开源地址：https://github.com/Vincent-Huang-2000/DATA301
感觉不错的话，可以在 Github 上给个 Star

本文档未完结，最新修订日期：2023年6月21日

Lab 0

Assuming we have 4 processing cores available, which of the following are true about the state of the Spark RDD after this line is executed (from the Background 1 lab section):

```
pA = sc.parallelize(A)
```

Select one or more:

a.pA is a pointer on the client to an in memory copy of all elements of A

b.data from A has been split into 4 equal sized chunks and each chunk transmitted over the network from client to different processing core

c.pA is a pointer on the processing cores to copies of A

d.pA is a pointer on the client to the addresses of the cores which contain chunks of A

e.each processing core gets a reference to A to request code to run on

解析：对于选项b，当我们执行"sc.parallelize(A)"，Spark会尝试将数据A分成若干个部分（在这个例子中是4个，因为有4个处理核心），并将这些部分发送到集群中的不同节点。因此，b选项是正确的。对于选项d，"pA"实际上是一个引用，或者说指针，指向集群中的数据。这个指针存在于客户端，但是它实际上是指向集群中各节点上的数据分片的。因此，d选项也是正确的。

选项a，c和e都有些误导。对于a和c，Spark并不会在客户端或处理核心上创建A的完整副本。相反，A的数据被分片并分布在集群中。对于e，每个处理核心并不会得到对A的引用以请求运行代码，而是会收到一部分的A的数据来进行处理。

What is a lambda expression?

Select one:

a.a hint that makes the python interpreter compile the code into assembly language

b.a mathematical symbol for summation

c.an expression that tells Spark to parallelize the code

d.an anonymous (unnamed) function that has one line of code

What does the flatMap function do?

Select one:

a.applies a function to a dataset and produces 0, 1, or many outputs for each input

b.transforms a nested dataset into a linear dataset

c.applies a dataset to a function and produces a new function

d.combines 0, 1, or many inputs into a map structure

What does the reduceByKey function do?

Select one:

a.Hashes each value and counts the values with the same hash

b.Hashes each key to a bucket and applies the reduction function to values that end up in the same bucket

c.Reduces each key to a single value

d.Combines each key and value to produce a new key and value pair

解析：
正确答案是b。在Spark中，reduceByKey函数将数据集中具有相同键的元素组合在一起，并使用指定的reduce函数对这些元素进行处理。这个过程可以看作是先将相同键的值哈希到一个桶中，然后应用reduce函数（比如求和，最大值，最小值等）对同一个桶中的所有值进行处理。

选项a和c并不准确。reduceByKey并不对值进行哈希计数（这更像是countByKey的功能），也不会把每个键简化为一个单一的值。相反，它将同一个键的所有值简化为一个值。

选项d描述的更像是map函数的功能，它接收一个键值对并返回一个新的键值对，而reduceByKey的目标是将同一键的值简化为一个值。

知识点补充1

reduceByKey 和 countByKey 都是 Apache Spark 中操作键值对 RDD（Resilient Distributed Dataset）的方法，但他们的功能和用途有显著的不同。

reduceByKey(func): 这个操作会对所有拥有相同键的元素应用一个指定的归约函数func，返回一个新的键值对RDD，其中的每一个键都对应一个值，这个值是原始RDD中与该键相关的所有值通过归约函数计算得到的结果。这个归约函数必须是可交换和可结合的，比如加法或乘法。

例如，如果我们有一个键值对RDD如下：

```
rdd = [("apple", 1), ("banana", 2), ("apple", 3), ("orange", 4),
("banana", 5), ("orange", 6)]
```

我们可以使用reduceByKey对所有的值进行求和：

```
rdd.reduceByKey(lambda a, b: a + b)
```

这将会返回一个新的键值对RDD：

```
[("apple", 4), ("banana", 7), ("orange", 10)]
```

countByKey(): 这个操作会返回一个字典，字典的键就是原始RDD中的键，字典的值是原始RDD中与该键相关的元素的数量。这个操作通常在原始RDD的数据量不大，能够在驱动程序中处理的情况下使用。

