1. Consider solving connected components via the Pregel framework. In general the naive algorithm works well because the diameter of the graph is low. However for arbitrary graphs the diameter can be $O(n)$. In this question we consider some attempts to reduce the worst-case runtime of the naive algorithm.

Recall that at step $s$ of the naive algorithm the state of node $i$ is the maximum ID of all nodes within distance $s$ of node $i$.

(a) Suppose we run the naive algorithm, but on the $k$th iteration we add an edge between each node $i$ and the node $j$, with the maximum ID within distance $k$ of $i$, if and only if the shortest path between $i$ and $j$ is of length exactly $k$. Once these edges have been added the naive algorithm completes running as normal. Provide an example of a graph with high diameter and give a description of the node IDs for which this algorithm would add $O(n - k)$ edges (given $n$ is large and the graph is sparse, this is undesirable as it significantly increases the shuffle size). Analyze the number of iterations required by this algorithm.

(b) Given the same algorithm in (a) and the same graph, identify the distribution of IDs that achieves the same number of iterations but with the fewest number of edges added. Analyze the number of iterations and identify the optimal $k$.

(c) Now suppose we run the naive algorithm, but on the $k$th iteration each edge selects a node $j$ at random with distance $k$ from itself. If the distance from $i$ to $j$ is greater than $\alpha k$ then we add an edge to the graph; otherwise we do nothing. Provide an algorithm that would implement this efficiently (assume $k$ and $\alpha$ are global parameters known to all nodes).

(d) Analyze the expected performance of the algorithm in (c) on the chain graph. What values of $\alpha$ and $k$ would you suggest? What kind of speed up is possible?

Solution:

The motivation for this question is as follows: Consider the chain graph. Note that, solving connected components using the naive algorithm requires $O(n)$ iterations on this graph. Number the nodes consecutively $1, \ldots, n$ starting from one of the leaf nodes. For $i = 1, \ldots, \sqrt{n}$, add edge $((i - 1)\sqrt{n}, i\sqrt{n})$ to the graph. We add $\sqrt{n}$ edges. The naive connected components algorithm now requires only $O(\sqrt{n})$ iterations, a significant improvement. The challenge is that we do not know the graph structure prior to running the algorithm and so must add extra edges during the run time of the algorithm.

(a) Consider the chain graph with node IDs ordered from highest to lowest. At the beginning of the algorithm the state of node $i$ is $i$. After $k$ iterations the state of node $i$ is $\max(i + k, n)$. Hence after $k$ iterations we add an edge between each node $i$ and $i + k$ ($i = n, \ldots, k + 1$). Therefore we add $O(n - k)$ edges.
The algorithm requires $O(k)$ iterations before new edges get added and $O(n/k)$ iterations once new edges have been added. So total number of iterations is $O(k + n/k)$ which is minimized for $k = \sqrt{n}$.

(b) Again consider the chain graph. Take the $n/k$ largest node IDs $(n, \ldots, n - \frac{n}{k} + 1)$. Assign these nodes IDs to the graph as follows:

i. Place ID $n$, the largest ID, on one of the leaf nodes
ii. Please ID $n - i$ at distance $ki$ from the largest ID.

Call the nodes with IDs assigned this way the large nodes. The remaining IDs may be assigned arbitrarily.

Under this assignment of IDs, after $k$ iterations the state of a node distance $j$ from the largest ID is $n - \lceil \frac{j}{k} \rceil + 1$. Hence after $k$ iterations we add an edge between each pair of consecutive large nodes. Therefore we add $O(n/k)$ edges.

The algorithm requires $O(k)$ iterations before new edges get added and $O(n/k)$ iterations once new edges have been added. So total number of iterations is $O(k + n/k)$ which is minimized for $k = \sqrt{n}$.

(c) For this algorithm we shall require the use of a hash function that takes two arguments: $f(i, u)$. Let the state of each node $i$ be given by 3 values:

i. $i.\text{maxJ}$: the ID of node $j$, the maximum ID seen by node $i$ so far;
ii. $i.\text{maxHash}$: the ID of node $v$, with maximum hash value so far;
iii. $i.\text{hashDist}$: the minimum distance between node $i$ and node $v$.

We decompose the algorithm into three phases: before, during and after the $k$-th iteration. During the first $k$ iterations:

i. message passing
   - each node $i$ passes $(i.\text{maxJ}, -1)$ to all its neighbors (this is the standard message used in the naive algorithm)
   - if iteration = 1: each node $i$ passes the message $(i, 1)$ to all its neighbors
   - if iteration $\geq 1$: for each message of the form $(u, d \geq 0)$ that node $i$ received, it passes the message $(u, d + 1)$ to all its neighbors.

ii. message processing
   - each node $i$ updates $i.\text{maxJ}$ to be the maximum of its current value and $\max(j : j \in (j, -1) \text{was a message received})$.
   - each node $i$ and message $(u, d)$ we compute the hash $f(i, u)$ is this value is strictly larger than the hash $f(i, i.\text{maxHash})$ then we update $i.\text{maxHash}$ to be $u$ and $i.\text{hashDist}$ to be $d$.

