ASSIGNMENT 2 – COMP 252

Alexandre St-Aubin & Jonathan Campana

January 30, 2024

1. ALGORITHM DESIGN. You are given n vectors $x_1,\dots,x_n$ in $\mathbb{Z}^n$. Design an efficient algorithm in the ram model for computing for each x_i one of its nearest neighbors among the other points, using the standard Euclidean metric to measure distances. You can't use real numbers, and operations like square root are not available. Nevertheless, show how this can be done in $o(n^3)$ worst-case time.

   Solution:
   We first note that minimizing the Euclidean distance between 2 vectors of length n is equivalent to minimizing the sum of squared differences between each individual components. The reason for this is that the square root is a monotone increasing function. Let $x_i = (x_{i,1},\dots,x_{i,n})$, $x_j = (x_{j,1},\dots,x_{j,n})$ both in $\mathbb{Z}^n$, and define

$$d_2^2(x_i,x_j) := \sum_{k=1}^{n}(x_{i,k}-x_{j,k})^2 = \sum_{k=1}^{n}(x_{i,k}x_{i,k}-2x_{i,k}x_{j,k}+x_{j,k}x_{j,k}) = \langle x_i,x_i\rangle + 2\langle x_i,x_j\rangle + \langle x_j,x_j\rangle \quad (1)$$

   It is obvious that $d_2^2 \sim O(n)$ in the RAM model , and that no real numbers, nor square roots were used to compute it. Now, we notice that $\frac{n(n-1)}{2}$ distances need to be computed, and in view of dynamic programming, we find a way to compute everything at once in order to reduce the complexity that would occur if we were to get each d_2^2 individually, namely, $O(n^3)$. Construct the following matrix,

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \cdots & x_{n,n} \end{pmatrix}$$

   Then,

$$X\cdot X^T = \begin{pmatrix} \sum_{k=1}^n x_{1,k}x_{1,k} & \sum_{k=1}^n x_{1,k}x_{2,k} & \cdots & \sum_{k=1}^n x_{1,k}x_{n,k} \\ \sum_{k=1}^n x_{2,k}x_{1,k} & \sum_{k=1}^n x_{2,k}x_{2,k} & \cdots & \sum_{k=1}^n x_{2,k}x_{n,k} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^n x_{n,k}x_{1,k} & \sum_{k=1}^n x_{n,k}x_{2,k} & \cdots & \sum_{k=1}^n x_{n,k}x_{n,k} \end{pmatrix} = \begin{pmatrix} \langle x_1,x_1\rangle & \langle x_1,x_2\rangle & \cdots & \langle x_1,x_n\rangle \\ \langle x_2,x_1\rangle & \langle x_2,x_2\rangle & \cdots & \langle x_2,x_n\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n,x_1\rangle & \langle x_n,x_2\rangle & \cdots & \langle x_n,x_n\rangle \end{pmatrix}$$

   where the entries of $X\cdot X^T$ are exactly the dot products needed in (1). Therefore, by employing STRASSEN'S algorithm, we can efficiently compute the product $X\cdot X^T$ with a time complexity of $O(n^{2.807})$. Subsequently, each of the $\frac{n(n-1)}{2}$ distances can be computed in $O(n^2)$ time, as computing one distance will take constant time by accessing the dot products in the matrix previously computed. Simultaneously, these distances can be added to an ordered list (corresponding to each individual vector) in constant time. Upon completion of this process, we will have the closest vector to any given vector readily accessible. The algorithm outlined above has a complexity of $O(n^{2.807})$, demonstrating that it can be accomplished in worst-case time less than $o(n^3)$.

2. DYNAMIC PROGRAMMING: COMPUTING THE OPTIMAL STAR.

Algorithm 1: Algorithm to compute the optimal star.

Input: A matrix $D[i,j]$ of pairwise distances between cities $1 \leq i,j \leq n$.
Output: The cost of the optimal star that links the given cities.

```
  // initialize M[v, S] for sets of size 1
1 for all $i \in [1,n]$ do
2     for all $j \neq i$ do
3         $M[i,\{j\}] \leftarrow D[i,j]$ ;

  // loop through each central city
4 for all $v \in [1,n]$ do
      // loop through each subset size
5     for all $k \in [1,n-1]$ do
          // loop through each subset of size k excluding the central city
6         for all S with $|S| = k,\ S \subseteq \{1,...,n\} \setminus \{v\}$ do
7             $M[v,S] \leftarrow \min_{l \in S}\{M[v,S \setminus \{l\}] + \min\{d(l,x)\ :\ x = v \text{ or } \deg(x) = 1\}\}$;

8 return $\min_{v \in [1,n]}\{M[v,S] : |S| = n\}$
```

Remark. The above is heavily inspired from the *Held–Karp algorithm*, which was presented in class.

