그래프 정제 및 삼각형 수 세기

이름 : 임현진 학번 : 20181684

Task 1. 그래프에서 중복된 간선과 loop를 모두 제거하는 기능을 MapReduce로 구현하기

Task1의 문제는 그래프에서 중복으로 존재하는 간선과 loop을 제거하는 것입니다. 여기서 말하는 중복과 loop은 예를 들어서,

0 1 / 1 0 , 100 200 / 200 100 처럼 앞 뒤만 바껴서 나오는 경우를 중복으로 간주하고,

0 0, 1 1, 3 3 과 같이 앞 뒤가 같은 경우를 loop이라고 생각 할 수 있습니다.

그렇기 때문에 이러한 중복과 loop을 제거하기 위해서 MapReduce작업을 실행하려고 합니다.

<mapper>

우선 mapper에서는 원래 데이터에 존재하는 간선들을 <작은거, 큰거>순으로 나열해주는 작업을 해줍니다. 이를 통해서 데이터에 존재했던 loop을 제거 할 수 있었습니다.

<작은거, 큰거>순으로 나열해주는 방법은 우선 데이터의 값을 읽어와서 StringTokenizer로 앞의 값과 뒤의 값을 나눠서 읽은 후, 각 각을 u, v라는 변수에 저장합니다.

만약 u<v 라면, 그대로 저장해서 reduce로 넘겨 주고 u>v의 라면, u와 v의 값의 위치를 바꿔서 저장한 후 reduce로 값을 넘겨줍니다.

```
public class TriMapper extends Mapper<Object, Text, IntWritable, IntWritable> {

	IntWritable ok = new IntWritable();
	IntWritable ov = new IntWritable();

	@Override
	protected void map(Object key, Text value, Mapper<Object, Text,
IntWritable, IntWritable>.Context context)
			throws IOException, InterruptedException {

		StringTokenizer st = new StringTokenizer(value.toString());

		int u = Integer.parseInt(st.nextToken());
		int v = Integer.parseInt(st.nextToken());
		if (u < v) {
			ok.set(u);
			ov.set(v);
			context.write(ok, ov);
		}
		else if (u > v) {
			ok.set(v);
			ov.set(u);
			context.write(ok, ov);
		}
	}
}
```

<Reduce>

reduce에서는 중복으로 존재하는 간선들을 제거할 수 있도록 했습니다. 우선 key값에 따라서 value값들을 Iterable을 사용하여 모두 읽어옵니다. 그렇게 되면 같은 key값을 가지기 때문에 서로 다른 value값을 가져야 중복이 발생하지 않는다는 것을 알 수 있습니다. 이를 이용하여 똑같은 값이 두번 나오게 되면 그냥 무시하고 넘어가고 처음 나오는 경우에는 neighbors라는 ArrayList에 저장해주었습니다. 여기서 두번 나오게 되는 경우를 아는 방법은 contains라는 함수를 사용하였습니다. contains라는 함수는 ArrayList에 값이 존재하면 true, 값이 존재하지 않으면 false를 반환해 줍니다.

그리고 ArrayList에 저장된 만큼 다시 반복문을 사용하여 <key, value>값으로 저장하여 중복과 loop이 제거된 데이터를 만들었습니다.

```
public class TriReducer extends Reducer<IntWritable,
IntWritable,IntWritable,IntWritable>{
	IntWritable ok = new IntWritable();
	IntWritable ov = new IntWritable();

	@Override
	protected void reduce(IntWritable key, Iterable<IntWritable> values,
			Reducer<IntWritable, IntWritable, IntWritable,
IntWritable>.Context context) throws IOException, InterruptedException {

		ArrayList<Integer> neighbors = new ArrayList<Integer>();

		//만약 값이 존재한다면 continue, 그렇지 않다면 저장
		for(IntWritable v : values) {
			if(neighbors.contains(v.get()))
			{
				continue;
			}
			neighbors.add(v.get());
		}

		ok.set(key.get());
		for(int u : neighbors)
		{
			if (key.get() < u )
			{
				ov.set(u);
				context.write(ok, ov);
			}
		}
	}
}
```

결과 값 : Task1으로 만든 데이터

(값이 너무 많아서 일부만 가져왔습니다.)

