Assignment-3 Submission

Indian Institute of Technology Delhi

COL351: Analysis and Design of Algorithms

Name: Dishant Dhiman Entry Number: 2019CS10347 Group: 03
Name: Deepanshu Entry Number: 2019CS50427 Group: 04

1 Convex Hull

The problem states that we need to find the convex polygon enclosing at points in P and having minimum area.

Algorithm 1: Finding upper tangent: upperTangent(A,B)

```
i ⟵ index of rightmost point of A ;
j ⟵ index of leftmost point of B ;
L ⟵ line joining A[i] and B[j];
while L crosses any of the polygons do
    while L crosses b do
        j++;
    end
    while L crosses a do
        i–;
    end
end
return (x_i, y_i), (x_j, y_j);
```

Algorithm 2: Finding lower tangent: lowerTangent(A,B)

```
i ⟵ index of leftmost point of A;
j ⟵ index of rightmost point of B;
L ⟵ line joining the A[i] and b[j];
while L crosses any of the polygons do
    while L crosses b do
        j–;
    end
    while L crosses a do
        i++;
    end
end
return (x_i, y_i), (x_j, y_j);
```

Algorithm 3: Merger function: merger(A,B)

```
upper1, upper2 = UpperTangent(A,B);
lower1, lower2 = LowerTangent(A,B);
start = upper1;
SOL = empty array;
while start != lower1 do
    SOL.add(start);
    start–;
end
SOL.add(lower1);
SOL.add(lower2);
start = lower2;
while start != upper2 do
    SOL.add(start);
    start++;
end
SOL.add(upper2);
return SOL;
```

Algorithm 4: Finding convex hull for points in P: findConvexHull(0, n-1)

P ⟵ Array of (x_i, y_i) that contains all the points sorted in increasing order of x_i;
n ⟵ length of P;
if $n \leq 3$ **then**
  return P;
end
start ⟵ 0;
end ⟵ n-1;
middle ⟵ (start + end)/2;
A ⟵ findConvexHull(start, middle);
B ⟵ findConvexHull(middle + 1, end);
return merger(A,B);

1.1 Proof of correctness

For proving that the obtained figure is convex hull, we need to show that it encloses all the vertices and that it has the smallest possible area.

1.1.1 Proving that obtained figure encloses all the vertices

Assume that left convex hull contains vertices as x_1, x_2 ..... x_n1 and that the right convex hull contains vertices as y_1, y_2 ..... y_n2.

1. From the recursive definition, we have two small convex hulls and then we are using merger to obtain the larger convex hull.
2. Note that using upper tangent to connect the two hulls will. ensure that all the vertices in $\{x_1, x_2 ..... x_{n1}\} \cup \{y_1, y_2 ..... y_{n2}\}$ will be below the upper tangent.
3. Similarly, all the vertices will be above the lower tangent.
4. Since on left and right sides, we have the smaller convex hulls, thus **all the vertices are bound on right side of left convex hull and left side of right convex hull.**
5. Hence, all the vertices will be enclosed by our figure.

1.1.2 Proving that obtained enclosing figure has smallest area

Prove by contradiction. Assume that there exist another convex hull H with smaller area. This means that H lies entirely inside our obtained figure. Also let the first point of difference of left convex hull and H be *v*. We can have 2 cases. *v* can either be at the right of upper tangent point or it can be at left of upper tangent point.
Case 1: *v* is at the left of upper tangent point.

1. This means that line joining left convex hull and right convex hull in H bend before reaching the upper tangent point and went on to join at right convex hull.
2. This would mean that some portion of left convex hull is cutted out by the diverging line.
3. This means we got a convex hull smaller in area than the left convex hull. But left convex hull being a convex hull, has the smallest area in its vertex domain. **We arrived at a contradiction**

Case 2: *v* is at the right of upper tangent point.

