哈尔滨工业大学

<<大数据分析>>

实验报告之三

(2019 年度春季学期)

姓名:	**王丙昊**
学号:	**1160300302**
学院:	**计算机学院**
教师:	**王金宝**

实验三 图数据分析

一、实验目的

掌握大图数据计算平台的原理、架构和工作机制，理解大图计算平台的各项功能，包括数据载入、大图数据划分和大图计算。能够编写大图数据计算平台中常用图分析法（PageRank，Single Source Shortest Path）程序。

二、实验环境

Windows 操作系统，Java 编程语言。

三、实验过程及结果

1. BSP 模型

Pregel 程序使用 **BSP（批量同步并行）计算模型**，将计算过程分为迭代的超级步(superstep)。在每一轮 SuperStep 中，每个顶点接收上一轮 SuperStep 的消息，执行用户自定义的 compute()函数，并发送信息给其他顶点，同时可能修改自身状态（用户自定义值、当前状态等）以及以它为顶点出边的状态。

BSP 模型具体实现：

- 在 Master 节点中编写 run()函数实现 BSP 计算模型。
- Pregel 计算过程从 SuperStep 0 开始，所有顶点都处于 active 状态。常在 SuperStep 0 中对顶点进行**初始化**（不同图分析程序对顶点自定义值和初始状态要求不同），并且简单的 Aggregator 常在 SuperStep 0 统计图的某些全局聚集信息。
- 在每个 SuperStep 中，Master 串行地调用每个 Worker 的 run()函数，Worker 的 run()负责串行地执行该 Worker 上所有 active 顶点的 compute()函数；然后 Worker 间进行通信，本轮各个顶点发送的消息将在下一个 SuperStep 中被接收。
- 如果在一轮 SuperStep 中的顶点计算完成后，没有要通信的消息并且所有顶点都处于 inactive 状态，BSP 计算过程结束。

2. Master 节点

Master 主要负责 **Worker 之间的工作协调**，每一个 worker 在其注册到 master 的时候会被分配一个唯一的 ID，Master 内部维护着一个当前活动的 worker 列表，该列表中包括每个 worker 的 ID 以及哪些 worker 被分配到了整个图的哪一部分。在每一个 SuperStep 中，Master 可以向所有的 Worker 发出指令，启动相应的操作。

Master 同时还保存着整个计算过程以及整个 graph 的**统计数据**，如图的总大小，处于

active 状态的顶点个数等。这个功能通过在 Master 中设置统计器，或者使用 Aggregator 类来实现。

Master 还控制系统的**输入、输出、图划分以及存储**等功能。

Master<V, L>类中，两个类型参数 V 和 L 分别表示用户自定义值的类型和消息的类型。

Master<V, L> 中的变量定义：

- **List<Worker<V, L>> workers**：Master 所维护的 Worker 列表。
- **Statistics statistics**：Master 中的统计器实例，用于统计每个 Worker 存储顶点、边的数量，以及各个 SuperStep 中每个 Worker 的计算时间、通信数量。
- **Boolean aggregator_flag**：Pregel 计算任务是否使用 Aggregator 的标志位。
- **Aggregator aggregator**：Pregel 计算任务所使用的聚集器。（具体实现见 3.7 Aggregator）

Master<V, L> 中的函数定义：

- **Public void setCombiner(Combiner<L> combiner)**

函数功能：为 Pregel 计算过程开启 Combiner 功能，并传入一个用户自定义的 Combiner 实例。在消息进入 Worker 的发送队列和接受队列时生效，根据用户自定义的 combine()函数对消息队列中的消息进行合并。

实现过程：通过将这个 Combiner 实例传给每个 Worker，为 Worker 开启 Combiner 功能。

- **Public void setAggregator(Aggregator aggregator)**

函数功能：为 Pregel 计算过程开启 Aggregator 功能。传入的 Aggregator 实例可以设置为在一轮 SuperStep 后默认关闭，也可以通过参数设置来一直开启 Aggregator 功能。

实现过程：将传入的 Aggregator 实例保存到 Master 节点中，然后为每个 Worker 传入一个相同类型的 Aggregator 实例，来为 Worker 开启 Aggregator 功能。

