CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Yokila Arora (yarora@stanford.edu)

HW#2 – Solution

1. **List Prefix Sums** As described in class, List Prefix Sums is the task of determining the sum of all the elements before each element in a list. Let us consider the following simple variation.

   - Select each element from the list randomly and independently with probability $1/\log n$ and add it to a set S. Add the head of the list to this set, and mark all these elements in the list.
   - Start from each element $s \in S$, and in parallel traverse the lists until you find the next element in S (by detecting the mark) or the end of the list. For $s \in S$, call this element found in this way $\texttt{next}(s)$. While traversing, calculate the sum from s to $\texttt{next}(s)$ (inclusive of s but exclusive of $\texttt{next}(s)$), and call this $\texttt{sum}(s)$.
   - Create a list by linking each $s \in S$ to $\texttt{next}(s)$ and with each node having weight $\texttt{sum}(s)$.
   - Compute the List Prefix Sums on this list using pointer jumping. Call the result $\texttt{prefixsum}(s)$.
   - Go back to the original list, and again traverse from each s to $\texttt{next}(s)$ starting with the value $\texttt{prefixsum}(s)$ and adding the value at each node to a running sum and writing this into the node. Now all elements in the list should have the correct prefix sum.

   Analyze the work and depth of this algorithm. These should both be given with high probability bounds.

   Solution

   Proposition 1. *With high probability* $|S| = O(n/\log n)$.

   Proof. We will show that the size of S is $O(n/\log n)$ with high probability using a Chernoff bound. Associate indicator variable X_i with the i-th element of the initial list to indicate whether it has been selected for inclusion in S. Since we sample each element with probability $1/\log n$, we see that X_i's are Bernoulli distributed where

   $$X_i = \begin{cases} 1 & \text{with probability } 1/\log n, \\ 0 & \text{otherwise.} \end{cases}$$

   Clearly, the X_i are independent. Since our X_i's are indicators, then their probability equals their expectation.[1] By linearity of expectations,

   $$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{\log n} = \frac{n}{\log n}.$$

[1] This follows trivially, since $\mathbb{E}[X_i] = 1 \cdot \Pr(X_i = 1) + 0 \cdot \Pr(X_i = 0) = \Pr(X_i = 1)$.

That is, in expectation $|S| = n/\log n$. Recall Chernoff's Bound, for $\delta \in (0,1)$,

$$\Pr(X > (1+\delta)\mu) < e^{-\delta^2\mu/3}.$$

We evaluate for $\delta = 1/2$ and $\mu = n/\log n$,

$$\begin{aligned}\Pr(X > 3n/(2\log n)) &< e^{\frac{-n}{12\log n}} \\ &< e^{-\log n} = \frac{1}{n}\end{aligned}$$

We have used the fact that for sufficiently large n, $\frac{n}{12\log n} > \log n$. Note that the analysis can be made more tight, but this probability bound will suffice to say that with high probability, $|S| = O(n/\log n)$. □

Total Work We see that steps a, b, c and e all require $O(n)$ work in the worst case. In step d, we use pointer jumping to perform prefix sums of our linked list. This algorithm, on a list of size n, requires $O(n\log n)$ work and $O(\log n)$ depth.[2] Therefore, step d requires $|S|\log|S|$ work. With high probability:

$$|S|\log|S| \leq \frac{3n}{2\log n}\log\left(\frac{3n}{2\log n}\right) = O(n).$$

Total work, then, is $O(n)$.

Total Depth Now we compute the total depth. Steps a and c have depth $O(1)$ because we may do all nodes in parallel. As we stated before, computing prefix sums using pointer jumping on linked lists takes depth $O(\log|S|) = O(\log(n/\log n)$. In steps b and e we need to traverse all elements in the original list between elements in S, so the depth of these steps will be the maximum length between elements in S. We will show with high probability this maximum length will be less than $4\log^2 n$. If this claim holds, then total depth for the algorithm will be $O(\log^2 n)$ with high probability.

Proposition 2. *The maximum length between elements in S is less than $O(\log^2 n)$ with high probability.*

Proof. Consider dividing the original list into chunks of size $2\log^2 n$ (so there will be $n/(2\log^2 n)$ chunks). For every element in the original list, the probability it is not chosen to S is $(1 - 1/\log n)$, independently. The probability that all of the elements in a particular chunk, c_i, of the original list are not in S is given by:

$$\begin{aligned}\Pr(e \notin S, \forall e \in c_i) &= (1-1/\log n)^{2\log^2 n} \\ &= ((1-1/\log n)^{\log n})^{2\log n} \\ &\leq (1/e)^{2\log n} < \frac{1}{n^2}\end{aligned}$$

[2]Please see the Wikipedia page on pointer jumping and/or `http://wwwmayr.informatik.tu-muenchen.de/lehre/2013WS/pa/split/sub-Prefix-Sum-single.pdf` for a discussion on pointer jumping for prefix sums.

