CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Anuj Nagpal (anujnag@stanford.edu)
HW#4 - Due 1st June

1. **PageRank Computation:** Consider a network of websites given to you as a directed graph $G = (V, E)$ with adjacency matrix A and a out-degree matrix D such that you can define $Q = D^{-1}A$:

   (a) Power Iteration: Assume that you have a $n \times n$ matrix M with all eigenvalues distinct. For any vector r_0, prove that the iterated product $M^k r_0$ converges to the eigenvector corresponding to largest eigenvalue of M as $k \to \infty$. Make sure to state any constraint on r_0 assumed for proving this convergence.

   (b) In the pagerank lecture notes, it is stated that: *The stationary distribution specifies what proportion of time on average is spent at specific node during an infinitely long random walk.*

   With this infinitely long random walk analogy with transition probability of the random walker given by matrix Q, describe two different scenarios where the random walker can 'get stuck' despite every node in the graph having atleast one incoming edge. Draw an example graph for each scenario and explain how adding some teleportation probability from each node to every other node will help.

   (c) Prove that the stationary distribution r of node probabilities for matrix

   $$P = \alpha Q + (1 - \alpha) \left[\frac{1}{N}\right]_{N \times N}$$

   satisfies the pagerank equation:

   $$r_j = \sum_{i \to j} \alpha \frac{r_i}{deg_i} + (1 - \alpha)\frac{1}{N}$$

   (d) Write a Spark program to compute pageranks for the following graph given as *(source, destination)* edge list. Report your rank value for each URL after 20 iterations. Assume you start with an r_0 having all 1-s and you compute rank vector iteratively as:

   $$(r_j)_{t+1} = \sum_{i \to j} 0.85 \frac{(r_i)_t}{deg_i} + 0.15$$

   ```
   google.com wikipedia.org
   stanford.edu google.com
   youtube.com stanford.edu
   youtube.com google.com
   wikipedia.org youtube.com
   wikipedia.org google.com
   ```

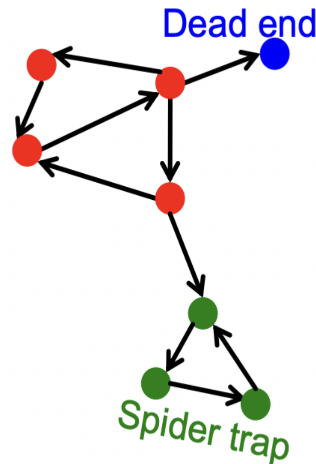
Solution:

(a) Since M has distinct eigenvalues $\lambda_1 > \lambda_2 > \ldots > \lambda_n$, it will have n linearly independent eigenvectors $x_1, x_2, \ldots, x_n$ which form a basis. So we can write r_0 as $c_1x_1 + c_2x_2 + \ldots + c_nx_n$, which gives:

$$\begin{aligned}
Mr_0 &= M(c_1x_1 + c_2x_2 + \ldots c_nx_n) \\
&= c_1(Mx_1) + c_2(Mx_2) + \ldots + c_n(Mx_n) \\
&= c_1(\lambda_1x_1) + c_2(\lambda_2x_2) + \ldots + c_n(\lambda_nx_n) \\
M^kr_0 &= c_1(\lambda_1^kx_1) + c_2(\lambda_2^kx_2) + \ldots + c_n(\lambda_n^kx_n) \\
&= \lambda_1^k\left[c_1x_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k x_2 + \ldots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k x_n\right]
\end{aligned}$$

Since $\lambda_1 > \lambda_i$ for $i = 2, 3, \ldots, n$, $\left(\frac{\lambda_i}{\lambda_1}\right)^k \to 0$ as $k \to \infty$ such that $M^kr_0 \approx c_1(\lambda_1^kx_1)$. Note that we assume $c_1 \neq 0$ for this convergence to largest eigenvalue.

(b) Any example of dead ends (some pages have no out-links) and spider traps (all out-links within the same group) counts as solution. Teleporting helps you to get out in a finite number of steps if you are stuck.



