Homework 3 example solution

Compiled by Nikita Alexandrov

December 4 2018

Question 1

1.1 Algorithm description

For simplicity we are going to consider only **full binary tree**: tree, where each node either is a leaf or has 2 children (note that it doesn't have to be balanced!). Trees that have nodes with 1 child can be transformed to full tree by merging such nodes with their child (see Figure 1). It doesn't affect our objective solution, since the node and its child have the same sum of variances regardless of cluster points selection.

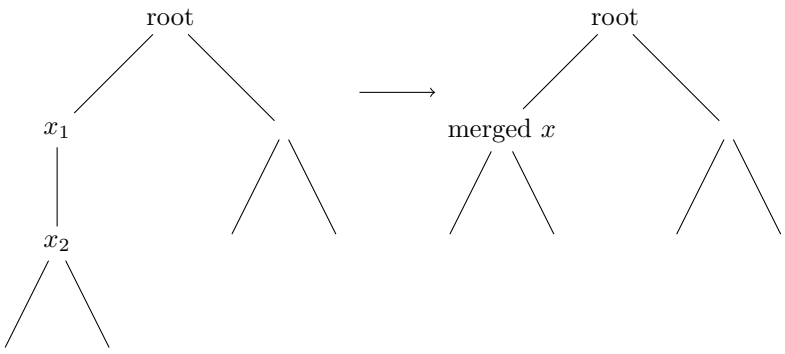

Figure 1: Transformation to full binary tree

The number of nodes in the full binary tree with n leaves is $N = 2n - 1$. So we can label nodes $v_1, v_2, \ldots, v_N$, where v_1 is a root of the tree and $v_n, \ldots, v_N$ are leaves. Also let $l(i)$ and $r(i)$ $(i < n)$ be functions that return index of left and right child of v_i accordingly.

Now we can construct the table of size $N \times k$, where each cell (i, j) stores the minimal value of objective function $s_{i,j}$ for subtree, rooted at node v_i and j is a number of clustering points. Our goal is to find $s_{1,k}$.

We are going to use the dynamic programming approach and $s_{i,j}$ can be calculated using the following formula:

$$s_{i,j} = \min_{0<j'<j} (s_{l(i),j'} + s_{r(i),j-j'}) \tag{1}$$

When $l(i)$ and $r(i)$ aren't defined (v_i is a leaf), we can set $s_{i,j} = +\infty$.

Note that this formula does not describe the case when $j = 1$: only subtree root can be a cluster center, therefore, $s_{i,1} = var(v_i)$. So we are almost ready to calculate $s_{1,k}$, but we need to calculate $var(v_i)$ at first.

Variances $var(v_i)$ for each node can be computed naively and it will still lead to polynomial time, but the most efficient way is bottom-up approach: first, we calculate from bottom to top three functions: $p_i = \sum_{x \in X(v_i)} x^2$ – sum of squares in $X(v_i)$, $q_i = \sum_{x \in X(v_i)} x$ – sum of all points in $X(v_i)$ and $r_i = \sum_{x \in X(v_i)} 1$ – number of points in $X(v_i)$, where $X(v_i)$ is the set of points that correspond to node v_i. Values of these functions in leaf nodes are x^2, x and 1 accordingly, where x is a single point from X, corresponding to the leaf. The value of each function in internal nodes is the sum of values for its children. The variance $var(v_i)$ can be expressed in terms of functions p, q and r:

$$var(v_i) = \sum_{x \in X(v_i)} ||x - c(v_i)||^2 = \sum_{x \in X(v_i)} \left|\left| x - \frac{q(v_i)}{r(v_i)} \right|\right|^2 = r(v_i) \left(\frac{q(v_i)}{r(v_i)}\right)^2 - 2\frac{q(v_i)}{r(v_i)} \sum_{x \in X(v_i)} x + \sum_{x \in X(v_i)} x^2 =$$

$$= \frac{q(v_i)^2}{r(v_i)} - 2\frac{q(v_i)^2}{r(v_i)} + p(v_i) = p(v_i) - \frac{q(v_i)^2}{r(v_i)}$$

When the node is a leaf (so, $p(v_i) = q(v_i)^2$ and $r(v_i) = 1$), we can verify that the formula gives 0.

So we found how to find $s_{1,k}$, but it doesn't return us the cluster centres! We only need to modify our data structure: in addition to keeping the minimal value of objective function $s_{i,j}$, we will also remember $d_{i,j} = \arg\min_{0<j'<j}(s_{l(i),j'} + + s_{r(i),j-j'})$. The information tells us about the number of cluster points in the left and right subtree in the best case. After we found the value $s_{1,k}$, we can traverse the tree starting from the root (state $(1,k)$): from the state (i,j) we go to $(l(i), d_{i,j})$ and $(r(i), j - d_{i,j})$. If during the traversal we came to state $(i,1)$, then we add v_i to the set of cluster centres.

1.2 Correctness of the algorithm

The fact that $s_{i,j}$ has the minimal value of objective function can be proven by induction on the number of cluster points j.

