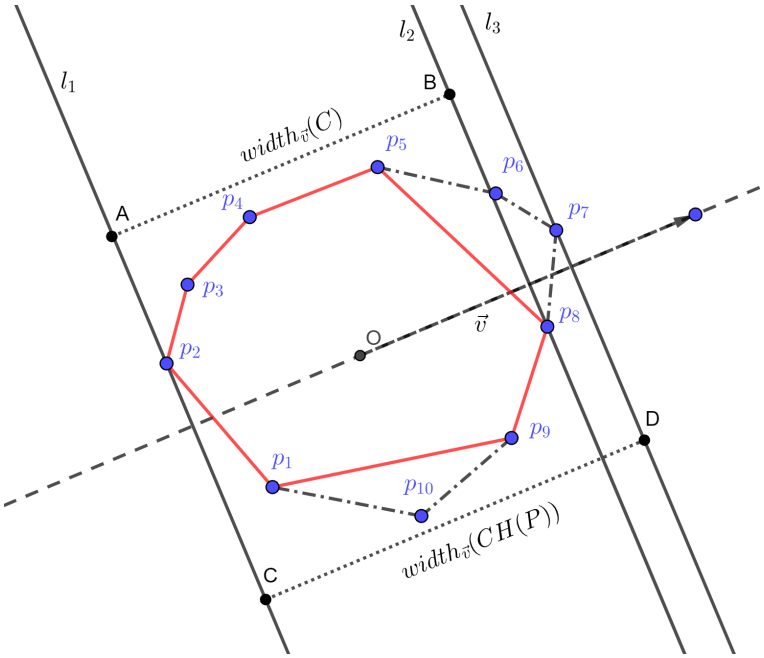


Figure 3.1: *$width_{\vec{v}}(C)$ and $width_{\vec{v}}(CH(P))$*

3.2 Designing Algorithm

First we discuss the existing algorithm developed by Alewijnse et al. [2] then present our randomized version of the algorithm.

3.2.1 Existing Algorithm

First we need a preprocessing step which will ensure that width of $CH(P)$ is roughly same in all directions. More precisely, $CH(P)$ will be *α-flat* for some constant $0 < \alpha < 1$ in the following sense: there are concentric circles C_{in} and C_{out} such that $C_{in} \subset CH(P) \subset C_{out}$ and $radius(C_{in}) = \alpha \cdot radius(C_{out})$. We can do so due to the following lemma [1].

Lemma 3.2.1. *For any set P of n points in $\mathbb{R}^2$ such that area of $CH(P)$ is nonzero, there is an affine transformation τ such that*

(i) $\tau(P)$ is α-flat and

(ii) For any $Q \subseteq P$ we have

$$\frac{width_{\vec{v}}(CH(Q))}{width_{\vec{v}}(CH(P))} = \frac{width_{\vec{v}}(CH(\tau(Q)))}{width_{\vec{v}}(CH(\tau(P)))}$$

Such a transformation, together with the circles C_{in} and C_{out} can be computed in $O(n)$ time.

We assume that C_{in} and C_{out} are centered at origin. Let $V(r)$ be the set of unit vectors we are working with in the r-th round and $P(r) \subseteq P$ be the set of extreme points, the algorithm selected during the computation of the r-th round. $C_r = CH(P(r))$ is the partial solution after r-th round of the progressive algorithm.

Constructing Initial Hull [2]

1. $V(1) = \{\hat{i}, \hat{j}, -\hat{i}, -\hat{j}\}$. Notice that they are equally spaced on circumference of a unit radius circle.

2. Let $p(\vec{v}) \in P$ denote the point that is extreme in the direction of $\vec{v}$, $\rho(\vec{v})$ denote the ray from origin with the direction same as $\vec{v}$ and $e(\vec{v})$ denote the edge of $CH(P)$ that is intersected by $\rho(\vec{v})$. One can find $e(\vec{v})$ using linear programming in $O(n)$ time [7] where the points in P induce constraints. Then initial extreme points are

$$P(1) := \{p(\vec{v}) : \vec{v} \in V(1)\} \cup \{p : p \text{ is an endpoint of } e(\vec{v}) \text{ for some } \vec{v} \in V(1)\}$$

3. $C_1 = CH(P(1))$ is the initial convex hull or the partial solution reported after round 1.

4. For each edge e of C_1, mark the points which belong to the cone joining origin with two endpoints of e but not within C_1. This step can be done in $O(n)$ time.

