HzGraphFlow 系统的实现

1 引言

在论文《GraphFlow:基于状态更新的动态图计算模型》(以下简称 GraphFlow)中，我们设计了一种新的面向流式数据的图计算模型，并且借助于 Flink 框架实现了 DD 和 TC 算法，但是发现，当实现更为复杂的 SSSP 和 PR 算法时，却无从下手，究其原因是 Flink 并没有提供灵活的分布式数据结构来存储图的状态。而之所以 DD 算法和 TC 算法能够在 Flink 上实现，是因为这两种算法都是局部算法,而且不需要进行迭代计算，因此很容易通过分流的方式存储局部计算结果，并且进行局部计算即可。而 SSSP 和 PR 算法是要在整个连通子图内多次迭代的进行计算，如果没有分布式数据结构来存储每个节点的状态，很难在整个连通子图内共享信息。而 Hazelcast 提供了这样的数据结构，因此本文希望通过在 Hazelcast 上构建整套模型，并且实现该四种算法。论文 GraphFlow 已经详细阐述了整个框架的设计理念，因此本文不再具体阐述，只是重点讲述如何实现这套框架及相关的动态图算法。

2 架构实现

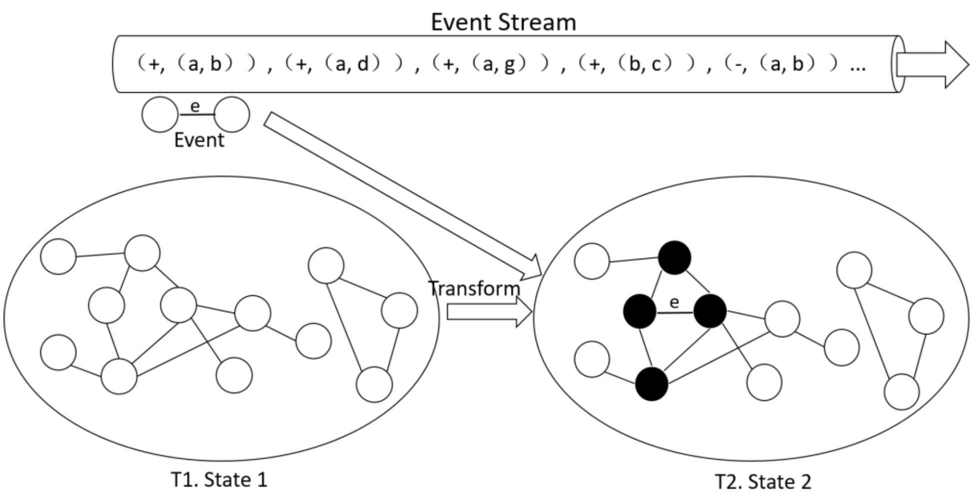


图 1 基于状态更新的动态图计算模型

如图 1 所示, 定义了 GraphFlow 的动态图计算模型的运行过程 :系统每次从 Event Stream 中读取一个 Event(如增加一条边这样的事件),利用该事件和图的原始状态 State1，在用户自定义的 Transform 函数的驱动下，转变为另一个状态 State2。因此需要实现该模型的这三个基本组件，并且在该组件之上定义图的算法。整个系统的架构如下图所示：

应用层	Application	
服务层	Library	
核心层	Graph	Components
引擎层	Hazelcast Engine	

- **Application**：面向用户的上层运用，这些运用涵盖了典型的使用场景，例如链接分析、欺诈检测、社区发现等，是针对某个具体问题的具体应用；
- **Library**：框架提供给用户使用的丰富的库函数和图算法，诸如 Degree Distribution, Triangle Count, Single Source Shortest Path, PageRank 等算法包都会在该层中实现；
- **Graph & Components**：提供了图的基本定义和组件的基本定义。该层是系统的核心层，也是模型的实现层，用户可以实现该层定义的接口来实现自定义的图算法。
- **Engine**：最底层的具体的引擎，本文使用 Hazelcast 这样的分布式数据结构框架作为整个系统的底层存储引擎。

