计算机体系结构课程实验报告

赵悦棋 517021910158

1. 实验环境

a) 处理器：2.3GHz Dual-Core Intel i5-7360U / 3.59GHz 6-Core AMD Ryzen 3500X
b) 内存：8GB 2133MHz LPDDR3 / 16GB 3600MHz DDR4
c) 操作系统：macOS 10.15.5 / Windows 10 Professional 2004
d) 监测工具：Activity Monitor / Performance Monitor
e) JDK：1.8.0_251
f) Scala：2.12.11
g) Spark：2.4.6
h) SBT：1.13.12

2. 环境搭建

初次搭建实验环境选择的是 Windows 平台，使用了 Win10 2004，但是在根据教程搭建环境的过程中遇到了诸多问题，比如：教程的时效性不强，本着用新不用旧的心态我一开始配置了最新版的 Scala 2.13，但后来发现即使是最新的 Spark 3.0 也只支持到 Scala 2.12，首次回滚；回滚完之后根据教程初始化了项目，但是在国内使用 Maven 拉取依赖速度非常慢，于是配置了 OpenWrt 的路由器做透明代理；拉取完依赖注意到 IDEA 自带的 Scala Archetype 已经很老了，自动生成的 pom 文件中的仓库地址直接访问会得到 404，于是照着教程覆盖了 pom 文件；为了尽快搭建出环境来此时我已经回滚 Scala 到和教程中一模一样的版本，并使用了教程中的 pom 文件，然后又注意到 pom 文件中使用 Scala 版本变量来拉取依赖，但是由于 Spark Core 同时兼容 Scala 2.10 和 2.11，所以依赖在仓库中的名称已经更新为 2.11 结尾，导致拉取依赖又出错，再次修改；花了很多时间才真的让项目导入完毕，但写了 demo 程序发现根本跑不起来……

在尝试了网络上的各种教程后我仍无法使用 Maven 在 Windows 平台搭建环境，于是转投了 macOS 平台，并改用 SBT 作为包管理工具，有了先前配置的路由器，通过 brew 安装 SBT 十分顺利，并且没有奇怪的依赖错误，最终我在 macOS 平台完成了实验环境的搭建，并成功把相同的工程文件在 Windows 环境运行，为了测试环境运行正常我根据教程中的代码写了 WordCount 来初上手 Spark 和 Scala，运行情况良好。

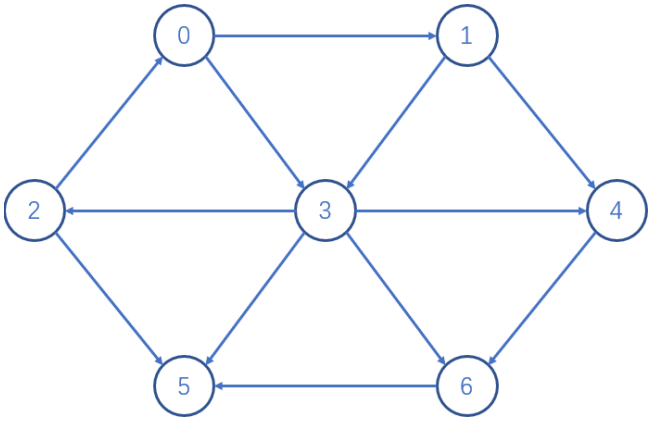
Spark 在运行过程中会有大量的 INFO 输出，为了在撰写报告的时候能截到比较美观的输出，需要在工程文件中手动引入 Sprak 默认的配置文件，并将输出从 INFO 修改为 ERROR，这样可以隐藏所有的运行信息仅保留我的代码输出。

```
# Set everything to be logged to the console
log4j.rootCategory=ERROR, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n
```

3. 实验一

单源最短路径问题指的是给定一个含权有向图，并指定其中一个顶点为源，计算从源到其他所有顶点的最短路径长度的问题。根据 GraphX 文档中 Graph Builders 板块的