By union bound, we know the probability that all the chunks have at least one element in S is $\leq n/(2\log^2 n) \cdot 1/n^2 \leq 1/n$. So with high probability the maximum length between elements in S is bounded by $4\log^2 n$ (the maximum interval between points in consecutive chunks). Thus, the depth of steps b and e is $O(\log^2 n)$ with high probability, and that is the total depth of the algorithm as well. □

2. **Random Mate on Graphs** In class we described a random-mate technique for determining graph connectivity. Each node flips a coin, and every edge from a head to a tail will attempt to hook the tail into the head (i.e., relabel the tail with the head pointer). Given a d-regular graph on n vertices, i.e. a graph in which every vertex has degree d, what is the expected number of vertices after one contraction step?

   Solution A vertex is relabeled if *and only if* its corresponding coin toss is Tails (probability 1/2), and *at least one* of its d neighboring vertices gets a Heads (probability $1-\left(\frac{1}{2}\right)^d$). Since each vertex flips a coin independently, we see that a vertex is relabeled with probability:

$$\begin{aligned}\Pr(\text{node } i \text{ relabeled}) &= \Pr(\text{node i is a follower } \textit{and} \text{ at least one neighbor a leader}) \\ &= \Pr(\text{node i is a follower}) \times \Pr(\text{at least one neighbor of node } i \text{ is a leader}) \\ &= \frac{1}{2}\left(1-\frac{1}{2^d}\right).\end{aligned}$$

   Let X_i denote the indicator variable which takes on value 1 if vertex i relabeled and 0 otherwise. Then the total number of vertices contracted is given by $X_n = \sum_{i=1}^n X_i$. Taking the expectation,

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i] = \sum_{i=1}^n \left(\frac{1}{2}\left(1-\frac{1}{2^d}\right)\right) = \frac{n}{2}\left(1-\frac{1}{2^d}\right).$$

   Hence, the expected number of vertices *remaining* after one contraction step is given by

$$n - \frac{n}{2}\left(1-\frac{1}{2^d}\right) = \frac{n}{2} + \frac{n}{2^{d+1}} = \frac{n}{2}\left(1+\frac{1}{2^d}\right).$$

3. **Minimum Spanning Tree** It turns out that Boruvka's algorithm for Minimum Spanning Trees is actually a parallel algorithm. The algorithm works as follows. Assume that the input graph is undirected.

Algorithm 1: Parallel MST
1 Start with an empty minimum spanning tree M
2 Every vertex determines its least weight incident edge and adds this to a set T and to our minimum spanning tree M (The set T forms a forest)
3 We run tree contraction on each tree in the forest to label all vertices in a tree with the same label
4 We update all edges so they point between new labels and delete self edges
5 If there are any edges left, return to the second step.

Analyze the work and depth of this algorithm in terms of the number of vertices n and the number of edges m.

Solution Our graph is stored in Adjacency-List form. We start by assuming the initial graph is connected.[3]

Finding Minimum Weight Incident Edge Each iteration, we find the minimum weight edge incident on each vertex. In the event that there are several min-weight edges at a vertex, we choose one arbitrarily. For *each* node, finding a min-weight incident edge requires $O(n)$ work, since in the worst case the node may be connected to each of the $n-1$ other nodes; in order to find the minimum we must consider each incident edge. Since `min` is an associative binary operator, we know how to compute the minimum of an n element list with linear work and logarithmic depth.[4]

Hence this step requires $W(n) = O(n), D(n) = O(\log n)$.

Realization: Adding Incident Edges yield another Forest Observe that at step 0, each node its own connected component, and we trivially have a forest of n singleton trees. Suppose we have found the minimum weight edge incident to each vertex; notice that by the Red-Rule, these edges must belong in the MST, hence they cannot form a cycle.[5] Further observe that adding in these edges yields yet another forest.

A Note on Data Structures We now introduce a few data-structures to make bookkeeping easier. We will use Star Contraction to contract our edges, and this requires us to toss a coin for each vertex. Imagine storing the results of the coin tosses in an n-bit sequence, where a 0 denotes the corresponding vertex flipped Tails and a 1 denotes a Heads. We note that allocating storage for this bit-sequence requires constant work and depth since we don't require the values to be initialized. In addition, we allocate storage for an n-element array in constant time and depth. This array will help us track (for each node) what its parent node is (i.e., if node u a follower and contracts into node v, a leader, then we write in position u of the array the value v).

Contracting Edges via Star Contraction We now focus on how to contract a tree using *Star Contraction*: each node flips a coin; if the coin lands Tails and at least one neighbor flips Heads, then we say the satellite node contracted to the center of the star. We may perform coin-flips in parallel, requiring $O(n)$ work and $O(1)$ depth. Each processor (corresponding to a single node) then writes down either a zero or one in the corresponding position of the n-element array.

