图计算实验部分

实验一 YiTu 安装与配置实验

【实验目的】

通过本实验，学习如何在 Linux 环境下配置图计算执行引擎 YiTu_XGraph 的安装与使用，为后续的图数据处理和分析工作做好准备。

【实验内容】

1. 安装 Linux 系统。
2. 安装 docker(可选)
3. 安装 YiTu 图计算编程框架

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：conda,python
3. 硬件要求：至少 1 台计算机或虚拟机，建议配置至少 4GB 内存和 100GB 的硬盘空间。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 用浏览器访问 https://github.com/CGCL-codes/YiTu，下载.zip 源代码文件。

2. 编译安装 YiTu：

通过 cmake 安装：

```
rm-rf~/temp
version=3.18
build=0
mkdir~/temp
cd~/temp
wgethttps://cmake.org/files/v$version/cmake-$version.$build-Linux-x86_64.sh
```

sudomkdir/opt/cmake

sudoshcmake-$version.$build-Linux-x86_64.sh--prefix=/opt/cmake--skip-license

sudoln-s/opt/cmake/bin/cmake/usr/local/bin/cmake

cd~

rm-rf~/temp

**(仅安装图计算部分)

定位到源码/examples/YiTu_GNN/NDP 目录下：

删除原有 pybind11 目录，直接下载 pybind11 源码

gitclonehttps://github.com/pybind/pybind11.git

cmake.

（如果报错，定位到 cmake 报错代码处，将 py 修改成 pybind11 即可）

make

cdnondp

make

图计算实验部分

实验二 DFS 实验

【实验目的】

本实验旨在通过编写和执行基于 YiTu_XGraph 的 DFS 程序，帮助学生深入理解图计算系统的工作原理，并学会使用 YiTu_XGraph 进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写简单的图算法程序以及运行它们在 YiTu_XGraph 系统中。

【实验内容】

1. 基于 YiTu_XGraph 实现 DFS 算法，掌握 DFS 算法和函数式编程思想。
2. 使用命令行执行 DFS 程序。
3. 查看程序执行结果。

【实验环境】

1. 操作系统：Linux（Ubuntu）
5. 软件环境：conda,python
2. 硬件要求：至少 1 台计算机或虚拟机，建议配置至少 4GB 内存和 100GB 的硬盘空间用于安装 Hadoop。
3. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 接下来，将以 YiTu_XGraph 中的 BFS 算法为例，讲解如何在 YiTu_XGraph 实现 BFS 算法并运行。

   以 BFS 算法为例的实现细节：执行 pythondemo.py--YiTu_GNN0--methodbfs–inputGraph.bcsr--source0

后调用<root>/YiTu/examples/YiTu_GNN/NDP/bfs_sig_async.cu 代码，需要注意，输入只支持.bcsr 和.el 格式的数据集

迭代方式的 BFS 算法的伪代码：

算法： BFS(G, v_0)

输入： 图$G=\{V，E\}$，指定源顶点v_0

输出： 图G所有顶点与v_0的深度向量val

初始化活跃顶点队列$active$和下一轮的活跃顶点队列$nextactive$;

初始化深度向量val;

将顶点v_0加入$active$;

while($active$不为空)**do**

Vertex$v=active$.front();

$active$出队;

while($\exists(v\rightarrow u)\in E\&\&val[v]$+1$<val[u]$)**do**

$val[u]\leftarrow val[v]$+1;//遍历到未被访问的邻居时，使邻居的深度加 1

顶点u加入队列$nextactive$;

endwhile

交换队列$active$和$nextactive$;

endwhile

returnval;

---

1) 初始化：初始化包括分析指令、初始化图结构、初始化图状态三个阶段。

分析指令时创建 ArgumentParser 对象分析 Linux 指令的输入参数；

```
ArgumentParser arguments(argc, argv, true, false);
```

初始化图结构创建GraphStructrue对象，并根据输入的图获取图的数据，如顶点数、边数以及分布情况。并根据输入图计算各顶点的度数，以 CSR 格式存储。

```
GraphStructure graph;
graph.ReadGraph(arguments.input);
```

初始化图状态根据图顶点数和算法需求创建 GraphStates 对象，并初始化活跃顶点

```
GraphStates<uint> states(graph.num_nodes, true, false, false);
```

