Homework #4

(Due: May 6)

Task 1. [160 Points] Sequence Alignment with General Gap Penalty.

The problem of *sequence alignment with general gap penalty (gap problem)* is a generalization of the edit distance problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1x_2 \dots x_m$ into another string $Y = y_1y_2 \dots y_n$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. An affine gap penalty function is predominantly used in bioinformatics. However, in many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case, two new cost functions w_{ins} and w_{del} are defined, where $w_{del}(p, q)$ $(0 \le p < q \le m)$ is the cost of deleting $x_{p+1} \dots x_q$ from X, and $w_{ins}(p, q)$ $(0 \le p < q \le n)$ is the cost of inserting $y_{p+1} \dots y_q$ into X. The substitution function $s(x_i, y_j)$ is the same as that of the standard edit distance problem. Let $G[i, j]$ denote the minimum cost of transforming $X_i = x_1x_2 \dots x_i$ into $Y_j = y_1y_2 \dots y_j$ (where $0 \le i \le m$ and $0 \le j \le n$) under this general setting. Then $G[0, 0] = 0$, $G[0, j] = w_{del}(0, j)$ for $1 \le j \le n$, and $G[i, 0] = w_{ins}(0, i)$ for $1 \le i \le m$. Otherwise,

$$G[i, j] = \min \left\{ \begin{array}{c} G[i-1, j-1] + s(x_i,\ y_j), \\ \min_{0 \le q < j} \{\ G[i, q] + w_{del}(q, j)\ \}, \\ \min_{0 \le p < i} \{\ G[p, j] + w_{ins}(p, i)\ \} \end{array} \right\}. \tag{1}$$

Figures 2 and 3 show two different approaches for evaluating the entries of $G[1 : m, 1 : n]$. While the pull-based approach shown in Figure 2 is the standard way of evaluating Recurrence 1, the non-standard push-based approach shown in Figure 3 seems more suitable for a MapReduce implementation.

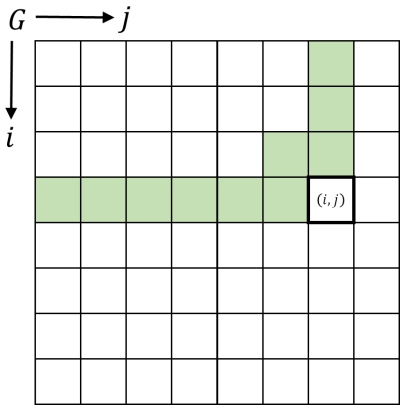

Figure 1: Dependency structure of the gap DP: cell $G[i, j]$ depends on the green cells.

```
LOOP-GAP-PULL( G, x, m, y, n ) {Inputs are sequences X = x_1 x_2 ... x_m and Y = y_1 y_2 ... y_n, and an (m+1) × (n+1)
                                matrix G[0 : m, 0 : n]. Row 0 and column 0 of G are assumed to be appropriately initialized.}
1.  for t ← 1 to m + n − 1 do
2.      for i ← max {1, t + 1 − n} to min {t, m} do
3.          j ← t + 1 − i
4.          G[i, j] ← G[i − 1, j − 1] + s(x_i, y_j)
5.          for q ← 0 to j − 1 do
6.              G[i, j] ← min {G[i, j], G[i, q] + w_ins(q, j)}
7.          for p ← 0 to i − 1 do
8.              G[i, j] ← min {G[i, j], G[p, j] + w_ins(p, i)}
9.  return G[m, n]
```

Figure 2: [PULL-BASED APPROACH] Looping code implementing Recurrence 1 in which each cell $G[i, j]$ pulls values from the cells it depends on.

```
LOOP-GAP-PUSH( G, x, m, y, n ) {Inputs are sequences X = x_1 x_2 ... x_m and Y = y_1 y_2 ... y_n, and an (m+1) × (n+1)
                                matrix G[0 : m, 0 : n]. Row 0 and column 0 of G are assumed to be appropriately initialized.}
1.  for i ← 1 to m do
2.      for j ← 1 to n do
3.          G[i, j] ← min {G[i, 0] + w_del(0, j), G[0, j] + w_ins(0, i)}
4.  for i ← 1 to m do
5.      G[i, 1] ← min {G[i, 1], G[i − 1, 0] + s(x_i, y_1)}
6.  for j ← 1 to n do
7.      G[1, j] ← min {G[1, j], G[0, j − 1] + s(x_1, y_j)}
8.  for t ← 1 to m + n − 1 do
9.      for i ← max {1, t + 1 − n} to min {t, m} do
10.         j ← t + 1 − i
11.         if i < m and j < n then G[i + 1, j + 1] ← min {G[i + 1, j + 1], G[i, j] + s(x_{i+1}, y_{j+1})}
12.         for q ← j + 1 to n do
13.             G[i, q] ← min {G[i, q], G[i, j] + w_ins(j, q)}
14.         for p ← i + 1 to m do
15.             G[p, j] ← min {G[p, j], G[i, j] + w_ins(i, p)}
16. return G[m, n]
```