- **Public void run(Vertex<V, L> vertex)**

函数功能：负责 BSP 模型的计算过程。

实现过程：

1. 在每个 SuperStep 中，串行地调用所有 Worker 的 run()函数，然后进行 Worker 之间的通信，过程中会将每个 Worker 的计算时间、通信数量记录到 Statistic（统计器）实例中。

2. 如果开启了 Aggregator 功能，Master 会搜集每个 Worker 上的聚集值，通过 Aggregator 上的 aggregate()方法对聚集值进行处理。

3. 在一轮 SuperStep 中所有的 Worker 计算完成后，如果 Worker 间没有通信并且所有的顶点都处于 inactive 状态，计算过程结束。

- **Public void initial()**

函数功能：初始化 Worker 节点，将图划分 load 到 Worker 中。

实现过程：Worker 在 Master 节点中注册，每个 Worker 调用 load()方法来加载所负责的图划分。加载过程中，可以将每个 Worker 存储顶点、边的数量记录到 Statistic（统计器）实例中。

- **Public void communication()**

函数功能：负责 Worker 节点之间的通信功能。

实现过程：每个 Worker 包括一个消息接收队列和消息发送队列，通信过程通过**变量赋**

值的方式实现。循环获取每个 Worker 的消息发送队列（消息队列是以 Map<String, List<L>> 的 Map 存储的，key 为目标顶点的 ID，value 为发往该顶点的消息列表），对消息发送队列中的每一项，判断目标顶点在哪一个 Worker 中，将消息列表赋值给该 Worker 中目标定点的接收消息队列中。

通信过程中，可以将每个 Worker 的通信数量记录到 Statistic（统计器）实例中。

如果启用了 Combiner 功能，在消息加入到 Worker 的消息接收队列时，会调用 Combiner 中用户自定义的 combine()函数，对消息队列中的消息进行 Merge。

- **Public Boolean isEnd()**

  函数功能：判断 BSP 计算过程是否结束。

  实现过程：判断依据是 Worker 间没有通信，并且所有的顶点都处于 inactive 状态。

  在 Worker 中实现一个方法来判断 Worker 节点是否活跃，判断依据是该 Worker 的消息发送队列是否为空且该 Worker 上所有顶点的状态。Master 判断 BSP 过程是否结束时，只需要判断每个 Worker 节点是否活跃，如果有活跃的 Worker 节点，计算过程就不会结束。

- **Public void partition()**

  函数功能：实现了基于 Edge-Cut 的 Random 划分算法。

  实现过程：首先将图中的所有顶点随机分配到每个 Partition 中；然后将所有的边按照边的起点加入到对应的 Partition 中。

- **Public void result(String filePath)**

  函数功能：将 Pregel 程序的结果持久化到磁盘。

  实现过程：这里只是简单的把图中顶点的 ID 以及计算结束后顶点的用户自定义值（value）输出到磁盘。

3. Worker 节点

一个 Worker 会在内存中维护分配到其上的 graph partition 的状态，可以简单地看作是一个**从顶点到顶点状态的 Map**，其中顶点状态包括：该顶点的当前值（用户自定义值），一个以该顶点为起点的出边列表，顶点当前是否 active 的标志位。同时每个 Worker 会维护一个接收消息队列和发送消息队列。

在每个 SuperStep 中，Worker 会循环遍历所有顶点，并调用每个顶点的 Compute()函数，传给该函数该顶点在上一轮 SuperStep 通信过程中接收到的消息。当顶点在 Compute 的过程中，如果要发送消息给其他顶点，会把该消息加入 Worker 的发送消息队列。

当 Worker 收到消息时，在实际的系统中可以做这样一个优化，首先判断目标顶点是属于远程的 Worker 机器，还是当前 Worker，如果在同一个 Worker 中，就直接把消息放到目标顶点的消息存储空间中（这样就需要两个消息接收队列！）；如果在不同 Worker 中，消息就放入消息发送队列中。

如果 Pergel 计算任务启动了 Combiner 功能，在消息被加入到发送队列或者到达输入队列时，会执行 Combiner 中的 Combine()函数。后一种情况不会节省通信代价，但是会节省消息存储的空间。

如果 Pregel 计算任务启动了 Aggregator 功能，Worker 会获取每个顶点上的聚集值，通过自定义的 Aggregator 中的 aggregate()方法来对聚集值进行处理，并将处理后的聚集值发送给 Master 节点。

