AUGSPLICING: Synchronized Behavior Detection in Streaming Tensors[1]

唐伟科

摘要

我们要如何才能追踪到时间戳元组流的同步行为?例如手机设备下载和卸载软件的同步行为,这种行为目的是为了提高目标软件在软件商城中的排名。我们将这些元组作为张量流中的一项，随着时间的推移增加属性的大小。同步行为趋向于在张量中形成一个密集快，这种信号显示异常行为或有趣的集体行为。然而，现有的密集块检测算法都是基于静态张量的或者在张量流的设定下低效的算法。因此，我们提出一个快速的流算法 AugSplicing，它能够通过拼接之前检测到的块和新到来的块去检测出最新的密集子块。相比较最新的算法，我们的方法对于检测现实中下载数据的欺诈行为和发现校园 WiFi 数据中有共同兴趣的学生群体是很有效的，并且由于拼接理论使其在密集块检测中具有健壮性。在与准确度相近的流算法比较时，我们的算法更加快速。

关键词：密集块检测；流张量；同步行为；

1 引言

在已有的基于时间戳元组流中，我们怎么样能够定位到其中的同步行为呢？

这个问题在现实生活中有着许许多多的应用。在在线评价网站，例如：Yelp，设表示用户，表示商品，表示评价分数，并且 t 为评价时间，最同步的高分评价行为表示为最可疑的评价欺诈。在应用程序的日志中,,，和 t 能够表示一台移动设备，一个应用程序，下载时间和卸载时间。来自一组设备的高度同步的下载和卸载行为能够揭示最可疑的刷分行为。在模式识别方面，在校园网连接日志中对同一 WiFi AP 的连接和断连行为可以发现有相同兴趣的学生。

设这类时间戳元组为多维张量中的一项，有用户，对象和时间三个维度。值得一提的是我们称每一个维度为一个模块，一个有两个模块的张量称为矩阵。因为张量允许我们去注意格外的信息，尤其是时间。最密集的块可以识别出带有时间戳元组中最同步的行为。

目前，流张量的密集子张量检测方法是必要的。这是因为随着技术的进步，收集大型数据集比过去要容易得多。不仅真实数据的规模非常大，而且它到达的速度也很块。例如，Facebook 用户每天发布数十亿条帖子，每天进行数十亿条信用卡交易。因此，整个数据可能太大，无法放入内存甚至磁盘上。另一方面，我们可以将这种数据生成看作是上面提到的流张量。因此，当张量随时间变化时，能够有效地更新其估计的方法对于密集子张量检测问题至关重要。然而，许多现有的关于密集子张量检测的工作都是针对一批给出的静态张量设计的，我们将它们称为批处理算法。尽管这些批处理算法与张量中元组的大小接近线性，但当我们遇到数量巨大的数据集和二次方的时间复杂度时，在流张量的每个时间步长重新运行算法可能会导致内存过载。由于对之前的元组需要不断重复的计算，这限制了在流数据下可处理数据的规模。例如最先进的流算法 DencseStream，不断维护一个细粒度的顺序从而方便寻找密集子块。但这个顺序在每一个新的元组到来都需要进行更新，这限制了其检测速度。因此，

我们提出 AugSplicing 算法，一个在流张量下快速的密集块检测增量算法。不需要像批算法一样进行大量的重复计算，我们探索的算法是基于拼接条件减少搜索空间的，算法会不断拼接之前检测到的拼接子块和正在到来的新块。我们算法的主要贡献如下：

1、快速的流算法：我们提出的快速密集块检测算法比最新的算法最高快 320 倍。

2、健壮性：AugSplicing 算法在增量拼接和快检测方面是有拼接理论支撑的。

3、有效且可解释性检测：我们的算法达到的准确率与当前最好的算法 DenseStream 相当。通过在现实世界的数据中进行同步安装和卸载，AugSplicing 可以发现可疑的移动设备，这些设备的行为是为了提高目标应用程序在推荐列表中的排名。

2 相关工作

多维度元组也可以表示为丰富图中的属性边，例如，将用户和对象表示为图节点，并将分数和时间评级为图边上的不同属性。因此，我们总结了同时使用图和张量（包括两个矩阵）的密集块检测的相关研究。

