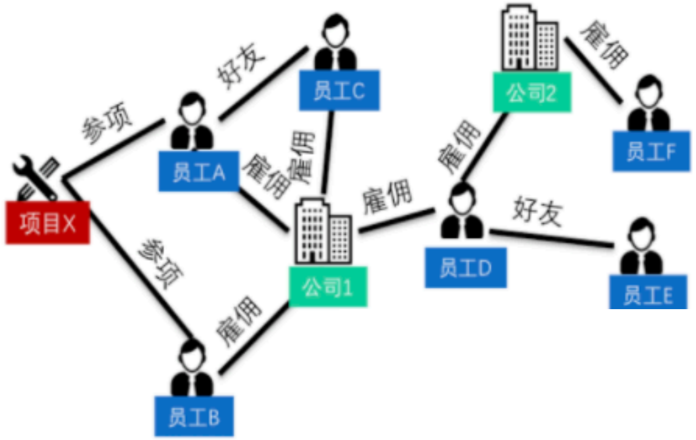
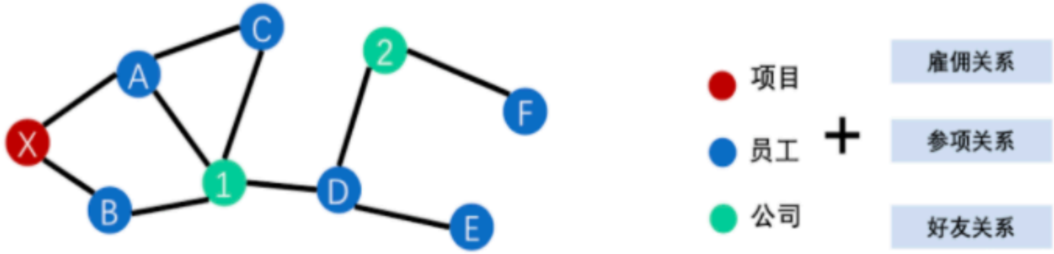
图数据分区策略的实现与优化

MENTOR / 胡英谦 陈　超
组 员 / 程苗苗 苏金涛 孙泽嵩

什么是图数据?

