Stanford CS149: Parallel Computing
Written Assignment 4

Artificial Intelligence

Problem 1. (30 points):

Deep learning strikes again! Researchers have created a new DNN that takes as input a list of student comments on the CS149 website, and outputs a predicted score for the student on the final exam. Prof Kayvon and Prof Olukotun immediately decide to run this new AI algorithm on that logs from the class website, so they write the following Spark program: (assume ID is represents the type of a student ID)

```
// RDD 1: load lines of file into RDD
// assume lines are of the format: [student_id comment_text]
RDD<string> comments = inputFile.load(‘‘weblog.txt’’);

// RDD 2: parse lines into student id + comment pairs: (ID, comment text)
// parser: string --> (ID, string)
RDD<ID,string> parsed = comments.map(parser);

// RDD 3: remove short comments, like ‘‘thanks’’
// long_enough: string --> bool
RDD<ID,string> filtered = parsed.filter(long_enough);

// RDD 4: group comments by student
RDD<ID, list<string>> per_student = filtered.group_by_key();

// RDD 5: turn each student’s list of comments into a Exam 2 prediction score
// prediction_ai: list<string> --> int
RDD<ID, int> feedback = per_student.map( prediction_ai );

// RDD 6: sort elements of the RDD by key (by student id).
// Assume the implementation is quicksort
RDD<ID, int> sorted = feedback.sort();

// save sorted sequence to a single continuous file on disk
sorted.save(‘‘results.txt’’);
```

Consider an implementation of Spark that attempts to be *as memory efficient as possible*. Please list which of the six RDDs must be fully materialized in memory to ensure correct execution of the program. For each RDD in your list, please give the reason why.

Hash Table Parallelization

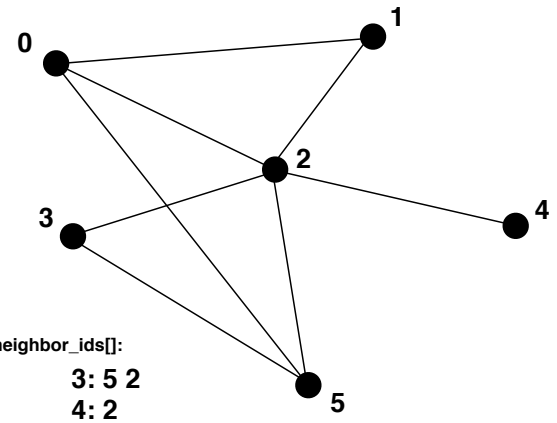
Problem 2. (40 points):

Below you will find an implementation of a hash table (a linked list per bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

```
struct Node {
  string value;
  Node* next;

};

struct HashTable {
   Node* bins[NUM_BINS];  // each bin is a singly-linked list


};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);
   bool result = false;




   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {



     insertToList(table->bins[idx1], s1);



     insertToList(table->bins[idx2], s2);



     result = true;
   }




   return result;
}
```

Updating a Graph

Problem 3. (30 points):




```
struct Graph_node {
   Lock   lock;
   float  value;
   int    num_edges;    // number of edges connecting to node (its degree)
   int*   neighbor_ids; // array of indices of adjacent nodes
};

// a graph is a list of nodes, just like in assignment 3
Graph_node graph[MAX_NODES];
```

Consider the undirected graph representation shown in the code above. The figure shows an example graph. In the bottom-left of the figure are the values in `neighbor_ids` for each node.

Your boss asks you to write a program that atomically updates each graph node's `value` field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows...

```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);

    // now perform computation...
```

Question on the next page...

You run your update code in parallel on nodes 0 and 2 and deadlock occurs. Please provide a solution (a sketch in pseudocode is fine) to the deadlock problem that maintains high concurrency and limits the amount of extra work that is performed in the `update` function. **Hint: your solution may involve preprocessing of the graph data structure (if so, describe it).**