2.1 静态张量和图

密集子图检测已经得到广泛的研究。Spoken 算法[2]，基于光谱分解的方法，使用基于 eigenvectors 生成的 EE-plot 的 EigenSpokes 来检测社交网络中的近端派系。欺诈者将节点和边缘可疑性都视为检测欺诈行为（即密集块）并且也是抵抗伪装的度量标准。CrossSpot 算法[3]提出了一个直观的、有原则的度量，满足任何怀疑的度量都应该遵守的公理，并设计了一个算法来发现按重要性顺序（“怀疑”）排序的密集块。HOSVD、CPD 和基于磁盘的算法通过张量分解发现密集子张量。M-ZOOM[4]和 D-CUBE[5]采用贪婪近似算法检测具有质量保证的密集子张量。CatchCore 算法设计了一个使用基于梯度的方法进行优化的统一度量来寻找分层密集的子张量。Liu，Hooi 和 Faloutsos 等人从拓扑、评分时间和分数优化了可疑度的度量。ISG+D-spot[6]算法构建了信息共享图，并为隐藏的最密集的块模式寻找密集的子图。Flock[7]在直播流媒体平台上检测步调一致的观众。然而，这些方法不考虑任何时间信息，或者只将时间箱视为一种静态模式

2.2 动态张量和图

在动态图方面，有一些方法可以监控整个图的演化，并检测子图的变化（密度或结构）。Spot-Light[8]算法利用图形草图来检测一段时间内图形快照的突然密度变化。SDRegion 算法[9]检测在遗传网络中持续变得密集或稀疏的块。EigenPulse（Zhang et al. 2019）[10]基于一种快速光谱分解方法，single-pass PCA（Yu et al. 2017）[11]来检测密度激增。其他方法，如 MIDAS（Bhatia 等人 2020a，c）[12]和 MSTREAM（Bhatia 等人 2020b）[13]检测到边缘流中突然到达的可疑相似的边缘组，但没有考虑图的拓扑结构。DENSESTREAM（Shin et al. 2017b）[14]为每个即将到来的元组增量地维护密集块，并在满足更新条件时更新密集子张量，从而限制了检测速度。对于基于聚类的方法，（Manzoor、Milajerdi 和 Akoglu 2016）将基于局部子结构的相对频率的图对发现异常的能力进行了比较。（Cao et al. 2014）发现了在持续一段时间内行为类似的恶意账户。基于张量分解的方法，如 SamBaTen 算法（Gujral、Pasricha 和 Papalexakis2018）[15]和 OnlineCP 算法（Zhou et al. 2016）[16]进行增量张量分解。基于总结的方法，例如（Shah et al. 2015）通过总结重要的时间结构来发现时间模式。（Araujo et al. 2014）使用迭代的秩为 1

的张量分解，结合 MDL（最小描述长度）来发现时间群落。我们的方法将时间戳元组定义为一个流张量，其时间模块不断增加，这样先前观察到的张量中的项数值就不会改变。我们在每个时间步长将输入的密集子张量与前面的子张量增量地拼接，以获得有效的结果。

3 本文方法

本文有许许多多的符号，所有符号的定义如表一所示：

Symbol	Definition
x , B	a tensor, and subtensor, i.e. block
x(t)	N-mode tensor up to time t
N	number of modes in x
$e_{i_1,\dots,i_N}(t)$	entry of x(t) with index i_1 , ... ,i_N
$I_n(.)$	set of mode-n indices of tensor
M(.)	mass of tensor i.e. sum of non-zero entries
S(.)	size of tensor
g(.)	arithmetic degree density of tensor
s	augmenting time stride
x(t, s)	N-mode augmenting tensor within time range (t,t + s]
k	number of blocks kept during iterations
[x]	u1, 2, ... ,x}

表 1: 符号定义表

3.1 本文方法概述

本算法流程图如下所示：

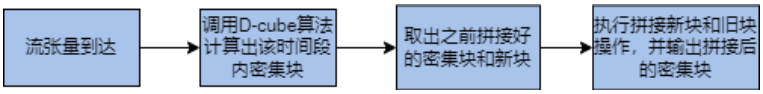


图 1: 方法示意图

3.2 拼接理论分析

我们分析这么一种理论条件，在合成两个密集块后，是否可以使密集块获得更高的密度。如图 2 所示，我们称这种合并为拼接。

定理 1（拼接条件）：给出两个块 B_1 和 B_2，并且 $g(B_1) >= g(B_2)$，要符合 $g(B_1 \cup \varepsilon) > g(B_2)$ 当且仅当满足下列不等式：

$$M(\varepsilon) > \sum_{n=1}^{N} r_n \cdot g(B_1) = Q \cdot g(B_1)$$

其中 $r_n = |I_n(\varepsilon)/I_n(B_1)|$，$r_n$ 为带入 B_1 块第 n 模块的索引值数量 $Q = \sum_{n=1}^{N} r_n$, 这表示带进 B_1 新索引的数量。