例如，如果我们有一个键值对RDD如下：

```
rdd = [("apple", 1), ("banana", 2), ("apple", 3), ("orange", 4),
("banana", 5), ("orange", 6)]
```

我们可以使用countByKey来计算每个键出现的次数：

```
rdd.countByKey()
```

这将会返回一个字典：

```
{"apple": 2, "banana": 2, "orange": 2}
```

所以，reduceByKey和countByKey的主要区别在于，reduceByKey对同一键的值应用一个归约函数，而countByKey计算的是每个键出现的次数。

groupByKey() 是Apache Spark中操作键值对RDD（Resilient Distributed Dataset）的一个方法。这个操作会根据键将数据进行分组，返回一个新的键值对RDD，每个键对应一个迭代器，包含了原始RDD中与该键相关的所有值。

例如，如果我们有一个键值对RDD如下：

```
rdd = [("apple", 1), ("banana", 2), ("apple", 3), ("orange", 4),
("banana", 5), ("orange", 6)]
```

我们可以使用groupByKey来将值按键分组：

```
rdd.groupByKey()
```

这将会返回一个新的键值对RDD：

```
[("apple", <iterator [1, 3]>), ("banana", <iterator [2, 5]>), ("orange",
<iterator [4, 6]>)]
```

这里的每一个迭代器代表了原始RDD中与该键相关的所有值。

groupByKey 的用途主要在于对同一键的所有值进行一些复杂的操作，这些操作可能无法通过reduceByKey来实现。例如，我们可能需要将同一键的所有值排序，或者将他们转换为一个列表或集合。

然而，需要注意的是，groupByKey 可能会导致大量的数据在网络之间进行传输，这可能会对性能产生负面影响。因此，如果可能，推荐优先使用 reduceByKey 或 aggregateByKey，这些操作可以在各个节点上对数据进行局部归约，减少数据传输量。

知识点补充2

当我们在Spark中调用 reduceByKey 操作时，首先会在每个分区内部进行局部归约。这就意味着如果我们有一个键在同一分区内有多个值，那么这些值会首先在他们所在的分区内进行归约操作，生成一个中间结果。然后，这些中间结果会被发送到其他节点上，进行全局的归约操作。这种方式可以减少网络传输的数据量，因为我们只需要传输每个键的中间结果，而不是所有的值。

举个例子，假设我们有以下的键值对数据，并且这些数据被分布在两个节点（Node1和Node2）上：

```
Node1: [("apple", 1), ("banana", 2), ("apple", 3)]
Node2: [("apple", 4), ("banana", 5), ("orange", 6)]
```

如果我们调用reduceByKey(lambda a, b: a + b)，首先会在每个节点内部进行局部归约：

```
Node1: [("apple", 4), ("banana", 2)]
Node2: [("apple", 4), ("banana", 5), ("orange", 6)]
```

然后，这些中间结果会被发送到其他节点上，进行全局的归约操作：

```
Result: [("apple", 8), ("banana", 7), ("orange", 6)]
```

可以看到，我们只需要传输每个键的中间结果，而不是所有的值，这大大减少了网络传输的数据量。

而对于 groupByKey，它不进行局部归约，而是将所有的键值对全部发送到其他节点上，然后在那里进行分组操作。

假设我们有以下的键值对数据，这些数据被分布在两个节点（Node1和Node2）上：

```
Node1: [("apple", 1), ("banana", 2), ("apple", 3)]
Node2: [("apple", 4), ("banana", 5), ("orange", 6)]
```

如果我们调用groupByKey()，那么会在各个节点上分别进行分组操作

然后，在每个节点上进行分组操作：

```
Node1: [("apple", <iterator [1, 3]>), ("banana", <iterator [2]>)])

Node2: [("apple", <iterator [4]>), ("banana", <iterator [5]>),
("orange", <iterator [6]>])
```

对于"apple"这个键，节点Node1和Node2都有关于"apple"的数据，所以在这两个节点上都会进行分组操作。然后，这两个节点上的结果（即每个键对应的值的迭代器）会被发送回驱动程序节点（主节点）。在驱动程序节点上，会将所有的结果合并在一起，形成最终的结果。