2) 子图划分：创建 Sugraph、SubgraphGenerator、Partitioner 等对象

```
Subgraph subgraph(graph.num_nodes, graph.num_edges);

SubgraphGenerator<uint> subgen(graph);

subgen.generate(graph, states, subgraph);
```

```
Partitioner partitioner;
```

3) 迭代：根据当前迭代是否存在活跃顶点进行迭代。每轮迭代执行 CUDA 核函数。对应核函数代码在 <root>/YiTu/examples/YiTu_GNN/NDP/shared/gpu_kernels.cu：

```
bfs_async<<<partitioner.partitionNodeSize[i] / 512 + 1, 512>>>(partitioner.partitionNodeSize[i],
                                                               partitioner.fromNode[i],
                                                               partitioner.fromEdge[i],
                                                               subgraph.d_activeNodes,
                                                               subgraph.d_activeNodesPointer,
                                                               subgraph.d_activeEdgeList,
                                                               graph.d_outDegree,
                                                               states.d_value,
                                                               d_finished,
                                                               (itr % 2 == 1) ? states.d_label1 : states.d_label2,
                                                               (itr % 2 == 1) ? states.d_label2 : states.d_label1);
```

```cpp
__global__ void bfs_async(unsigned int numNodes,
                          unsigned int from,
                          unsigned int numPartitionedEdges,
                          unsigned int *activeNodes,
                          unsigned int *activeNodesPointer,
                          OutEdge *edgeList,
                          unsigned int *outDegree,
                          unsigned int *dist,
                          bool *finished,
                          bool *label1,
                          bool *label2)
{
    //GPU计算线程ID
    unsigned int tId = blockDim.x * blockIdx.x + threadIdx.x;

    if(tId < numNodes)
    {
        unsigned int id = activeNodes[from + tId];
        if(label1[id] == false)
            return;
        label1[id] = false;
        //根据活跃顶点表的顶点、边偏移获取顶点的邻居索引
        unsigned int sourceWeight = dist[id];
        unsigned int thisFrom = activeNodesPointer[from+tId]-numPartitionedEdges;
        unsigned int degree = outDegree[id];
        unsigned int thisTo = thisFrom + degree;
        unsigned int finalDist;
        //从邻居的起始偏移开始遍历邻居
        for(unsigned int i=thisFrom; i<thisTo; i++)
        {
            finalDist = sourceWeight + 1;//每轮迭代使深度+1
            if(finalDist < dist[edgeList[i].end])
            {
                atomicMin(&dist[edgeList[i].end] , finalDist);//原子操作更新顶点值
                *finished = false;//有顶点发生更新，存在活跃顶点，因此迭代继续
                label2[edgeList[i].end] = true;//标记顶点为活跃
            }
        }
    }
}
```

运行指令和输出结果：

编译完成后执行 Pythondemo.py–YiTu_GNN0–method<app-name>--input<输入数据集>--source<源顶点>

```
(xgnn) chengjian@node27:~/Yitu/YiTu/examples/YiTu_GNN/NDP$ python demo.py --YiTu_GNN 0  --method bfs --input /home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/Live
Journal.bcsr --source 0
['demo.py', '--YiTu_GNN', '0', '--method', 'bfs', '--input', '/home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/LiveJournal.bcsr', '--source', '0']
bfs single-async!
Reading the input graph from the following file:
>> /home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/LiveJournal.bcsr
Done reading.
Number of nodes = 4847571
Number of edges = 68993773
Graph Reading finished in 0.2339 (s).
Processing finished in 0.055179 (s).
Results of first 30 nodes:
[0:0 1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1 10:1 11:1 12:1 13:1 14:1 15:1 16:1 17:1 18:1 19:1 20:1 21:1 22:1 23:1 24:1 25:1 26:1 27:1 28:1 29:1]
```

运行 bfs 的输出结果

接下来，请同学们将以 YiTu_XGraph 中的 BFS 算法为例，在 YiTu_XGraph 实现 DFS，并成功正确地跑出结果。

（注意事项）

```
if(algorithm == "bfs") {
    ArgumentParser arguments(argc, argv, true, false);
    cout << "bfs single-async!" << endl;
    bfs_sig_async(arguments);
}
else if(algorithm == "bc") {
    ArgumentParser arguments(argc, argv, true, false);
    cout << "bc single-async!" << endl;
    bc_sig_async(arguments);
}
```