0	10772
1	598775
1	170193
1	2
1	471455
3	1101828
3	1101924
3	1100919
3	1101709
3	1101827
3	1102561
3	1101945
3	1101808
3	1101724
3	1102853
4	1102708
4	1102701
4	1101498
4	1101709
4	1101894
4	1103745
5	1101894
5	1101550
5	1101324
5	1101709
5	1102159
5	1101987
5	1101498
6	1101660
6	1102234
7	1102709
7	1103797
8	1101498
8	1103797
8	1102709
8	1102155
8	1101610
8	1101555
8	1101548
8	1102847
9	1102709
9	1101496
9	10
9	1103943
10	1102021

Task 2. 간선 (u, v)에 대해서 다음 조건에 따라 u와 v의 순서를 변경

- degree(u) < degree(v) 이거나, degree(u) == degree(v) 이면서 id(u) < id(v) 인 경우 → 그대로 둠 (즉, (u, v)를 출력) 그 밖의 경우에는 u와 v의 순서를 바꿈 (즉, (v, u)를 출력)

Task2의 경우에는, Task1의 결과를 통해 만들어진 data로 이제 Task2를 진행합니다.
Task2에서는 각 node와 연결된 간선의 수를 먼저 세어줍니다. 이것을 계산하기 위해서는 총 2단계를 거쳐야합니다. 그렇게 하는 이유는 노드 u,v가 있을 때, 각 각의 degree정보를 구해서 비교하는게 한번에 되지 않기 때문에 2단계로 나누는 것입니다.

우선 첫번째 단계에서는 노드 a, b,가 있을 때, key를 a로 하고 value를 (a,b)로 한 값과 key를 b로 하고 value를 (a,b)로 한 값을 만듭니다. 그러고나서 reduce에서 key값에 해당하는 간선의 수를 모두 세어줍니다. 이때 key값이 a이고 value값이 (a,b)이면 key값을 (a,b)로 하고 value값을(a의 degree수, " ")로 output을 만들고, key값이 b이고 value값이 (a,b)이면 key값을 (a,b)로 하고 value값을 (" ", b의 degree수)로 output을 만듭니다.

두번째 단계에서는 key값이 (a,b) 이고 value = (a의 degree수, " ")와 value = (" ",b의 degree의 수)를 합쳐서 key = (a,b), value = (a의 degree수, b의 degree수)로 data를 만들어서 위의 조건에 해당하도록 u,v의 값을 만들어줍니다.
각 각의 단계에 맞는 mapper와 reduce파일을 통해서 좀 더 자세히 설명하도록 하겠습니다.

<mapper – 첫번째 단계>

Mapper에서는 Task1에 의해서 만들어진 데이터를 가지고 우선 StringTokenizer를 통해서 서로 연결된 노드의 값을 각 각 u와 v라는 변수에 저장합니다. 그리고 나서 key = u로 하고 value = (u,v)인 값과 key = v이고 value = (u,v)인 값을 reduce로 넘겨줍니다.

```
public class TriMapper1 extends Mapper<Object, Text, IntWritable, Text> {

	IntWritable ok = new IntWritable();
	Text ov = new Text();

	@Override
	protected void map(Object key, Text value, Mapper<Object, Text,
IntWritable, Text>.Context context)
			throws IOException, InterruptedException {

		StringTokenizer st = new StringTokenizer(value.toString());

		int u =  Integer.parseInt(st.nextToken());
		int v =  Integer.parseInt(st.nextToken());
		ok.set(u);
		ov.set(u+" "+v);
		context.write(ok,ov);

		ok.set(v);
		ov.set(u+" "+v);
```

```
                context.write(ok,ov);
        }
}
```

<reducer – 첫번째 단계>

Reduce파일에서는 mapper를 통해서 받은 key = u, value=(u,v)인 값과 key = v, value=(u,v)인 값에 대해서 각 각의 key에 해당하는 값이 data에 몇 개 존재하는지 세기 위해서, 우선 key값에 따라서 존재하는 (u,v)의 값을 Iterable하게 하여 찾아줍니다. 그리고 찾은 (u,v)의 값을 StringTokenizer을 통해서 합쳐져 있던 (u,v)를 u, v로 쪼개줍니다.

