CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)

HW#2 – Due at the beginning of class Thursday May 3

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.

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?

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.

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()
```