- 图由节点和边组成，边代表两个节点之间的关系
- 图数据是指存储内在关系是图的数据

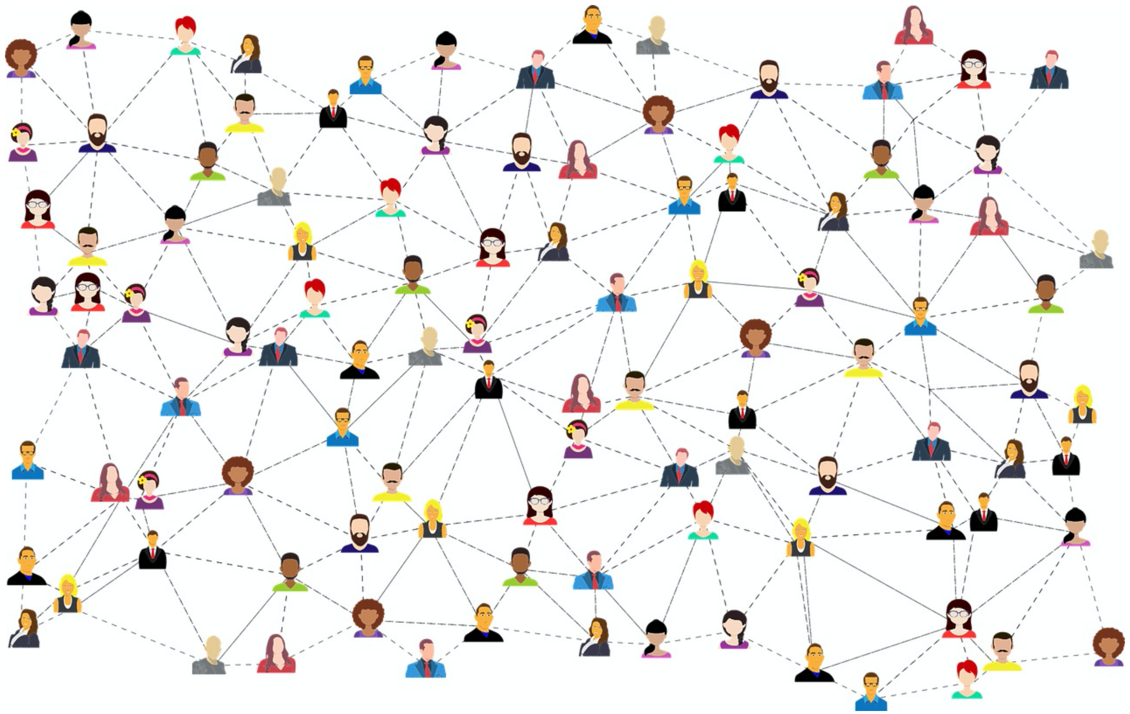




图数据的应用场景

大规模图计算

- 计算Page Rank
- 计算图的特征，做社区检测
- 计算聚类系数



大规模图神经网络

- 搜索推荐
- 药物预测

图数据库

- 用户信息、用户和用户的关系（关注、好友等）
- 内容（视频、文章、广告等）
- 用户和内容的联系（点赞、评论、转发、点击广告等）

图数据库

社交媒体推荐系统——召回

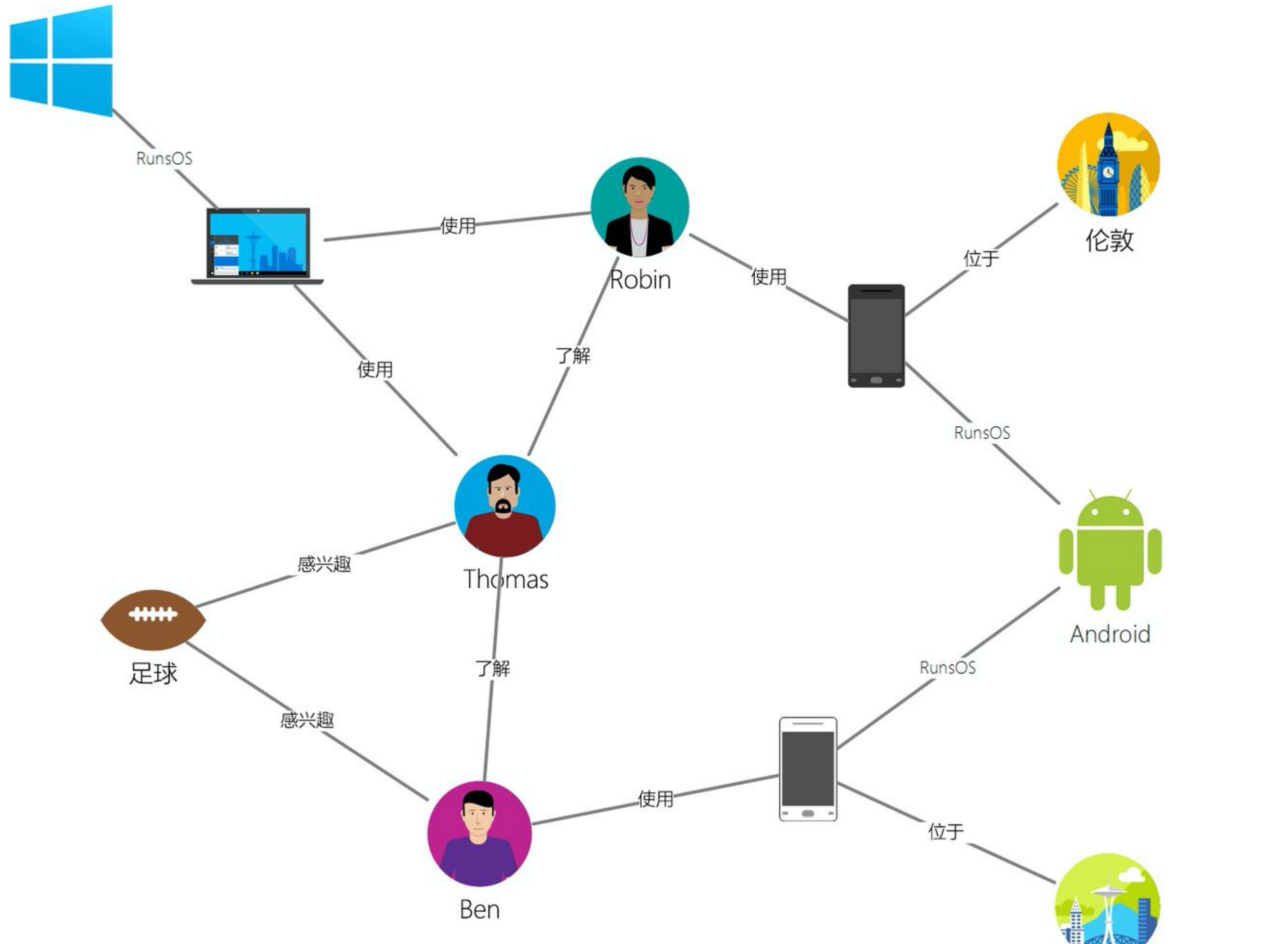
- 通过社交关系进行好友发掘、关注页可能感兴趣的视频推荐
- 一度或两度查询，需要通过亲密度等属性进行计算