指导，我为下图写了一个可以被 GraphLoader.edgeListFile 加载的文件（data/Graph.txt）。



由于 GraphLoader.edgeListFile 规定了格式只能是源点和终点的组合，我无法在使用它加载图的条件下为每一条边增加权重，因此所有的边的权重都是默认的 1。通过打印 iptGraph 的内容，验证了自定义的图已经被正确地加载。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_251.jdk/Contents/Home/bin/java ...
[info]<--------Edge Information------> [info]<-------Vertex Information----->
Edge(0,1,1)                           (4,1)
Edge(0,3,1)                           (0,1)
Edge(1,3,1)                           (6,1)
Edge(1,4,1)                           (2,1)
Edge(2,0,1)                           (1,1)
Edge(2,5,1)                           (3,1)
Edge(3,2,1)                           (5,1)
Edge(3,4,1)
Edge(3,5,1)
Edge(3,6,1)
Edge(4,6,1)
Edge(6,5,1)
```

接下来就要解决 SSSP 问题，解 SSSP 用到了 Dijkstra 算法的思想。在原图的基础上生成一个新的图，每个顶点的权重用来记录从源点到该顶点的最短路径，因此在初始化阶段除了源点外其余所有顶点的权重均设置为 INF（取 Int 的最大值）。然后利用 Pregel 开始求解，核心思路就是对于每一个顶点，如果从我出发到另一个顶点的路径长度比它本身记录的路径长度还要短，那么就更新那个顶点的属性，直到图的状态不再变化为止。

Pregel 的格式为 Pregel(initialMsg)(vprog, sendMsg, mergeMsg)。其中 initialMsg 部分为初始化消息，这个消息会发给每一个顶点用于初始化属性，因此也为 INF；vprog 根据收到的消息更新属性，根据核心思路，此处取最小值；sendMsg 为消息发送函数，该函数的运行参数是一个代表边的上下文，同样的根据核心思路可以写出条件语句，但是注意由于迭代并不是从源点开始，且 INF 并不是真的无穷大而是整形的最大值，因此**对于当前权重为 INF 的顶点必需跳过，否则加上边的权重之后会溢出变为负边，导致最短路径可以到无穷小（INT_MIN）的错误，并且在图较大的时候可以导致卡死**；mergeMsg 是收到多条消息时的合并逻辑，不难想到收到多条消息时应选择其中值最小的接收。最终输出如上图所示，笔头验证无误，详见 SSSP.scala 中的代码和注释。