Remark. *Since there are* 4 *unit vectors in* $V(1)$*, so there can be at most* 4 *extreme points in those directions and we can have at most total* 8 *endpoints of some edge* $e(\vec{v})$*. Thus* $P(1)$ *contains not more than* 12 *points of* P *and* $C_1 = CH(P(1))$ *can be found in* $O(1)$ *time. Obviously, total time to construct initial hull is* $O(n)$*.*

Constructing C_r from C_{r-1} [2]

In a generic round r, we are given the polygon C_{r-1}. For each edge of C_{r-1}, a set $P(e)$ is defined as follows. Consider the rays from the origin through the vertices of C_{r-1}. These rays partition $\mathbb{R}^2$ into cones. *cone*(e) corresponds to the edge e of C_{r-1}. Now $P(e)$ contains the points from P that lie in *cone*(e) but outside C_{r-1}. Basically, points in $P(e)$ can still appear as vertices of $CH(P)$, other points that are vertices of C_{r-1} will be vertices of $CH(P)$ as well, and the points which are inside C_{r-1} are eliminated from the contest. Now proceed as follows.

1. Determine $V(r)$, set of 2^{r+1} many unit vectors such that they are equally spaced on the circumference of a unit radius circle and $\hat{i}$ must belong to the set.

2. For each new vector $\vec{v} \in V(r) \setminus V(r-1)$ find the edge e of C_{r-1} such that $p(\vec{v}) \in P(e)$. This entire step can be done in $O(n)$ time for all the new vectors by walking around C_{r-1}. There can be at most one vector $\vec{v}$ per edge e of C_{r-1}

because in between any two new direction vector there is a direction vector in $V(r)$.

3. For each new direction $\vec{v} \in V(r) \setminus V(r-1)$ find the edge e of C_{r-1} which is intersected by the ray $\rho(\vec{v})$. Here also we can have only one such edge.

4. For each edge e of C_{r-1}, if $P(e) \neq \phi$ find a ray $\vec{\rho}_e$ from origin such that it contains at most $|P(e)|/2$ many points from $P(e)$ on either side.

5. Now for each edge e with $P(e) \neq \phi$ do the following.

   - If in Step 2 we found a vector $\vec{v}$ with $p(\vec{v}) \in P(e)$, then compute the extreme point $p(\vec{v})$ of $P(e)$ in the direction of vector $\vec{v}$.
   - If in Step 3 we found a vector $\vec{v}$ which intersects e then compute the edge $e(\vec{v})$ of $CH(P)$ that is intersected by the ray $\rho(\vec{v})$ using linear programming where the points in $P(e)$ together with the endpoints e induce the constraints.
   - Also find the edge of $CH(P)$ that is intersected by $\vec{\rho}_e$ as found in Step 4 again using linear programming.

6. Step 5 gives upto five new vertices per edge e of C_{r-1}. These are inserted in between e to obtain the convex chain replacing e in new intermediate solution C_r. Now for each new edge e', we also need to compute $P(e')$. For this, we can check each point p of old set $P(e)$, if p lies inside C_r, then just remove it, otherwise find e' from the new edges in this chain such that $p \in P(e')$ and insert it to $P(e')$. This will take $O(|P(e)|)$ time per old chain of C_{r-1}.

Remark. *All of the steps individually can be done in $O(n)$ time. Thus each round of this progressive algorithm can be done and partial solution can be reported in $O(n)$ time.*

3.2.2 Randomized Progressive Algorithm

The above algorithm chooses the set of vectors (which are computed in each round) in deterministic way. We wanted to see if randomization helps and compute the set of vectors.

Computing set $V(r)$ at round r

Suppose we are given set of unit vectors $V(r-1)$ and we want to compute a set of unit vectors for the r-th round. We do as follows.

1. For each $\vec{v}$ of $V(r-1)$ find the point of intersection between unit radius circle centered at origin and a ray from origin in the direction of $\vec{v}$. Let $U(r-1) = \{u_1, u_2, \dots\}$ be the set of all such points in the same order as the corresponding vectors.

2. Initialize $U(r) = \phi$. For each of the two consecutive points u_i, u_{i+1} generate two points in between using the following rule.
   - Let the short arc length joining u_i and u_{i+1} be θ.
   - Generate two point uniformly randomly on the arc. These two points divide the arc into three smaller arcs. If length of any of these new three arcs is greater than $\theta/2$ then repeat this step, otherwise insert those two points along with u_i and u_{i+1} to the set $U(r)$ in order of their position in the arc.
3. Remove duplicates from $U(r)$ by a linear scan. Construct $V(r)$ according to the position vectors of the points of $U(r)$.