3 算法实现

3.1 Degree Distribution

节点的度分布算法，是用来统计无向图中各个节点的度。如图 2.所示，图中数字表示各个节点的度，当增加一条新边时，将这条边的两个顶点的度各加 1 即可。

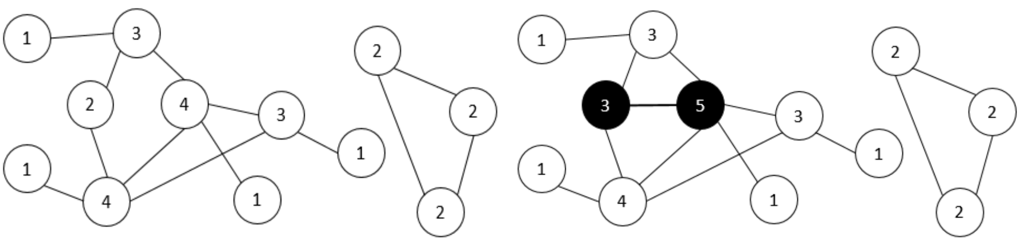


图 2 DD Algorithm

核心代码如下表。

DD algorithm

```
public boolean increase(KV id){
    if(state.containsKey(id)){
        state.lock(id);
        set(id,state.get(id)+1);
        state.unlock(id);
    }else
        set(id,1L);
```

```
    return true;
}

public boolean decrease(KV id){
    if(state.containsKey(id)){
        state.lock(id);
        long count = state.get(id);
        if(count > 0) set(id,count-1);
        else return false;
        state.unlock(id);
    }
    return false;
}
```

3.2 Triangle Count

TC 算法是用来统计无向图中不同三角形的数目。如图 3 所示，图中节点编号表示节点拥有三角形的数目。当增加一条边时，找出这条边的两个顶点的公共邻接点，即为新增的三角形的数目。

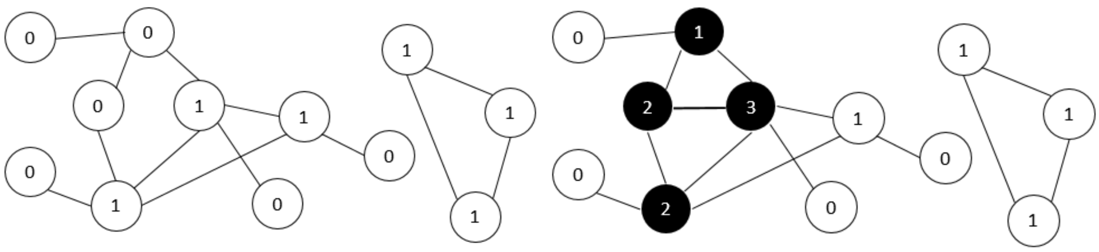


图 3 TC Algorithm

核心代码如下所示:

TC Algorithm

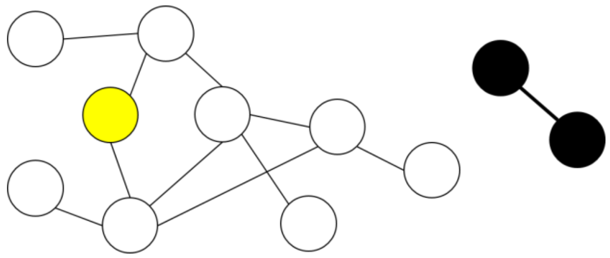
```
public boolean update(EdgeEvent<KV,EV> event) {
    EventType type = event.getType();
    Edge<KV,EV> edge = event.getValue();
    switch(type){
        case ADD:
            KV source = edge.getSource();
            KV target = edge .getTarget();

            outNeighborState.update(event);
            outNeighborState.update(new EdgeEvent<KV,
EV>(type,edge.reverse()));

            outNeighborState.lockKey(source);
```

```
outNeighborState.lockKey(target);

            Set<KV> sn = outNeighborState.get(source);
            Set<KV> tn = outNeighborState.get(target);

            int increased = 0;
            if(sn.size() < tn.size()){
                for(KV vertex : sn)
                    if(tn.contains(vertex)) increased++;
            }else{
                for(KV vertex : tn)
                    if(sn.contains(vertex)) increased++;
            }
            counter.addAndGet(increased);

            outNeighborState.unlockKey(source);
outNeighborState.unlockKey(target);
            return true;
        default:
            return false;
    }
}
```

3.3 Single Source Shortest Path

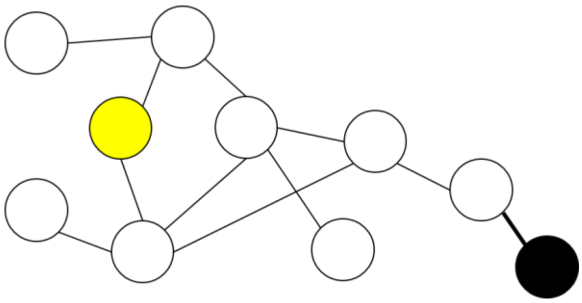
单源点最短路径算法，是在有向图中，给定一个源点，就该源点到图中其他各点的最短路径。下图中黄色顶点为给定的源点，白色顶点为再处理新增边之前已经存在而且处理好的顶点，黑色顶点和边为新增的顶点。当增加一条边时，这条边有三种可能：

