MS&E 317/CS 263: Algorithms for Modern Data Models, Aut 2011-12
Course URL: `http://www.stanford.edu/~ashishg/amdm`.
Instructor: Ashish Goel, Stanford University.

Lecture 8, 10/19/2011. Scribed by Pong Eksombatchai.

In the first part of this lecture, we will continue to improve on our Naive Algorithm to keep track of the connected components of an undirected graph and modify it into the Union-Find Algorithm. Then in the second part, we will introduce Local Sensitive Hashing.

Finding Connected Components of an Undirected Graph

We are given an initial set of the nodes V of an undirected graph G and edges (u_t, v_t) appearing at time t. We have to support the ability to find the number of connected components in G and check if nodes u and v are in the same connected component at any given time. The basic idea is that we have multiple arrays; COLOR is to keep track the connected component each node belongs to and the others to keep track of the nodes in each component. We will suppose that $V = \{1, 2, ..., N\}$. Before any edges arrive, we have N non-empty arrays corresponding to the N components consisting of individual vertices. After each edge arrival, we update the arrays so that there are exactly as many non-empty arrays as the number of connected components. This gives the following simple algorithm.

Naive Algorithm

State
Store an array COLOR of size N and arrays $A_1, A_2, ..., A_N$, where $A_i = \{u \,|\, \text{COLOR}[u] = i\}$

Initialization
Set COLOR$[u] = u$ and $A_u = \{u\}$ for all $u \in V$.

```
Update($t, u_t, v_t$)
    If COLOR[$u_t$] $\neq$ COLOR[$v_t$] then
        Set $i$ = COLOR[$u_t$] and $j$ = COLOR[$v_t$]
        If $|A_j| > |A_i|$ then
            Swap($i, j$)
        Concatenate $A_i$ with $A_j$
        Set COLOR[$v$] = $i$ for all $v \in A_j$
        Set $A_j = \{\}$
```

Query: Number of Connected Components
Count the number of A_i for all integers i in the range $[1, N]$ such that A_i is not empty.

Query: Is Connected(u, v)
Return true if COLOR$[u]$ is equal to COLOR$[v]$, otherwise return false.

The state space size is $O(N)$ because we have the array COLOR and the total size of all the A_i is N at any given time. Suppose that M in the total number of edges in graph G when all of the edges are given, we have that the total running time of this algorithm is $O(M + N \log N)$. The $O(N \log N)$ comes from the part when we need to update COLOR to the correct values after we concatnate A_i with A_j. It may seem at first that this process takes $O(N^2)$ in total, but we have that each COLOR$[u]$ is updated at most $O(\log N)$ times, because the connected component that u is in at least doubles in size everytime it is merged into another component; making the total running time of merging $O(N \log N)$.

Note that when we are dealing with very large graphs such as a real world social network graph, we cannot keep the whole COLOR array in memory and need to store portions of it across different machines. After we have all the edges of G, we know that we have to look up COLOR$[u]$ at most $O(\deg(u) + \log N)$ total number of times, because we need to look at COLOR$[u]$ at least once for every edge (u, v) in the update step and the number of times COLOR$[u]$ can be updated is $O(\log N)$. This may seem that the load for each machine is quite balanced, but in real world networks, we often have nodes with very high degree. We will end up hitting machines that contain these high degree nodes a lot more often than others, which is not the behavior we want.

Union-Find Algorithm

Although the Naive Algorithm works quite well, we will now develop it into a simple Union-Find Algorithm. The idea behind it is to represent each set by a tree, where each node keeps a reference to its parent and the representative of the set is the root of the tree. The simple algorithm, which excludes the Path Compression improvement, works as follow:

State
Store an array parent and an array size of length N, where parent$[u]$ contains the parent of u in the tree and size$[u]$ is the size of the set in which u is the representative of. Note that size$[u]$ is undefined if u is not a representative of a set. For initialization, we will set parent$[u] = u$ and size$[u] = 1$ for all $u \in V$.

```
Union($t, u_t, v_t$)
    Set $r_1$ = Find($u_t$) and $r_2$ = Find($v_t$)
    If $r_1 \neq r_2$ then
        If size[$r_1$] < size[$r_2$] then
            Swap($r_1$,$r_2$)
        parent[$r_2$] = $r_1$
        size[$r_1$] = size[$r_1$] + size[$r_2$]
```

```
Find(u)
    while parent[u] ≠ u
        u = parent[u]
    return u
```

Query: Number of Connected Components
Count the number of u for all $u \in V$ such that parent$[u] = u$.

Query: Is Connected(u, v)
Call Find(u) and Find(v) and if the return values are the same return true, otherwise return false.

The state space size is $O(N)$ because we have two arrays of length N; parent and size. We also have that the running time for each update step is $O(\log N)$ because most of our work comes from the call to Find which has a runtime of $O(\log N)$. For each u, the update to the connected component U that node u is in, makes the path length from u to the root either the same when a smaller connected component is merged into U or increase by one when U is merged into a connected component at least the size of U. In the worst case we have that path length from u to the root increases by one in each update, but this can only happen $O(\log N)$ times because U at least doubles in size each time this happens. We have that the path length from any node to the root is $O(\log N)$ and thus the runtime of each Find is $O(\log N)$.

In a distributed network, most of our load will come from the looking up parent$[u]$ in Find. There is still a problem with this because we need to look up the nodes which are representatives of the set a lot more often than others. Fortunately, there are techniques that can solve this problem which will be discussed later in class.