Immediately after the $k$-th iteration:

i. each node $i$ compares the distance between itself and node $v$ with the value $\alpha k$
ii. if the distance is at least $\alpha k$ then we add the edge $(i, v)$ to the graph
iii. otherwise node $i$ does nothing
From the $k + 1$-th iterations onwards we resume the naive algorithm but now including the additional edges. Once the algorithm completes we delete the additional edges.

Note that during phase (i) there is a lot of message passing. However, if $k$ is small and the graph is sparse the number of messages passed will remain small. Some supplementary techniques can be used to reduce the number of messages passed, but for now we shall ignore this concern.

(d) We begin by noting that node $i$ is responsible for the create of a new edge with probability $1 - \alpha$, hence the algorithm adds $(1 - \alpha)n$ new edges in expectation (by Chernoff bounds the actual number of edges added will be close to its expectation). The key thing we need to analyze is how much does adding an edge reduce the diameter of the graph?

Consider an interval of length $\alpha k$, each node in this interval has probability $1 - \alpha$ of creating a new edge. In the chain graph, there are $2(1 - \alpha)k$ nodes that could send an edge to node $i$. These nodes send an edge with probability $2/k$. Hence the probability another node creates an edge to node $i$ is $1 - \alpha$. Therefore the expected number of new edges on a node $i$ is $2(1 - \alpha)$ and the expected number of edges with one end point in an interval of length $\alpha k$ is $2(1 - \alpha)\alpha k$.

Consider the overlap of two intervals of length $\alpha k$. If the two intervals overlap by less than a third of their length then the reduction in diameter from having both intervals, instead of just one, is at least $1/3\alpha k$. Given two intervals that overlap, the probability they overlap by less than a third of their length is $1/3$ (under assumptions of uniform random starting point for each interval). Hence in expectation each edge added to the graph reduces the diameter by at least $c\alpha k$, where $c \in (0, 1)$ is a constant.

Therefore the total number of iterations needed is upper bounded by:

$$n - c(1 - \alpha)\alpha nk + k$$

And the number of additional edges added is $(1 - \alpha)n$. For $k = O(\sqrt{n})$ and $\alpha = O(1/\sqrt{n})$ we have an upper bound of $O(\sqrt{n})$ iterations while adding only $O(\sqrt{n})$ edges.

2. For The Perceptron, we proved in class that if all points are scaled to have unit norm, and the two classes can be separated by margin at least $\gamma$, then the perceptron will make at most $1/\gamma^2$ mistakes. Notice that the margin changes depending on the scaling of the points.

Prove that if all the points have norm at most $R$ (i.e. $|x_i| < R$ for all $i$ - all points live within radius $R$ of the origin), then the perceptron makes at most $R^2/\gamma^2$ mistakes.

Solution:

Let $w_i$ denote the $i$th cutting plane, $w^*$ denote the optimum cutting plane and $x$ denote a data point.

Then if $w_i \cdot x$ is a false negative we have:

$$w_{i+1} \cdot w^* = (w_i - x) \cdot w^* = w_i \cdot w^* - x \cdot w^* \geq w_i \cdot w^* + \gamma$$
And also:
\[ \|w_{i+1}\|^2 = \|w_i - x\|^2 = \|w_i\|^2 - 2w_i \cdot x + \|x\|^2 \leq \|w_i\|^2 + \|x\|^2 = \|w_i\|^2 + R^2 \]

Similarly, if \( w_i \cdot x \) is a false positive we have:
\[ w_{i+1} \cdot w^* = (w_i + x) \cdot w^* = w_i \cdot w^* + x \cdot w^* \geq w_i \cdot w^* + \gamma \]
And also:
\[ \|w_{i+1}\|^2 = \|w_i + x\|^2 = \|w_i\|^2 + 2w_i \cdot x + \|x\|^2 \leq \|w_i\|^2 + \|x\|^2 = \|w_i\|^2 + R^2 \]

Combining these two results after \( M \) mistakes and assuming \( w_0 = 0 \) gives:
\[ M\gamma \leq \|w_{M+1} \cdot w^*\| \leq \|w_{M+1}\| \leq R\sqrt{M} \]

Hence
\[ M\gamma \leq R\sqrt{M} \quad \Rightarrow \quad M \leq \frac{R^2}{\gamma^2} \]

3. In class we saw how to compute highly similar pairs of \( m \)-dimensional vectors \( x, y \) via sampling in the mappers, where the similarity was defined by cosine similarity: \( \frac{x^T y}{|x||y|} \).