```
[info]<---------Shortest Path-------->
(4,2)
(0,0)
(6,2)
(2,2)
(1,1)
(3,1)
(5,2)
```

4. 实验二

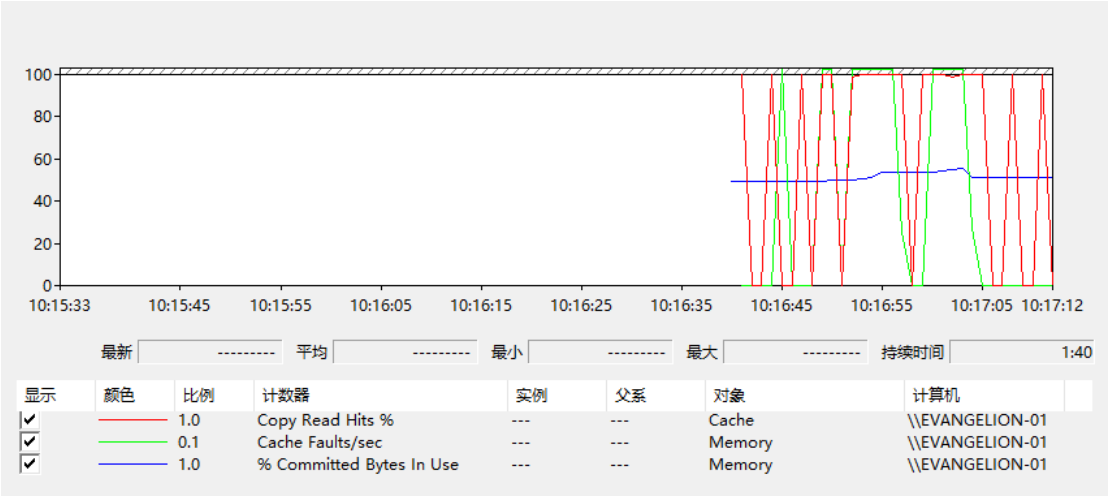
开始前先把 PageRank 写出来，好在允许使用 org.apache.spark.graphx.lib，在加载完图之后可以用自带的算法直接实现 PageRank，其中代表迭代停止标志的 tol（两次迭代 Rank 之差的绝对值）我设置为 0.0001，这样可以让迭代的总次数相对较大。使用我自定义的图测试下来没有问题，若使用给出的两个数据集，需要**禁止 Spark 的 INFO 输出否则会因为大量的日志导致运行效率低下甚至卡死**。

在 macOS 平台，由于性能监测生态不是很完善，而且权限管控严格，第三方软件很难直接获得系统数据，在多次尝试后发现大部分软件记录的内存图为内存压力图，和内存占用图并不一致，不论是跑 Google 图还是 Wikipedia 图均不会导致爆内存，因此**内存压力图在任何一组测试中均为接近水平的直线**。为了更好地监测内存，我采用了如下方式：调整系统 Activity Monitor 的数据采集间隔为 1 秒，在每次运行程序前退出全部 java 进程，**开启录屏，运行程序，记录所有 java 进程的内存占用，根据视频时间轴誊写进 Excel 后作图**。得到结果如果下：

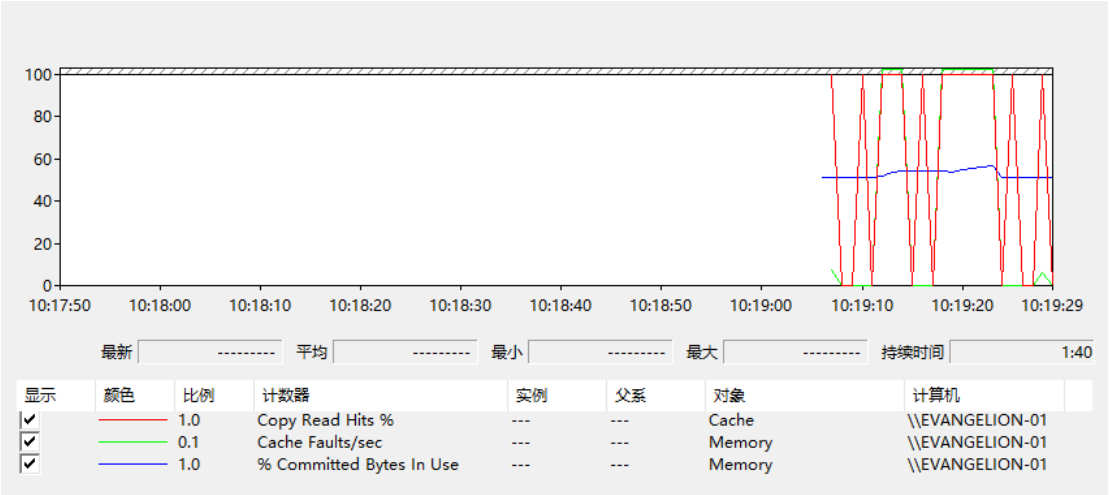
从图中可以看出，两种程序的内存初始占用量均为 0MB，与 Java 冷启动的实验设计吻合；两者均在启动后 9-10 秒达到第一个峰值，从运行时录制的视频可以看出这一

段是 Java 环境初始化以及 Scala 程序初始化的过程，第一次的峰值对应加载完图的时间点；在峰之后紧跟着一个低谷，推测是图加载完毕后释放了一些临时资源；之后内存占用资源缓慢攀升，该段对应 SSSP 的 Pregel 和 PageRank 的迭代过程；最后得出结果释放资源，出现快速内存用量的快速下降；由于 Java 不会主动结束全部进程，最后维持了一个空载的状态，呈水平线。

在 Windows 平台，采用如下方式：开启 Performance Montior，在计数器中添加 Copy Read Hits、Cache Faults/sec、Committed Bytes In Use，待图像撑满画面后开始运行程序并截图。得到结果如下：