Worker<V, L>中两个类型参数 V 和 L 分别表示用户自定义值的类型和消息的类型。

Worker<V, L> 中的变量定义：

- **Set<Vertex<V, L>> vertices**：Worker 所维护的顶点集合，每个 Vertex 上的信息包括顶点的 ID，当前的用户自定义值，顶点当前的活跃状态等。
- **Map<String, List<L>> receive_messages**：Worker 的消息接收队列。
- **Map<String, List<L>> send_messages**：Worker 的消息发送队列。
- **int superstep**：Worker 当前所处的 SuperStep 轮数。
- **Boolean combiner_flag**：Worker 是否启用了 Combiner 功能的标志位。
- **Boolean aggregator_flag**：Worker 是否启用了 Aggregator 功能的标志位。
- **Combiner combiner**：Pergel 计算任务中使用的 Combiner 实例。
- **Aggregator aggregator**：Pregel 计算任务中使用的 Aggregator 实例。

Worker<V, L> 中的函数定义：

- **Public void run(Vertex<V, L> vertex)**

  函数功能：负责 BSP 模型的每个 SuperStep 在每个 Worker 中的操作。

  实现过程：

  1. Worker 在每个 SuperStep 中，会循环遍历 Worker 上存储的所有顶点，如果顶点处于 active 状态，则执行其 Compute()函数，执行时需要传入该 Vertex 在上一轮 SuperStep 中接收到的消息。

  2. 在 Vertex 的 Compute()函数执行完后，将 Vertex 发送的消息加入到 Worker 的消息发送队列中。如果开启了 **Combiner 功能**，每当消息加入到消息发送队列时，都会调用用户自定义的 combine()方法，对消息发送队列中的消息进行 Merge。

  3. 如果 Worker 开启了 **Aggregator 功能**，Worker 在执行过程中会获取 Vertex 上 report 的聚集值，并通过 aggregate()方法对搜集到的聚集值进行处理。

  Aggregator 可以分为在 SuperStep 0 后自动关闭，或一直开启这两种。如果 Aggregator 自动关闭，在 SuperStep0 中 Worker 处理完 Vertex 上的聚集值后，向 Master 发送该聚集值，之后的 SuperStep 中，不会对聚集值进行操作；如果一直开启 Aggregator 功能，每一轮 SuperStep 中 Worker 都会处理 Vertex 上的聚集值，并发送给 Master。

- **Public void load(String filePath)**

  函数功能：从磁盘读取图划分结果，将图数据存储于内存。

  实现过程：读文件，根据读取到的内容构造 Vertex 实例，Vertex 实例中必须包括 Vertex 的 ID 以及从该 Vertex 出发的边。

  在这里没有必要对 Vertex 的用户自定义值以及初始状态(active 或 inactive)进行初始化，因为不同的图分析算法对 Vertex 的 value 以及 state 的初始化要求不同。对于不同图分析算法 Vertex 初始化的问题，可以在用户自定义 Vertex 的 Compute()方法中，在 SuperStep0 中对 Vertex 的 value 及 state 进行初始化。

- **Public Boolean idEnd()**

  函数功能：判断 Worker 节点是否活跃。

  实现过程：**非活跃**的判断依据是 Worker 没有要发送的消息，以及存储在 Worker 上的 Vertex 的状态都为 inactive。

- **Public void setCombiner(Combiner<L> combiner)**

  函数功能：为 Worker 开启 Combiner 功能。

  实现过程：将 Worker 的 Combiner_flag 置为 true，将传入的 Combiner 实例赋值给 Worker 中的 Combiner 变量。

- **Public void setAggregator(Aggregator aggregator)**

  函数功能：为 Worker 开启 Aggregator 功能。

  实现过程：将 Worker 的 Aggregator_flag 置为 true，将传入的 Aggregator 实例赋值给 Worker 的 Aggregator 变量。

- **Public Boolean contain(String vertex_id)**

  函数功能：判断一个 Worker 上是否存储 ID 为 vertex_id 的顶点。

  实现过程：为了加快该函数的执行速度，在每个 Worker 中维护一个从 Vertex 的 ID 到 Vertex 的 Map（**HashMap<String, Vertex>**），这样既可以加快该函数的执行过程，也方便由 vertex_id 迅速找到对应的 Vertex 实例。后一种作用更为重要。