新添加算法后需要在<root>/YiTu/examples/YiTu_GNN/NDP/YiTu_GP.cpp 中算法的 ifelse 中添加新增加的算法，使其定位到新添加的算法中。并在<root>/YiTu/examples/YiTu_GNN/NDP/CMakeLists.txt 中的 set 中添加新增的算法文件，并重新进行编译

```
set(dso_SOURCE_FILES
    ${PROJECT_SOURCE_DIR}/shared/globals.hpp
    ${PROJECT_SOURCE_DIR}/shared/gpu_kernels.cu
    ${PROJECT_SOURCE_DIR}/shared/partitioner.cu
    ${PROJECT_SOURCE_DIR}/shared/subgraph_generator.cu
    ${PROJECT_SOURCE_DIR}/shared/subgraph.cu
    ${PROJECT_SOURCE_DIR}/shared/subway_utilities.cpp
    ${PROJECT_SOURCE_DIR}/shared/argument_parsing.cu
    ${PROJECT_SOURCE_DIR}/shared/graph.cu
    ${PROJECT_SOURCE_DIR}/shared/timer.cpp

    ${PROJECT_SOURCE_DIR}/ndp/bc_sig_async.cu
    ${PROJECT_SOURCE_DIR}/ndp/bfs_sig_async.cu
    ${PROJECT_SOURCE_DIR}/ndp/cc_sig_async.cu
    ${PROJECT_SOURCE_DIR}/ndp/pr_sig_async.cu
    ${PROJECT_SOURCE_DIR}/ndp/sssp_sig_async.cu
    ${PROJECT_SOURCE_DIR}/ndp/sswp_sig_async.cu
)
```

图计算实验部分

实验三 SCC 实验

【实验目的】

本实验旨在通过编写和执行基于 YiTu_XGraph 的 SCC 程序，帮助学生深入理解图计算系统的工作原理，并学会使用 YiTu_XGraph 进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写简单的图算法程序以及运行它们在 YiTu_XGraph 系统中。

【实验内容】

4. 基于 YiTu_XGraph 实现 SCC 算法，掌握 SCC 算法和函数式编程思想。
5. 使用命令行执行 SCC 程序。
6. 查看程序执行结果。

【实验环境】

4. 操作系统：Linux（Ubuntu）
6. 软件环境：conda,python
5. 硬件要求：至少 1 台计算机或虚拟机，建议配置至少 4GB 内存和 100GB 的硬盘空间用于安装 Hadoop。
6. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 接下来，将以 YiTu_XGraph 中的 CC 算法为例，讲解如何在 YiTu_XGraph 实现 CC 算法并运行。

2.以 CC 算法为例的实现细节：执行 pythondemo.py--YiTu_GNN0--methodCC–inputGraph.bcsr--source0。与 BFS 算法的主要区别在于核函数的实现方式的不同

1）初始化：初始化包括分析指令、初始化图结构、初始化图状态三个阶段。

分析指令时创建 ArgumentParser 对象分析 Linux 指令的输入参数；

```
ArgumentParser arguments(argc, argv, true, false);
```

初始化图结构创建 GraphStructrue 对象，并根据输入的图获取图的数据，如顶点数、边数以及分布情况。并根据输入图计算各顶点的度数，以 CSR 格式存储。

```
GraphStructure graph;
graph.ReadGraph(arguments.input);
```

初始化图状态根据图顶点数和算法需求创建 GraphStates 对象，并初始化活跃顶点

```
GraphStates<uint> states(graph.num_nodes, true, false, false);
```

2) 子图划分：创建 Sugraph、SubgraphGenerator、Partitioner 等对象

```
Subgraph subgraph(graph.num_nodes, graph.num_edges);

SubgraphGenerator<uint> subgen(graph);

subgen.generate(graph, states, subgraph);
```

```
Partitioner partitioner;
```