Constructing Initial Hull

For constructing initial hull we follow the same procedure as the previous algorithm (at section 3.2.1). Notice that during the procedure we set $V(1) = \{\hat{i}, \hat{j}, -\hat{i}, -\hat{j}\}$ which is necessary to generate set of vectors for upcoming rounds. Thus algorithm reports C_1.

Constructing C_r from C_{r-1}

We generate the set of vectors $V(r)$ using the method discussed above. Other steps will be quite similar to the earlier one (at section 3.2.1). Earlier we were having only one new vector in between old vectors, here we have two new vectors in between the vectors of the previous round. We do as follows.

1. For each vector $\vec{v} \in V(r) \setminus V(r-1)$ find the edge e of C_{r-1} such that $p(\vec{v}) \in P(e)$.
2. For each direction $\vec{v} \in V(r) \setminus V(r-1)$ find the edge e of C_{r-1} which is intersected by the ray $\rho(\vec{v})$.
3. For each edge e of C_{r-1}, if $P(e) \neq \phi$ find a ray $\vec{\rho}_e$ from origin such that it contains at most $|P(e)|/2$ many points from $P(e)$ on either side.
4. Now for each edge e with $P(e) \neq \phi$ do the following.
   - If in Step 1 we found some vector $\vec{v}$ with $p(\vec{v}) \in P(e)$, then compute the extreme point $p(\vec{v})$ of $P(e)$ in the direction of each such vector $\vec{v}$.
   - If in Step 2 we found some vector $\vec{v}$ which intersects e then compute the edge $e(\vec{v})$ of $CH(P)$ that is intersected by the ray $\rho(\vec{v})$ using linear programming where the points in $P(e)$ together with the endpoints e induce the constraints.
   - Find the edge of $CH(P)$ that is intersected by $\vec{\rho}_e$ as found in Step 3 again using linear programming.

5. Above step gives upto eight new vertices per edge e of C_{r-1}. These are inserted in between e to obtain the convex chain replacing e in new intermediate solution C_r. Now for each $p \in P(e)$, check whether it lies inside C_r. Otherwise find e' from these new edges such that $p \in P(e')$ and insert it to that set. This will take $O(|P(e)|)$ time per old chain of C_{r-1}.

3.3 Analysis of Algorithm

3.3.1 Convergence Function

Notice that, while constructing initial hull our set of vectors was $V(1) = \{\hat{i}, \hat{j}, -\hat{i}, -\hat{j}\}$. Angle between two consecutive vector is $\pi/2$. While constructing $V(r)$ from $V(r-1)$ our randomized algorithm ensures angle between two consecutive vector is upper bounded by half of the same of previous round. Combining these two facts, we can say that at round r, angle between two consecutive vector is upper bounded by $\pi/2^r$. Now we can use the lemma by Alewijnse et al. [2] to get the following lemma.

Lemma 3.3.1. *For round r, partial solution C_r reported by the randomized progressive algorithm satisfy the following property*

$$err_{C_r} = O(1/2^{2r}) \tag{3.2}$$

where err_C is as defined as in equation 3.1

Therefore, the convergence function for the randomized progressive algorithm is $f_{conv}(r) = O(1/2^{2r})$ with respect to the error function.

3.3.2 Expected Running Time

For determining the expected running time of the randomized progressive algorithm, we need the expected running time per round and a upper bound on number of rounds.

Lemma 3.3.2. *Maximum expected running time per round $\mathbb{E}_{max}[T(n)]$ of the progressive randomized algorithm discussed above is $O(n)$.*

Proof. For $r = 1$, we are constructing the initial hull following the same procedure as the existing algorithm 3.2.1. So worst case running time for $r = 1$ is $O(n)$.

For $r > 1$ we can divide each round in two parts.

- Computing V_r from V_{r-1}
- Constructing C_r from C_{r-1}

The time spent for constructing C_r from C_{r-1} at Step 1-5 is $O(n + \sum_e |P(e)|)$, where the sum is taken over all the edges e of C_{r-1}. Since $\sum_e |P(e)| < n$, the time for constructing C_r from C_{r-1} given set of vectors $V(r)$ is $O(n)$ in worst case.

Now it remains to find the expected time for computing V_r from V_{r-1}. Suppose we have arc of length θ. We take two random points x, y on the arc, both uniform on $[0, \theta]$. Let $A = min(x, y), B = max(x, y)$ and $C = max(A, B - A, \theta - B)$.