그러고 나서 key값이 u노드랑 같은지 v노드랑 같은지 비교해줍니다. 이때, neighbors라는 ArrayList타입 변수, check라는 hashmap타입 변수 그리고 map이라는 hashmap타입 변수를 사용했습니다. neighbors라는 변수는 key=u일 때, v값을 저장하고 key=v일 때 u값을 저장합니다. check라는 변수는 neighbors라는 변수에 저장된 값이 u인지 v인지 식별하기 위해서, v라면 0을 넣고, u라면 1을 넣어서 값을 구분할 수 있도록 해줬습니다. 마지막으로 map이라는 변수는 해당 key값이 u또는 v와 일치 할 때, 값이 처음 나왔다면 1을 저장해주고 처음나온게 아니라면 원래 가지고 있던값에 1을 더해서 저장해줍니다.

마지막으로 neighbors변수가 가지고 있는 값의 수만큼 for문을 돌려서 neighobrs안에 들어있는 값을 check라는 변수에 넣어서 만약 값이 0이라면 이 노드는 key = u 이기 때문에 key=(u,v), value=(u의 갯수, "-1")로 하고 만약 check라는 변수에 넣었는데 값이 1이라면 이 노드는 key = v 이기 때문에 key=(u,v), value=("-1",v의 갯수)로 하여 output으로 내보내 줍니다.

여기서 "–1"로 한 이유는 큰 이유는 없지만, 나중에 stringtokenizer를 통해서 좀 더 값을 쉽게 뽑기위해서 "-1"로 설정하였습니다.

```
public class TriReducer1 extends Reducer<IntWritable,Text, Text, Text>{
        Text ok = new Text();
        Text ov = new Text();
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();

        @Override
        protected void reduce(IntWritable key, Iterable<Text> values,
                        Reducer<IntWritable, Text, Text, Text>.Context context)
throws IOException, InterruptedException {

                int k = key.get();
                ArrayList<Integer> neighbors = new ArrayList<Integer>();
                Map<Integer, Integer> check = new HashMap<Integer, Integer>();

                for(Text t : values)
        {
            StringTokenizer st = new StringTokenizer(t.toString()," ");
            int u = Integer.parseInt(st.nextToken());
            int v = Integer.parseInt(st.nextToken());

            if(k == u)
            {
                neighbors.add(v);
                check.put(v,0);
                if(map.containsKey(u))
```

```
                {
                        int num = map.get(u);
                        map.put(u, num+1);
                }
                else
                {
                        map.put(u,1);
                }
            }
            else if (k == v)
            {
                neighbors.add(u);
                check.put(u,1);
                if(map.containsKey(v))
                {
                        int num = map.get(v);
                        map.put(v, num+1);
                }
                else
                {
                        map.put(v,1);
                }
            }
        }
                for(int u : neighbors)
                {
                        if(check.get(u)==0)
                        {
                                ok.set(k+","+u);
                                ov.set(map.get(k)+","+"-1");
                        }
                        else if(check.get(u)==1)
                        {
                                ok.set(u+","+k);
                                ov.set("-1"+","+map.get(k));
                        }
                        context.write(ok,ov);
                }
        }
}
```

결과 값 : 결과값이 너무 커서 열어볼수가 없었습니다...

<mapper – 두번째 단계>

mapper에서는 첫번째 단계에서 만든 데이터를 그대로 읽어와서 reduce로 넘겨줍니다.

```
public class TriMapper2 extends Mapper<Object, Text, Text, Text> {

	Text ok = new Text();
	Text ov = new Text();

	@Override
	protected void map(Object key, Text value, Mapper<Object, Text, Text,
Text>.Context context)
			throws IOException, InterruptedException {
		StringTokenizer st = new StringTokenizer(value.toString());
		String k = st.nextToken();
		String v = st.nextToken();
		ok.set(k);
		ov.set(v);
		context.write(ok,ov);

	}
}
```