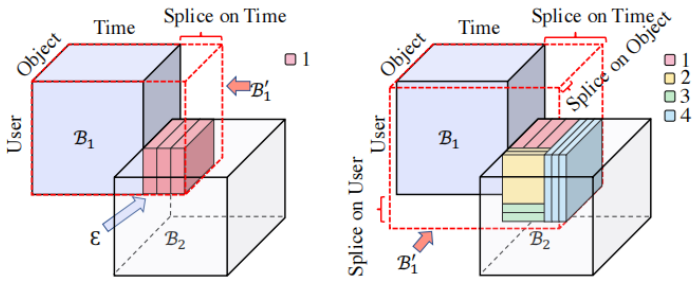
证明过程如下：

$$g(B_1 \cup \varepsilon) = \frac{M(B_1) + M(\varepsilon)}{S(B_1) + Q} > \frac{M(B_1) + Q \cdot g(B_1)}{S(B_1) + Q}$$

$$= \frac{S(B_1) \cdot g(B_1) + Q \cdot g(B_1)}{S(B_1) + Q} = g(B_1)$$

这里我们证明了必要条件，相似的也可证明充分条件。

我们能够看到当拼接两个块时，有 Q 个新索引被拼接到的一些维度上，并且只有当拼接块有足够大的质量时才可进行拼接。基于这个理论，我们设计一种有效的算法去拼接块。



(a) Splicing on mode *time* (b) Splicing result of Algorithm 1

图 2: 两个三维块拼接图示

3.3 如何拼接两个块

拼接两个块的目的是为了通过移动一个块的某些项到另一个块去，从而发现一个更高密度的块。因此，基于以上分析，一个大小较小的新块和一个质量较大的拼接快可以逐步的增加拼接的密度。我们算法对于拼接两个密集块设计如 Algorithm 1 所示：

Procedure 1 Splice two dense blocks

Input: two dense block:B_1 and B_2, with $g(B_1) > g(B_2)$
Output: new dense blocks
repeat
  $/* minimize\ the\ size\ of\ new\ indices,\ Q */$
    q - get set of modes that have to bring new indices into B_1 for splicing
    Q - |q|
    H ← an empty max heap for blocks and ordered by block mass
for *each combination of new indices*$(i_{q1}, \cdots, i_{qQ}), q \in$ ***q do*** **do**
  $\xi \leftarrow block\ with\ entries\ \{e_{i_1}, \cdots, e_{i_n} \in B_2 | \forall n \in [N] \backslash q, i_n \in I_n(B_1)\}$
  $push\ \xi\ into$ **H**
end
$/* maximize\ M(E),\ given\ Q */$
for $\xi \leftarrow$ ***H***$.top()$ **do**
  if $M(\xi) > Q \cdot g(B_1)$ **then**
    $B_1, B_2 \leftarrow update B_1 \cup \xi, B_2 \backslash \xi$
  end
  else
    break
  end
end
until *no updates on* B_1 *and* B_2
return *new block* B_1 *of higher density, and residual dense block* B_2

在给出的 $g(B_1) > g(B_2)$ 中，这个算法的想法是交替的找到新索引的最小大小和最大质量的块 ε 并将这些新的索引进行拼接。

对于 Q，我们首先需要找到会带入新索引到块 B_1 中模块 **q** 的集合。因为 B_1 和 B_2 块在维度 q 中没有共同的索引值，所以至少有一个新的索引值会被加入到 B_1 的维度 q 中，然后我们把 q 加入 **q**。因此 $Q = |q|$, 如果所有模块都有共同的索引，那么 **q** 为空且我们继续下列操作：

1、让块 $\varepsilon \in B_2$，由共同索引的项所组成

2、移动所有 ε 中的非零项到 B_1 中，这些项能够在不带入新索引值的情况下增长 B_1 的密度。

3、选择一个模块 q 进行拼接。对于每个模块，我们通过在该模式上选择一个新的索引来生成 B_2 的子块，该块在其它模块中所有索引都与 B_1 重叠。

在质量最大化方面，我们采用了大顶堆通过块的质量来组织。大顶堆的顶部块是最大质量的块。然后我们枚举所有新索引的合并去建立一个大顶堆 **H**，这些索引来自于 **q** 中的模块。因为现实数据通常模块的数量在 3-5 之间，并且合并的次数大概在 $S(B_2)$,B_2 是原始张量 **X** 中的一个较小的块。同时根据不等式 (1)，只有这些较大质量的块能够被添加进大顶堆 **H** 中。然后我们拼接大顶堆堆顶的块，迭代的增加 $g(B_1)$ 并且合并下一个堆顶的块直到没有满足质量条件的块，这些块需要满足的质量条件 $M(s) > Q \cdot g(B_1)$。