Base: $j = 1$. There is only one possible way to assign a cluster point for the subtree, rooted at node v_i: only v_i can be chosen and $s_{i,1} = var(v_i)$.

Induction step: $j - 1 \to j$. Let us assume that we store the minimal value in $s_{i,j'}$ for all $1 \le i \le N, 1 \le j' < j$ and we are calculating the value for $s_{i,j}$. Since $j > 1$ and each internal node has 2 children, we should have some non-zero cluster points in the left subtree as well as in the right subtree. In the formula 1, we check all possible combinations of distributing cluster centres in left and right subtrees and we choose the one with minimal sum. Since the number of cluster points in the left subtree j' is between 1 and $j - 1$, the values of $s_{l(i),j'}$ and $s_{r(i),j-j'}$ are minimal based on our assumption. Therefore, $s_{i,j}$ stores the minimal value of the variances sum.

Since we remember how many cluster points are located in each subtrees, we can explicitly return them.

1.3 Complexity of the algorithm

We have $N = 2n - 1 = O(n)$ nodes in our tree, and in order to compute the variance of all nodes, we have to complete only one traversal of tree with calculation of p, q and r functions. Therefore, it takes $O(nd)$ time to complete it, where d is the dimension of points in X.

Formula for calculating $s_{i,j}$ requires $O(j)$ time. That's why we can fill the rest of the table using $O(n(1 + 2 + \ldots + (k-1)) = O(nk^2)$ time. And also we need $O(n)$ time to recover the answer.

The total time complexity of algorithm is $O(nd + nk^2 + n) = O(n(k^2 + d))$

Question 2

2.1 At most k clusters

Suppose at some point the algorithm produces k clusters with $\mathbf{c}^{'} = \left\{c_1^{'}, \ldots, c_k^{'}\right\}$ centers.

Under this case, we claim: for the clusters $\mathbf{c}^* = \{c_1^*, \ldots, c_k^*\}$ in the optimal clustering, there is a *one to one* mapping from $\mathbf{c}^{'}$ to $\mathbf{c}^*$. In other words, each cluster in the optimal clustering contains a unique cluster center by the algorithm.

Proof: if this is not the case, there must be some $c \in \mathbf{c}^*$ that contains at least two centers $c_i^{'}, c_j^{'} \in \mathbf{c}^{'}$. Because they are in the same cluster of the optimal solution, then $d(c_i^{'}, c_j^{'}) \leq 2d^*$. However, this is contradictory to the rule $d(c_i^{'}, c_j^{'}) > 2d^*$, which the algorithm uses to create new cluster.

Then using the above result, when there are k centers, we end up with the case in Figure 2, where x has its nearest center $c^{'}$ given by the algorithm and the nearest optimal center c^* given by the optimal clustering.

If x creates a new cluster, then $d(c^{'}, x) > 2d^*$. Because of optimality, we have $d(x, c^*) < d^*$ and $d(c^*, c^{'}) < d^*$. Adding them together, we have

$$d(c^*, c^{'}) + d(x, c^*) < 2d^* < d(c^{'}, x)$$

which contradicts with triangle inequality.

Thus, the algorithm produces at most k clusters.

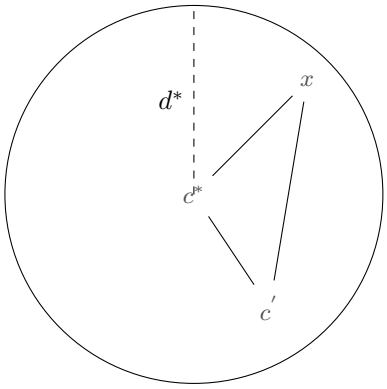

Figure 2: Given a new point x, $c^{'}$ is the nearest neighbor to x given by the algorithm. c^* is the nearest neighbor of the optimal clustering to x.

2.2 Factor-2 approximation

Using the above result, it is obvious to see all points have distances smaller than $2d^*$ to their nearest neighbors. Thus, the algorithm is a factor-2 approximation algorithm.

2.3 Realistic case

The basic principle is: update the cluster centers (merging, repartition, etc) when $|C| > k$ and update the distance threshold (d^*) appropriately.

For example, Doubling algorithm by Charikar et al.[1] is one option. The basic idea is:

- when $|C| <= k$, keep taking in new elements in the same as STREAMING-FURTHEST until $|C| > k$.
- when $|C| > k$, we re-select the cluster centers from C so that each pair of them are even further away. This can be seen as merging clusters in C. Then, the distance threshold used for creating new cluster is doubled.

Interestingly, the algorithm given in Algorithm 1 is factor-8 approximation algorithm.