- **Public void receiveMessage(String dest_vertex, List<L> messages)**

  函数功能：接收从其他 Worker 发送过来的消息。

  实现过程：在执行这个函数之前，必须要保证执行该函数的 Worker 存储了 ID 为 dest_vertex 的 Vertex。将 List<L> messages 加入到 dest_vertex 的消息接受队列中，并且**将 dest_vertex 的状态改为 active**。如果开启了 Combiner 功能，每当一个顶点接收消息时，都会调用 Combiner 中自定义的 combine()函数，对该顶点的接收消息队列进行合并。

- Worker 中变量的一些 get()方法或初始化方法，这里不再一一列出

4. Vertex 基本类

编写 Pregel 程序需要继承该 Vertex<V, L>类，并重写里面的 Compute()方法。其中 V 表示用户自定义值的类型，L 表示消息的类型。

在 Vertex 基本类中存储了一个 Vertex 上的信息，包括 Vertex 的 ID，用户自定义值，以该顶点为起点的边的集合，当前的状态（active or inactive）。

用户重写 Vertex 的 Compute()函数，该函数会在每一个 SuperStep 中对每一个顶点进行调用。预定义的 Vertex 基本类中实现了方法，允许 Compute()方法可以通过调用 GetValue()方法来得到当前顶点的值，或者通过调用 MutableValue()方法来修改当前顶点的值，还可以通过 GetEdges()方法来遍历出边列表等。

Vertex<V, L> 变量定义：

- **String id**：顶点的标识符。
- **V value**：顶点当前的用户自定义值。
- **Set<String> edges**：已该顶点为起点的出边列表。
- **Boolean state**：顶点当前的状态（active or inactive）。
- **Int superStep**：Pregel 计算过程中该顶点目前所处的超步轮数。
- **Map<String, List<L>>send_messages**：顶点上临时的消息发送队列。

Vertex<V, L> 方法定义：

- **Public void compute(Vertex<V, L> vertex, List<L> messages)**

函数功能：所有图分析算法要重写的方法。在该方法内根据顶点的当前值及接收的消息，计算得到新的顶点值，可以对顶点值进行更新；可以对顶点的状态进行改变；可以发送消息给其他顶点；可以在 SuperStep 0 中根据不同图分析算法的要求，对顶点进行初始化。

- **Public V getValue()**

函数功能：获取顶点当前的用户自定义值。

- **Public void mutableValue(V value)**

函数功能：修改顶点当前的用户自定义值。

- **Public Set<String> getEdges()**

函数功能：获取以该顶点为起点的出边列表。

- **Public void sendMessageTo(String dest_vertex, List<L> message)**

函数功能：从当前顶点发送消息给目标顶点；

实现过程：在本次实现中，首先把要发送的消息存储在**当前顶点的临时消息发送队列**中，当 Worker 对该顶点的 compute()函数调用完成后，由 Worker 来获取该顶点所要发送的所有消息，将它们加入到 Worker 的消息发送队列中。如果开启了 Combiner 功能，每当消息加入 Worker 的消息发送队列时，都会调用自定义的 combine()方法对消息进行 Merge。

- **Public void voltToHalt()**

函数功能：将顶点自身的 status 设置成 **halt** 来表示它已经不再 active。这就表示该顶点没有进一步的计算需要执行，Pregel 框架将不会在接下来的 SuperStep 中对该顶点执行 compute()，**除非该顶点接收到其他传送的消息**（此时会调用 voltToActive()）。

- **Public void voltToAvtive()**

函数功能：在初始化或者当顶点接收到其他顶点传来的消息时使用，将顶点自身的 status 设置成 active，表明在下一轮的 SuperStep 中，该顶点需要参与运算。