3.4 算法概括

在这一部分，我们描述算法的轮廓，算法主要做的是在时间间隔 t+s 中增量检测最密集块。

假定 B(t) 和 C(t) 是之前的 X(t) 块和即将到来的 X(t,s) 块的前 k+l 个密集块。其中 l 是一个松散常量，用于寻找 l 个近似前 k 个密集块的块。我们算法整体如下：

(a) 拼接两个密集块：我们迭代的选择两个候选块来自 B(t)C(t)，取名为 B_1 和 B_2，其中它们满足 $g(B_1) \geqslant g(B_2)$。使用算法 1 合并它们。

(b) 迭代和输出：直到没有块能够被拼接或者达到最大拼接次数我们才停止拼接迭代。然后，AugSplicing 算法输出在时间间隔 t+s 中的 k+l 个密集块的前 k 个，并且移动携带这这 k+l 个快移动到下一个时间间隔。

4 复现细节

4.1 与已有开源代码对比

本次复现论文有开源代码，代码地址为 https://github.com/BGT-M/AugSplicing。本次论文主要的改进有两点：1、更改算法对最终结果的密度块列表的插入策略，由原先的顺序插入改为二分查找插入。2、更改其拼接前拼接块的选择策略。3、优化 AugSplicing 算法的结果展示，原先是需要打断点或看输出文件来评估结果，改进后可以通过控制台的数据来评估其结果。详细的改进思路将在 4.4 创新点处详细阐述。

下面展示三种改进所编写的代码：

1、插入策略改进代码：

```
#二分查找插入策略
def insertBlockbyDensity(block, blocklist):
    density = block.getMass() / block.getSize()
    left, right = 0, len(blocklist) - 1
    while left <= right:
        mid = int(left + (right - left) / 2)
        midBlockDen = blocklist[mid].getMass() / blocklist[mid].getSize()
        if midBlockDen > density:
            left = mid + 1
        else:
            right = mid - 1
    if left == len(blocklist):
        blocklist.append(block)
    else:
        blocklist.insert(left, block)
    return blocklist
```

图 3: 插入策略改进算法

2、拼接块选择策略改进代码：

```python
#核心函数，计算出topk个密集子块
def calTopkBlock(blocklist1, blocklist2, k, l, maxSp, N):
    G, blockNumdict, edgesDensity = init_graph(blocklist1, blocklist2, N, k)
    print(edgesDensity)
    if len(G.nodes) == 0:
        return getResultBlocks(G, blockNumdict, k, l)
    lastden = min(blocklist1[-1].getDensity(), blocklist2[-1].getDensity())
    i, fail, index = 0, 0, 0
    while i < maxSp and fail < 5 * maxSp and index < len(edgesDensity):
        cnode, neighnode = random.choice(list(G.edges()))
        cnode, neighnode = edgesDensity[index][0]
        density = edgesDensity[index][1]
        index += 1
        if density < lastden or G.__contains__(cnode) == False or G.__contains__(neighnode) == False:
            continue
        block1 = blockNumdict[cnode]
        block2 = blockNumdict[neighnode]
        if block1.getDensity() >= block2.getDensity():
            sflag, takenBlock, leftBlock = stb.splice_two_block(block1, block2, N)
            label = '1'
        else:
            sflag, takenBlock, leftBlock = stb.splice_two_block(block2, block1, N)
            label = '2'
        if sflag:
            i += 1
            if label == '2':
                cnode, neighnode = neighnode, cnode
            blockNumdict[cnode], blockNumdict[neighnode] = takenBlock, leftBlock
            if leftBlock.getSize() != 0 and leftBlock.getDensity() >= lastden:
                'update rule changes'
                G = update_graph(G, blockNumdict, cnode, neighnode, N)
            else:
                G = remove_update_graph(G, blockNumdict, cnode, neighnode, N)
        else:
            fail += 1
    return getResultBlocks(blockNumdict, k, l)
```

图 4: 拼接块选择策略修改函数 1

```python
#初始化函数：创建一个图，图的边表示了两个新旧密集块有共同的属性值
def init_graph(blocklist1, blocklist2, N, k):
    edges = []
    blockNumdict = {}
    edgesDensity = {}
    for idx, block in enumerate(blocklist1):
        blockNumdict[idx] = block
    for idx, block in enumerate(blocklist2):
        blockNumdict[idx + k] = block
    for idx1, block1 in enumerate(blocklist1):
        modeToAttVals1 = block1.getAttributeDict()
        for idx2, block2 in enumerate(blocklist2):
            modeToAttVals2 = block2.getAttributeDict()
            for dimen in range(N):
                insec_dimens = modeToAttVals1[dimen] & modeToAttVals2[dimen]
                if len(insec_dimens) != 0:
                    edges.append((idx1, k+idx2))
                    density = block1.getDensity() + (block2.getDensity() - block1.getDensity()) / 2
                    edgesDensity[(idx1, k+idx2)] = density
                    break
    G = nx.Graph()
    edgesDensity = sorted(edgesDensity.items(), key=lambda x:x[1], reverse=True)
    G.add_edges_from(edges)
    return G, blockNumdict, edgesDensity
```

