COL351 : Analysis and Design of Algorithm
Assignment 3

Gaurav Jain (2019CS10349) & T Abishek (2019CS10407)

1 Convex Hull

The Divide and Conquer algorithm to compute Convex Hull is present below:

Algorithm 1 Convex Hull Algorithm

```
1:  procedure ConvexHull(P = $\{(x_1,y_1),(x_2,y_2)....(x_n,y_n)\}$)
2:      if n $\leq$ 5 then
3:          Solve using Brute Force method
4:      Sort P in increasing order according to x coordinate
5:      Divide P into A and B such that: A contains the left $\lfloor \frac{n}{2} \rfloor$ points and B contains the right $\lceil \frac{n}{2} \rceil$ points.
6:      $Q_1 \leftarrow$ ConvexHull(A)
7:      $Q_2 \leftarrow$ ConvexHull(B)
8:      $Q \leftarrow$ Merge($Q_1$,$Q_2$)
9:      return $Q$

10: procedure Merge(A,B)
11:     $(x_1, y_1) \leftarrow$ UpperTangent(A,B)
12:     $(x_2, y_2) \leftarrow$ LowerTangent(A,B)
13:     U $\leftarrow$ {}
14:     $u \leftarrow x_1$
15:     while $u \neq x_2$ do
16:         Add $u$ to U
17:         $u \leftarrow$ next element in A in counter clockwise direction
18:     $u \leftarrow y_1$
19:     while $u \neq y_2$ do
20:         Add $u$ to U
21:         $u \leftarrow$ next element in B in clockwise direction
22:     return U

23: procedure UpperTangent(A,B)
24:     a $\leftarrow$ Rightmost point of A
25:     b $\leftarrow$ Leftmost point of B
26:     T $\leftarrow$ (a,b)
27:     while T = ab not a tangent to both A and B do
28:         while T not a tangent to A do
29:             a $\leftarrow$ next element in A in counter clockwise direction
30:         while T not a tangent to B do
31:             b $\leftarrow$ next element in B in clockwise direction
32:     return T

33: procedure LowerTangent(A,B)
34:     a $\leftarrow$ Index of rightmost point of A
35:     b $\leftarrow$ Index of leftmost point of B
36:     T $\leftarrow$ ab
37:     while T = ab not a tangent to both A and B do
38:         while T not a tangent to A do
39:             a $\leftarrow$ next element in A in clockwise direction
40:         while T not a tangent to B do
41:             b $\leftarrow$ next element in B in counter clockwise direction
42:     return T
```

Runtime Analysis:

The time complexity of the divide and conquer algorithm to compute convex hull is calculated using the recurrence relation: T(n) = 2T(n/2)+O(n)+merge operation time complexity. Median is calculated in O(n) using k^{th} smallest element problem discussed in lectures. The brute force method is used for base case when $n \leq 5$ which take $O(n^3)$ time for n points. Thus, there is only a small increase in runtime.
The time taken in merge operation depends on the time taken to calculate upper tangent and lower tangent of the two convex hulls to be merged and the time taken to merge these two convex hulls and the two tangents which is performed in a while loop iterating over all elements of A and B.
The lower tangent and upper tangent function has same time complexity which is $O(sizeA)$ for convex hull A. Thus, the time complexity of merge operation is $O(sizeA)$ + $O(sizeB)$ + $O(n)$ = $O(n)$.

Correctness of Algorithm:

Claim 1
The line given by the LOWERTANGENT procedure touches A and B at their lower halves and does not intersect A and B at any other points.
Proof
The procedure tends to move both points located on A and B in the same direction and this leads to a line segment which passes through the lowest end of the lower halves of A and B.
Due to this movement, line T becomes a direct tangent to the convex hulls A and B. This line also never intersect at any other point of A and B. □
Claim 2
The polygon formed after the merging step of two convex hulls A and B is a convex hull.
Proof Proof by Contradiction
Since A and B are convex and they are connected using upper and lower tangents, the merged polygon is also convex in nature. Suppose the polygon formed after merging A and B is not a convex hull. This means that there is at least one extra point in the merged polygon.
So after removing this point, the new polygon is a convex hull and the removed point should lie inside the new polygon. This implies that the removed point was part of a concave edge which is a contradiction as A and B are convex in nature. So the above claim holds. □

We can proof the correctness of the algorithm using induction on number of points present in the plane. By dividing the plane into smaller and smaller parts, we reach the base case which is solved using brute force method. Using claim 1 and 2, merging the smaller convex hulls results in the final answer.