社交平台服务端的用户关系和行为记录

- 用户点赞、关注、转发等行为关系记录和判断
- 请求规模极大，多为一度查询，对延迟极为敏感

知识图谱

- 用于搜索实体推荐、知识库问答等场景
- 边类型和点类型较多，多度复杂查询



图分区

为什么要图分区？

以分布式图数据库为例，其性能要求为：

- **海量数据存储：** 百亿点、万亿边的数据规模；
- **海量吞吐：** 最大集群 QPS 达到数千万；
- **低延迟：** 要求访问延迟 pct99 需要限制在毫秒级

图分区

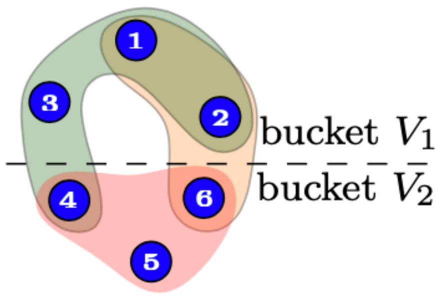
为什么要图分区?

以分布式图数据库为例，其性能要求为:

- 海量数据存储：百亿点、万亿边的数据规模；
- 海量吞吐：最大集群 QPS 达到数千万；
- 低延迟：要求访问延迟 pct99 需要限制在毫秒级

如何进行图分区?

- 基本思想：将图的顶点划分为多个大小近似的分量，以使分量之间交叉的边的数量最小
- 一个良好的图分区算法可以解决节点间的数据通信开销和负载均衡

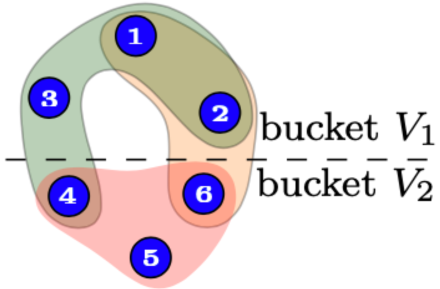


如何衡量图分区算法的性能?