(c) For stationary distribution r, we can say $Pr = r$ which gives:

$$\begin{aligned}
r &= \alpha Qr + (1-\alpha)\left[\frac{1}{N}\right]_{N\times N} r \\
r_j &= \sum_{i\to j} \alpha\frac{r_i}{deg_i} + (1-\alpha)\frac{1}{N}
\end{aligned}$$

Using the fact that $\sum r_i = 1$

(d) Following spark code will read the text file and compute the pagerank vector:

```
val lines = sc.textFile("url.txt")
val links = lines.map{ s =>
  val parts = s.split("\\s+")
  (parts(0), parts(1))
}.distinct().groupByKey().cache()

var ranks = links.mapValues(v => 1.0)
val iters = 20

for (i <- 1 to iters) {
  val contribs = links.join(ranks).values.flatMap{ case (urls, rank) =>
    val size = urls.size
    urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}

val output = ranks.collect()
output.foreach(tup => println(tup._1 + " has rank: " + tup._2 + "."))
```

Output:

```
youtube.com has rank: 0.7323900229505396.
google.com has rank: 1.4357617405523626.
wikipedia.org has rank: 1.3705281840649928.
stanford.edu has rank: 0.4613200524321036.
```

2. **Singular Value Decomposition:**

(a) Write a Spark program to compute the Singular Value Decomposition of the following 10×3 matrix M:

```
-0.5529181 -0.5465480 0.009519836
-0.5428579 -1.5623879 0.982464609
-1.3038629  0.5715549 0.499441144
 0.6564096  1.1806877 0.495705999
-1.2061171  1.3430651 0.153477135
 0.2938439 -1.7966043 0.914381381
-0.2578953  0.2596407 0.815623895
 0.9659582  2.3697927 0.320880634
-0.4038109  0.9846071 0.488856619
 0.6029003 -0.3202214 0.380347546
```

The matrix M should be stored in a row matrix format i.e. the rows should be split up and inserted into an RDD. Report all singular vectors and values and submit your Spark program. You can use Spark's MLlib library for this problem.

(b) The eigenvalue decomposition of a real, symmetric, and square matrix A (of size $d \times d$) can be written as the following product: $A = Q\Lambda Q^T$ where $\Lambda = diag(\lambda_1, \dots, \lambda_d)$ contains the eigenvalues of A (which are always real) along its main diagonal and Q is an orthogonal matrix containing the eigenvectors of A as its columns.
Can we do eigenvalue decomposition of $M^T M$?

(c) Prove that the nonzero eigenvalues of MM^T are the same as the nonzero eigenvalues of $M^T M$. You may ignore multiplicity of eigenvalues. Are their eigenvectors the same?

(d) Compute the eigenvalue decomposition of $M^T M$. What is the relationship (if any) between the eigenvalues of $M^T M$ and the singular values of M?

Solution:

(a) The following Spark code computes the singular value decomposition for given matrix M:

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix

val data = Array(
  Vectors.dense(-0.5529181, -0.5465480, 0.009519836),
  Vectors.dense(-0.5428579, -1.5623879, 0.982464609),
  Vectors.dense(-1.3038629,  0.5715549, 0.499441144),
  Vectors.dense(0.6564096,  1.1806877, 0.495705999),
  Vectors.dense(-1.2061171,  1.3430651, 0.153477135),
  Vectors.dense(0.2938439, -1.7966043, 0.914381381),
  Vectors.dense(-0.2578953, 0.2596407, 0.815623895),
  Vectors.dense(0.9659582,  2.3697927, 0.320880634),
  Vectors.dense(-0.4038109,  0.9846071, 0.488856619),
  Vectors.dense(0.6029003, -0.3202214, 0.380347546))

val rows = sc.parallelize(data)

val mat: RowMatrix = new RowMatrix(rows)

// Compute the top 3 singular values and
// corresponding singular vectors.
val svd: SingularValueDecomposition[RowMatrix, Matrix]
            = mat.computeSVD(3, computeU = true)
val U: RowMatrix = svd.U
val s: Vector = svd.s
val V: Matrix = svd.V
```

Output:

```
s: org.apache.spark.mllib.linalg.Vector =
[4.042401823284167,2.4291077696900882,1.8100732530241168]
V: org.apache.spark.mllib.linalg.Matrix =
0.06659805628006435    -0.9690419589598888   0.2377443599223723
0.9957712030226099     0.0796612820078603    0.04575796520572747
-0.06328040874253092   0.23369159574970882   0.9702493637953764
```

(b) $(M^TM)^T = M^T(M^T)^T = M^TM$, so it is symmetric.
M^TM is square with dimension 3×3 in this case.
Since all entries in M are real, M^TM will also have real entries.
Since M^TM is real, symmetric and square, we can do its eigendecomposition as stated in the problem statement.

(c) For an eigenvector v of MM^T such that $MM^Tv = \lambda v$, we can multiply both sides by M^T to get:

$$M^TM(M^Tv) = \lambda(M^Tv)$$

Hence λ is also an eigenvector of M^TM with corresponding eigenvector being M^Tv. This eigenvector need not be the same except when $M = I$

(d) The singular values of M are square root of eigenvalues of M^TM

```
>>> import numpy as np
>>> M = np.array([[-0.5529181,  -0.5465480,  0.009519836],
                  [-0.5428579,  -1.5623879,  0.982464609],
                  [-1.3038629,   0.5715549,  0.499441144],
                  [0.6564096,    1.1806877,  0.495705999],
                  [-1.2061171,   1.3430651,  0.153477135],
                  [0.2938439,   -1.7966043,  0.914381381],
                  [-0.2578953,   0.2596407,  0.815623895],
                  [0.9659582,    2.3697927,  0.320880634],
                  [-0.4038109,   0.9846071,  0.488856619],
                  [0.6029003,   -0.3202214,  0.380347546]])
>>> A = np.matmul(M.T, M)
>>> w, v = np.linalg.eig(A)
>>> w
array([16.3410125 ,   5.90056456,   3.27636518])
```

3. Given a matrix M in row format as an RDD[Array[Double]] and a local vector x given as an Array[Double], give Spark code to compute the matrix vector multiply Mx.

Solution:

```
x_bc = sc.broadcast(x)
output = M.map(lambda row: np.dot(row, x_bc.value)).collect()
```

4. 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: $\frac{x^Ty}{|x|_2|y|_2}$.

Show how to modify the sampling scheme to work with overlap similarity, defined as

$$\text{overlap}(x, y) = \frac{x^T y}{\min(|x|_2^2, |y|_2^2)}$$

(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:

Algorithm 1 DIMSUMOverlapMapper(r_i)

1: **for** all pairs (a_{ij}, a_{ik}) in r_i **do**

2: With probability $\min\left(1, \gamma \frac{1}{\min(\|c_i\|_2^2, \|c_j\|_2^2)}\right)$

3: emit $((j, k) \to a_{ij}a_{ik})$

4: **end for**

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.

$$\begin{aligned}
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{\#(c_i,c_j)} P(\text{DIMSUMOverlapEmit}(c_i, c_j)) \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \#(c_i, c_j) P(\text{DIMSUMOverlapEmit}(c_i, c_j)) \\
&\leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \gamma \frac{\#(c_i, c_j)}{\min(\|c_i\|_2^2, \|c_j\|_2^2)} \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \gamma \frac{\#(c_i, c_j)}{c_i^T c_i} \\
&\leq \gamma \sum_{i=1}^{n} \frac{1}{c_i^T c_i} \sum_{j=1}^{n} \#(c_i, c_j) \\
&\leq \gamma \sum_{i=1}^{n} \frac{1}{\#(c_i) H^2} L\#(c_i) \\
&= \gamma Ln/H^2
\end{aligned}$$

The fourth equality comes from assuming WLOG $\|c_i\|_2^2 \leq \|c_j\|_2^2$.

(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 $O(\min(Bn^2, nL\gamma/H^2))$.

5. **Shallow Graphs** For an undirected graph $G = (V, E)$ with n vertices and m edges ($m \geq n$), we say that G is shallow if for every pair of vertices $u, v \in V$, there is a path from u to v of length at most 2 (i.e. using at most two edges).

   (a) Give an algorithm that can decide whether G is shallow in $O(n^{2.376})$ time.

   (b) Given an $n \times r$ matrix A and an $r \times n$ matrix B where $r \leq n$, show that we can multiply A and B in $O\left((n/r)^2 r^{2.376}\right)$ time. Hint: use the fact that we can multiply two $r \times r$ matrices in $O(r^{2.376})$ time.

   (c) Give an algorithm that can decide whether G is shallow in $O(m^{0.55}n^{1.45})$ time. Hint: consider length-2 paths that go from low-degree vertices and length-2 paths that go through high-degree vertices separately. Use result from part (b).

   Solution:

   (a) Consider the adjacency matrix A for G. A_{ij} contains the number of paths of length 1 from node i to j. similarly, A^2_{ij} contains the number of paths of length 2 from node i to j. Thus $(A^2 + A)_{ij}$ contains the number of paths of length at most 2 from node i to j. Our algorithm will compute $A^2 + A$ and return true if and only if all non-diagonal entries of $A^2 + A$ are non-zero. A^2 can be computed in $O(n^{2.376})$ using Strassen's algorithm. A be be computed in $O(n^2)$ time, for a total running time of $O(n^{2.376} + n^2) = O(n^{2.376})$.

   (b) We simply split up the $n \times r$ matrix into n/r $r \times r$ matrices, and use block matrix multiplication. In the case that r does not divide n exactly, we can simply add rows of zeros to the left-hand multiplicand matrix, and add columns of zeros to the right-hand multiplicand matrix and then remove extraneous rows and columns from the result.
   We perform $\lceil n/r \rceil \times \lceil n/r \rceil$ block matrix multiplications, each taking $O(r^{2.376})$ time.
   The runtime will be $O(\lceil n/r \rceil^2 r^{2.376}) = O((n/r + 1)^2 r^{2.376}) = O((n/r)^2 r^{2.376})$.

   (c) We will maintain a boolean matrix M that will have $M_{ij} = 1$ if and only if there is a path of length at most 2 between node i and j. We initialize $M = A$, the adjacency matrix for G, leaving only paths of length 2 to be considered. At the end, we check each entry of M and claim the graph is shallow if and only if all non-diagonal entries of M are positive. Since M is initialized to A, it already contains paths of length 1. We will continuously update M to take into account paths of length 2. To do that, we look at all possible ordered triples (u, v, w). Each triple defines a path of length 2 going from u to w, through v.
   We split the vertex set into two sets:

$$V_H = \{v \in V | deg(v) > d\}, V_L = \{v \in V | deg(v) \leq d\}$$

   Consider each ordered triple (u, v, w) defining a path from u to v to w. Either $v \in V_L$ or $v \in V_H$.

Algorithm 2