1. This mean that the line joining left convex hull and right convex hull is at right of upper tangent point.
2. This means that there is lesser distance to cover the same height now in H. This would mean that the slope of such line would be larger in magnitude than the line joining the two smaller hulls.
3. This results in more angle difference as we move to the left.
4. This eventually leads to concavity in the shape at *v* only. **Hence we arrived at a contradiction**.

Since we are getting contradiction in every possible case. Thus, out assumption is wrong. Hence, obtained enclosing figure has the smallest possible valid area.

Hence, the obtained convex figure encloses all the vertices and has the smallest possible area. **This means that the obtained figure is the correct convex hull and this completes the proof.**

1.2 Time complexity of the algorithm

Note the following observations in context of time complexity:

1. We are storing the points in P using a vector list.
2. For any call, we need to perform the following tasks:
   - Partition the point set
   - Recursively calculate the convex hull.
   - Merge the two convex hulls to get the final convex hull of P.
3. Partitioning the vector is same as changing the start and end variables. Thus, partitioning takes constant i.e O(1) time.
4. We initialise the variables as rightmost point of A and leftmost point of B (See Algorithm-1). These can be found in linear time by comparing the x-coordinates of the vertices in the list.

5. Testing whether the line crosses A or B can be done by comparing the slopes of the lines. This can be done in constant time.

6. Thus, overall scanning is bounded by O(n) for upper tangent. Similarly for lower tangent, the time involved is bounded linearly.

7. Thus, we get the following recrusive relation:

$$T(n) = \begin{cases} 1 & \text{if n} \leq 3 \\ \text{n + 2T(n/2)} & \text{otherwise} \end{cases}$$

8. This is the same recurrence that arrives in MergeSort.

9. **Thus, T(n) = O(nlogn)**.

2 Particle Interaction

The problem states that we need to calculate the coulomb force on each particle by other particles in a 1-D array. The expression of Force on particle j, F_j can be expressed as

$$F_j = \Sigma_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \Sigma_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

Let N be the number of particles in the array. We will also index the values starting from 1 instead of 0. Then above expression can be obtained from the convolution of two vectors P and Q which are as follows:

$$P = (q_1, q_2, ..., q_N)$$

$$Q = (\frac{-1}{(n-1)^2}, \frac{-1}{(n-2)^2}, ..., \frac{-1}{4}, -1, 0, 1, \frac{1}{4}, ..., \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2})$$

We know that the convolution of two vectors x and y of size n and m respectively is:

$$(x * y)(\tau) = \Sigma_{i=1}^{N} x_i y_{\tau - i}$$

Thus, we can show algebraically that

$$F(j) = (P * Q)(j + N + 1).Cq_j$$

To calculate the above term for all particles, we can do this efficiently using Fast Fourier Transform as discussed in class. The algorithm is as follows:

1. Obtain the Fourier Transform for array P and Q in O(nlogn) time using the algorithm discussed in class.
2. The convolution can be written as product in the Fourier Transform domain. We calculate the product of FT(P) and FT(Q) in O(n) time.
3. Calculate the Inverse Fourier Transform of the product to obtain the convolution of P and Q. This step can be completed in O(nlogn) time.
4. From the convolution we obtain every term using the expression of F(j). This step takes O(n) time.

Overall Time complexity of the algorithm becomes O(nlogn) + O(n) + O(nlogn) + O(n) which is O(nlogn).

3 Distance Computation using Matrix Multiplication

We are given a unweighted undirected Graph G(V,E). D_G is the distance matrix of G and D_H is the distance matrix of H.