图 5: 拼接块选择策略修改函数 2

3、结果展示功能代码：

```
def optimiAlgo(inputfile, outpath, s, k, l, maxSp, N, delimeter, steps):
    augmented_lines, accum_lines = [], []
    sindex = 0
    mints = 0
    with open(inputfile, 'r') as f:
        for line in f:
            user, obj, ts, v = map(int, line.strip().split(delimeter))
            augmented_lines.append(line)
            accum_lines.append(line)
            if ts - mints >= s:
                dcube_file = os.path.join('augfile.txt')
                f = open(dcube_file, 'w')
                f.writelines(augmented_lines)
                f.close()
                dcube_output = os.path.join('augfile_dcube_output', str(sindex))
                if not os.path.exists(dcube_output):
                    os.makedirs(dcube_output)
                os.system('cd ./dcube-master && ./run_single.sh' + ' ../' +
                          dcube_file + ' ../' + dcube_output + ' ' + str(N) + ' ari density ' + str(k+l))
                if sindex == 0:
                    curr_output = os.path.join(outpath, str(sindex))
                    if not os.path.exists(curr_output):
                        os.makedirs(curr_output)
                    for fn in os.listdir(dcube_output):
                        file = os.path.join(dcube_output, fn)
                        file2 = os.path.join(curr_output, fn)
                        shutil.copy(file, file2)
                else:
                    dcubeBlocks = util.readBlocksfromPath(dcube_output, k+l)
                    past_output = os.path.join(outpath, str(sindex - 1))
                    pastBlocks = util.readBlocksfromPath(past_output, k+l)

                    curr_output = os.path.join(outpath, str(sindex))
                    if not os.path.exists(curr_output):
                        os.mkdir(curr_output)
                    currBlocks = Caltopk.calTopkBlock(dcubeBlocks, pastBlocks, k, l, maxSp, N)
                    print("\n\nrunning the AugSplicing algorithm")
                    for idx, block in enumerate(currBlocks):
                        tuplefile = 'block_' + str(idx + 1) + '.tuples'
                        print("block: " + str(idx + 1))
                        util.writeBlockToFile(curr_output, block, tuplefile)
                mints = ts
                augmented_lines = []
                sindex += 1
                if sindex == steps:
                    print('***end***')
                    break
```

图 6: 结果展示优化修改函数 1

```
#将block写入tuplegilename文件中
def writeBlockToFile(path, block, tuplefilename):
    tuplefile = os.path.join(path, tuplefilename)
    tuples = block.getTuples()
    print("Size: " + str(block.getSize()))
    print("Mass: " + str(block.getMass()))
    print("Density: " + str(block.getDensity()))
    with open(tuplefile, 'w') as tuplef:
        for key in tuples:
            words = list(key)
            words.append(str(tuples[key]))
            tuplef.write(','.join(words))
            tuplef.write('\n')
    tuplef.close()***end***')
                    break
```

图 7: 结果展示优化修改函数 2

4.2 实验环境搭建

实验环境：

操作系统版本：Window10

虚拟机使用工具：VMware Workstation Pro

虚拟机环境：Ubuntu-22.04.1

开发软件：pycharm、java 17.0.4.1

4.3 使用说明

1、代码下载并解压

2、将代码导入 pycharm

3、打开运行文件 run_beer 或 run_yelp 运行

4、在 output 目录下查看算法导出的结果

4.4 创新点以及实验结果展示

4.4.1 插入策略改进

改进缘由：该算法的目的是输出张量流中的前 k + l 个密集子张量，输出的方法是将所有拼接好的块按照密度从大到小逐一插入一个列表中 (blocklist)，最终取 blocklist 列表的前 k + l 个块输出到指定文件中。在将块插入列表的过程中使用的是顺序插入，思想是在 blocklist 列表中从头至尾的遍历查找，直到找到第一个比插入块密度小的块的位置，在这个位置执行插入操作。在这种插入操作中，顺序查找效率比较低。

改进思路：对于该操作我将插入策略中查找的策略改为二分查找，使用二分查找来定位第一个比当前密度块密度小的块的位置。在查询位置使用二分查找代替顺序查找可以加快查询速度。