If $C > \theta/2$, randomized algorithm will discard that x, y and will generate the points again. So $C \leq \theta/2$ is favourable case for our algorithm. Now,

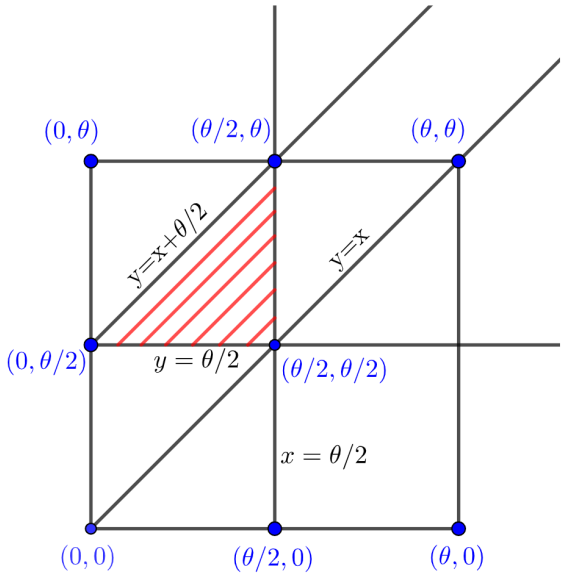


Figure 3.2: *Representing probability as area in the square of side θ*

$$\begin{aligned} Pr(C \leq \theta/2) &= Pr(max(A, B - A, \theta - B) \leq \theta/2) \\ &= Pr(A \leq \theta/2, B - A \leq \theta/2, \theta - B \leq \theta/2) \\ &= 2.Pr(x \leq \theta/2, y - x \leq \theta/2, \theta - y \leq \theta/2, x < y) \\ &\qquad\qquad \text{[From symmetry of } x, y\text{]} \\ &= 2.Pr(x \leq \theta/2, y \leq x + \theta/2, y \geq \theta/2, x < y) \end{aligned}$$

To calculate $Pr(x \leq \theta/2, y \leq x + \theta/2, y \geq \theta/2, x < y)$ we take a square of side θ. Any point (x, y) inside the square represents uniformly randomly selected two points on the arc of length θ. It is trivial that $x, y \in [0, \theta]$. Ratio of an area of a region to the area of the square denotes the probability of that region.

We plot $x \leq \theta/2, y \leq x + \theta/2, y \geq \theta/2, x < y$ and get their intersection as the triangle shaded in the Figure 3.2. Area of the triangle is $\frac{1}{2} \times \frac{\theta}{2} \times \frac{\theta}{2} = \frac{\theta^2}{8}$. Thus,

$$\begin{aligned} Pr(C \leq \theta/2) &= 2 \times Pr(x \leq \theta/2, y \leq x + \theta/2, y \geq \theta/2, x < y) \\ &= 2 \times (\frac{\theta^2}{8} : \theta^2) \\ &= 2 \times \frac{1}{8} = 0.25 \end{aligned}$$

Favourable or success probability for selecting two uniformly random points on the arc to satisfy the criteria is 0.25. Therefore, the expected number of trials for the points to satisfy the criteria is $1/0.25 = 4$. Since number of vectors in the set V_{r-1} in worst case is bounded by $O(n)$, so expected time for computing V_r from V_{r-1} is $O(n)$.

Thus maximum of the expected running time per round $\mathbb{E}_{max}[T(n)]$ of the progressive randomized algorithm is $O(n)$.

□

Similar to a theorem by Alewijnse et al. [2] we can get the following theorem.

Theorem 3.3.1. *There is a randomized progressive algorithm to compute the convex hull of a set of n points in $\mathbb{R}^2$ that runs in at most $k := \lfloor \log n \rfloor$ rounds, maximum expected running time of each round being $O(n)$, with convergence function $f_{conv}(r) = O(1/2^{2r})$ with respect to the error function defined in equation 3.1.*

Proof. The bound on convergence rate follows from Lemma 3.3.1. Maximum of the expected running time we get from Lemma 3.3.2. Step 3 of the progressive algorithm makes sure that $|P(e)| < n/2^r$ for any edge e of C_r. So total number of rounds is at most $\log n$. □

Chapter 4

Convex Hull in $\mathbb{R}^3$

We have described the problem in section 1.2.2. In this chapter we shall extend the progressive algorithm for convex hull in $\mathbb{R}^2$ by Alewijnse et al. [2] to points in $\mathbb{R}^3$. We interpret the notations in terms of this problem. During analysis, we shall see the bottleneck of our algorithm in terms of the running time. Here we have used an observation to upper bound the error of a partial solution. There is no formal proof for the result. Though our observation is backed up by experimental results.