Analysis of the algorithm

Allow Ω to represent the set of all vertices (cities). The outer loop iterates through each potential central vertex v of the star, while the subsequent loops systematically explore subsets of $\Omega \setminus v$ in a bottom-up fashion. Starting with subsets of size 1 (i.e., $S = s$), we know $M[v,S] = d(v,s)$. We then progress through sets of increasing size, using our knowledge from preceding, smaller sets to obtain an optimal set length, until we reach $M[v,\Omega \setminus v]$, thereby identifying the optimal star configuration with the central vertex v.

Let's delve into the time complexity analysis. Line 4 executes n times. Lines 5 and 6 represent all possible subsets of a set with $n-1$ vertices, resulting in 2^{n-1} iterations. We can safely assume line 7 to be $O(n)$. Combining these components, the overall time complexity is given as $O(n^2 \cdot 2^n)$. We note that lines 1 & 8 run in $O(n)$ time, so they play no significant role in the overall time complexity of the algorithm.

3. INDUCTION. We are given the recurrence

$$T_n = 2T_{\frac{n}{a}} + 7T_{\frac{n}{a^2}} + 1,$$

where $a \geq 2$ is a given integer, and n is restricted to be a power of a. We also know that $T_1 = T_a = 1$.

(a) $T_n = \Omega(n^c)$ for some constant c.

Proof. For the base case, let $n = a^2$, then,

$$T_{a^2} = 2T_a + 7T_1 + 1 = 2 + 7 + 1 = \Omega(1) = \Omega(n^0),$$

Now, assume $T_{a^k} = \Omega((a^k)^c)$, for any $2 \leq k < n$, then we have

$$\begin{aligned} T_{a^n} &= 2T_{a^{n-1}} + 7T_{a^{n-2}} + 1 \\ &\leq 2((a^{n-1})^c) + 7((a^{n-2})^c) + 1 \quad \text{[by I.H.]} \\ &\leq 9(a^{n-1})^c \\ &= \frac{9}{a^c}(a^n)^c \end{aligned}$$

Where $\frac{9}{a^c}$ is a constant so we conclude that $T_{a^n} = \Omega((a^n)^c)$. Remains to find the largest such c. □

4. SORTING WITH DUPLICATES.

   (i) Give a divide-and-conquer algorithm that sorts n numbers with $1 \leq k \leq n$ uniques. This can be done in time $O(n\log_2(k+1))$ with a ternary comparison oracle.

   See Algorithm 2 on next page.

   (ii) Prove the complexity claim.

Proof. For the sake of simplicity, we assume that $k = 2^i$ for some $i \in \mathbb{N}$. To prove that the complexity of our algorithm is $O(n\log(k+1))$, we count the number of times that it uses the ternary oracle (number of comparisons made), as the rest costs 0, by the oracle model of complexity. We notice that the comparisons occur only at the *merge* step of the algorithm, so we'll start counting at the first merge, namely when we have n subarrays of size 1. We assume the worst case, i.e. that no duplicates are found in the merges until level $\log_2 k$ is reached (see Figure 1). When this level is reached, we have $\frac{n}{k}$ sets of length at most k, and the next step will surely find duplicates, thereby reducing the running time compared to plain old merge sort.

Let's count the number of comparison needed to get to level $\log_2 k$ of the merging. In order to get one subarray of length k, k subarrays of length 1 must be merged into 1. This process will take $\log_2 k$ merges, each one costing $\frac{(2^i-1)k}{2^i}$ comparisons, that is,

$$\sum_{i=1}^{\log_2 k} \frac{(2^i-1)k}{2^i} \leq \sum_{i=1}^{\log_2 k} k = n\log_2 k \text{ comparisons.}$$

And, since there are $\frac{n}{k}$ such subarrays, it follows that the complexity of the above process is $O(n\log_2 k)$.