扇出 (fanout)

对于给定的分区 $P = \{V_1, \ldots, V_k\}$ 和查询顶点 $q \in Q$，我们将 q 的扇出定义为具有入射到 q 的数据顶点的不同存储桶的数量:

$$\mathrm{fanout}(P, q) = |\{V_i : \exists\{q, v\} \in E, v \in V_i\}|.$$

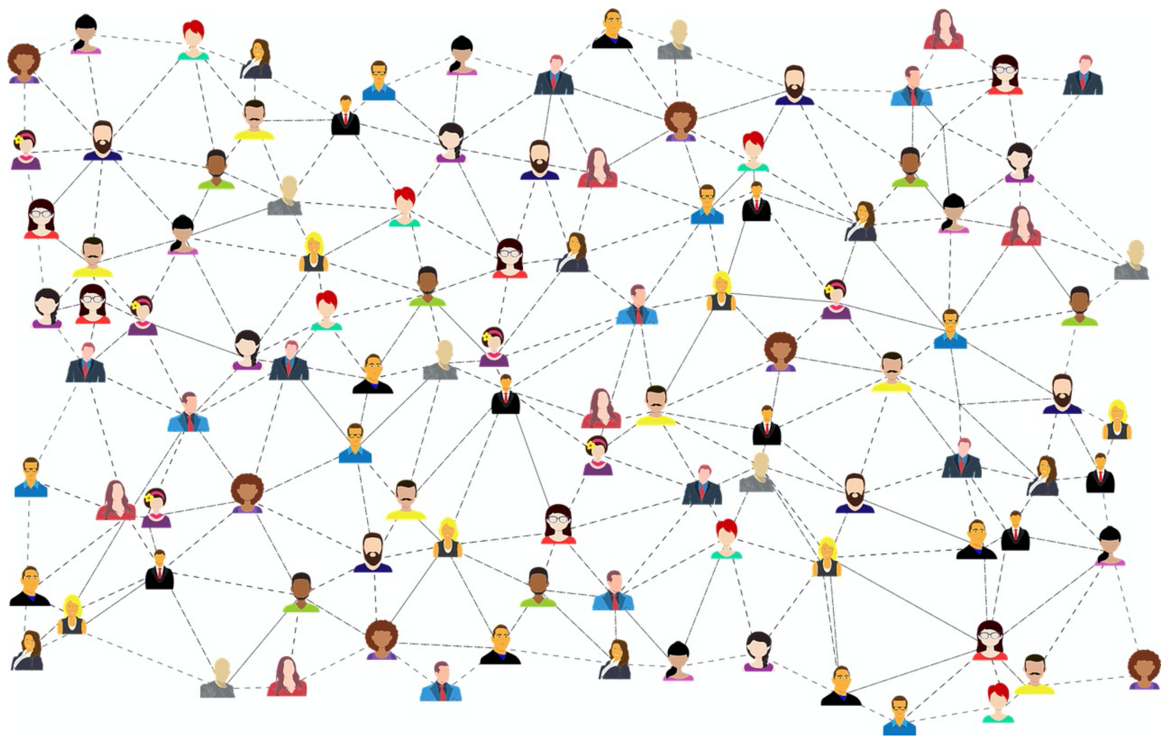


如何衡量图分区算法的性能?

- **可扩展性 (scalability)**

  随着图规模和分区数的增大，算法的时间复杂度、计算复杂度、空间复杂度和通信等指标可能会影响图分区算法的结果，影响算法的**可扩展性**

项目贡献



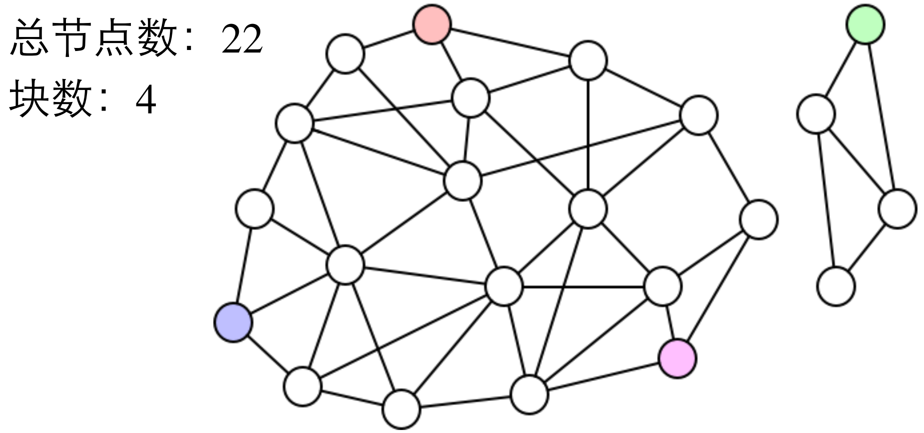
- 在阅读大量论文的基础上，实现了random，BDG，Social Hash三种图分区算法，并且在**真实世界数据集**上进行了验证