5. Communiaction

消息的**类型**由 Vertex<V, L>中的 L 类型参数来决定。

在本次实验中，使用变量赋值来模拟消息发送。**这个功能由 Vertex，Worker 以及 Master 协同实现：**

- 在一个 SuperStep 中，一个顶点在执行它的 Compute()方法时，该顶点可以发送任意多条消息。这些消息首先存储在 **Vertex 类中的临时消息发送队列**中。
- Worker 每调用完一个顶点的 compute()方法时，会将该顶点的临时消息发送队列中的消息加入到 **Worker 的消息发送队列中**。如果启用了 Combiner 功能，每当有消息进入到消息发送队列时，会根据用户自定义的 combiner()函数对消息进行 Merge。
- 在一轮 SuperStep 中，Master 串行地调用所有 Worker 的 run()函数。当所有 Worker 执行完成后，开始调用 **Master 中的 communication()**函数，进行 Worker 节点之间的通信。

- **Master 中的 communication()执行过程：**每个 Worker 包括一个消息接收队列和消息发送队列，通信过程通过**变量赋值**的方式实现。循环获取每个 Worker 的消息发送队列（消息队列是以 Map<String, List<L>>的 Map 存储的，key 为目标顶点的 ID，value 为发往该顶点的消息列表），对消息发送队列中的每一项，判断目标顶点在哪一个 Worker 中，将消息列表赋值给该 Worker 中目标定点的接收消息队列中。如果启用了 Combiner 功能，每当消息加入到接收队列时，都会调用 combine()方法。

6. Combiner

发送消息时会产生一些开销，并且消息在 Worker 节点上的存储会占用一定的存储空间。在一些特定的图分析算法中，可以借助用户的协助来降低这些开销。

通过定义一个 Combiner 接口，用户在实现特定的图分析算法时，可以实现该接口中的 combine()方法。

Combiner 功能在默认情况下关闭，因为找不到一种对所有顶点的 Compute()函数都合适的 Combiner；如果开启了 Combiner 功能，在消息到达 Worker 的消息发送队列或消息接收队列时，会自动调用 combine()方法，对消息进行 Merge。

Combiner<L> 接口：在该接口中仅有一个抽象方法。

- **Public List<L> combine(List<L> messages)**

  函数功能：对消息队列中的消息进行 Merge，并将合并后的结果返回。

在单源最短路径算法中，可以开启 Combiner 功能，使用 ShortestPathCombiner 类来实现 Combiner<Integer>，并重写 combine()方法。该函数的作用是在消息队列中仅保留消息队列中值最小的消息。以 SuperStep10 的通信量为例进行比较。

Single-Source-Shortest-Path 未启用 Combiner 功能情况下的通信量：

```
superstep: 10 communicating.....
worker1 communication size: 62325
worker2 communication size: 63916
worker3 communication size: 61965
worker4 communication size: 65126
worker5 communication size: 61525
worker6 communication size: 63354
worker7 communication size: 62277
worker8 communication size: 61811
total communication size: 502299
superstep: 10 communication complete
```

Single-Source-Shortest-Path 启用 Combiner 功能情况下的通信量：

```
superstep: 10 communicating.....
worker1 communication size: 38641
worker2 communication size: 39413
worker3 communication size: 37671
worker4 communication size: 39846
worker5 communication size: 38289
worker6 communication size: 38966
worker7 communication size: 38855
worker8 communication size: 37943
total communication size: 309624
superstep: 10 communication complete
```

在 PageRank 算法中，也可以开启 Combiner 功能，使用 PageRankCombiner 类来实现 Combiner<Double>，并重写 combine()方法。该函数的作用是在消息队列中保留所有消息的加和。比较 PageRank 算法是否启用 Combiner 功能的通信量。

PageRank 未启用 Combiner 功能每轮的通信量：

```
superstep: 0 communicating……
worker1 communication size: 635685
worker2 communication size: 641098
worker3 communication size: 636479
worker4 communication size: 644138
worker5 communication size: 639293
worker6 communication size: 639902
worker7 communication size: 637305
worker8 communication size: 631139
total communication size: 5105039
superstep: 0 communication complete
```

PageRank 启用 Combiner 功能每轮的通信量：

```
superstep: 0 communicating……
worker1 communication size: 259503
worker2 communication size: 260028
worker3 communication size: 258090
worker4 communication size: 261513
worker5 communication size: 260196
worker6 communication size: 259404
worker7 communication size: 259402
worker8 communication size: 258114
total communication size: 2076250
superstep: 0 communication complete
```