这种在各个节点上进行分组，然后在驱动程序节点上进行合并的过程，有时也被称为shuffle。

Lab 1

Spark's functions count, min, max are examples of

Select one:

a.lambdas

b.datasets

c.actions

d.transformations

相关知识点：

RDD提供了两种类型的操作：transformation 和 action

所有的transformation都是采用的懒策略，如果只是将transformation提交是不会执行计算的，计算只有在action被提交的时候才被触发。

1）transformation操作：得到一个新的RDD，比如从数据源生成一个新的RDD，从RDD生成一个新的RDD

2）action操作：action是得到一个值，或者一个结果（直接将RDD cache到内存中）

一些教程：https://blog.csdn.net/qq_16365849/article/details/50698261

几个常用 PySpark 函数的使用（查看更多）

count() 返回 RDD 中的所有元素的个数。

take(n) 返回 RDD 的前 num 个元素。

takeOrdered(num, key=None) 从按升序排列或按可选键函数指定的顺序排列的 RDD 中获取 N 个元素。

```
sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6,
key=lambda x: -x)
# [10, 9, 7, 6, 5, 4]
```

top() 从 RDD 中获取前 N 个元素。(即降序)这会返回按降序排序的列表。

```
sc.parallelize([10, 4, 2, 12, 3]).top(1)
# [12]
sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
# [6, 5]
sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
# [4, 3, 2]
```

reduce() 函数从 RDD 中取两个元素作为输入参数，然后输出与输入元素类型相同的结果。我们可以往 RDD 中添加元素，然后统计单词数量。它接受交换与结合运算作为参数。

```
from operator import add
sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
# 15
```

takeSample() 操作返回一个数组，其中包含来自数据集的随机元素样本。

intersection() 求交集

```
x = sc.parallelize(['A','A','B'])
y = sc.parallelize(['A','C','D'])
z = x.intersection(y)
print(z.collect())
# ['A']
```

Code such as

```
(access_logs
             .map(lambda log: (log.endpoint, 1))
             .reduceByKey(lambda a, b : a + b)
             .cache())
```

Select ALL that are correct

Select one or more:

a.modifies the access_logs dataset

b.is a short form code style used instead of using temporary variables on each line to store partial results

c.specifies several transformations and allows Spark to figure out what order to apply them

d.chains together output of one transformation to input of the next transformation

以下是每个选项的解析：

a. 这个答案是错误的。Spark的操作分为两类：转换操作和行动操作。转换操作（如map和reduceByKey）都是惰性的，它们并不会立即执行，也不会修改原始的数据集。在这段代码中，access_logs 数据集并没有被修改。

b. 这个答案是正确的。这段代码是一种流畅的编程风格（fluent style），它允许我们将多个操作链接在一起，而无需在每一步都创建临时变量。这可以让代码更加简洁和易读。

c. 这个答案是错误的。虽然这段代码指定了几个转换操作，但是Spark并不会改变它们的顺序。这是因为每个转换操作的结果都依赖于前一个操作，所以必须按照特定的顺序执行。

d. 这个答案是正确的。这段代码的确将一个转换操作的输出链到下一个转换操作的输入。例如，map操作的输出成为了reduceByKey操作的输入，reduceByKey操作的输出又成为了cache操作的输入。

Lab2

无选择题

Lab 3

In Market Basket Analysis we might be given the following data:

B1 = {m, c, b} B2 = {m, p, j}

B3 = {m, b} B4= {c, j}

B5 = {m, p, b} B6 = {m, c, b, j}

B7 = {c, b, j} B8 = {b, c}

Which is the best term to describe what B4 is?