Figure 3: [PUSH-BASED APPROACH] Looping code implementing Recurrence 1 in which each cell $G[i, j]$ pushes its finalized value to all cells that depend on it. This approach is better suited for MapReduce.

In the rest of the task we will assume $m = n$ for simplicity. We will also assume $w_{ins}(i, j) = j^2 - i^2$, $w_{del}(i, j) = j^3 - i^3$, and $s(x_i, y_j) = 0$ when $x_i = y_j$ and 1 otherwise.

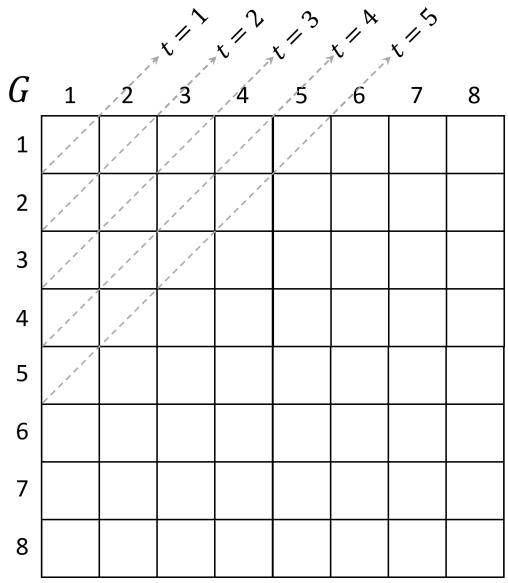

Figure 4: The dashed arrows show the cells that push their finalized values to other cells for various values of t in the approach given in Figure 3. For each value of t all cells to the right and below of the corresponding dashed arrow get updated due to the pushes.

(a) [**80 Points**] Design a MapReduce algorithm for solving Recurrence 1 based on the push-based approach given in Figure 3. Write pseudocode, and analyze its time and space complexities. Modify your algorithm to reduce space complexity.

(b) [**80 Points**] Implement your algorithm from part (a) on Hadoop. As a practical performance optimzation you may want to avoid executing the last few values of t as MapReduce rounds because the number of emits will be very low. Similarly for the first few values of t. Find the smallest value of n for which you are unable to allocate the entire matrix G on a single machine. Let n' be that value. Now use your Hadoop implementation for solving the problem for $n = n'$ and $n = 2n'$. Report running times using 10 data nodes.

Task 2. [90 Points] Floyd-Warshall's All-Pairs Shortest Path (APSP).

This task is an extension of Task 1 of Homework 1.

Consider a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by a real-valued length $l(v_i, v_j)$. If $(v_i, v_j) \notin E$, $l(v_i, v_j)$ is assumed to have a value $+\infty$. The length of a path is defined as the sum of the lengths of the edges in the path, taken in order. For each pair $v_i, v_j \in V$, $\delta[i, j]$ is defined to be the smallest of the lengths of all paths going from v_i to v_j. We assume that G does not have any negative length cycle.

Floyd-Warshall's algoithm for computing the length of the shortest path between every pair of vertices of G has a standard iterative implementation as shown in Figure 5. The input is an $n \times n$ matrix $D[1 \ldots n, 1 \ldots n]$ in which for every pair $i, j \in [1, n]$, $D[i, j]$ is initialized with $l(v_i, v_j)$. After the algorithm terminates $D[i, j] = \delta[i, j]$ for all $i, j \in [1, n]$.

```
LOOP-FLOYD-WARSHALL-APSP-PULL(D, n)
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

Figure 5: [PULL-BASED APPROACH] Looping code implementing Floyd-Warshall's APSP algorithm.

(a) [**90 Points**] Design a MapReduce algorithm for Floyd-Warshall's APSP shown in Figure 5. Write pseudocode, and analyze its time and space complexities. Modify your algorithm to reduce space complexity.

APPENDIX 1: What to Turn in

One compressed archive file (e.g., zip, tar.gz) containing the following items.

- Source code, makefiles and job scripts.
- A PDF document containing all answers and plots.