We now consider the second and last part of our computation. At this step, we have $\frac{n}{k}$ sets of size k. In the normal merge sort algorithm, the next step would generate half as many sets of double the size ($2k$). However, our algorithm combines duplicate terms into a single tuple element, and since there are only k elements that are different in our set, k is an upper bound for the size of a subarray. Hence, we instead obtain half the amount of sets, but still of size at most k. This is also true for all the following merges.

Let's compute the time complexity of merging $\frac{n}{k}$ subarrays until we have 1 array of length k. Well, it is clear that each merge will cost at most k comparisons, as in the worst case, we are merging two arrays of length k, which must therefore be identical, as they are ordered and contain the same k elements. Thus, the oracle will yield k equalities, which suffices to merge the arrays. So, the total number of comparisons is given by

$$k\sum_{i=0}^{\log_2 \frac{n}{k}} 2^i = k(2^{\log_2 \frac{n}{k}} - 1) = k\left(\frac{n}{k} - 1\right) = n - k$$

Combining both processes, it is easy to see that the total time complexity is

$$O(n - k + n\log_2(k)) = O(n\log_2(k+1))$$

Remark. The reason why the complexity is $O(n\log_2(k+1))$, and not $O(n\log_2(k))$ is that in the case where $k = 1$, if we need the complexity to be $O(n\log_2(1))$, not $O(0)$.

□

Algorithm 2: DC algorithm to sort a list of n elements with $1 \leq k \leq n$ duplicates.

```
Input: An arbitrary array of integers.
Output: The sorted array.
// Initialize the array with Tuples
for $i = 0$ to $l$(array) do
    array$[i] \leftarrow$ new Tuple$(i, 1)$;
// A function to compare tuples, each use costs 1.  Anything else is free.
Function TernaryOracle(tuple1, tuple2):
    if tuple1[0] = tuple2[0] then
        return 0;
    else if tuple1[0] < tuple2[0] then
        return-1;
    else
        return 1;
Function MergeSort(array):
    if $l$(array) < 2 then
        return array;
    else
        mid $\leftarrow \lfloor l(\text{array})/2 \rfloor$;
        return Merge (MergeSort (array [:mid ]), MergeSort (array [mid:]));
Function Merge(Left, Right):
    $i, j \leftarrow 0$;
    array $\leftarrow$ new array;
    while $i \leq l$(Left) && $j \leq l$(Right) do
        switch TernaryOracle(Left [i], Right [j]) do
            case 0 do
                array$[i + j] \leftarrow$ new Tuple(Left$[i][0]$, Left$[i][1]$ + Right$[i][1]$) ;
                $i{+}{+}$, $j{+}{+}$;
            case 1 do
                array$[i + j] \leftarrow$ Left$[i]$ ;
                $i{+}{+}$;
            case $-1$ do
                array$[i + j] \leftarrow$ Right$[j]$ ;
                $j{+}{+}$;
        // then add the rest of the array that we didn't reach the end of to
           the current array:
    array $\leftarrow$ array&$(i \leq l(\text{Left})) *$ Left$[i : l(\text{Left})]$&$(j \leq l(\text{Right})) *$ Right$[j : l(\text{Right})]$
```

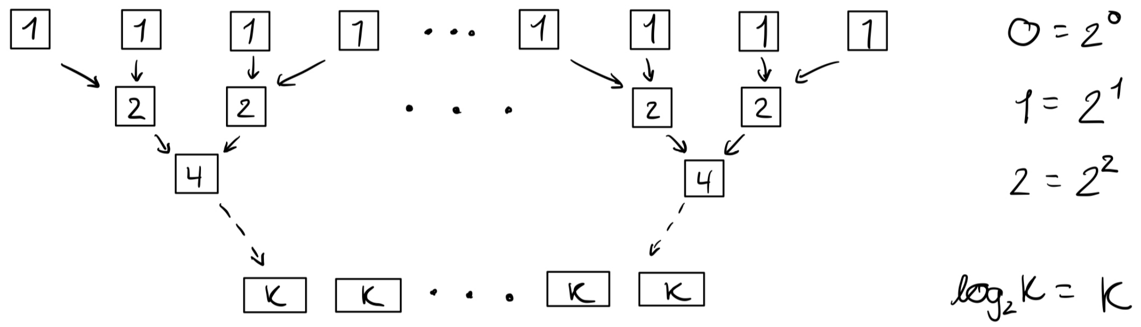

Figure 1: The merging of subarrays.