<Reducer – 두번째 단계>

Reducer에서는 key=(u,v)에 따라서 value=("u의 개수", "-1")와 value=("-1", "v의 개수")를 통해서 각 각, u의 개수와 v의 개수를 구해줍니다. 이 개수는 각 각의 degree수를 의미합니다. 이때 value값을 stringtokenizer를 통해서 u와 v로 나눴을 때, -1이 아닌 값이 실제 존재하는 개수 라는 것을 알 수 있습니다. 이렇게 u와 v의 개수를 각 각 구했다면,

조건)

degree(u) < degree(v) 이거나, degree(u) == degree(v) 이면서 id(u) < id(v) 인 경우 → 그대로 둠

그 밖의 경우에는 u와 v의 순서를 바꿈 (즉, (v, u)를 출력)

다음과 같은 조건을 적용시켜서 u와 v값의 순서를 정하여 output으로 내보내줍니다.

```
public class TriReducer2 extends Reducer<Text,Text, Text, Text>{
	Text ok = new Text();
	Text ov = new Text();

	@Override
	protected void reduce(Text key, Iterable<Text> values,
			Reducer<Text, Text, Text, Text>.Context context) throws
IOException, InterruptedException {


		int u1=-1;
		int v1=-1;

		for(Text t : values)
	{
	    StringTokenizer st = new StringTokenizer(t.toString(),",");
	    int u = Integer.parseInt(st.nextToken());
	    int v = Integer.parseInt(st.nextToken());
```

```
            if(u==-1)
            {
                v1 = v;
            }
            else if(v==-1)
            {
                u1 = u;
            }
        }
            StringTokenizer s = new StringTokenizer(key.toString(),",");
            int k = Integer.parseInt(s.nextToken());
            int v = Integer.parseInt(s.nextToken());
            if(u1<v1 || (u1==v1 && k<v))
            {
                    ok.set(""+k);
                    ov.set(""+v);
            }
            else
            {
                    ok.set(""+v);
                    ov.set(""+k);
            }
            context.write(ok,ov);
        }
}
```

결과값 : **: Task2로 만든 데이터**

(값이 너무 많아서 일부만 가져왔습니다.)

0	10772
1	170193
2	1
1	471455
1	598775
10	1101387
10	1101491
10	1102021
10	1102159
10	1105121
100	1167999
100	1168749
100	433013
100	433384
100	433392
100	649490
1000	12545
1000	1291934
1000	1660538
1000	6214
10976	10000
109985	10000
10000	1233699
10000	1233949
1292532	10000
10000	1650933
10000	727234
10000	842034
10000	842200
10000	847261
890098	10000
100000	100001
100000	1108125
1000002	1000000
1000000	1000015
1000000	1000034
1000040	1000000
1000000	1000056
1000000	1184026
1000000	1184369
1000000	1184562
1000000	1184622
1000002	1000001
1000001	1000004
1000017	1000001
1000001	1026895
1000001	1060102
1000001	1174785
1000001	1177913
1000001	1178389
1000001	1179890

*여기서 task1을 통해서 만든 결과의 개수와 task2를 통해서 만든 결과의 개수가 같은걸로 보아서 결과가 잘 만들어졌다는 것을 알 수 있을 것 같습니다.

Task 3. 삼각형 MapReduce 알고리즘에 1) Task 1의 결과 그래프와 2) Task 2의 결과 그래프를 각각 입력했을 때, 성능 비교하기

(기존의 삼각형 mapreduce 알고리즘에서 수정한 파일만 설명하도록 하겠습니다.)