3.1 Part(a)

Let *n* be the number of vertices in the graph and *m* be the number of edges in the graph. We define E(G) as the *n x n* adjacency matrix of the Graph G and e_{ij} is the $(i,j)^{th}$ term of the matrix E(G). From the definition of adjacency matrix, define e_{ij} as follows

$$e_{ij} = \begin{cases} 1 & \text{if there is an edge(i,j) in the graph G or when i = j} \\ 0 & \text{otherwise} \end{cases}$$

Let,

$$A = E(G).E(G)$$

We can write the term a_{ij} as follows:

$$a_{ij} = \Sigma_{k=1}^{n} e_{ik} * e_{kj}$$

This indicates that the term a_{ij} will be non-zero when $i = j$. This will happen only when there is atleast one pair of edges (i,k) and (k,j) in graph G.
If we update the matrix A using the following rule:

$$a_{ij} = \begin{cases} 1 & \text{if } a_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

By applying the above rule, the matrix A will correspond to the adjacency matrix of graph H. This is because it will satisfy the condition that $a_{ij} = 1$ when either $e_{ij} is 1$ or $e_{ik} = 1$ and $e_{kj} = 1$ for some k.
Since we can always generate a matrix A from the Graph G, this implies that we can always have his matrix A correspond to a graph H that we wish to get as the final result.

3.1.1 Time complexity

1. The time complexity for the matrix multiplication of E(G) is O(n^{ω}), $w > 2$ and ω being the exponent of matrix multiplication.
2. the time complexity for the Updation of matrix A is O(n^2).
3. Thus, the overall time complexity of the above algorithm to obtain graph H from G is O(n^{ϵ}).

3.2 Part(b)

Lemma 1 *Reducing the number of vertices in the minimum path from x to y in G gives the minimum path from x to y in H.*

Proof: We prove the above statement by contradiction. Let P be the path obtained by halving the number of vertices in the min-path from x to y in G. Let Q be the min-path in H from x to y. From the structure property of Q, we can say that every edge(u,v) in Q is present only if that edge is present in the graph G or there is another vertex w such that there are edges (u,w) and (w,v). i.e. the path from x to y corresponding to path Q will be double the size of path Q. But if such a path exists, then it should be the min-path otherwise the path obtained by halving the number of vertices in any other path should be smaller than Q.
This leads us to contradiction. Hence, the path obtained by reduction of the min-path in G will lead to the min-path in H.

Proposition: $\forall$ *x,y* $\in$ *V,* D_H*(x,y)* = $ceil(\frac{D_G(x,y)}{2})$

Proof: Let $D_G(x, y) = k$. Then the path P from x to y in graph G can be represented as x, v_1, v_2, ..., v_{k-1}, y.
Case 1: k is even
If k is even, then the number of vertices in the path P are even. Thus, the graph H can be formed using the vertices x, v_2, v_4, ..., v_{k-2}, y. i.e. the number of vertices are exactly halved because the graph H has an edge (x,v_2) because the graph G has the edges (x,v_1) and (v_1,v_2). Thus the number of vertices in the path is equal to $ceil(\frac{D_G(x,y)}{2})$.
Case 2: k is odd
If k is odd, then the number of vertices in the path P are odd. Thus, we can write the path in graph H as x, v_2, v_4, ..., v_{k-3}, v_{k-1}, y. The number of vertices in the path is equal to $ceil(\frac{D_G(x,y)}{2})$.
Hence, Proved.

3.3 Part(c)

It is given that M = $D_H.A_G$. This can be written mathematically that

$$M(x, y) = \Sigma_{v \in G} D_H(x, v) A_G(v, y) = \Sigma_{v \in N_G(y)} D_H(x, v)$$

where $N_G(y)$ is the set of neighbours of y in G.
We have two cases here,
Case 1: $D_G(x, y)$ is even
Then, we can write $D_H(x, y) = \frac{D_G(x,y)}{2}$ using 3.2. Now the distance to neighbours of y is either $D_G(x, y) + 1$ or $D_G(x, y) - 1$, we can say that

$$D_G(x, v) = 2 * D_H(x, y) + 1 \text{ or } 2 * D_H(x, y) - 1 \implies D_G(x, v) \geq 2 * D_H(x, y) - 1$$

This can further imply that

$$D_H(x, v) = ceil(\frac{D_G(x, y)}{2}) \geq ceil(\frac{2 * D_H(x, y) - 1}{2}) = D_H(x, y)$$