2 Particle Interaction

Observation:

$\mathrm{F_j} = \mathrm{C}\ \mathrm{q_j} \left(\sum_{i<j} \frac{q_\mathrm{i}}{(j-i)^2} - \sum_{i>j} \frac{q_\mathrm{i}}{(j-i)^2} \right)$
$\mathrm{F_j} = \mathrm{C}\ \mathrm{q_j}\ \mathrm{c_j}$

Where,
$\mathrm{c_k} = \sum_{i<j} \frac{q_\mathrm{i}}{(j-i)^2} - \sum_{i>j} \frac{q_\mathrm{i}}{(j-i)^2}$ for all $\mathrm{k} \in [1,\mathrm{n}]$ (1)

Consider the two polynomials,
$\mathrm{A(x)} = \mathrm{q_1x} + \mathrm{q_2x^2} + \ldots + \mathrm{q_nx^n}$

$\mathrm{B(x)} = -\frac{1}{(n-1)^2}x - \ldots - \frac{1}{4}x^{\mathrm{n-2}} - x^{\mathrm{n-1}} - 0.x^{\mathrm{n}} + x^{\mathrm{n+1}} + \frac{1}{4}x^{\mathrm{n+2}} + \ldots + \frac{1}{(n-1)^2}x^{\mathrm{2n-1}}$

Consider the product, P(x) = A(x)∗B(x).
Co-efficient of $\mathrm{x^k}$ for k in range (n,2n+1) $= \sum_{l=1}^{n} a_\mathrm{l}\mathrm{b_{k-l}}$ (2)

From equation (1) and equation (2), it is observable that coefficient of x^{n+j} in P(x) = c_j. So, to obtain c_j, we need P(x) which can be calculated using Fast Fourier Transform discussed in the lectures. The time complexity here is $O(nlogn)$.

Now, using $\mathrm{c_j}$ and equation (1), we can obtain $\mathrm{F_j}$ for all the particles $(j \in [1, n])$ within $O(nlogn)$ time complexity.

Algorithm 2 Particle Interaction Algorithm

```
 1: procedure PARTICLEINTERACTION
 2:     $A \leftarrow [q_1, q_2, ..., q_\mathrm{n}]$
 3:     $B \leftarrow [-\frac{1}{(n-1)^2}$, - $\frac{1}{(n-2)^2}, -1, 0, 1, \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2}]$
 4:     $P \leftarrow$ MULTIPLY(A, B)
 5:     $TotalForce \leftarrow$ Array of size n
 6:     for $i \leftarrow 1$ to $n$ do
 7:         $TotalForce[i] \leftarrow C * A[i] * P[n+i]$
 8: function MULTIPLY($A, B$)
 9:     $dft_A \leftarrow DFT(A)$
10:     $dft_B \leftarrow DFT(B)$
11:     $dft_P \leftarrow$ Array of size 2n+1
12:     for $i \leftarrow 1$ to $len(dft_A)$ do
13:         $dft_P[i] \leftarrow dft_A[i] * dft_B[i]$
14:     $P \leftarrow InvDFT(dft_P)$
15:     return $P$
```

Runtime Analysis:
The above algorithm returns $TotalForce$ array which has j^{th} element as the final totalforce acting on j^{th} particle. The MULTIPLY function invloves calculating DFT for 2 vectors, each has time complexity of O(nlogn) as discussed in the lecture.

Next we have a for loop which has O(n) time complexity. Further, there is InverseDFT which again takes O(2nlog2n). Next, after the MULTIPLY function we calculate the $TotalForce$ using elements of P and A. This for loop takes O(2n) time complexity. So,

$$T(n) = 2*O(nlogn) + O(n) + O(2nlog2n) + O(2n) = O(nlogn)$$

3 Distance Computation using Matrix Multiplication

(a) **Prove that the graph H = (V, E_H) can be computed from G in O(n^ω) time, where ω is the exponent of matrix-multiplication.**

Solution: Using the property of transitive closure(discussed in lecture notes). Assume A is the adjacency matrix of graph G. Then, $(A^k)_{ij} > 0$ iff there is a walk of length exactly k from i to j. Here edges exist between vertices in H, if there is a walk of length 2 or 1 in graph G. (A^2) gives the adjacency matrix of edges who had walks of length of 2 in G. So, Adjacency matrix of H is given by (A^2) + A. Matrix Addition is O(n^2) and Matrix Multiplication is O(n^ω). So, overall time complexity is O(n^ω) if $\omega > 2$

(b) **Argue that for any x, y $\in$ V , D_H(x, y) = $\lceil D_G(x, y)/2 \rceil$.**

Solution: Assume for some vertice x,y the distance in graph G is k. The Path is (x,a_1, a_2, a_3 .... $a_(k-1)$, y). As all the vertices who have walk of distance 2 between them get edges in the graph H, x and a_2, a_2 and a_4.... all have edges between them. So the path between x and y becomes (x, a_2, a_4 .... $a_(k-2)$, y) if k is even and (x, a_2, a_4 .... $a_(k-1)$, y) if k is odd. The distance is k/2 if k is even and k/2 + 1 if k is odd. $\implies$ D_H(x, y) = $\lceil D_G(x, y)/2 \rceil$ Proved.