4.1 Error Function

Given $P = \{p_1, p_2, \ldots, p_n\}$, a set of n points in $\mathbb{R}^3$ and a partial solution $C \subseteq CH(P)$, the error function we have used is

$$err_C = \max_{\vec{v}} \left\{ 1 - \frac{width_{\vec{v}}(C)}{width_{\vec{v}}(CH(P))} \right\} \tag{4.1}$$

$\vec{v}$ is any vector of the form $x\hat{i} + y\hat{j} + z\hat{k}$ and $width_{\vec{v}}(C)$ is the distance between two tangent planes of the convex polygon C whose normal vector is $\vec{v}$.

4.2 Designing Algorithm

Like what we did when the points were in $\mathbb{R}^2$, here also we preprocess the points so that $CH(P)$ becomes α-flat. We use following lemma by Agarwal et al. [1].

Lemma 4.2.1. *[1] For any set P of n points in $\mathbb{R}^3$ such that volume of $CH(P)$ is non zero, there is an affine transformation τ such that,*

(i) $\tau(P)$ is α-flat, i.e., for some constant $0 < \alpha < 1$ there are concentric spheres C_{in}, C_{out} centered at origin such that $C_{in} \subset CH(\tau(P)) \subset C_{out}$ and $radius(C_{in}) = \alpha \cdot radius(C_{out})$.

(ii) For any $Q \subseteq P$ we have

$$\frac{width_{\vec{v}}(CH(Q))}{width_{\vec{v}}(CH(P))} = \frac{width_{\vec{v}}(CH(\tau(Q)))}{width_{\vec{v}}(CH(\tau(P)))}$$

Such a transformation, together with the circles C_{in} and C_{out} can be computed in $O(n)$ time.

This preprocessing step will make sure that width of the convex hull is roughly same in all direction. Now we define few notations that will be useful to present the progressive algorithm.

- Let $P = \{p_1, p_2, \ldots, p_n\}$ be given set of points after applying transformation τ where $p_i \in \mathbb{R}^3$ for $i = 1, 2, \ldots, n$. We assume that points are in *general position* i.e., no three points lie on same line, no four points lie on the same plane.
- $p(\vec{v}) \in P$ denotes extreme point with respect to the direction of $\vec{v}$. It is extreme in the sense that if we take the plane with normal vector $\vec{v}$ and passing through the point $p(\vec{v})$ then all the points of P will belong to same side of the plane.
- $s(\vec{v})$ denotes the surface of $CH(P)$ that is hit by the ray that was shot from the origin with direction of $\vec{v}$.
- $P(r) \subseteq P$ denotes the set of extreme points after r-th round of the algorithm.
- Let $V(r)$ denote the set of vectors the algorithm uses during r-th round of computation and $V'(r)$ be the set of tuples during r-th round. Each tuple is of the form $t = (\vec{v_1}, \vec{v_2}, \vec{v_3})$. The set $V'(r)$ helps to compute $V(r+1)$.
- Let C_r be the convex polygon reported as partial solution of r-th round. Due to restriction of point set P, each surface is triangular. For each surface if we draw a plane taking a side of the surface and passing through origin then entire $\mathbb{R}^3$ will be divided into hollow cones. We denote the cone corresponding to surface s by $cone(s)$ and $P(s)$ is the set of points from P which are in $cone(s)$ but outside the convex polygon C_r.

Constructing Initial Hull C_1

1. Take $V(1) = \{\hat{i}, -\hat{i}, \hat{j}, -\hat{j}, \hat{k}, -\hat{k}\}$ and $V'(1) = \{(\hat{i}, \hat{j}, \hat{k}), (-\hat{i}, \hat{j}, \hat{k}), (-\hat{i}, -\hat{j}, \hat{k}), (\hat{i}, -\hat{j}, \hat{k}), (\hat{i}, \hat{j}, -\hat{k}), (-\hat{i}, \hat{j}, -\hat{k}), (-\hat{i}, -\hat{j}, -\hat{k}), (\hat{i}, -\hat{j}, -\hat{k})\}$. Notice that the angle between any two of the vectors in a tuple is $\pi/2$.
2. Find $p(\vec{v}) \in P$ for each $\vec{v} \in V(1)$.
3. Find $s(\vec{v})$ for each $\vec{v} \in V(1)$. We can find $s(\vec{v})$ using linear programming for three dimension [7] in $O(n)$ time.