a. 这条边的两个顶点都是最新出现的



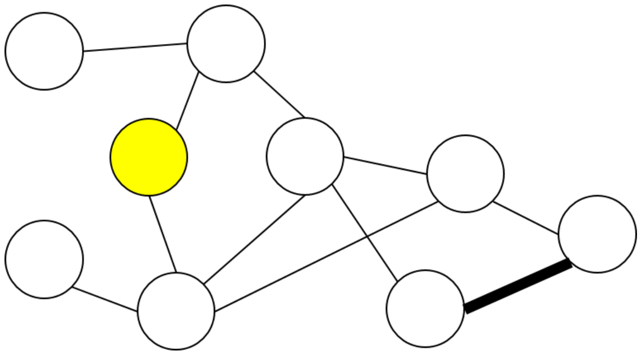
因为新增的这条边的两个顶点都是最新出现的，因此原图中的任何顶点都无法与之建立连接，即这两个顶点是不可达的。所以他们的 SSSP 值为无穷大。

b. 这条边的两个顶点有一个是最新出现的



假设新增的这条边为（v1,v2,distance），如果 v1 是原图中已经有的，v2 是新增加的节点，v1 指向 v2,则 v2 的 SSSP 值为 v1+distance；反过来，如果 v1 是新增加的，v2 是已经有的，则图中没有节点指向 v2，即 v2 是不可达的，SSSP 值为无穷大。

c. 这条边的两个顶点都是原图已经存在的顶点



假设新增的边为(v1,v2,distance)，v1 和 v2 均为原图中已经存在的点，且边的方向为 v1 指向 v2。此时，因为指向 v1 的点集没有发生改变，所以 v1 的 SSSP 的值不会发生改变，而指向 v2 的点新增了 v1 这个点，有可能导致从 v1 走向 v2 距离更短。假设 v2 的原来的 SSSP 的值为 oldDis，如果 v1+distance < oldDis，则更新 v2 的值为 v1+distance,因为 v2 被更新，所以 v2 的后续节点可能也会被更新，则继续更新 v2 的后续节点，如果 v1+distance>=oldDis,则新增的这条边不会更新 v2 的值，即 v2 保持不变。

算法和核心代码如下：

SSSP Algorithm

```
public boolean update(EdgeEvent<KV, EV> event) {
    EventType type = event.getType();
    Edge<KV, EV> edge = event.getValue();
    KV source = edge.getSource(), target = edge.getTarget();
    switch (type){
        case ADD:
            neighborState.update(event); //update the neighbors.
            if(state.containsKey(source)){
                Long newValue = get(source) +
edge.getEdgeValue().longValue();
                spread(target,newValue);
            }
            return true;
        default:
            throw new UnsupportedOperationException("The delete and
```

```
update type events are not supported by now.");
    }
}
```

```
public void spread(KV id,Long value){
    //if the vertex is not already in state and its closer to original vertex, we will change nothing.
    if(state.containsKey(id) && state.get(id) <= value)
        return;
    set(id,value);
    Set<Edge<KV, EV>> neighbors = neighborState.get(id);
    KV target; Long tarOldValue,tarNewValue;
    if(neighbors == null) return;
    for(Edge<KV,EV> edge : neighbors){
        target = edge.getTarget();
        if(state.containsKey(target)){//if this vertex has already calculated.
            tarOldValue = get(target).longValue();
            tarNewValue = value + edge.getEdgeValue().longValue();
            if( tarNewValue < tarOldValue){ // if the new value is smaller.
                spread(target,tarNewValue);
            }
        }else{//else the vertex is reachable now.
            tarNewValue = value + edge.getEdgeValue().longValue();
            spread(target,tarNewValue);
        }
    }
}
```

3.4 PageRank

DD,TC 和 SSSP 算法能够用流式的增量计算模型是显而易见的。但 PageRank 是否可以呢？我们假设原图为G_0,原图的初始状态为x_0，新增一条边之后，现在的图为G_1，新增节点的初始状态为x_1。PageRank 算法为f。

则有

$$R_0 = f(x_0, G_0)$$
$$R_1 = f(R_0 + x_1, G_1)$$
$$R_* = f(x_0 + x_1, G_1)$$

R_0表示在原始状态为x_0的图G_0中进行若干次迭代之后的计算结果；R_1表示以原始图的计算结果R_0和新增节点的初始状态x_1作为新图G_1的初始状态，经过若干次迭代之后的运行结果；R_*表示直接在最初的x_0和x_1上进行若干次迭代之后的运行结果，如果有R_1=R_*，则证明 PageRank 算法可以用流式的增量计算模型进行计算，而且能够得到准确结果。感谢 Larry

Page 和 Sergey Brin，他们从理论上证明了不论初始值如何选取，PageRank 算法都保证了计算结果能够收敛到他们的真实值。因此 PageRank 算法可以使用流式的增量计算模型。如果初始值越接近真实值，那么算法的收敛就越快，而利用上一次的计算结果作为下一次的初始值，显然要比从头开始计算收敛的要快。

4 算法测试

上述四种算法，均完成了功能性的测试，测试结果无误。后续会展开进行性能测试。测试过程带后续补充。