[3] If the graph is *not* connected, then there does not exist any spanning tree.

[4] See Lecture 1, or problem set 1 question 5.

[5] For a proof of the Red-Rule, refer to Lecture 5, theorem 2.1.

In the event that we contract an edge (i.e. two vertices become merged into a super-vertex), then we must update our data structures. Suppose we contract edge (u, v), where u a follower and v a leader. We first update our n-element array to indicate that node u belongs to node v's connected component. We then must add all neighbors of u to the neighborhood list of v.

Naively, we may look through each edge of the follower nodes and add it to the neighborhood list of its corresponding leader node. In the worst case, we might have to add $O(m)$ edges to some neighborhood; this neighborhood is a BST with height $O(\log n)$. Realize, however, that each edge may be contracted in parallel, thanks to our Star Contraction algorithm which ensures that each node contracts into at most one other node (each iteration). Hence tree contraction requires $O(m)$ work and $O(\log^2 n)$ depth.[6]

Relabeling Edges Realize that our edges now must be relabeled, since it's possible that for an edge, call it (u, v), that it itself was *not* contracted, and yet one of its incident nodes was chosen to be contracted into another super-vertex. In this case, edge label (u, v) is no longer accurate, instead we need to find u's parent (call it w) and relabel the edge (u, v) to (w, v).

To do this, we must consider each edge, hence we require $O(m)$ work. However, each edge can be relabeled in constant time thanks to our array data-structure which tracks children-parent node relationships. Hence we only require $O(1)$ depth to relabel.

Self Loops We must now also take care to discard self loops; i.e., suppose we choose to contract edge (u, v). Clearly, u, v belonged to different components at the start of the iteration. It's possible that a node in u's component, call it x, has a neighbor in v's component, call it y. When we contract edge (u, v), we effectively have created a self-loop via edge (x, y). This is simple to remove. Thanks to our array data structure, we may check in constant time what x's "parent" is, and what y's "parent" is. If the parents are identical, then we have a self loop and the edge may be discarded.

Realize that we may do this in parallel for each edge. Hence to discard self-loops requires $O(m)$ work and $O(1)$ depth.

Total Number of Iterations Required In each iteration, the vertex shrinks by half because a forest constructed by selecting the minimum weight edge for each vertex of a connected graph can not have a disconnected component that consists of a single vertex. Hence, each connected component of such a forest contains at least two vertices, and the forest of x vertices contains not more than $x/2$ connected components, which after a tree contraction yields a graph on less than $x/2$ vertices.

Hence, total number of iterations required is $O(\log n)$.

[6]For more detail, see Blelloch and Maggs Chapter 17 Section 17.3, or lecture 5.

Total Work and Depth Hence, the max number of iterations is $\log n$, hence the total work is $O(m \log n)$ and the depth is $O(\log^3 n)$.

4. Implement logistic regression using tensorflow. Use the following code to generate train and test data. Note that we have set seed (using "random_state=42"). Use cross-entropy loss and gradient descent optimizer with a learning rate of 0.01. Use batch_size of 100, and run for 500 steps. Report the accuracy on test set.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Generate data
X_data, y_data = make_classification(n_samples=200, n_features=2,
n_redundant=0, random_state=42)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
test_size=0.2, random_state=42)

# Plot training data
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.show()
```

Solution

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

# Generate data
X_data, y_data = make_classification(n_samples=200, n_features=2,
                                     n_redundant=0, random_state=42)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
test_size=0.2, random_state=42)

# Plot training data
#plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
#plt.show()

y_train = y_train.reshape((-1,1))
```

```
y_test = y_test.reshape((-1,1))

# Define parameters
learning_rate = 0.01
batch_size = 100
num_steps=500
n_samples=X_train.shape[0]

# Define placeholders for input
X = tf.placeholder(tf.float32, shape=[None, 2])
y = tf.placeholder(tf.float32, shape=[None, 1])

# Define variables to be learned
W = tf.get_variable("weights", (2,1), initializer =
 tf.random_normal_initializer())
b = tf.get_variable("bias", (1,), initializer =
tf.constant_initializer(0.0))
y_pred = tf.matmul(X,W)+b
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=
y_pred, labels=y))

# Define optimizer
opt = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

with tf.Session() as sess:
    #Initialize Variables in graph
    sess.run(tf.global_variables_initializer())
    for _ in range(num_steps):
        # Select random minibatch
        indices = np.random.choice(n_samples, batch_size)
        X_batch, y_batch = X_train[indices,:], y_train[indices]
        # Do gradient descent step
        _, loss_val = sess.run([opt, loss], feed_dict={X: X_batch,
        y: y_batch})
        print(loss_val)
    # Test model
    correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1)
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print("Accuracy:", accuracy.eval({X: X_test, y: y_test}))
```