3) 迭代：根据当前迭代是否存在活跃顶点进行迭代。每轮迭代执行 CUDA 核函数。CC 执行的核函数如下，对应代码在 <root>/YiTu/examples/YiTu_GNN/NDP/shared/gpu_kernels.cu：

```
cc_async << < partitioner.partitionNodeSize[i] / 512 + 1, 512 >> > (partitioner.partitionNodeSize[i],
    partitioner.fromNode[i],
    partitioner.fromEdge[i],
    subgraph.d_activeNodes,
    subgraph.d_activeNodesPointer,
    subgraph.d_activeEdgeList,
    graph.d_outDegree,
    states.d_value,
    d_finished,
    (itr % 2 == 1) ? states.d_label1 : states.d_label2,
    (itr % 2 == 1) ? states.d_label2 : states.d_label1);
```

```cpp
__global__ void cc_async(unsigned int numNodes,
                         unsigned int from,
                         unsigned int numPartitionedEdges,
                         unsigned int *activeNodes,
                         unsigned int *activeNodesPointer,
                         OutEdge *edgeList,
                         unsigned int *outDegree,
                         unsigned int *dist,
                         bool *finished,
                         bool *label1,
                         bool *label2)
{
    unsigned int tId = blockDim.x * blockIdx.x + threadIdx.x;

    if(tId < numNodes)
    {
        unsigned int id = activeNodes[from + tId];
        if(label1[id] == false)
            return;
        label1[id] = false;
        unsigned int sourceWeight = dist[id];
        unsigned int thisFrom = activeNodesPointer[from+tId]-numPartitionedEdges;
        unsigned int degree = outDegree[id];
        unsigned int thisTo = thisFrom + degree;

        for(unsigned int i=thisFrom; i<thisTo; i++)
        {
            if(sourceWeight < dist[edgeList[i].end])
            {
                atomicMin(&dist[edgeList[i].end] , sourceWeight);
                *finished = false;
                label2[edgeList[i].end] = true;
            }
        }
    }
}
```

3. 运行指令和输出结果：

编译完成后执行 Pythondemo.py–YiTu_GNN0–method<app-name>--input<输入数据集>--source<源顶点>

```
(xgnn) chengjian@node27:~/Yitu/YiTu/examples/YiTu_GNN/NDP$ python demo.py --YiTu_GNN 0  --method cc --input /home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/LiveJ
ournal.bcsr --source 0
['demo.py', '--YiTu_GNN', '0', '--method', 'cc', '--input', '/home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/LiveJournal.bcsr', '--source', '0']
cc single-async!
Reading the input graph from the following file:
>> /home/chengjian/Yitu/YiTu/examples/YiTu_GNN/NDP/nondp/LiveJournal.bcsr
Done reading.
Number of nodes = 4847571
Number of edges = 68993773
Graph Reading finished in 0.224365 (s).
Processing finished in 0.092218 (s).
Results of first 30 nodes:
[0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0 12:0 13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0 25:0 26:0 27:0 28:0 29:0]
```

运行 CC 算法的输出结果

接下来，请同学们将以 YiTu_XGraph 中的 CC 算法为例，在 YiTu_XGraph 实现 SCC，并成功正确地跑出结果。

图计算实验部分

可选实验：GraphX 实验

任务：由于基于 YiTu 的图计算实验需要安装 CUDA 编程库，有 GPU 资源的同学可以按上述的任务书来完成，没有条件的同学采用 Spark 的 GraphX 来实现图的深度优先搜索 DFS 和强连通分量 SCC 算法。