改进结果：两者执行的时间相差不多

总结：

1、由于在该整体的插入操作中，插入所占用的时间远比查询的时间开销大很多。因为每插入一次可能需要移动很多个块，和这种时间开销相比，前面的查询操作所花费的时间可以忽略。这也就是改进后两者时间相差不大的原因之一，两者时间都在 83500ms 左右波动。

2、改进后时间优化不多的另一个原因是因为 blocklist 块列表的规模不够大，由于在本示例算法中只需要求前 15 个密集子张量，blocklist 列表的规模也只有 O(100) 左右。若数据量很大，要求算密度最大的 1 万个密集子张量。则效率的提升也就变得十分可观了。

4.4.2 拼接块选择策略改进

对于 AugSplicing 算法，在拼接张量流中的旧块和新块的时候使用的策略是从构造好的关联图中随机选择一条边，也就是随机选择旧块和新块进行拼接（前提是这两个块有关列，即在某一个维度上有相同的属性值）。然后拼接 maxSp 次或者拼接失败 5 * maxSp 次。

改进缘由：我认为这样的随即拼接虽然方便操作，但是拼接后的块密度是不稳定的。有时候可能出现拼接的新块和旧块都是属于各自块列表内低密度的块。从而导致拼接出来的块的密度也不是很大，也不可能是前 k 个密集子张量。这样的拼接对于结果来说，无疑只是浪费了时间。因此，我将尝试更换拼接块的选择策略。

改进思路：

1、在建立关联图的时候，不仅要将关联边加入图中。还需要将边对应的两个顶点，也就是新块列表和旧块列表中的两个块的密度的平均值作为该边的密度（也可称为权值）。在拼接时，我们将尽量选择先拼接密度较大的边。

2、其次是数据结构的选择，我们将选择字典来保存两个顶点和边密度之间的映射关系。值得考虑的是是将两个顶点作为键还是值作为键？当然，我们只能选择两个顶点作为键。因为边密度和顶点之间存在一对多的关系，一个边密度可能可以映射到很多个顶点对上。所以，我们选择顶点作为键。最后在 python 中输出该字典时需要按密度排序输出，从而又变成一个列表（字典是无序的）。

3、我们将按边密度从大到小选择两个快进行拼接，而不是随机拼接。其次，当密度小于 1 时，我们认为其不再是密集块。因此，不断拼接直到拼接次数达到 maxSp、失败拼接次数达到 5 * maxSp 或者遍历到了列表尾部。

4、在改进的策略中，进行拼接之后可能会出现原本和被拼接块有关联的边消失了，被拼接块也可能消失了，而拼接块和其它块多了不少新的关联边。本次设计的策略会忽略这些情况，当由于拼接导致原本的关联边以及被拼接块消失了。那么，当我们遍历到这样的边时，我们将会跳过。也就是说，我们的拼接策略是面向拼接开始前的状态的，而不是随着拼接过程动态变化的。

改进结果：我们将给出改进前后算法最后拼接出来的前五个块和后五个快以及时间，然后就数据进行分析对比其优劣。

改进前：

```
running the AugSplicing algorithm
block: 1
Size: 2626
Mass: 101890.0
Density: 38.8004569687738
block: 2
Size: 3029
Mass: 66503.0
Density: 21.95543083525916
block: 3
Size: 4441
Mass: 69355.0
Density: 15.616978158072506
block: 4
Size: 2088
Mass: 16375.0
Density: 7.842432950191571
block: 5
Size: 4225
Mass: 33013.0
Density: 7.813727810650888
```

```
block: 11
Size: 13
Mass: 18.0
Density: 1.3846153846153846
block: 12
Size: 13
Mass: 16.0
Density: 1.2307692307692308
block: 13
Size: 31
Mass: 38.0
Density: 1.2258064516129032
block: 14
Size: 893
Mass: 841.0
Density: 0.9417693169092946
block: 15
Size: 30
Mass: 28.0
Density: 0.9333333333333333
end
消耗时间为75529毫秒
```