<TriStep1Mapper >

우선 TriStep1Mapper에서 노드의 크기를 비교하여 u와 v를 바꾸는 부분이 있었는데 이미 task1과 task2데이터 이러한 과정을 거쳤기 때문에 필요없어서 지워줬습니다.

```
public class TriStep1Mapper extends Mapper<Object, Text, IntWritable, IntWritable>
{

	IntWritable ok = new IntWritable();
	IntWritable ov = new IntWritable();

	@Override
	protected void map(Object key, Text value, Mapper<Object, Text,
IntWritable, IntWritable>.Context context)
			throws IOException, InterruptedException {

		StringTokenizer st = new StringTokenizer(value.toString());

		int u = Integer.parseInt(st.nextToken());
		int v = Integer.parseInt(st.nextToken());

		ok.set(u);
		ov.set(v);

		context.write(ok, ov);
		}

}
```

<TriStep2MapperForEdges>

edge와 wedge를 조인할 때, wedge와 edge가 일치하도록 하기 위해서 edge부분에서 u<v가 되도록 코드를 수정해줬습니다.

```
public class TriStep2MapperForEdges extends Mapper<Object, Text, IntPairWritable,
IntWritable>{

	IntPairWritable ok = new IntPairWritable();
	IntWritable ov = new IntWritable(-1);
	@Override
	protected void map(Object key, Text value, Mapper<Object, Text,
IntPairWritable, IntWritable>.Context context)
			throws IOException, InterruptedException {

		StringTokenizer st = new StringTokenizer(value.toString());
		int u = Integer.parseInt(st.nextToken());
		int v = Integer.parseInt(st.nextToken());
		if (u < v) {
			ok.set(u,v);
			context.write(ok, ov);
		}
		else if (u > v) {
			ok.set(v,u);
```

```
                        context.write(ok, ov);
                }
        }
}
```

<TriStep2MapperForWedges>

edge와 wedge를 조인할 때, wedge와 edge가 일치하도록 하기 위해서 wedge부분에서도 u<v가 되도록 코드를 수정해줬습니다.

```
public class TriStep2MapperForWedges extends Mapper<IntPairWritable, IntWritable,
IntPairWritable, IntWritable>{

        IntPairWritable ok = new IntPairWritable();
        @Override
        protected void map(IntPairWritable key, IntWritable value,
Mapper<IntPairWritable, IntWritable, IntPairWritable, IntWritable>.Context
context)
                        throws IOException, InterruptedException {
                StringTokenizer st = new StringTokenizer(key.toString());
                int u = Integer.parseInt(st.nextToken());
                int v = Integer.parseInt(st.nextToken());
                if (u < v) {
                        ok.set(u,v);
                        context.write(ok, value);
                }
                else if (u > v) {
                        ok.set(v,u);
                        context.write(ok, value);
                }
        }
}
```

Wiki-topcats로 비교를 하려고 했지만, 데이터가 너무 커서 디스크 용량을 자꾸 초과하여 잘안되 가지구 snap.standford.edu에 존재하는 다른 데이터 셋을 이용하여 비교를 해보았습니다.

사용한 데이터 : com-amazon.ungraph.txt

1. **Wedge의 수 비교하기**

(conf값을 3으로 하여 파일을 3개로 나누어서 결과값이 나오도록 했습니다.)

Task1의 결과로 삼각형 mapreduce 알고리즘을 돌리면 총 3개의 파일이 나옵니다.

파일의 각 각의 크기가, 20,361KB, 20,717KB, 21,645KB이고

파일의 각 각의 개수가1042230개, 1048576+a, 1048576+a입니다. 따라서 총 개수가 3,139,382+a 개가 나온다고 볼 수 있습니다.

Task2의 결과로 삼각형 mapreduce 알고리즘을 돌려도 총 3개의 파일이 나옵니다.

파일의 각 각의 크기가 7,431KB, 7,390KB, 7,424KB이고

파일의 각 각의 개수가 390388개, 378257개, 380017개 입니다. 따라서 총 개수가 1,138,662개가 나온다고 볼 수 있습니다.

Task1과 Task2의 결과로 wedge를 만들면 Task2의 결과를 사용하여 만든 wedge의 수가 더 작다는 것을 알 수 있었습니다.
=> 3,139,382+a > 1,138,662 (거의 3배 가까이 차이가 납니다....!!)