(c) **Let A_G be adjacency matrix of G, and M = $D_H \star A_G$. Prove that for any x, y $\in$ V, the following holds.**

$$D_G(x,y) = \begin{cases} 2D_H(x,y) & M(x,y) \geqslant \mathbf{degree_G(y) \cdot D_H(x, y)} \\ 2D_H(x,y) - 1 & M(x,y) \lneq \mathbf{degree_G(y) \cdot D_H(x, y)} \end{cases}$$

(d) **Use (c) to argue that D_G is computable from D_H in O(n^ω) time.**

From (c) we have $D_G(x,y) = \begin{cases} 2D_H(x,y) & M(x,y) \geqslant \text{degree}_G(y) \cdot D_H(x, y) \\ 2D_H(x,y) - 1 & M(x,y) \lneq \text{degree}_G(y) \cdot D_H(x, y) \end{cases}$ where A_G is the adjacency matrix of G and M = $D_H \star A_G$.

As, D_G is computable using D_H which can be computed in O(n^ω) time. M can be computed in O(n^ω) time too. So, the overall time complexity is O(n^ω).

(e) **Prove that all-pairs-distances in n-vertex unweighted undirected graph can be computed in O(n^ωlog n) time, if ω is larger than two.**

Algorithm 3 Pair Distance

```
procedure SHORTESTDISTANCE($A_G$)

    if $A_G$ elements are non-empty then
        return $A_G$

    else
        $A_H$ = ($A_G^2$) + $A_G$
        $D_H$ = SHORTESTDISTANCE($A_H$)
        return dpart($D_H$)
```

Runtime Analysis:
This is a recursive algorithm, where T(n) = T(n/2) + O(n^ω), as each time part d is called, the graph's distance halves. This when solved, the time complexity becomes O(n^ωlog n) time

Correctness of Algorithm:
As, D_H gives D_G using computation done in part D, using that we can say that, if D_H is correct, then D_G calculated will also be correct. In the base case, when the all the vertices are connected, that itself is returned. So, the base case is correct. In the induction step, when until n paths are calculated, the next step would give the correct distance too. Therefor, by induction, proved.

4 Hashing

(a) The approximate probability can be calculated by:

$$P = \frac{n * n^{n-\log_2 n}}{n^n} * \binom{n}{\log_2 n}$$

In the denominator, we have n^n because, n numbers are being assigned and each number has n possible positions that it can go into.
In the numerator, we have n, as the maximum chain, could be any one of the n positions in the hash table. and $n^{\text{(n-logn)}}$ because $n - logn$ numbers can be assigned to any position and the rest is assigned to the single position. It is also multiplied with n choose $(logn)$ because from the n numbers, logn numbers are choosen to be used in the max chain. This final calculation is actually an approximation which is greater than the actual probability, because the expression above actually recounts the same cases multiple times due to the $n^{\text{(n-logn)}}$ factor. And below, we prove that this bigger approximation is lesser than 1/n and thus the probability is also lesser than 1/n.

$$P = \frac{n}{n^{\log_2 n}} * \frac{n!}{(\log_2 n)!(n-\log_2 n)!}$$

$$P = \frac{n}{(\log_2 n)!} * \frac{n(n-1)(n-2)....(n-\log_2 n+1)}{n^{\log_2 n}}$$