图 8: 改进前算法运行结果

改进后：

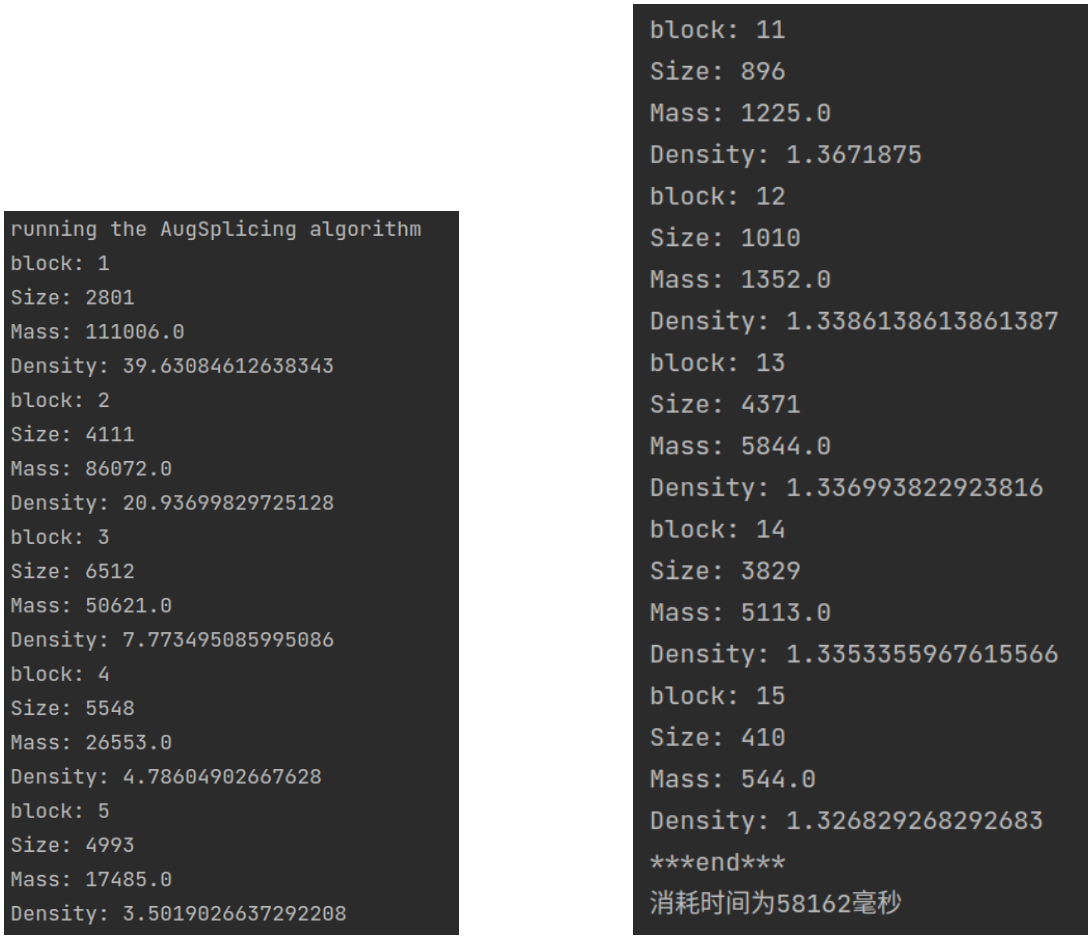


图 9: 改进后算法运行结果

结果分析：

首先对于改进前的 AugSplicing 算法，由于其拼接是随机拼接。因此，其结果是不可再现的。每一次的结果都会有不小差异，这也是其的缺点之一。而我们改进后的算法在拼接块选择策略上不是随机的。因此，结果也是可以再现的。接下来我们对比两个算法，由上图可以看出，改进前算法执行完毕的时间为 75s 左右，改进后的算法执行完毕的时间为 58s 左右，改进后的算法在时间上有一个极大的提升。这得益于我们的拼接块选择策略，我们没有花太多的时间去拼接一些最终不可能成为密集块的块上。

其次，我们观察其前五个密集子块的密度。可以发现我们改进后的算法，在前五个块的拼接上做的没有改进前算法那么好。从第三个密集块开始，拼接出来的密度比改进前的算法相差不少。这个结果应该是由于我们拼接策略没有随着拼接过程动态变化所导致的。

最后，我们观察其后 5 个密集块。我们可以看出，在这一点上改进后的算法明显比改进前的算法做的要好。第一、我们改进后的算法维持了一个下届，保证块的密度不会小于 1。小于 1 的块我们不将其纳入密集子块的选择范围内，这一点在代码上也深有体现。改进前的算法最后两个块的密度都已经低于 1，这是一个不理想的结果。

4.4.3 结果展示功能代码

改进缘由：对于初始的 AugSplicing 算法，其结果是没有可视化而且也没有代表其效果的参数显示到控制台的。原始的算法只将中间的 D-cube 算法的结果显示到了控制台，并且这个显示也是封装

好的 D-cube 算法实现的。因此，在复现过程中只能不断打断点或者看输出文件评估其效果。鉴于此，我仿照 D-cube 算法的输出格式设计了 AugSplicing 算法的结果展示功能。从而通过一些显示在控制台上的参数方便对其效率进行评估。