```
1: for edge (v, w) ∈ E do
2:     if v is low-degree then
3:         for each neighbor u of v do
4:             M_uw = 1
5:             M_wu = 1
6:         end for
7:     end if
8:     if w is low-degree then
9:         for each neighbor u of w do
10:            M_uv = 1
11:            M_vu = 1
12:        end for
13:    end if
14: end for
```

Case: $v \in V_L$, i.e. the middle vertex is low degree This step takes at most $O(md)$ time since for each edge we check at most d neighbors.

Case: $v \in V_H$, i.e. the middle vertex is high-degree. We construct a matrix B with dimensions $n \times r$ where $r = |V_H|$. Each row corresponds to a node in V and each column corresponds to a node in V_H. $B_{ij} = 1$ if and only if there is an edge between arbitrary node i and V_H-member j. Thus BB^T gives us the number of paths of length 2 from arbitrary node i to arbitrary node j that go through some high-degree node as the middle node. We can do the BB_T computation in $O((n/r)^2 r^{2.376})$ time. We then update M to $M = M + BB^T$.

Since $2m =$ sum of all degrees $\geq |V_H|d = rd$. Thus $r \leq 2m/d$. So the computation takes $O((n/r)^2 r^{2.376}) = O(n^2 r^{0.376}) = O(n^2 (m/d)^{0.376})$.

So now we've covered all cases, M accounts for all possible paths of length 2 going through high-degree or low-degree vertices.

Finally we traverse M and claim the graph is shallow if and only if all non-diagonal entries of M are non-zero. This $O(n^2)$ will be dominated by $O(n^{1.45} m^{0.55})$, since $m \geq n$.

Thus total running time is $O(md + n^2 (m/d)^{0.376})$. We now minimize this bound with respect to d. Setting $md = n^2 (m/d)^{0.376}$ gives $d^* = n^{1.45} m^{-0.45}$. Subbing back in gives a bound of $O(md^* + n^2 (m/d^*)^{0.376}) = O(n^{1.45} m^{0.55})$.