结论：在 PageRank 以及单源最短路径算法中，通过开启 Combiner 功能，可以极大地减少 Worker 之间的通信量。

7. Aggregator

Pregel 的 Aggregator 是一种提供全局通信，监控和数据查看的机制。在一个 SuperStep 中，每个顶点都可以向 Aggregator 提供一个数据，系统会使用一种 aggregate()操作来负责聚合这些值。在有些计算中，产生的聚集值可能会在下一轮 SuperStep 中使用。

Aggregator 功能默认关闭，如果要使用 Aggregator 功能，需要实现 Aggregator 接口并重写里面的方法，这里可以定义一些通用的 Aggregator。

Aggregator 这里分为两种，一种是执行一轮 SuperStep 后自动关闭，默认是这种情况；另一种是在 Pregel 的计算过程中一直开启。Aggregator 的种类由 isAutoClosed()方法来决定。

Aggregator 接口：

- **Public double report(Vertex vertex)**
  函数功能：指定每个顶点发送的内容
- **Public void aggregate(double aggregate_value)**
  函数功能：接收发送过来的聚集值，并对聚集值进行处理。
- **Public Aggregator getInstance(Boolean autoClose)**
  函数功能：获取相同类型的一个新的 Aggregator 实例。
- **Public Boolean isAutoClosed()**

函数功能：判断 Aggregator 的类型，是仅执行一轮 SuperStep 后自动关闭，还是一直开启。

这里实现了两类 Aggregator，仅执行一轮 SuperStep，随后自动关闭的，如统计图中边的个数或顶点的个数（VertexNumberAggregator、EdgeNumberAggregator）；Aggregator 在计算过程中一直开启的，如统计每一轮 active 顶点的个数（ActiveVertexNumberAggregator）。

1. 对图中顶点的个数进行统计，在 PageRank 算法的执行中加入 VertexNumberAggregator，输出结果如下：

```
Aggregate Value: 875713.0
```

分析：图中顶点的数量为 875713

2. 对单源最短路径算法中每一轮 SuperStep 的 active 顶点进行统计：

```
superstep: 0 Aggregate Value: 875713.0
superstep: 1 Aggregate Value: 1.0
superstep: 2 Aggregate Value: 4.0
superstep: 3 Aggregate Value: 33.0
superstep: 4 Aggregate Value: 169.0
superstep: 5 Aggregate Value: 521.0
.................
.................
superstep: 33 Aggregate Value: 9.0
```

分析：在 SuperStep 0 中，对所有顶点的 value 及 status 进行初始化，所以 active 顶点的数量为 875713；在 SuperStep 1 中，仅有源点活跃，active 顶点的数量为 1；源点向与之邻接的顶点发送消息，与“0”邻接的顶点由 4 个，所以在 SuperStep 2 中，active 顶点的数量为 4.

8. 图分析算法实现

- **PageRank**
  - **PageRankVertex**，继承自 Vertex 类，并重写了 compute()方法。在 compute()方法中，如果顶点处于 SuperStep 0，则对顶点进行初始化，将所有顶点的 value 初始化为 1(可以是用户自定义的其他值)，status 初始化为 active；若顶点处于规定的迭代轮数内，会根据当前值及收到的消息，计算新的 value 并更新，并向邻接顶点发送新的 value 值；如果超过了迭代轮数，不发送任何消息，且将自身的 status 置为 inactive。
  - **PageRankCombiner**，实现了 Combiner 接口，并重写了 combine()方法。在该方法中仅保留消息队列中的消息的和。
  - **PageRank 结果**，将所有顶点的 value 初始化为 1，迭代轮数设为 10，在 BSP 计算过程执行完成后，将每个顶点的 ID 及其当前 value 值输出到 output/pagerank/。

```
897213  0.18817560154783589
633576  0.15
417877  0.2277706941195945
417879  0.24151595831866574
897211  0.1925
417870  0.742291621459163
114062  0.351897616852514
36364   0.15
897216  0.225149539360968
897225  0.350189712834299
417886  0.8994914399692927
633589  0.485760265999185
442823  16.49186100554055
114058  0.17350756635888917
```

- **PageRank 结果分析：**每个顶点的 Value 表示了该顶点的重要程度，所有顶点的 Value 初始化为 1，随着不断地迭代，value 会发生变化，value 越大表示该顶点越重要。