Algorithm 1: Doubling algorithm. Line 5-10 takes new elements until $|C| > k$. Line 11-13 updates the new cluster centers so that each pair of centers in the new clustering has distance larger than $2d^{'}$. Note $d(c, C^{'}) = \min_{c^{'} \in C^{'}} d(c, c^{'})$

Data: a stream of data points X
Result: clustering of points in X in k clusters
1 read $\{x_1, \ldots, x_k\}$, the first k points in the stream;
2 define the set of cluster centers as $C \leftarrow \{x_1, \ldots, x_k\}$;
3 $d^{'} \leftarrow$ the maximum distance of pairs in C;
4 **repeat**
5 **while** $|C| \leq k$ **do**
6 read next point x;
7 compute the distance d_m from x to its closest cluster center x_i in C;
8 **if** $d_m > 2d^{'}$ **then**
9 take x to be a new cluster center;
10 $C \leftarrow C \cup \{x\}$;
11 define new clustering $C^{'} = \{c_1\}$;
12 **while** *there exists* $c_i \in C$ *such that* $d(c_i, C^{'}) > 2d^{'}$ **do**
13 $C^{'} \leftarrow C^{'} \cup \{c_i\}$;
14 $C \leftarrow C^{'}$;
15 $d^{'} \leftarrow 2d^{'}$
16 **until** *stream becomes empty*;
17 **return** C

Another interesting paper to read is [2].

Question 3

3.1 Algorithm description

In order to recall, we have $|V| = N$ vertices and set of active edges $E(T, W) = e_{\max(T-W+1,1)}, \ldots, e_T$ at time T . The connected graph should have at least $N - 1$ edges, consequently, when $W < N - 1$, the graph is not connected.

When $W \geq N - 1$, we will maintain a **spanning tree**. When the graph isn't connected, we consider the spanning trees of graph components. We use $ST(T)$ to denote the spanning tree (or the spanning trees in unconnected case) at time T. $ST(T)$ can be represented with adjacency lists: for each vertex, we store a list of the vertices adjacent to it and also the timestamp, when the related edge was added.

Algorithm 2: isConnected

Data: set of spanning trees ST at the moment of time $T - 1$ ($ST(T - 1)$)
Result: true or false

```
1  T ← T + 1;
2  if T > W then
3      remove e_{T−W} from ST if it's possible
4  read new edge e_T from data stream;
5  add this edge to ST;
6  begin start DFS from any vertex of added edge:
7      if found a cycle during traversal then
8          remove from ST the oldest edge of this cycle
9      return n ← number of visited nodes
10 if n = N then
11     return true
12 else
13     return false
```

3.2 Correctness of the algorithm

First of all, at any given moment of time, edges of $ST(T)$ belong to the list of active edges $E(T, W)$, because we delete the oldest edge (line 2-3). Therefore, $ST(T)$ is always a subgraph of $G(T, W)$. As a result, if $ST(T)$ is connected ($n = N$, line 10), then $G(T, W)$ is connected too.

Now let us assume that $ST(T)$ is not connected ($n \neq N$), but $G(T, W)$ is. It means that $ST(T)$ has at least 2 components ($ST_1, ST_2, \ldots$). If $G(T, W)$ is connected, it means that there is an edge $e_i, i > T - W$ that connects ST_1 and ST_2. Our algorithm always adds the coming edge to ST, therefore, e_i was contained in ST at some moment and was deleted later as the oldest edge in the cycle (line 8). Then, this cycle has to have some another edge $e_j, j > i$ that connects components ST_1 and ST_2. If e_j still exists in $ST(T)$, then $ST(T)$ is actually connected, but if it isn't, then there is another edge $e_k, k > j$ that connects these components and etc. Our window W is limited, therefore, we will end up with edge e_T that just has been recently added to $ST(T)$ and can't be removed. Therefore, ST_1 is connected with ST_2 in $ST(T)$. **Contradiction**

3.3 Space complexity and 3.4 Time complexity

ST is always a tree (or trees), since at any time T new coming edge e_T can create no more than one cycle that we immediately break. Therefore the number of edges there is no more than $N - 1$ (to be more accurately, if ST has k components, then ST has exactly $N - k$ edges). Therefore, our DFS complexity is $O(N + N - 1) = O(N)$. In accordance with the implementation of the data structure for adjacency lists (hash-table, deque, double-linked list, sets/maps, heap for timestamps and etc), the complexity of searching or removing the oldest edge globally (line 3) and the oldest edge from the cycle (line 8) can be $O(1)$, $O(\log N), O(N)$. It doesn't affect the time complexity, therefore, the answer is $O(N)$, but it can affect the space complexity:

- We can keep in memory the set of active edges $E(T, W)$ in the deque (the newest edge is added to the deque end, the oldest edge is extracted from the front of deque) and hash-table for graph ST. Then, all operations of adding and removing will cost $O(1)$ time, but the space complexity is $O(N + W)$.

- Alternatively, we can keep the adjacency lists in deques: each node has a deque of adjacent vertices, sorted in the order when edges arrived. In this case the space complexity is $O(N)$, but the search of the oldest edge in the graph requires $O(N)$ time.

P.S. The naive algorithm to do DFS on the whole graph $G(T, W)$ will lead to time complexity $O(N+W)$, therefore, such solutions didn't receive full points.

References

[1] Moses Charikar, Chandra Chekuri, Tomas Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33(6):1417–1440, 2004.

[2] Richard Matthew McCutchen and Samir Khuller. Streaming algorithms for k-center clustering with outliers and with anonymity. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 165–178. Springer, 2008.