Select one:

a. algorithm

b.market

c.basket

d.item

Which is the best term to describe what j is?
Select one:

a. item

b.customer

c.basket

d.market

Apriori算法是一种经典的关联规则学习算法，常用于挖掘交易数据中的频繁项集，也就是在交易中经常一起出现的物品组合。Apriori算法基于一个简单的原则：如果一个物品集是频繁的，那么它的所有子集也一定是频繁的。相反，如果一个物品集是非频繁的，那么它的所有超集也一定是非频繁的。

以下是一个简单的例子来解释Apriori算法：

假设我们有以下的交易数据：

```
T1: {牛奶,面包,尿布}
T2: {可乐,面包,尿布,啤酒}
T3: {牛奶,尿布,啤酒,鸡蛋}
T4: {面包,牛奶,尿布,啤酒}
T5: {面包,牛奶,尿布,可乐}
```

首先，我们需要设置一个最小支持度，假设是60%。支持度是一个物品集在所有交易中出现的次数与总交易数的比值。只有支持度大于或等于最小支持度的物品集才被认为是频繁的。

然后，我们先找出所有单个物品的支持度，然后剔除掉支持度小于最小支持度的物品。在这个例子中，所有单个物品（如牛奶，面包，尿布等）的支持度都大于60%，所以都保留下来。

接着，我们找出所有两个物品的组合的支持度，然后剔除掉支持度小于最小支持度的组合。在这个例子中，{牛奶,尿布}，{面包,尿布}和{牛奶,面包}的支持度都大于60%，所以都保留下来。

然后，我们找出所有三个物品的组合的支持度，然后剔除掉支持度小于最小支持度的组合。在这个例子中，只有{牛奶,面包,尿布}的支持度大于60%，所以只保留这个组合。

最后，我们得到的频繁项集就是{牛奶}，{面包}，{尿布}，{牛奶,尿布}，{面包,尿布}，{牛奶,面包}和{牛奶,面包,尿布}。

B1 = {m, c, b} B2 = {m, p, j}

B3 = {m, b} B4= {c, j}

B5 = {m, p, b} B6 = {m, c, b, j}

B7 = {c, b, j} B8 = {b, c}

Given the above records B1 through B8, what is the interest that {m} -> {p}?

这里老师给的答案是 0.15（这题不懂先看后面的例题 ）

先计算 conf({m} -> {p}) = 2 / 5 = 0.4

Pr({p}) = 2 / 8 = 0.25

0.4 - 0.25 = 0.15

Support, Confidence, Interest

Support for itemset I: Number of baskets containing all items in I

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Support of
{Beer, Bread} = 2

{Beer, Bread} 这个组合在所有的 backets 中出现了2次，所有 support 为2

- **Confidence** of this association rule is the probability of j given $I = \{i_1,\ldots,i_k\}$

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

- **Interest** of an association rule $I \rightarrow j$: difference between its confidence and the fraction of baskets that contain j
  - Interesting rules are those with high positive or negative interest values (usually above 0.5)

$$\text{Interest}(I \rightarrow j) = \text{conf}(I \rightarrow j) - \Pr[j]$$

例题：

B1 = {m, c, b} B2 = {m, p, j}
B3 = {m, b} B4= {c, j}
B5 = {m, p, b} B6 = {m, c, b, j}
B7 = {c, b, j} B8 = {b, c}

Association rule: {m, b} →c
Confidence = 2/4 = 0.5
Interest = |0.5 – 5/8| = ⅛

这个问题涉及到数据挖掘中的关联规则学习，特别是关于关联规则的"置信度"和"兴趣度"的计算。

关联规则中的 "置信度" (Confidence) 和 "兴趣度" (Interest) 是评估关联规则有效性的两个重要指标。

在这个特定的问题中，我们有一个关联规则 {m, b} → c，以及一组集合 B1, B2, ..., B8。

- 置信度 (Confidence) 的计算是基于规则的前项 {m, b} 在所有集合中出现的频率，以及规则的全部项 {m, b, c} 在所有集合中出现的频率。在这个问题中，前项 {m, b} 在 B1, B3, B5, B6 中出现，所以出现频率为 4。全部项 {m, b, c} 在 B1 和 B6 中出现，所以出现频率为 2。因此，置信度为 2/4 = 0.5。