Windows 平台对 Wiki 图运行 SSSP



Windows 平台对 Wiki 图运行 PageRank

由于 Performance Monitor 无法仅针对某个程序使用的资源进行监控，因此不能排除其它进程占用的内存基数，因此 Windows 平台的内存线并不从源点开始。可以看到内存占用的曲线与 macOS 下手动记录的大致一致，因此相关分析不再赘述。可以注意到缓存命中率在运行过程中的变动非常剧烈，上升和下降都十分极速，但可以注意到在 SSSP 和 PageRank 运行时缓存命中率均存在两段持续的时间维持在 100%，即一条水平线段，推测同样是加载图后读取图以及 SSSP 的 Pregel 和 PageRank 两段过程中，由于处理的数据是固定的图，并且 Ryzen 3500X 有 32MB 的 L3 缓存，3MB 的 L2 缓存和 384KB 的 L1 缓存，因此可以容纳大量的数据放在高速缓存中，导致在 1 次 Cold Miss（快速下降）后可以持续命中（快速上升到封顶）。比较意外的是 Cache Faults/sec 的线和 Cache Read Hits 的曲线几乎重合，但是广泛地搜索后并没有找到关于 Cache Faults/sec

的具体定义，可能和我本来以为的不命中有所出入，导致结果并不符合预期，在之后的实验中将不再讨论该参数。

5. 实验三

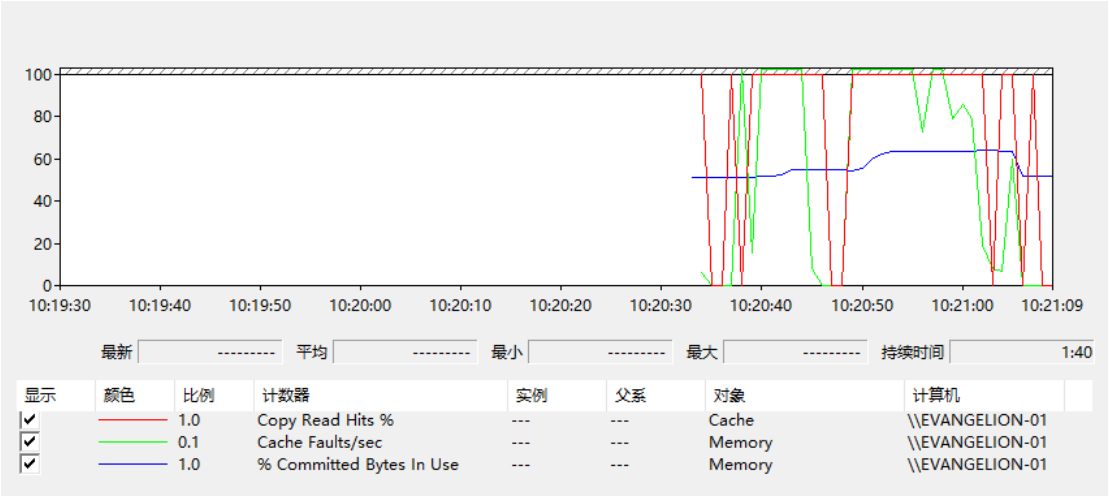
同实验二，将数据集替换为 Google 数据集后重新在两个平台分别运行了 SSSP 和 PageRank，十分幸运地没有遇到内存不够的情况，避免了数据分割。

在 macOS 平台，测得 SSSP 的内存占用与 PageRank 的内存占用分别如下：

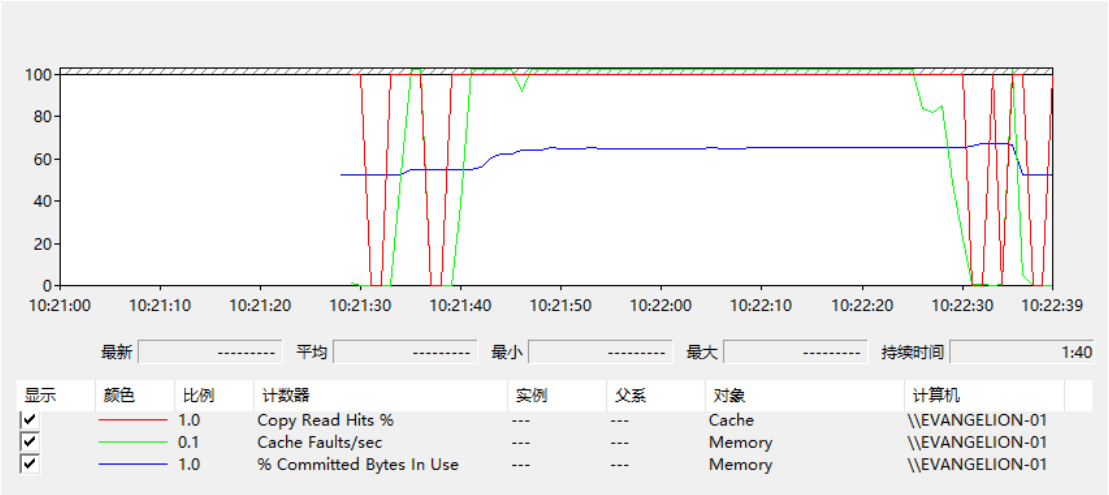
从图中可以看出，两种程序的内存初始占用量仍为 0MB，再次验证了 Java 冷启动的状态；两者均在 5-10 秒达到第一个峰值，由于时间轴密度关系峰在 SSSP 的体现可能不是特别的明显，这一部分我认为仍对应 Java 环境和 Scala 程序的初始化过程；二者均在 14-15 秒达到第二个高峰，同 Wiki 图，第二此峰值对应加载图的过程；在高峰过后紧跟着一个低谷，同 Wiki 图，推测是对应加载完毕图之后释放临时资源的过程，由于 SSSP 和 PageRank 到这一步为止的行为几乎一致，因此二者在前 15 秒的内存占用图高度相似；二者在图加载完毕后的一段时间内内存占用量快速上升，并且上升速度不断放缓最后趋于平稳，对于 SSSP，快速上升的部分可能对应的是初始阶段每一次迭代都会产生大量的消息，随着图区域稳定，消息减少，内存的新增占用量也就越来越少，最后图