数据集：https://github.com/databricks/spark-training/tree/master/data/graphx

相关教程：https://spark.apache.org/docs/latest/graphx-programming-guide.html

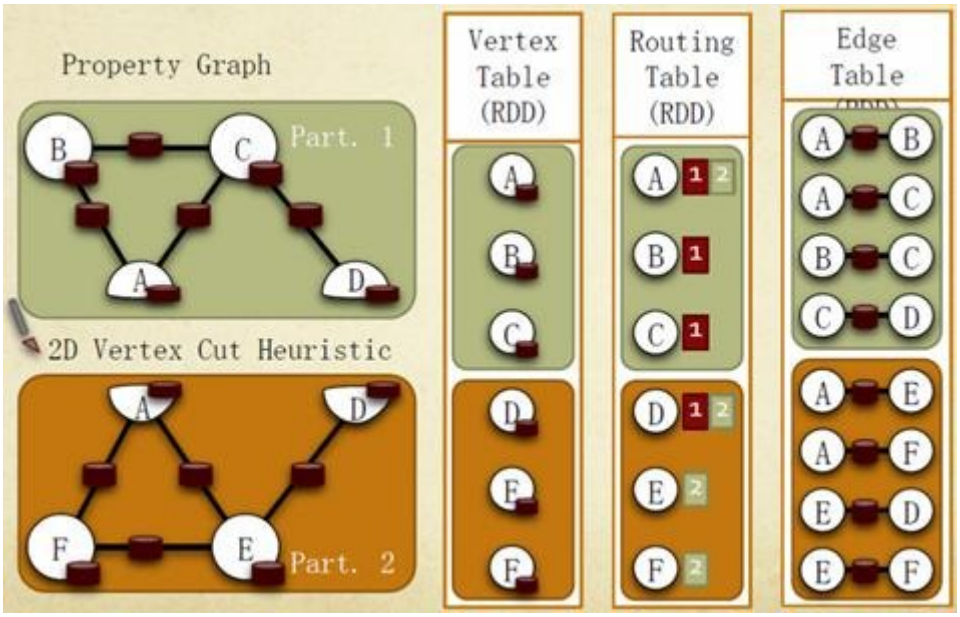
GraphX 存储模式

Graphx 借鉴 PowerGraph，使用的是 Vertex-Cut(点分割)方式存储图，用三个 RDD 存储图数据信息：

VertexTable(id,data)：id 为 Vertexid，data 为 Edgedata

EdgeTable(pid,src,dst,data)：pid 为 Partionid，src 为原定点 id，dst 为目的顶点 id

RoutingTable(id,pid)：id 为 Vertexid，pid 为 Partionid



GraphX 计算模式

GraphX 的 Graph 类提供了丰富的图运算符，大致结构如下图所示。可以在官方 GraphXProgrammingGuide 中找到每个函数的详细说明。

对 Graph 视图的所有操作，最终都会转换成其关联的 Table 视图的 RDD 操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性：Immutable、Distributed 和 Fault-Tolerant，其中最关键的是 Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。

两种视图底层共用的物理数据，由 RDD[Vertex-Partition]和 RDD[EdgePartition]这两个 RDD 组成。点和边实际都不是以表 Collection[tuple]的形式存储的，而是由 VertexPartition/EdgePartition 在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在 RDD 转换过程中是共用的，降低了计算和存储开销。

图的缓存机制：每个图是由 3 个 RDD 组成，所以会占用更多的内存。相应图的 cache、unpersist 和 checkpoint，更需要注意使用技巧。出于最大限度复用边的理念，GraphX 的默认接口只提供了 unpersistVertices 方法。如果要释放边，调用 g.edges.unpersist()方法才行，这给用户带来了一定的不便，但为 GraphX 的优化提供了便利和空间。参考 GraphX 的 Pregel 代码，对一个大图，目前最佳的实践是：

```
var g=...
var prevG: Graph[VD, ED] = null
while(...){
  prevG = g
  g = doSomething(g)
  g.cache()
  prevG.unpersistVertices(blocking=false)
  prevG.edges.unpersist(blocking=false)
}
```

根据 GraphX 中 Graph 的不变性，对 g 做操作并赋回给 g 之后，g 已不是原来的 g 了，而且会在下一轮迭代使用，所以必须 cache。另外，必须先用 prevG 保留住对原来图的引用，并在新图产生后，快速将旧图彻底释放掉。否则，十几轮迭代后，会有内存泄漏问题，很快耗光作业缓存空间。

邻边聚合机制：mrTriplets（mapReduceTriplets）是 GraphX 中最核心的一个接口。Pregel 也基于它而来，所以对它的优化能很大程度上影响整个 GraphX 的性能。mrTriplets 运算符的简化定义是：

```
def mapReduceTriplets[A](
  map: EdgeTriplet[VD, ED] =>
Iterator[(VertexID, A)],
  reduce: (A, A) => A)
  : VertexRDD[A]
```