- **单源最短路径**

  ShortestPathVertex 继承自 Vertex 类，并重写了 compute()方法

  - **ShortestPathVertex**，继承自 Vertex 类，并重写了 compute()方法。在 compute()方法中，如果顶点处于 SuperStep 0，则对顶点进行初始化，将源点的 value 初始化为 0，status 初始化为 active，其他顶点的 value 初始化为无穷大，status 初始化为 inactive；在每轮超步中，active 顶点将接收到的消息与自身 value 比较，如果 value 更新则向邻接顶点发送消息，然后将 status 改为 inactive，如果 status 不变，直接将 status 改为 inactive。
  - **ShortestPathCombiner**，实现了 Combiner 接口，并重写了 combine()方法。在该方法中仅保留消息队列中的最小值。
  - **ShortestPath 结果**，取 ID 为 0 的点为源点，在 BSP 计算过程停止后，将计算结果输出到 output/sssp/sssp_result.txt 中，value 为顶点距离源点的跳步数，如果为 INF 表示该点不可达。

```
897213  12
633576  INF
417877  10
417879  12
897211  INF
417870  11
114062  10
36364   INF
897216  12
897225  INF
417886  INF
633589  14
442823  10
114058  11
```

  - **ShortestPath 结果分析：**取 ID 为 0 的点为源点，除了不可达的顶点外，顶点距离源点的最远距离中，最大值为 32.

9. 图划分算法

Edge-Cut 的 random 划分算法。

在 Master 类中编写 partition()函数来实现图数据的划分，以及将划分后的结果持久化到磁盘；在 Worker 类中编写 load()函数来载入所应负责的各个划分。

- **Public void partition()**

  函数功能：实现了基于 Edge-Cut 的 Random 划分算法。

  实现过程：首先将图中的所有顶点随机分配到每个 Partition 中；然后将所有的边按照边的起点加入到对应的 Partition 中。

- **Public void load(String filePath)**

  函数功能：从磁盘读取图划分结果，将图数据存储于 Worker 的内存。

  实现过程：读文件，根据读取到的内容构造 Vertex 实例，Vertex 实例中必须包括 Vertex 的 ID 以及从该 Vertex 出发的边。

10. 统计功能

定义一个 Statistic 基本类，在 Master 中加入一个该类的实例。在程序运行过程中，统计每个 Worker 存储顶点、边的数量以及各个 SuperStep 中每个 Worker 的计算时间、通信数量。这里拿单源最短路径算法的运行举例，在运行过程中会把这些信息存储到 Statistic 实例当中：

- 每个 Worker 存储顶点、边的数量

```
worker1 load completely.
worker1 vertex number: 109220, edge number: 635685
worker2 load completely.
worker2 vertex number: 109489, edge number: 641098
worker3 load completely.
worker3 vertex number: 109608, edge number: 636479
worker4 load completely.
worker4 vertex number: 109571, edge number: 644138
worker5 load completely.
worker5 vertex number: 110105, edge number: 639293
worker6 load completely.
worker6 vertex number: 109512, edge number: 639902
worker7 load completely.
worker7 vertex number: 109196, edge number: 637305
worker8 load completely.
worker8 vertex number: 109012, edge number: 631139
```

- 每轮 SuperStep，每个 Worker 的计算时间、通信数量

```
------------------------superstep: 4---------------------------
superstep: 4 running……
worker1 running time: 0.014661395s
worker2 running time: 0.016079161s
worker3 running time: 0.014655225s
worker4 running time: 0.016217441s
worker5 running time: 0.014757008s
worker6 running time: 0.014744157s
worker7 running time: 0.013929892s
worker8 running time: 0.015220172s
superstep: 4 run complete, time: 0.121119841s
superstep: 4 communicating……
worker1 communication size: 106
worker2 communication size: 68
worker3 communication size: 79
worker4 communication size: 86
worker5 communication size: 107
worker6 communication size: 102
worker7 communication size: 56
worker8 communication size: 126
total communication size: 730
superstep: 4 communication complete
superstep: 4 Aggregate Value: 169.0
```

四、实验心得

本次实验就是在理解了 Pregel 计算模型之后，一个简单的计算系统的实现。如果熟练掌握 Java 编程的一些基本技巧，如面向 ADT 的编程和一些设计模式的使用，实现过程的难度并不大。