**(a)** Since nodes without incoming edges are excluded, we actually only have to consider the target nodes and the weight associated with each of them. Using the provided example graph (shown in Table 1 *Map* Input), the MapReduce procedure is described as follows.

The *Mapper* function is currently implemented by using the *map* method, which processes the input graph line by line. By using the Java built-in method *StringTokenizer* and specifying the delimiter (i.e., tab "\t"), the function splits each line into separated tokens. The first token which represents the source node is simply omitted, while the last two tokens for the target node and weight are processed and stored as a key-value pair. The input and output of the *Mapper* function are shown below in the Table 1. Specifically, the input is a set of strings containing source/target nodes and weight, and output is a set of intermediate key/value pairs which are sorted by key.

**Table 1**

| *Map* Input | *Map* Output | |
|---|---|---|
| String set | key | value |
| "100 10 3" | 10 | 3 |
| "110 10 3" | 10 | 3 |
| "200 10 1" | 10 | 1 |
| "150 130 30" | 101 | 15 |
| "110 130 67" | 130 | 30 |
| "10 101 15" | 130 | 67 |

**Table 2**

| *Reduce* Input | | *Reduce* Output | |
|---|---|---|---|
| key | value | key | value |
| 10 | 3 | 10 | 1 |
| | 3 | | |
| | 1 | | |
| 101 | 15 | 101 | 15 |
| 130 | 30 | 130 | 30 |
| | 67 | | |

The *Reducer* function is implemented via the *reduce* method. The input for the *Reducer* is a group of {key, (list of values)} pairs, each of which contains a list of intermediate values which share the same key (e.g., {10, (3, 3, 1)}). In the current implementation, the function loops over all the values for each key and finds the smallest one among these values. The output is a set of key/value pairs. Table 2 shows the input and output of the *Reducer* function for the given example.

**(b)** The algorithm is proposed based on the following idea: we first map over the whole collection of records and extract the *Department_ID* (i.e., key for join) as the intermediate key, and lump together the remaining attributes in a record as a tuple which will be the value to that intermediate key. As Hadoop guarantees that all values with an equal key are grouped together and sent to the same *Reducer*, all the tuples associated with a join key will be gathered. Also, the intermediate key/value pairs outputted by the *Mapper* function are sorted by key, hence it is not necessary to post-process the output records regarding the orders. After we have a group of {key, (list of tuples)} pairs (as we assume there is enough space to store the data), we can simply search the tuple list for department and student names, and output records with the specified format. A pseudo-code for this algorithm is provided below.

**Table 3**

| *Map* Input | *Map* Output | |
|---|---|---|
| String set | key | value |
| "Student, Alice, 1234" | 1123 | (Student, Carol ) |
| "Student, Bob, 1234" | 1123 | (Department, CSE) |
| "Department, 1123, CSE" | 1234 | (Student, Alice) |
| "Department, 1234, CS" | 1234 | (Student, Bob) |
| "Student, Carol, 1123" | 1234 | ( Department, CS) |

```
Given: An unordered collection of records
Function Mapper(args)
    Initialize variables tokens, dept_id, rest_attr
    Function map(value, collector, args)
        tokens = value.toString().split(",")
        if tokens[0] == "Student"
            dept_id = tokens[2].toInteger()
            rest_attr = (tokens[0], tokens[1])
        else if tokens[0] == "Department"
            dept_id = tokens[1].toInteger()
            rest_attr = (tokens[0], tokens[2])
        collector.collect(dept_id, rest_attr)

Function Reducer(args)
    Initialize variables dept
    Function reduce(key, values, collector, args)
        for (tuple tpl : values)
            if tpl[0] == "Department"
                dept = tpl[1]
                break
        for (tuple tpl : values)
            if tpl[0] == "Student"
                collector.collect(key, (tpl[1], dept))

Function main(args)
    Set job configuration
    Pass Mapper implementation to job
    (Pass Combiner implementation to job)
    Pass Reducer implementation to job
    Pass output implementation to job
    Specify file input format
    Specify file output format
```

The input and output of the *Mapper* function are shown in Table 3. The input is a set of strings which contains the unordered collection of records. For each record, the *Mapper* function extracts the *Department_ID* as the key and arranges the rest of information into a tuple. The output is then a set of intermediate key/value pairs which are sorted by key, where the key is *Department_ID*, and the value for each key is a tuple of the form (*Department*, *Department_Name*) or (*Student*, *name*).

The input for the *Reducer* is a group of {key, (list of tuples)} pairs. The list is a collection of tuples which share the same key (e.g., [(*Student*, *Carol*), (*Department*, *CSE*)]). For each key, the *Reducer* function searches along the list of tuples and finds the name of department first. Then a second scan is performed to gather the key, student name and department name into a new key/value pair, which has a form of (*Department_ID*, (*name*, *Department_Name*)). The output is a group of these key/value pairs. The input and output of the *Reducer* function are summarized in Table 4.

**Table 4**

| *Reducer* Input | | *Reducer* Output | |
|---|---|---|---|
| key | values | key | values |
| 1123 | (Student, Carol ) | 1123 | (Carol, CSE) |
| | (Department, CSE) | | |
| 1234 | (Student, Alice) | 1234 | (Alice, CS) |
| | (Student, Bob) | 1234 | (Bob, CS) |
| | ( Department, CS) | | |