改进思路：主要就是设计结果输出格式，并且对算法的执行时间进行统计。

改进结果：结果就是上述图 8-图 11 所展示的图，这给我对比改进前后的算法的效率提供了很大的便利。

5 总结与展望

在本次论文复现改进中，我认为 AugSplicing 算法最大的缺陷就是其拼接块选择策略上做的不是很好，由于其选择策略采取了随机选取从而导致了其结果的不可再现性。因此，我觉得其后续发展的方向很大可能是设计出一个更加合理、且更具有理论性的密集块拼接选择策略。这不仅仅可以提升其运行速度，还有可能对最终拼接出来的密集子块密度的提升有一个正向的影响。

最后，我总结一下在这次课程作业中的感悟。在这门课中，我觉得最重要的是学会如何去思考改进一个算法。当然，由于所复现的算法大部分为顶刊中的算法，更多的可能是改退。但是，学会去判断改哪里、去思考如何改才是这门课最重要的一点。对于我，首先会通过其算法将其划分为多个小的部分，然后不断思考每一部分是否有可能进行改进、如何改进。这是一个很漫长的过程，也是一个很考验耐心的过程。因此，这门课所教会我的不仅仅是复现了一篇论文并加以改进。更多的是让我体会到了如何科研，如何对这些已发表论文进行批判性的思考。而不是像以前一样，看到这些顶刊的论文就将其视为真理，丝毫不怀疑里面的设计、想法。因此，我感谢这一门课以及参与的老师。这让我受益匪浅，共勉！

参考文献

[1] ZHANG J, LIU S, HOU W, et al. AugSplicing: Synchronized Behavior Detection in Streaming Tensors [C]//Proceedings of the AAAI Conference on Artificial Intelligence: vol. 35: 5. 2021: 4653-4661.

[2] PRAKASH B A, SRIDHARAN A, SESHADRI M, et al. Eigenspokes: Surprising patterns and scalable community chipping in large graphs[C]//Pacific-Asia conference on knowledge discovery and data mining. 2010: 435-448.

[3] JIANG M, BEUTEL A, CUI P, et al. A general suspiciousness metric for dense blocks in multimodal data[C]//2015 IEEE International Conference on Data Mining. 2015: 781-786.

[4] SHIN K, HOOI B, FALOUTSOS C. M-zoom: Fast dense-block detection in tensors with quality guarantees[C]//Joint european conference on machine learning and knowledge discovery in databases. 2016: 264-280.

[5] SHIN K, HOOI B, KIM J, et al. D-cube: Dense-block detection in terabyte-scale tensors[C]//Proceedings of the Tenth ACM International Conference on Web Search and Data Mining. 2017: 681-689.

[6] YIKUN B, XIN L, LING H, et al. No place to hide: Catching fraudulent entities in tensors[C]//The

World Wide Web Conference. 2019: 83-93.

[7] SHAH N. Flock: Combating astroturfing on livestreaming platforms[C]//Proceedings of the 26th International Conference on World Wide Web. 2017: 1083-1091.

[8] ESWARAN D, FALOUTSOS C, GUHA S, et al. Spotlight: Detecting anomalies in streaming graphs[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018: 1378-1386.

[9] WONG S W, PASTRELLO C, KOTLYAR M, et al. Sdregion: Fast spotting of changing communities in biological networks[C]//Proceedings of the 24th ACM SIGKDD international conference on Knowledge discovery & data mining. 2018: 867-875.

[10] ZHANG J, LIU S, YU W, et al. Eigenpulse: Detecting surges in large streaming graphs with row augmentation[C]//Pacific-Asia Conference on Knowledge Discovery and Data Mining. 2019: 501-513.

[11] YU W, GU Y, LI J, et al. Single-pass PCA of large high-dimensional data[J]. arXiv preprint arXiv:1704.07669, 2017.

[12] BHATIA S, HOOI B, YOON M, et al. Midas: Microcluster-based detector of anomalies in edge streams [C]//Proceedings of the AAAI Conference on Artificial Intelligence: vol. 34: 04. 2020: 3242-3249.

[13] BHATIA S, JAIN A, LI P, et al. MStream: Fast Streaming Multi-Aspect Group Anomaly Detection[J]., 2020.

[14] SHIN K, HOOI B, KIM J, et al. Densealert: Incremental dense-subtensor detection in tensor streams [C]//Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2017: 1057-1066.

[15] GUJRAL E, PASRICHA R, PAPALEXAKIS E E. Sambaten: Sampling-based batch incremental tensor decomposition[C]//Proceedings of the 2018 SIAM International Conference on Data Mining. 2018: 387-395.

[16] ZHOU S, VINH N X, BAILEY J, et al. Accelerating online cp decompositions for higher order tensors [C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016: 1375-1384.