이렇게 wedge의 수가 줄었다고 삼각형의 수가 줄어들었을까요? 그렇지 않습니다.
Task1의 결과로 만들어진 파일의 크기는 각 각, 4,439KB, 4,423KB, 4,421KB였고 데이터의 개수는 각 각, 222,947개, 222,138개, 222,044개 였습니다. 이걸 다 합하면 삼각형의 개수가 667,129개 라라는 것을 알 수 있습니다.
Task2의 결과로 만들어진 파일의 크기는 각 각, 4,448KB, 4,420KB, 4,415KB였고 데이터의 개수는 각 각, 223,417개, 221,988개, 221,724개 였습니다. 이걸 다 합하면 삼각형의 개수가 667,129개 라라는 것을 알 수 있습니다.
이처럼 만들어진 3개의 파일의 크기가 다를수는 있지만, 모두 합하면 삼각형의 개수가 같다는 것을 알 수 있습니다.
=>667,129 = 667,129

2. 실행시간 비교하기

cm-amazon.ungraph.txt파일의 크기가 그렇게 크지 않아서 실행 시간이 5초~10초 정도의 차이가 밖에 나지 않았지만, 그래도 task1의 결과로 삼각형 mapreduce 알고리즘을 돌리는 것보다 task2의 결과로 삼각형 mapreduce 알고리즘을 돌리는게 항상 일찍 끝났습니다. 왜그런가 생각을 해보니깐 wedge를 만드는 부분에서 굉장히 차이가 많이나서 그렇다는 것을 알 수 있습니다.
지금은 데이터가 작은데도 3배 가까이가 차이났는데, 데이터가 이보다 훨씬 크다면 차이가 더 크게 날거라고 생각합니다.
그렇다면 왜 wedge를 만드는데서 차이가 나는지 생각해봤습니다. 자세히 들여다보니 Task1의 경우에는 wedge를 만들 때, undirect그래프로 노드 자신과 연결된 모든 간선을 통해서 wedge를 만듭니다. 그렇기 때문에 for문이 굉장히 오래도록 돈다는 것을 알 수 있습니다.
하지만 Task2의 경우에는 wedge를 만들 때, direct그래프로 이미 방향성이 정해져 있고, degree수와 id의 크기에 따라서 데이터가 전처리 되어 있기 때문에 노드 자신과 연결된 모든 간선을 통해서 wedge를 만들지 않습니다. 그렇기 때문에 for문이 task1에 비해서 굉장히 적게 돈다는 것을 알 수 있습니다.
이처럼 task1과 task2에서 wedge를 만들 때, 반복되는 for문의 차이 때문에 task1에 비해서 task2의 실행시간이 더 빠르게 끝난다는 결과를 도출 할 수 있었습니다.

이를 바탕으로 데이터가 큰 wiki-topcats에서 task1을 통해서 만들 때는 disk용량이 부족하여 만들지 못하지만 task2를 만들때는 disk용량이 부족하지 않아서 만들수 있는 점을 설명할 수 있습니다. 그이유는 task1을 만들때는 wedge를 만들 때, 연결된 모든 간선에 대해서 wedge를 만들기

때문에 for문이 굉장히 많이 돌게되고 그만큼 데이터가 만들어지기 때문에 Disk용량 부족 문제가 발생합니다. 하지만 task2에서는 task1에 비해서 훨씬 적게 for문을 사용하고 그렇기 때문에 훨씬 적게 wedge만들어서 디스크 용량 문제가 발생하지 않고, task1에 비해서 빠르게 작업을 끝낼 수 있게 됩니다.
(저는 task2도 디스크 용량 문제로 못했습니다...하지만 결과가 이렇다고 설명은 할 수 있을 것 같습니다.)

느낀점.

Mapreduce를 이용하여 서울 공기질 분석하기 과제는 쉬운 과제였구나라는 것을 이번 과제를 하면서 깨달았습니다. 또한 삼각형 만들기가 굉장히 단순한 문제인줄 알았는데, 이렇게 다양한 접근을 통해서 만들 수 있다는 점이 놀라웠습니다.
이것 말고도 다양한 삼각형 만들기 최적화 방법이 존재하는 것 같은데 기회가 된다면 어떤 방법이 또 존재하는지 배워보고 싶습니다. 이번 과제를 통해서 mapreduce에 대해서 더 알게 된 것 같아서 기분이 좋습니다.