达到稳定后内存占用几乎不再变化，对于 PageRank，快速上升的部分可能对应的是迭代初始阶段两次 Rank 的差额的绝对值远高于预定义的 tol，因此每次有大量的顶点属性需要被修改，后期 Rank 在迭代的过程中的变化会越来越小，因此需要修改的顶点数下降，内存占用趋于平稳；最后得出结果后释放资源，出现内存用量的快速下滑，恢复到 Java 默认运行状态后不再变化，呈水平线。

在 Windows 平台，测得情况如下：



Windows 平台对 Google 图运行 SSSP



Windows 平台对 Google 图运行 PageRank

同样可以看到，内存占用曲线与 macOS 下手动记录的十分接近，此处不再赘述。缓存命中率也与之前的 Wiki 图类似，大致分为两段，分别对应图加载和 PageRank 算法运行的过程，第一段从 0 快速上升到 100 和加载图的 Cold Miss 有关，第一段与第二段中间的命中率的急速下降应该和进入 PageRank 算法出现 Cold Miss 有关，大缓存使得在 PageRank 算法长时间的运行过程中始终维持了 100%的命中率。

6. 问答题

- SSSP 的运行流程与算法设计：
  a) 使用 GraphLoader 加载图
  b) 指定源点的 VertexId
  c) 利用加载的图生成新的图，源点权重置为 0，其余置为 INF（INT_MAX）
  d) 利用 Pregel 计算 SSSP（详见实验一）

大部分设计在实验一中已经阐述清楚，此处扩充一下 sendMsg 的部分，在实验一中有提到该函数的运行参数是一个代表边的上下文 triplet，此处以边代称。若边的源点权重为 INT_MAX 也就是所谓的 INF，则必须跳过该边，因为在加载图的过程中由于不支持指定边的权重，所有的边默认权重为 1，而在核心思路中必须先计算从源点出发到目标点的距离，对 INT_MAX 执行加 1 操作将溢出得到 INT_MIN，使得边的权重从 1 变成了负无穷大，这样的跳过操作会在遇到源点时停止，所以并不会有意外发生。对于满足安全条件的边，若源点加边长小于目标点当前的权重，意味着经过源点过去更快，于是向目标点发送消息让它更新，否则什么事都不做。若干次次迭代后图上所有顶点的权重将不再变化，即任何一条边都不满足源点加边长小于目标点权重，不难推出此时所有顶点的权重就是 SSSP 的结果。