4. $P(1) := \{p(\vec{v}) : \vec{v} \in V(1)\} \cup \{p : p \text{ is a vertex of } s(\vec{v}) \text{ for some } \vec{v} \in V(1)\}$.

5. There can be at most *twenty four* points in $P(1)$. Report $C_1 = CH(P(1))$ as the initial hull or the partial solution after round 1. For each surface s of C_1 determine the set $P(s)$ (points in $cone(s)$ but outside C_1) as it will be helpful from next round.

Computing set $V(r)$ at round r

Now we present our idea to determine the set of vectors for r-th round. We already have set of tuples $V'(r-1)$ from previous round. Our objective is to bound the angle between two vectors in a tuple by half of the previous round. Initialize $V(r) = V(r-1)$ and $V'(r) = \phi$. For each $t = (\vec{v_1}, \vec{v_2}, \vec{v_3}) \in V'(r-1)$ we do as follows.

1. Shoot three rays from the origin with direction vectors $\vec{v_1}, \vec{v_2}, \vec{v_3}$, respectively and find their intersection with unit radius sphere centered at the origin. Let the points be A_1, A_2, A_3, respectively. We can find the points simply by dividing the vectors by their corresponding norms.

2. Construct the triangle $\triangle A_1A_2A_3$ and find the point within triangle whose distance from its closest vertex is maximum.

3. Without loss of generality, let $A_1A_2 \geq A_2A_3 \geq A_3A_1$. Also let position vector of the mid-point of the sides be $\vec{v}_{a_3}, \vec{v}_{a_1}, \vec{v}_{a_2}$, respectively. We now have three cases.

   (a) If $\triangle A_1A_2A_3$ is acute-angled triangle, then their circumcenter is the required point which we can find easily as the coordinates of A_1, A_2, A_3 are known. Let the position vector of the point be $\vec{v_c}$.
   - Set $V'(r) = V'(r) \cup \{(\vec{v_c}, \vec{v_1}, \vec{v}_{a_3}), (\vec{v_c}, \vec{v_2}, \vec{v}_{a_3}), (\vec{v_c}, \vec{v_2}, \vec{v}_{a_1}), (\vec{v_c}, \vec{v_3}, \vec{v}_{a_1}), (\vec{v_c}, \vec{v_3}, \vec{v}_{a_2}), (\vec{v_c}, \vec{v_1}, \vec{v}_{a_2})\}$.
   - Set $V(r) = V(r) \cup \{\vec{v_c}, \vec{v}_{a_1}, \vec{v}_{a_2}, \vec{v}_{a_3}\}$.

   (b) If $\triangle A_1A_2A_3$ is right-angled triangle, then circumcenter is the required point and it is the midpoint of the largest side, i.e., position vector is $\vec{v}_{a_3}$.
   - Set $V'(r) = V'(r) \cup \{(\vec{v}_{a_3}, \vec{v}_{a_2}, \vec{v_1}), (\vec{v}_{a_3}, \vec{v}_{a_2}, \vec{v_3}), (\vec{v}_{a_3}, \vec{v}_{a_1}, \vec{v_3}), (\vec{v}_{a_3}, \vec{v}_{a_1}, \vec{v_2})\}$.
   - Set $V(r) = V(r) \cup \{\vec{v}_{a_1}, \vec{v}_{a_2}, \vec{v}_{a_3}\}$.

   (c) Finally, if $\triangle A_1A_2A_3$ is obtuse-angled triangle then the required point will lie on largest side A_1A_2. Intersection of the side A_1A_2 with perpendicular bisector of the second largest side A_2A_3 is the point whose closest distance from any of the vertices is maximum. Let the position vector of the point be $\vec{v_c}$.
   - Set $V'(r) = V'(r) \cup \{(\vec{v_c}, \vec{v}_{a_1}, \vec{v_2}), (\vec{v_c}, \vec{v}_{a_1}, \vec{v}_{a_3}), (\vec{v}_{a_1}, \vec{v}_{a_2}, \vec{v}_{a_3}), (\vec{v}_{a_3}, \vec{v}_{a_2}, \vec{v_1}), (\vec{v}_{a_1}, \vec{v}_{a_2}, \vec{v_3})\}$.

- Set $V(r) = V(r) \cup \{\vec{v_c}, \vec{v}_{a_1}, \vec{v}_{a_2}, \vec{v}_{a_3}\}$.