- 兴趣度 (Interest) 公式。

$$\text{Interest}(I \rightarrow j) = \text{conf}(I \rightarrow j) - \Pr[j]$$

Confidence({m, b} →c) = 0.5,

{c} 在 B1, B4, B6, B7, B8 中出现，出现频率为 5/8。
所以 Pr({c}) = 2/8 = 0.25
因此，兴趣度为 |0.25 - 0.5/8| = 1/8。

The A-Priori algorithm is a two-step algorithm. Why is it necessary to perform two steps instead of doing it all in one step?

a.the dataset is too large to process in one step

b.first we have a map and then a reduce

c.pyspark doesn't automatically transfer data to each worker node so we must broadcast it first

d.we need to output pairs so that requires two items and two steps

e.we don't know what items are frequent until we've read through the whole dataset once

A-Priori 算法是一种用于寻找频繁项集的算法，常用于关联规则学习中。

在算法的第一步，算法会通过数据集，计算各个项目（即单项集）的支持度（出现的频率），然后根据用户设置的最小支持度阈值，筛选出频繁的项目。这个过程称为 "发现频繁项集"。

在第二步，算法会基于第一步发现的频繁项集，生成候选的多项集（如对，三项集等），然后再通过数据集，计算这些多项集的支持度，并同样根据最小支持度阈值，筛选出频繁的多项集。这个过程我们称为 "生成关联规则"。

所以说，A-Priori 算法的两步操作主要是为了发现频繁项集并基于这些项集生成关联规则，而非由于数据集过大（选项 a）、需要映射和归约操作（选项 b）、Pyspark 的数据传输问题（选项 c）或需要输出对（选项 d）。

Lab 4

协同过滤（Collaborative Filtering）是一种推荐系统算法，其目标是根据用户的历史行为和兴趣，预测他们可能喜欢的其他项目或产品。协同过滤算法的基本思想是通过分析用户与项目之间的关系来进行推荐，而不依赖于项目的属性或用户的个人信息。

在协同过滤中，主要有两种类型的协同过滤方法：基于用户的协同过滤和基于项目的协同过滤。

1. 基于用户的协同过滤（User-Based Collaborative Filtering）：

2. 基于项目的协同过滤（Item-Based Collaborative Filtering）：

协同过滤的优点是可以在不需要事先了解项目或用户的特征的情况下进行推荐，只需要用户的历史行为数据即可。然而，协同过滤算法也存在一些挑战，例如稀疏性问题（用户对大多数项目都没有评分）、冷启动问题（新用户或新项目缺乏足够的历史数据）、规模扩展性（随着用户和项目数量的增加，计算相似度变得更加复杂）等。

为了克服这些挑战，研究人员提出了许多改进的协同过滤算法，如基于模型的协同过滤、混合方法和基于内容的推荐等。这些方法可以结合其他信息（如项目的属性、用户的个人信息）来提高推荐的准确性和多样性。

Cosine similarity is defined as

$$cos\text{-}sim(u,v) = \frac{u \cdot v}{\|u\| \|v\|}$$

Given two identical vectors, such as u = [1, 0, 1] and v = [1, 0, 1] the angle between them is 0 degrees. What is the cosine similarity?

Select one:

a.0

b.1/3

c.pi

d.1

余弦相似度的计算

```
import math

def cosine_similarity(u, v):
    dot_product = sum(i * j for i, j in zip(u, v))
    norm_u = math.sqrt(sum(i * i for i in u))
    norm_v = math.sqrt(sum(i * i for i in v))
    return dot_product / (norm_u * norm_v)
```

假设 u = [1, 2, 3] v = [4, 5, 6]

上面的代码中，计算 dot_product 其实是将两个列表中每一元素都对应地相乘

```
dot_product = [ 1*4, 2*5, 3*6 ]
```