它的计算过程为：map，应用于每一个 Triplet 上，生成一个或者多个消息，消息以 Triplet 关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每一个 Vertex 上，将发送给每一个顶点的消息合并起来。

mrTriplets 最后返回的是一个 VertexRDD[A]，包含每一个顶点聚合之后的消息（类型为 A），没有接收到消息的顶点不会包含在返回的 VertexRDD 中。

Pregel 模式：GraphX 中的 Pregel 接口，并不严格遵循 Pregel 模式，它是一个参考 GAS 改进的 Pregel 模式。定义如下：

```
def pregel[A](initialMsg: A, maxIterations:
  Int, activeDirection: EdgeDirection)(
  vprog: (VertexID, VD, A) => VD,
  sendMsg: EdgeTriplet[VD, ED] =>
Iterator[(VertexID,A)],
  mergeMsg: (A, A) => A)
  : Graph[VD, ED]
```

这种基于 mrTrilets 方法的 Pregel 模式，与标准 Pregel 的最大区别是，它的第 2 段参数体接收的是 3 个函数参数，而不接收 messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个 Ghost 副本收到的消息聚合后，发送给 Master 副本，再使用 vprog 函数来更新点值。消息的接收和发送都被自动并行化处理，无需担心超级节点的问题。

常见的代码模板如下所示：

```
// 更新顶点
vprog(vid: Long, vert: Vertex, msg: Double):
Vertex = {
  v.score = msg + (1 - ALPHA) * v.weight
}
// 发送消息
sendMsg(edgeTriplet: EdgeTriplet[…]):
Iterator[(Long, Double)]
  (destId, ALPHA * edgeTriplet.srcAttr.
score * edgeTriplet.attr.weight)
}
// 合并消息
mergeMsg(v1: Double, v2: Double): Double = {
  v1+v2
}
```

可以看到，GraphX 设计这个模式的用意。它综合了 Pregel 和 GAS 两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的 Pregel 版本是最简单的一个版本。对于复杂的业务场景，根据这个版本扩展一个定制的 Pregel 是很常见的做法。

GraphX 算法工具包：GraphX 也提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持 PageRank、数三角形、最大连通图和最短路径等 6 种经典的图算法。这些算法的代码实现，目的和重点在于通用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。研读这些代码，也是理解 GraphX 编程最佳实践的好方法。

例子：基于 GraphX 的 pagerank 算法

数据集：https://github.com/databricks/spark-training/tree/master/data/graphx

测试数据为顶点数据 graphx-wiki-vertices.txt 和边数据 graphx-wiki-edges.txt，graphx-wiki-vertices.txt 顶点格式为顶点编号和网页标题，graphx-wiki-edges.txt 边数据由两个顶点构成。

示例代码：

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

object PageRank {
def main(args: Array[String]) {
//屏蔽日志
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

//设置运行环境
val conf = new SparkConf().setAppName("PageRank").setMaster("local")
val sc = new SparkContext(conf)
```

```
//读入数据文件
val articles: RDD[String] =
sc.textFile("/home/Hadoop/IdeaProjects/data/graphx/graphx-wiki-vertices.txt")
val links: RDD[String] =
sc.textFile("/home/hadoop/IdeaProjects/data/graphx/graphx-wiki-edges.txt")

//装载顶点和边
val vertices = articles.map { line =>
val fields = line.split('\t')
(fields(0).toLong, fields(1))
}

val edges = links.map { line =>
val fields = line.split('\t')
Edge(fields(0).toLong, fields(1).toLong, 0)
}

//cache 操作
//val graph = Graph(vertices, edges,
"").persist(StorageLevel.MEMORY_ONLY_SER)
val graph = Graph(vertices, edges, "").persist()
//graph.unpersistVertices(false)

//测试
println("**********************************************************
")
println("获取 5 个 triplet 信息")
```

```
println("*************************************************************
")
graph.triplets.take(5).foreach(println(_))

//pageRank 算法里面的时候使用了 cache()，故前面 persist 的时候只能使用 MEMORY_ONLY
println("*************************************************************
")
println("PageRank 计算，获取最有价值的数据")
println("*************************************************************
")
val prGraph = graph.pageRank(0.001).cache()

val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {
(v, title, rank) => (rank.getOrElse(0.0), title)
}

titleAndPrGraph.vertices.top(10) {
Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))

sc.stop()
}
}
```