Remark. *For each tuple we can do the above steps in $O(1)$ time. The number of tuples for round 1 is $|V'(1)| = 8$. Each tuple can generate at most six tuples in the worst case following above procedure. Thus we have $|V'(r)| \leq 8 \times 6^{r-1}$.*

By design, the above procedure tries to halve the angle between any two vectors for a tuple in next round. Let θ^r_{max} be the maximum angle between two vectors within a tuple for r-th round. Experimentally we have seen $\theta^r_{max} < 2\pi/2^r$ for $r = 1, 2, \ldots, 30$. Moreover, for each $r = 2, 3, \ldots, 29$, it is observed that $\theta^r_{max} \geq 2 \cdot \theta^{r+1}_{max}$. Though a formal proof would have been better, for our analysis, we shall assume $\theta^r_{max} < 2\pi/2^r$ as observed from the experiment.

Constructing C_r from C_{r-1}

1. Determine the set $V(r)$ using the above procedure.
2. For each direction $\vec{v} \in V(r) \setminus V(r-1)$, find the surface s of C_{r-1} such that $p(\vec{v}) \in P(s)$.
3. For each direction $\vec{v} \in V(r) \setminus V(r-1)$ find the surface s of C_{r-1} which is intersected by the ray $\rho(\vec{v})$.
4. Now for each s with $P(s) \neq \phi$ do the following.
   - If in Step 2 we found $\vec{v}$ with $p(\vec{v}) \in P(s)$, then compute the extreme point of $P(s)$ with respect to the direction of $\vec{v}$.
   - If in Step 3 we found $\vec{v}$ intersecting this surface s, then compute $s(\vec{v})$ as defined before.
   - Thus we get at most four new points for the surface s. Insert these vertices to replace s with a new set of surfaces. For each point in $P(s)$ determine the new set $P(s')$ where s' is a new surface.
   - For each new surface check if there exists any s' such that $|P(s')| > |P(s)|/2$. There can be at most one such surface. If s' exists, then replace it using median trick as discussed below.
5. Once algorithm computes the above steps, report current convex polygon C_r as the partial solution for r-th round.

Median Trick

In order to bound the number of rounds of the progressive algorithm, we want to make sure that for each surface s, the points outside hull, i.e., $P(s)$ reduces by at least half of its size in next round. If for some s', it violates, i.e., $|P(s')| > |P(s)|/2$, then we do as follows.

1. Find a plane passing through origin and intersecting surface s' such that both side of the plane contains at most $|P(s')|/2$ many points from $P(s')$.
2. Take the line segment of intersection of the plane and surface s'.
3. Find the position vector of the mid point of the line segment. Let it be $\vec{v}$.
4. Compute $s(\vec{v})$, surface of $CH(P)$ intersected by the ray $\rho(\vec{v})$. Thus we get three new extreme points along with three extreme points of s'.
5. Replace the old surface s' with new surfaces of convex hull of the six extreme points.
6. For each new surface s'' determine $P(s'')$. If there exist any s'' such that $|P(s'')| > |P(s)|/2$ then follow median trick again for s''.

4.3 Analysis of Algorithm

4.3.1 Convergence Function

For finding convergence function of the progressive algorithm with respect to the error function we follow the technique used by Alewijnse et al. [2]. Let $\vec{v}$ be any arbitrary direction, then we need to bound $1 - \frac{width_{\vec{v}}(C_r)}{width_{\vec{v}}(CH(P))}$ for partial solution after round r. Let $plane_O$ be the equation of a plane passing through origin and normal to $\vec{v}$. If we rotate the plane anti-clockwise, let $\vec{v}^*$ be the vector from $V(r)$ it intersects first after crossing $\vec{v}$.

According to our selection of vectors we know that the angle between $\vec{v}$ and $\vec{v}^*$ is bounded by $2\pi/2^r$. Again let, $p(\vec{v}) \in cone(s)$ of C_r. If we take any extreme point from $cone(s)$ which is also a vertex of C_r then the angle between their position vector will again be bounded by $2\pi/2^r$. Therefore, following the method analogous to points in $\mathbb{R}^2$ we obtain

$$err_{C_r} = O(1/2^{2r}) \tag{4.2}$$

Thus error at the r-th round of the algorithm can be bounded by the convergence function $f_{conv}(r) = O(1/2^{2r})$.