而这行代码 norm_u = math.sqrt(sum(i * i for i in u)) 则是：

计算了向量 u 中每个元素的平方，并且将所有平方值相加。

相当于：

```
norm_u = math.sqrt( 1*1 + 2*2 + 3*3 )
```

这个结果就是向量u的长度或范数。

---

zipWithIndex() 函数

zipWithIndex() 函数的使用，返回值是 pyspark.rdd.RDD[Tuple[T, int]]

```
sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()
# [('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

zipWithIndex 是一个用于给 RDD 或 DataFrame 的每个元素分配唯一索引的函数。它将元素和索引值作为元组进行组合，并返回一个新的 RDD 或 DataFrame。

Given a matrix M=

1	1/3	1/2
0	1/3	0
0	1/3	1/2

and a vector R=

1/3

1/3

1/3

If we multiply them together we will get a result

x

y

z

What is the value of x?

What is the value of x?

Select one:

- a. 1/2
- b. 7/18
- c. 1/3
- d. 11/18 ✔

To find the value of `x`, you would do a matrix-vector multiplication. In this case, you would multiply the elements of the first row of the matrix M by the corresponding elements of the vector R, and then sum the results.

Here is how it is calculated:

x = (1 * 1/3) + (1/3 * 1/3) + (1/2 * 1/3) = 1/3 + 1/9 + 1/6

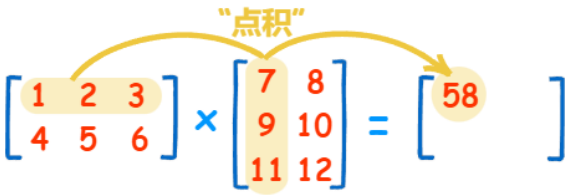
To add these fractions, it's helpful to have a common denominator. The least common multiple of 3, 9, and 6 is 18, so we'll use that:

x = 6/18 + 2/18 + 3/18 = 11/18

So the answer is (d) 11/18.

矩阵乘法

求 **第一行** 和 **第一列** 的答案：



"点积" 是把 **对称的元素相乘**，然后把结果加起来：

$$(1, 2, 3) \cdot (7, 9, 11) = 1\times7 + 2\times9 + 3\times11 = 58$$

我们把第一个元素相配（1 和 7），然后相乘。第二个元素（2 和 9）和第三个元素（3 和 11）也一样，然后把结果加起来。

PageRank

PageRank是由谷歌的创始人拉里·佩奇和谢尔盖·布林在1996年开发的一种算法，用于对互联网上的网页进行排名。

PageRank的核心思想是：一个网页的重要性可以通过查看链接到它的其他网页的重要性来决定。也就是说，如果一个重要的网页链接到了另一个网页，那么被链接的网页就可能也很重要。

PageRank将互联网视为一个有向图，其中的每个网页是一个节点，每个链接是一个有向边。PageRank算法的工作原理是对这个图进行多次迭代，每次迭代都会更新每个节点的PageRank值，直到这些值收敛为一个稳定的值。最后，这些稳定的值就是每个网页的PageRank值，也就是它们的重要性。

PageRank的计算过程可以用一个简单的例子来说明。假设我们有四个网页A、B、C和D。开始时，每个网页的PageRank值都是1。然后，我们进行以下步骤：

1. 对于每个网页，计算它的出度（即它链接到的其他网页的数量）。例如，如果网页A链接到了网页B和C，那么A的出度就是2。

2. 对于每个网页，将它的PageRank值平均分配给它链接到的其他网页。例如，如果网页A的PageRank值是1，它的出度是2，那么它就会给每个链接的网页（如B和C）提供0.5的PageRank值。

3. 对于每个网页，将它接收到的所有PageRank值相加，得到它的新的PageRank值。例如，如果网页B接收到了来自网页A的0.5的PageRank值，来自网页D的0.3的PageRank值，那么B的新PageRank值就是0.5 + 0.3 = 0.8。

4. 重复以上步骤，直到每个网页的PageRank值稳定下来。

PageRank算法的一个关键的概念是“随机冲浪者”模型。在这个模型中，一个“冲浪者”在互联网上随机地点击链接，从一个网页跳转到另一个网页。PageRank值就是这个“冲浪者”在任何给定的时间点停留在一个特定网页的概率。

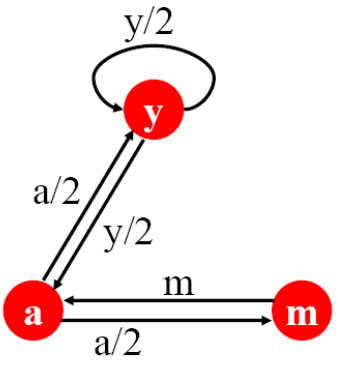
PageRank: The "Flow" Model

- A "vote" from an important page is worth more
- A page is important if it is pointed to by other important pages
- Define a "rank" r_j for page j

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

d_i … out-degree of node i

The web in 1839



"Flow" equations:

$r_y = r_y/2 + r_a/2$