Thus,

$$M(x,y) = \Sigma_{v \in N_G(y)} D_H(x,v) \geq \Sigma_{v \in N_G(y)} D_H(x,y) = D_H(x,y) * degree(y)$$

Case 2: $D_G(x,y)$ is odd
Then, we can write $D_H(x,y) = \frac{D_G(x,y)+1}{2}$ using 3.2(ceiling can be taken as adding one to numerator). Now the distance to neighbours of y is either $D_G(x,y)+1$ or $D_G(x,y)-1$, we can say that

$$D_G(x,v) = 2 * D_H(x,y) \text{ or } 2 * D_H(x,y) - 2 \implies D_G(x,v) < 2 * D_H(x,y)$$

This can further imply that

$$D_H(x,v) = ceil(\frac{D_G(x,y)}{2}) < ceil(\frac{2 * D_H(x,y)}{2}) = D_H(x,y)$$

Thus,

$$M(x,y) = \Sigma_{v \in N_G(y)} D_H(x,v) < \Sigma_{v \in N_G(y)} D_H(x,y) = D_H(x,y) * degree(y)$$

Hence Proved

3.4 Part(d)

We can calculate the matrix M in O(n^ω) time. Degree of all the nodes in graph G can be obtained in O(n^2) time. From the matrix M, we can calculate $D_G(x,y)$ in O(1) time, Thus we can calculate the entire matrix in O(n^2) time. Thus, we are left with the overall time complexity of O(n^ω).

3.5 Part(e)

We have written the algorithm to calculate the matrix H from matrix G in O(n^ω) time. We name that procedure as findH(G). We can use this function to calculate the distance between every pair.
Algorithm:

1. We assign the distance of every node as 1 which has an edge directly connected to them in graph G.
2. We calculate the findH(G). Any new edge that is formed here is assigned a distance of 2. findH(G) matrix is now assigned to K.
3. We multiply K and G to obtain the next H and assign the distance to newly formed edges as 3.
4. We repeat the step 3 until all the edges become connected.

Proof Of Correctness: Proof by induction
Base case: True for graph G. the distance between nodes which have an edge between them is 1.
Induction Hypothesis: We get the distance i when the graph is not connected in A_g^{i-1} but connected in A_g^i.

To get a distance of i+1, we look at the connectivity of of the graph A_G^{i+1}. If any new vertex gets connected to other vertex, this implies that there is a path of length i between them. No other path existed between them before. Hence the distance between them is assigned as i.
Hence Proved.

4 Universal Hashing

4.1 Part(a)

Let a_j be the length of chain at slot j in the hash table of S under hash function H(.).

Let x_i be the random variable denoting the slot of an element i distributed uniformly over the set U.
Thus,

$$a_j = \Sigma_{i=1}^{\infty} H(x_i) = j \ \forall j \in \{0, 1, 2, ..n-1\}$$

1. Since it is a uniform distribution, the random variable distributes it randomly.
2. Thus, the probability that H(x_i) = j is approximately equal to 1/n for large enough n.
3. Now we have to prove that Pr(max(a_j) > logn) $\leq$ 1/n.
4. We know that Pr(max(a_j) > logn) $\leq \Sigma_{j=0}^{n-1} H(x_i)$
5. This implies that Pr(max(a_j) > logn) $\leq {}^nC_{logn} * (\frac{1}{n})^{logn}$.
6. We claim that ${}^nC_{logn} \leq \frac{n^{logn}}{logn!}$. This can be proven by:

$$ {}^nC_{logn} = \frac{n!}{logn!(n-logn)!} \leq \frac{n^{logn}}{logn!}$$

$$\text{Thus,} Pr(max(a_j) > logn) \leq \frac{1}{logn!}$$

4.2 Part(b)

1. We have M elements in total.
2. Note that in H_r(x), there will be n modulo classes i.e 0,1,2,..... n-1.
3. We have to put M elements in n modulo classes. Since M¿n, we can use pigeon hole principle to get that there will be one class with at least M/n elements.
4. Now from this class containing at least M/n elements, we can choose n elements in ${}^{M/n}C_n$ ways.
5. All these sets will have max chain length of n. Since, n is Θ(n), thus they will have length as Θ(n).
6. Thus, there exists at least ${}^{M/n}C_n$ subsets of U of size n in which maximum chain length in hash-table corresponding to H_r(x) is Θ(n).