- 针对Social Hash，**提出并实现了两点改进**（最终实验运行速度提升 **3 倍**，实验指标fanout 提升 **7 个点**），并提出其他未来可以进行的工作

策略1 基于块的决定性贪婪(BDG)分区方法

图着色：将输入图切成细粒度的块以避免破坏局部性

- 为了限制块的大小和确保所有非连通分量被染色，可以将BFS从每个随机选择的源采取的步数设置为较小的值；重复上述过程，直到所有顶点都着色为止



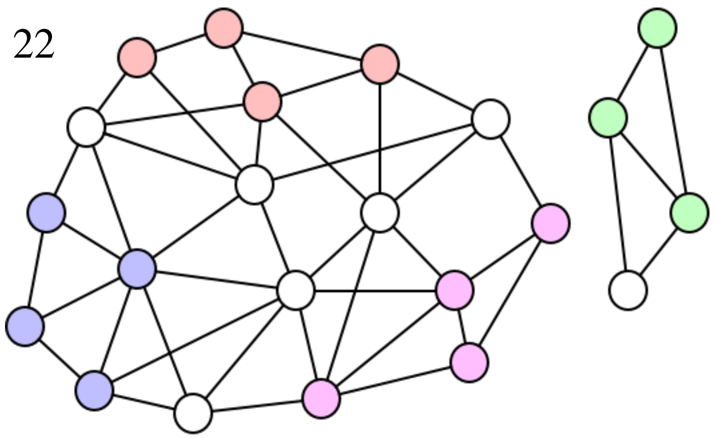
随机选择多个源点

策略1 基于块的决定性贪婪(BDG)分区方法

图着色：将输入图切成细粒度的块以避免破坏局部性

- 为了限制块的大小和确保所有非连通分量被染色，可以将BFS从每个随机选择的源采取的步数设置为较小的值；重复上述过程，直到所有顶点都着色为止

总节点数：22
块数：4



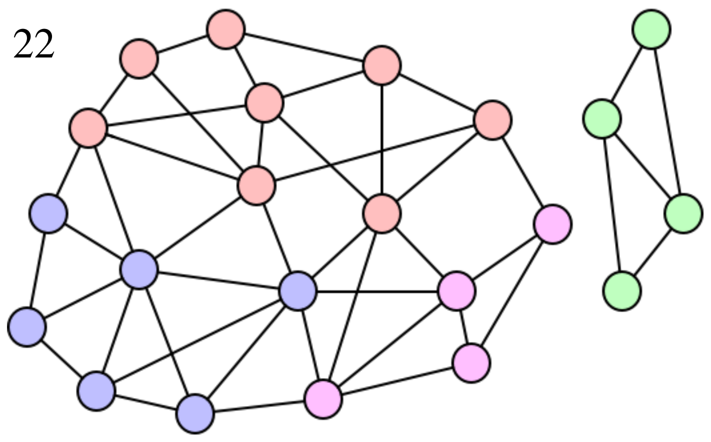
将颜色广播给邻居

策略1 基于块的决定性贪婪(BDG)分区方法

图着色：将输入图切成细粒度的块以避免破坏局部性

- 为了限制块的大小和确保所有非连通分量被染色，可以将BFS从每个随机选择的源采取的步数设置为较小的值；重复上述过程，直到所有顶点都着色为止