$r_a = r_y/2 + r_m$

$r_m = r_a/2$

其中 r_i 是其他网页 i 的 PageRank 值

d_i 是其他网页 i 的链接数量

Given the following file that contains a directed graph

1 2
2 3
3 4
4 1
4 2
4 5
5 1
5 2
5 3
5 5

What does the second row of the PageRank matrix of links look like? Be careful to write an exact match using 0 and truncate floating point numbers at 2 digits followed by spaces. For example on question 4 the third row would be

0 0.33 0.5

链接的 PageRank 矩阵的第二行是什么样的？小心使用 0 编写精确匹配，并截断 2 位后跟空格的浮点数。例如，对于问题 4，第三行将是 0 0.33 0.5

The correct answer is: 1 0 0 0.33 0.25

在给定的图中，我们可以看到以下的链接指向页面2：

节点1到节点2（第一行）
节点4到节点2（第五行）
节点5到节点2（第八行）

所以，我们需要计算每个页面链接到页面2的概率。这个概率是每个节点的出度（即链接出去的边的数量）的倒数。我们会发现如果这个页面链接出去的数量越多，则访问某个页面的概念会降低，我们可以这样去记忆。

节点1有1个出度（到节点2），所以概率为1。
节点4有3个出度（到节点1，节点2，节点5），所以概率为1/3≈0.33。
节点5有4个出度（到节点1，节点2，节点3，节点5），所以概率为1/4=0.25。

所以，根据这个计算，PageRank矩阵的第二行应该是：

1 0 0 0.33 0.25

上面的值分别对应节点一到五，没有pagerank的用0来表示。

Lab 5

Point to point communication in MPI is

Select one:

a.a broadcast message with a root and destinations

b.a network connection between Process A and Process B

c.a message sent directly from Process A to Process B

d.the timing mechanism used to make sure Process A does not overwrite the memory of process B

In MPI, deadlock occurs when

Select one:

a.one of the processes crashes because of a programming error like division by zero

b.Two or more processes try to write to the same area of memory at the same time

c.at least one process is blocked waiting for a communication it will never receive

d.all processes are trying to send at the same time

e.a comm.Barrier() is used

Deadlock

Given the following code which produces a deadlock, which line would a minor change allow it to send and receive the messages correctly?

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
randNum = numpy.zeros(1)
if rank == 0:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=1)
        comm.Recv(randNum, source=0)
        print("Process", rank, "received the number", randNum[0])

if rank == 1:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=0)
        print("Process", rank, "received the number", randNum[0])
        randNum *= 2
        comm.Send(randNum, dest=0)
```

Select one:

a.comm.Send(randNum, dest=0)
b.comm.Recv(randNum, source=0)
c.comm.Send(randNum, dest=1)
d.randNum = numpy.zeros(1)
e.if rank == 1:

解析：这段代码中，进程0和进程1都在尝试发送和接收消息，但由于顺序问题，造成了死锁。问题就是进程0先发送了消息到进程1，然后试图从进程0（也就是它自己）接收消息，这在没有其他进程向进程0发送消息的情况下是不可能的。另一方面，进程1在接收到进程0的消息后，试图向进程0发送消息，但此时进程0正在等待从自己那里接收消息，所以这个消息发送不出去，造成了死锁。
解决方法是要更改进程0的接收源。进程0应该从进程1那里接收消息，而不是从自己那里接收。因此，应该更改以下代码行：

```
comm.Recv(randNum, source=0)
```

改成

```
comm.Recv(randNum, source=1)
```