4.3.2 Running Time

- While constructing C_r from C_{r-1} the "median trick" ensures that $|P(s)| < n/2^r$ for any surface s of C_r. This implies that the number of rounds of the progressive algorithm will be at most $k := \lfloor \log n \rfloor$.
- Preprocessing step can be done in $O(n)$ time and initial hull can also be computed in $O(n)$ time.

- As discussed earlier, computing the set $V(r)$ at round r takes $O(6^r)$ time. Since $r < \log n$, worst case time to compute $V(r)$ will be $O(n)$.
- Now coming to the construction time of C_r from C_{r-1}. Up to Step 3, we can do in $O(n)$ time. To bound the number of rounds, we are doing "median trick" at the next step. Since the procedure is recursive and it can continue till number of points for a surface reduces to half of the previous round, it can take in worst case $O(n \log n)$ time. This is the bottleneck of our algorithm, which increases total running time of the progressive algorithm.

We can summarize our result with the following theorem.

Theorem 4.3.1. *There is a progressive algorithm to compute the convex hull of a set of n points in $\mathbb{R}^3$ following general position assumption that runs in at most $k := \lfloor \log n \rfloor$ rounds, each taking at most $O(n \log n)$ time with convergence function $f_{conv}(r) = O(1/2^{2r})$ with respect to the error function defined in this chapter.*

Chapter 5

Conclusion and Future Works

For the huffman coding problem our progressive algorithm ensures $err_{C_r} \leq n/2^{r-2}$ but there is no guarantee that solutions are getting better and better. Precisely we can not say without proof that $err_{C_1} \geq err_{C_2} \geq \ldots \geq err_{C_k}$. It would be interesting to develop a progressive algorithm which actually ensures this monotonicity. In case of convex hull, we can clearly see this property holds as convex hull generated in current round will be subset of the convex hull to be generated in the next round.

For the convex hull in $\mathbb{R}^3$ we have imposed general position assumption for the point set. So there is a scope of improvement when we relax this restriction. Also we see overall running time of the algorithm is $O(n(\log n)^2)$. But there are already better algorithms for finding convex hull for points in $\mathbb{R}^3$. So we can think upon coming up with a progressive algorithm with overall running complexity that matches with the time complexity of best known algorithm.

There is a nice relationship between Voronoi diagrans and Delaunay triangulations of points in $\mathbb{R}^2$ and the convex hulls of a particular set of points in $\mathbb{R}^3$. So once we can develop an efficient progressive algorithm for convex hull for any set of points in $\mathbb{R}^3$, that would in turn help to design and analysis of progressive algorithms for Voronoi diagrams and Delaunay triangulations in 2-dimension.

Bibliography

[1] Pankaj K Agarwal, Sariel Har-Peled, Kasturi R Varadarajan, et al. Geometric approximation via coresets. *Combinatorial and computational geometry*, 52:1–30, 2005.

[2] Sander P. A. Alewijnse, Timur M. Bagautdinov, Mark de Berg, Quirijn W. Bouts, Alex P. ten Brink, Kevin Buchin, and Michel A. Westenberg. Progressive geometric algorithms. *JoCG*, 6(2):72–92, 2015. doi: 10.20382/jocg.v6i2a5. URL https://doi.org/10.20382/jocg.v6i2a5.

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996. doi: 10.1145/235815.235821. URL https://doi.org/10.1145/235815.235821.

[4] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7 (4):448 – 461, 1973. ISSN 0022-0000. doi: 10.1016/S0022-0000(73)80033-9. URL http://www.sciencedirect.com/science/article/pii/S0022000073800339.

[5] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952. doi: 10.1109/JRPROC.1952.273898. URL https://doi.org/10.1109/JRPROC.1952.273898.

[6] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. Efficient and progressive group steiner tree search. In *Proceedings of the 2016 International Conference on Management of Data*, pages 91–106, 2016.

[7] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31(1):114–127, 1984. doi: 10.1145/2422.322418. URL https://doi.org/10.1145/2422.322418.

[8] Amir Mesrikhani and Mohammad Farshi. Progressive sorting in the external memory model. *The CSI Journal on Computer Science and Engineering*, 15(2), 2018.

[9] Amir Mesrikhani, Mohammad Farshi, and Mansoor Davoodi. Progressive algorithm for euclidean minimum spanning tree. In *First Iranian Conference on*, page 29, 2018.

[10] Jan van Leeuwen. On the construction of huffman trees. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*, pages 382–410. Edinburgh University Press, 1976. URL http://www.staff.science.uu.nl/~leeuw112/huffman.pdf.