$$P \le \frac{n}{(\log_2 n)!}$$

Using Sterling Approximation,

$$\log(k!) = k * log(k) - k$$

So, $\log(\log_2 n!) = (\log_2 n)(\log(n/e))$

$$\frac{\log(\log_2 n!)}{\log n} = \frac{(\log_2 n)(\log(n/e))}{\log n}$$

$$\frac{\log(\log_2 n!)}{\log n} = \frac{(\log(n/e))}{\log 2}$$

For large n,

$$\frac{(\log(n/e))}{\log 2} > 2$$

$$\frac{\log(\log_2 n!)}{\log n} > 2$$

$$\log(\log_2 n!) > 2 * \log n$$

$$\log(\log_2 n!/n^2) > 1$$

$$\log(\log_2 n!/n^2) > 1$$

So, $\frac{n}{(\log_2 n)!} < \frac{1}{n}$

(b) Prove that for any given r ∈ [1,p-1], there exists at least $\binom{M/n}{n}$ subsets of U of size n in which maximum chain length in hash-table corresponding to $H_r(x)$ is $\Theta(n)$.
Solution:
As x is invertible with rx mod p with the given constraints(given in lecture), rx mod p ∈ [0,M-1].
Take y = rx mod p, then $H_r(x) = H(y)$.
The aim is to have a set S such that, when hashed using H(y), all the numbers go to the same spot and the maximum-chain length is n, which is $\Theta(n)$.
Assume S = $\{a_1, a_2.....a_n\}$, which implies that, a_1 mod n = a_2 mod n, a_2 mod n = a_3 mod n .....
$\implies$ $a_1 - a_2$ mod n = 0 ....
$\implies$ $a_1 - a_2 = cn$ where c is an integer.
$\implies$ $a_1 = b + c_1 n$ where b (fixed for a specific set) ∈ [0,n-1] and c_1 ∈ [0, $(M-1)/n$] as a_1 ∈ [0,M].
All elements in S can be represented in the above form. And all the elements with above form can be put in a set S which satisfies the set constraints we are looking for.
As, c_1 can have M/n unique values, a set S can choose from M/n numbers while forming the set of n numbers.
$\implies$ the possible number of subsets with a fixed b, is $\binom{M/n}{n}$, whose maximum length is n ∈ $\Theta(n)$.
$\implies$ there exists atleast $\binom{M/n}{n}$ subsets that whose maximum chain length is $\Theta(n)$. □

(c) The graph plotted is shown below:

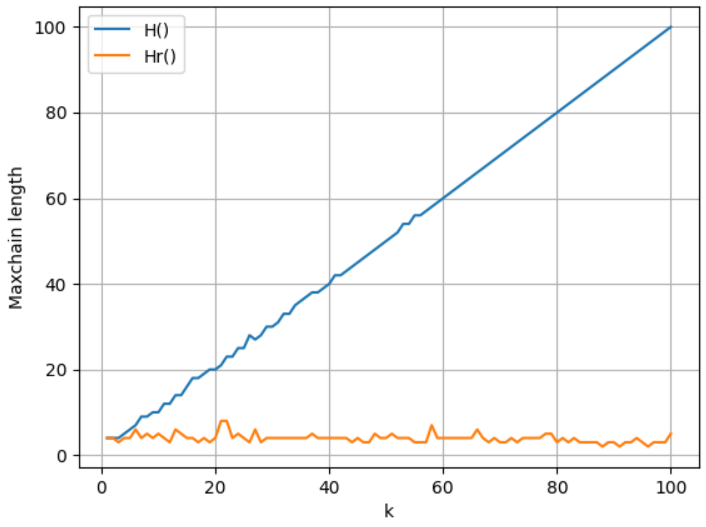

Figure 1: Graph of Maxchain length.

We see that the H() hashing function gives a linear line whereas the Hr() hashing function's max chain seems to be almost constant. This is because, as we increase k, the numbers in the set whose hashing table position is same increases linearly. Due to this, the max chain length too increases. But in the Hr() hashing function, the rx mod p part, helps "randomize" the number before applying the hashing function, thus reducing the max chain length and almost keeping it at a constant level.

The code used to plot is pasted below:

```
import random
import matplotlib.pyplot as plt

random.seed(10)
n=100
M=10000
listset=[]
X=[]
Y1=[]
Yr=[]
p=16747
for i in range(1,n+1):
    r = random.randint(1,p-1)
    set1 = set()
    dic = {}
    X.append(i)
    for j in range(i):
        set1.add(j*n)
    for j in range(n-i):
        set1.add(random.randint(0,M-1))
    for k in set1:
        x = k%n
        if x in dic:
            dic[x]+=1
        else:
            dic[x]=1

    Keymax = max(zip(dic.values(), dic.keys()))[1]
    Y1.append(dic[Keymax])
    dic={}
    for k in set1:
        x = ((r*k)%p)%n
        if x in dic:
            dic[x]+=1
        else:
            dic[x]=1
    Keymax = max(zip(dic.values(), dic.keys()))[1]
    Yr.append(dic[Keymax])
plt.plot(X,Y1, label = 'H()')
plt.plot(X,Yr, label = 'Hr()')
plt.grid()
#plt.title("")
plt.legend()
plt.xlabel('k')
plt.ylabel('Maxchain length')
#plt.title('PMF')
plt.savefig('plot5.png')
print("Plot saved")
```