4.3 Part(c)

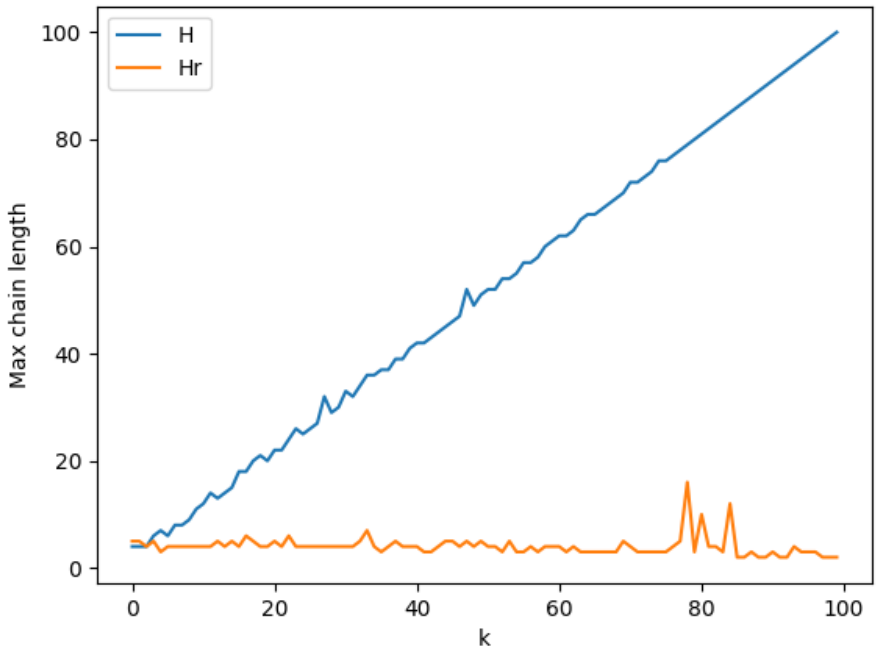

Figure 1: Max Chain Length v/s Set

Explanation of the plots:

1. In H, elements of similar classes lands at same hash value. Hence, it leads to larger max chain length.
2. Whereas in universal hashing, it accounts for elements belonging to same class. Hence, the value of max chain length is lesser for Hr.

```
import matplotlib.pyplot as plt
import random, math


def max_chain(k, primes, n, M):
    hash_h, hash_hr = [0 for i in range(n)], [0 for i in range(n)]
    p = primes[random.randint(0, len(primes)-1)]
    r = random.randint(1, p-1)
    max_h, max_hr = 0, 0
    for i in range(n):
        if i<k:
            x = i*n
        else:
            x = random.randint(0, M-1)
    hash_h[x%n ] += 1
    hash_hr[((r*x)%p)%n] += 1
    max_h = max(max_h, hash_h[x%n])
    max_hr = max(max_hr, hash_hr[((r*x)%p)%n])

    return max_h, max_hr
n, M = 100, 10000
primes = []
for i in range(M, 2*M+1):
    prime = True
    for j in range(2, int(math.sqrt(i)+1)):
        if i%j == 0:
            prime = False
            break
    if prime:
        primes.append(i)
max_h, max_hr = [], []
for i in range(n):
    x, y = max_chain(i+1, primes, n, M)
    max_h.append(x)
    max_hr.append(y)
plt.plot(max_h,  label='H')
plt.plot(max_hr, label='Hr')
plt.xlabel('k')
plt.ylabel('Max chain length')
plt.legend()
plt.savefig('plot_max_chain_length.png')
```