总节点数：22
块数：4



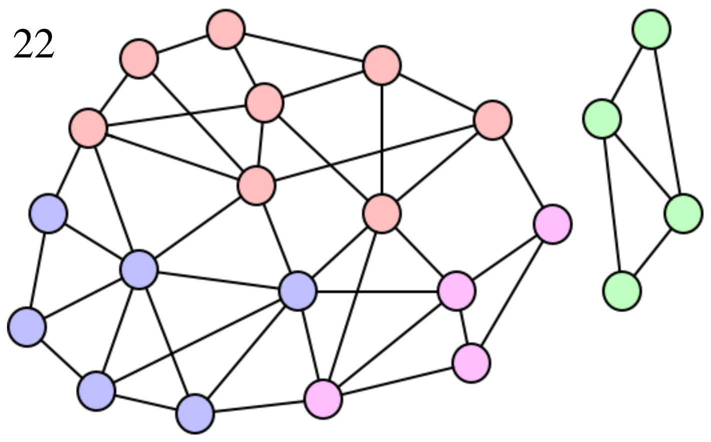
直到所有顶点被着色

策略1 基于块的决定性贪婪(BDG)分区方法

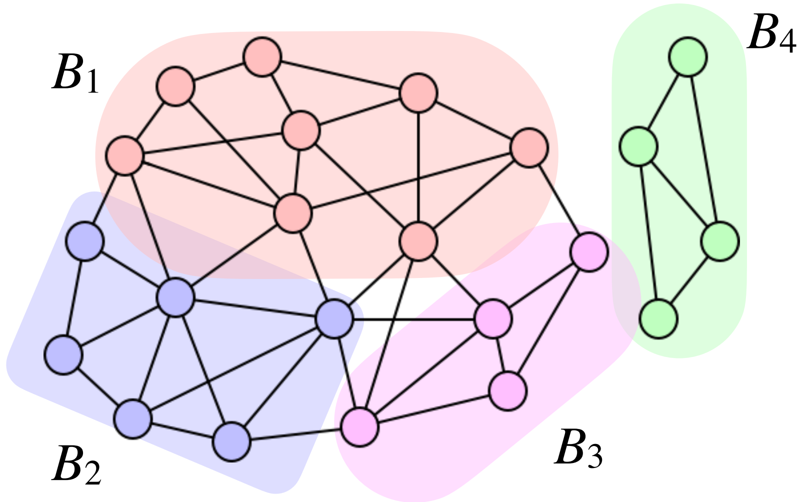
图着色：将输入图切成细粒度的块以避免破坏局部性

- 为了限制块的大小和确保所有非连通分量被染色，可以将BFS从每个随机选择的源采取的步数设置为较小的值；重复上述过程，直到所有顶点都着色为止

总节点数：22
块数：4



直到所有顶点被着色



输入图被分为几个互不相交的块

策略1 基于块的决定性贪婪(BDG)分区方法

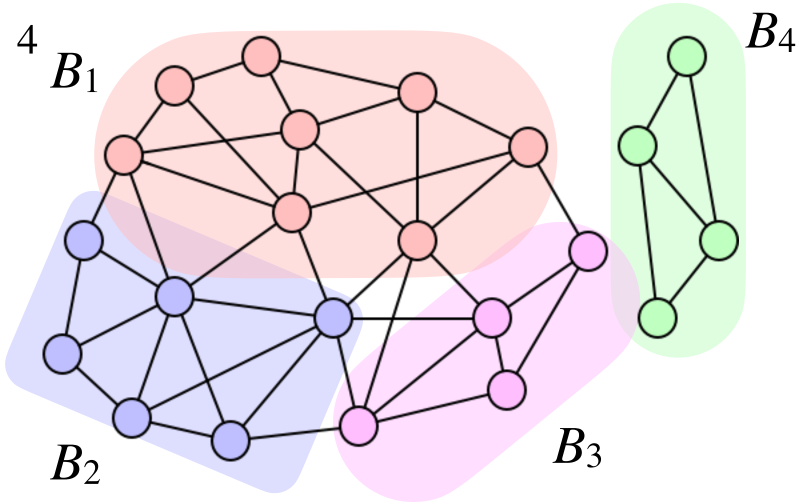
决定性分配：根据决定性贪婪算法将块分配

- 块的分配顺序可以影响最终分区结果，按照块大小的降序对块进行排序，然后从最大的块开始分配
- 确保将每个块 B 分配给与之相似性高的分区 $P(j)$，并且仍然具有足够的可用容量来容纳 B
- 将 $|V|$ 个顶点分配进入 k 个分区(bucket)，
  则每个分区的预期容量为 $C = |V| / k$
  $\Gamma(B)$ 为块 B 的1度邻居块，$P(i)$ 为属于已分配给分区 i 的块的顶点