- PageRank 的运行流程与算法设计：
  a) 使用 GraphLoader 加载图
  b) 调用 GraphX 库中自带的 PageRank 算法，设置 tol 为 0.0001

  因为用了库的算法所以没有特别的实现方面的内容可以讲，主要谈一下我对 tol 这个变量的理解。在 PageRank 算法中每一轮迭代中当前顶点的权重都为所有指向自己的顶点在上一轮迭代中的权重值与他们的出度的商之和，因此两轮迭代的数据会存在一个差值。控制迭代次数的方式有很多，比如在循环体的条件中写明循环 100 次退出，而 GraphX 的库则是用两次迭代间的插值范围来控制迭代次数，只要变化小于 0.0001 就停止，这样就可以动态的控制迭代次数并且保证了精度，是十分精妙的设计。
- 实验二与实验三的比较：

  相同程序，Google 图比 Wiki 图要耗费更长的时间，占用更多的内存资源；相同的图，PageRank 要比 SSSP 耗费更长的时间，占用更多的内存资源。前者比较好理解，从外部数据来看 Wiki 的图只有 1MB，而 Google 的图高达 70MB，在 push 到仓库的过程中甚至触发了 GitHub 的大文件警告，会更耗费资源是完全可预期的。后者我觉得原因会多一些，一个是因为 SSSP 的结果是固定的，不存在精度的问题，因此运行时间相对固定，并且因为有消息机制，在运行过程中并不需要完整记录上一轮迭代的图的状态，而 PageRank 受到精度参数的控制，时间和资源的开销弹性较大，并且从它的计算方式可以看出每一轮迭代都完整依赖上一轮迭代的结果因此时刻要保存两份图的状态，这明显会更加耗费资源。
- 遇到的困难和解决方式：

  遇到最大的困难就是环境的搭建了，解决办法在环境搭建中已经阐明。代码实现的部分基本没有遇到特别大的困难，因为有一个自定义的很小的数据集，因此 Debug 的成本也没有很高，主要的问题就是权重溢出前面也已经多次提到。未提及的最大的困难就是在 macOS 下的性能监测，由于 macOS 的生态问题我尝试了很多的软件都达不到理想的效果，最理想的选择是 PCM 但是配置过于复杂经过多次尝试并没有成功，最终解决方案是采用实施录制占用率并根据时间轴手动填写数据到表格中的方式完成了制图，并且好在我在 macOS 下搭建的项目在 Windows 平台成功地运行了，使得可以有双平台的数据互相印证，总体完成度得到了保证。