Locality Sensitive Hashing

Given a set of points V and a distance metric d, we are often interested in finding out if there are any points $x \in V$ such that x is close to the query point q. This problem can be solved quite well in low dimension spaces by using space partitioning techniques, but in high dimension spaces it is impossible to do that. However, fortunately, we will be able to use LSH to solve some types of these problems, one of which is the (c, R)-Near Neighbor problem. The problem is defined as follow:

Input
The input to this problem which is given up front is a set of points V of size N and a distance metric d.

Query(q)
For each query point q, if there exists some $x \in V$ such that $d(x, q) \leq R$, then the algorithm must output $x' \in V$ such that $d(x', q) \leq cR$ with probability at least $1 - \delta$.

After defining the problem that we are trying to solve, we will define the tool that will be used to solve this problem.

Locality Sensitive Hashing Family: A family of hash functions H that is said to be (c, R, P_1, P_2)-sensitive for a distance metric d, when:
(1) $Pr_{h \sim H}[h(x) = h(y)] \geq P_1$ for all x and y such that $d(x, y) \leq R$
(2) $Pr_{h \sim H}[h(x) = h(y)] \leq P_2$ for all x and y such that $d(x, y) > cR$

Example:
Let $V \subseteq \{0,1\}^J$ and $d(x, y) =$ Hamming distance between x and y. If $R << J$ and $cR << J$, one example LSH family is $H = \{h_i\}$ for all integers i from the range $[1, J]$, where $h_i(x) = x_i$. Therefore, the family H is (c, R, P_1, P_2)-sensitive for the distance metric d, when $P_1 = 1 - \frac{R}{J}$ and $P_2 = 1 - \frac{cR}{J}$.

After defining LSH families, the algorithm that we will use to solve the (c, R)-Near Neighbor problem will be as follow:

Locality-Sensitive Hashing for solving (c, R)-Near Neighbor

Suppose that we have a LSH family H that is (c, R, P_1, P_2)-sensitive for the distance metric d. Let $h_{i,j}$ be hash functions such that $h_{i,j} \sim H$ for all integers i and j such that $1 \leq i \leq K$ and $1 \leq j \leq L$. We will define $g_j(x)$ to be $< h_{1,j}(x), h_{2,j}(x), ..., h_{K,j}(x) >$ for all intergers j such that $1 \leq j \leq L$.

Preprocessing
For all $x \in V$ and for each integer j in the range $[1, L]$, we add x to the $bucket_j(g_j(x))$.

```
Query(q)
    for $j = 1$ to $L$
        for all $x \in bucket_j(g_j(q))$
            if $d(x, q) \leq cR$ then return $x$
    return none
```

It is easy to see that the total runtime to preprocess all of the points is $O(NKL)$ and the space required is $O(NL)$. The runtime for each query is $O(KL + NLF)$, where F is the probability that a point x is hashed using g_j into a $bucket_j$ that q is also hashed into, but $d(x, q) > cR$. To find F, from the definition of LSH family, we notice that:

$$\begin{aligned} Pr[g_j(x) = g_j(q) \,|\, d(x,q) > cR] &= Pr[\cap_{i=1,...,K} h_{ij}(x) = h_{ij}(q) \,|\, d(x,q) > cR] \\ &= \Pi_{i=1,...,K} Pr[h_{ij}(x) = h_{ij}(q) \,|\, d(x,q) > cR] \\ &\leq P_2^K \end{aligned}$$

Hence, we have that the running time is $O(KL + NLP_2^K)$. Next, we try to analyze the success probability of the algorithm. We can find the lower bound of it, which is the probability that a point x such that $d(x,q) \leq R$ is hashed using any of the g_j into a $bucket_j$ that q is also hashed into. First, we calculate the probability $Pr[g_j(x) = g_j(q) \,|\, d(x,q) \leq R]$.

$$\begin{aligned} Pr[g_j(x) = g_j(q) \,|\, d(x,q) \leq R] &= Pr[\cap_{i=1,\ldots,K} h_{ij}(x) = h_{ij}(q) \,|\, d(x,q) \leq R] \\ &= \Pi_{i=1,\ldots,K} Pr[h_{ij}(x) = h_{ij}(q) \,|\, d(x,q) \leq R] \\ &\geq P_1^K \end{aligned}$$

Next, we calculate the probability of success, which is $Pr[\cup_{j=1,..,L} g_j(x) = g_j(q) \,|\, d(x,q) \leq R]$.

$$\begin{aligned} Pr[\cup_{j=1,..,L} g_j(x) = g_j(q) \,|\, d(x,q) \leq R] &= 1 - Pr[\cap_{j=1,..,L} g_j(x) \neq g_j(q) \,|\, d(x,q) \leq R] \\ &= 1 - \Pi_{j=1,\ldots,L} Pr[g_j(x) \neq g_j(q) \,|\, d(x,q) \leq R] \\ &\geq 1 - (1 - P_1^K)^L \end{aligned}$$

Last, if we set $L = 1/P_1^K$ and $K = \log N / \log(1/P_2)$, we will have that $L = N^\rho$ where $\rho = \log P_1 / \log P_2$. From this we have that the probability of success is $\geq 1 - (1 - \frac{1}{L})^L$ which is approximately $1 - \frac{1}{e}$. Hence, if we let $\delta = \frac{1}{e}$, we have fulfilled the requirement for the (c, R)-Near Neighbor Problem.