Show how to modify the sampling scheme to work with overlap similarity, defined as
\[ \text{overlap}(x, y) = \frac{x^T y}{\min(|x|^2, |y|^2)} \]

(a) Prove shuffle size is still independent of \( m \), the dimension of \( x \) and \( y \).
(b) Assuming combiners are used with \( B \) mapper machines, analyze the shuffle size.

Solution:

(a) We modify the DIMSUM mapper as follows:

\begin{algorithm}
\caption{DIMSUMOverlapMapper\( (r_i) \)}
1: for all pairs \( (a_{ij}, a_{ik}) \) in \( r_i \) do
2: \quad With probability \( \min(1, \gamma \min(\|c_i\|^2, \|c_j\|^2)) \)
3: \quad emit \( ((j, k) \rightarrow a_{ij}a_{ik}) \)
4: end for
\end{algorithm}

The shuffle size of this scheme is \( O(nL\gamma/H^2) \) where \( H \) is the smallest nonzero element of \( A \) in magnitude. To show this we start with the expected contribution
from each pair of columns.

\[
\sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{\#(c_i, c_j)} P(\text{DIMSUMOverlapEmit}(c_i, c_j))
\]

\[
\sum_{i=1}^{n} \sum_{j=i+1}^{n} \#(c_i, c_j) P(\text{DIMSUMOverlapEmit}(c_i, c_j))
\]

\[
\leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \gamma \frac{\#(c_i, c_j)}{\min(||c_i||^2, ||c_j||^2)}
\]

\[
\sum_{i=1}^{n} \sum_{j=i+1}^{n} \gamma \frac{\#(c_i, c_j)}{c_i^T c_i}
\]

\[
\leq \gamma \sum_{i=1}^{n} \frac{1}{c_i^T c_i} \sum_{j=1}^{\#(c_i, c_j)}
\]

\[
\leq \gamma \sum_{i=1}^{n} \frac{1}{\#(c_i)^2} L \#(c_i)
\]

\[
= \gamma \frac{L}{H^2}
\]

The fourth equality comes from assuming WLOG $\|c_i\|^2 \leq \|c_j\|^2$.

(b) In the naive case with combiners, each of the $B$ machines will emit at most $n^2$ pairs — one for each element in $A^T A$. However, without combiners we know that DIMSUM will have a shuffle size of at most $nL\gamma/H^2$. Thus the shuffle size is at most $O(\min(Bn^2, nL\gamma/H^2))$.

4. Assume we are optimizing a separable function of $n$ separable parts, where the function has $p$ parameters to be tuned.

(a) Recall that Parallel Stochastic Gradient Descent for strongly convex functions needs one shuffle of the data and one Reduce to compute an average. What is the total communication cost for this, and how much time does it take when $k$ processors are used?

(b) Similarly, ADMM requires $T$ iterations, where the communication cost of each iteration consists of an AllReduce. With $T$ iterations, what is the total communication cost of ADMM, and how much time does it take?

Finally, notice that the amount of communication performed by ADMM is independent of $n$, whereas Parallel SGD is not.

**Solution:**

Throughout this analysis we shall assume that $k \leq n$ so some machines run 2 or more separable functions.

(a) Parallel Stochastic Gradient Descent:
For the shuffle, in the worst case we must shuffle all the data. So communication cost for shuffle is $O(n)$.

Assume that any machines processing more than one separable function combine their outputs prior to the AllReduce. Then for the AllReduce:

i. In the first stage $k/2$ machines send $p$ data to another $k/2$ machines.
ii. In the second stage $k/4$ machines send $p$ data to another $k/4$ machines.
iii. ...and so forth...
iv. In the last stage 1 machine sends $p$ data to 1 other machine.

Communication cost for AllReduce is: $O(kp)$.

Therefore total communication cost for PSGD algorithm is $O(n + kp)$ each iteration.

To achieve $1/\varepsilon$ error we require $O(\log(1/\varepsilon))$ iterations and each iteration involves $O(p)$ work. For the AllReduce there are $\log(k)$ stages, each of which involves $O(p)$ computation. Therefore the total computation time for an AllReduce is $O(p \log(k))$.

(b) Alternating Direct Method of Multipliers:

As per the PSGD method, each AllReduce has communication cost $O(kp)$. Note that once the average has been computed it must then be distributed to the original $k$ machines. This doubles the amount of communication that takes place. One AllReduce is required each iteration and there are $T$ iterations.

Therefore the total communication cost for ADMM algorithm is $O(Tkp)$ per (outer) iteration.

To achieve $1/\varepsilon$ error we require $O(\log(1/\varepsilon))$ outer iterations and each iteration involves $O(p)$ work. As per the PSGD method, each AllReduce has $\log(k)$ stages, each of which involves $O(p)$ computation. Therefore the total computation time for an AllReduce is $O(p \log(k))$. 