$$j = \arg\max_{i \in [k]} \{|P(i) \cap \Gamma(B)| * (1 - \frac{|P(i)|}{C})\}$$

总节点数：22

块数：4



策略2 Social Hash分区方法

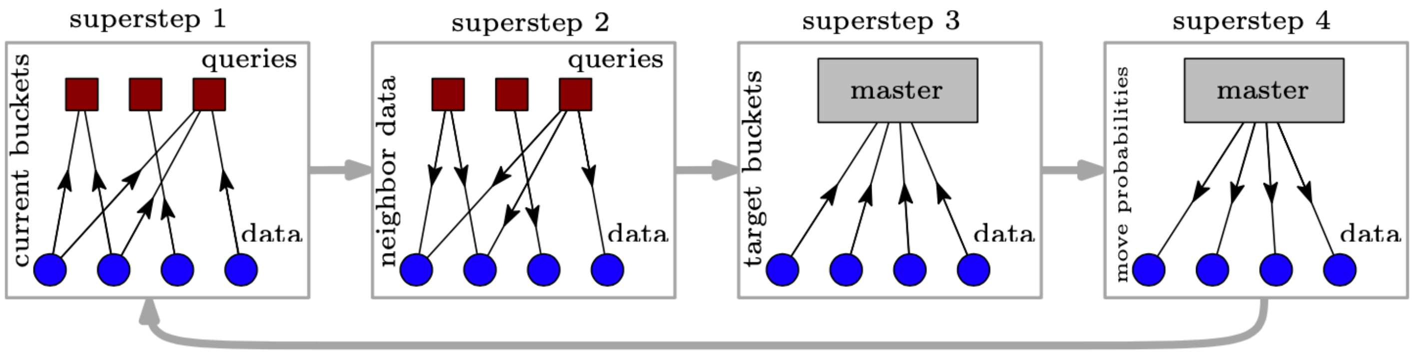
Social Hash分区算法的实现在以顶点为中心的框架中包括四个步骤:

（1）收集查询邻居数据;

（2）计算移动增益 (moving gain);

（3）选出候选目标桶;

（4）计算交换概率并执行移动



Algorithm 1: Fanout Optimization

Input : graph $G = (\mathcal{Q} \cup \mathcal{D}, E)$, the number of buckets k, imbalance ratio ε

Output: buckets $V_1, V_2, \ldots, V_k$

for $v \in \mathcal{D}$ **do** `/* initial partitioning */`

  $bucket[v] \leftarrow \text{random}(1, k)$;

repeat `/* local refinement */`

  for $v \in \mathcal{D}$ **do**

    for $i = 1$ **to** k **do**

      $gain_i[v] \leftarrow ComputeMoveGain(v, i)$;

    `/* find best bucket */`

    $target[v] \leftarrow \arg\max_i gain_i[v]$;

    `/* update matrix */`

    if $gain_{target[v]}[v] > 0$ **then**

      $S_{bucket[v],target[v]} \leftarrow S_{bucket[v],target[v]} + 1$;

  `/* compute move probabilities */`

  for $i, j = 1$ **to** k **do**

    $probability[i, j] \leftarrow \frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$;

  `/* change buckets */`

  for $v \in \mathcal{D}$ **do**

    if $gains[v] > 0$ **and** $\text{random}(0, 1) < probability[bucket[v], target[v]]$ **then**

      $bucket[v] \leftarrow target[v]$;

until *converged* **or** *iteration limit exceeded*;

策略2 Social Hash分区方法

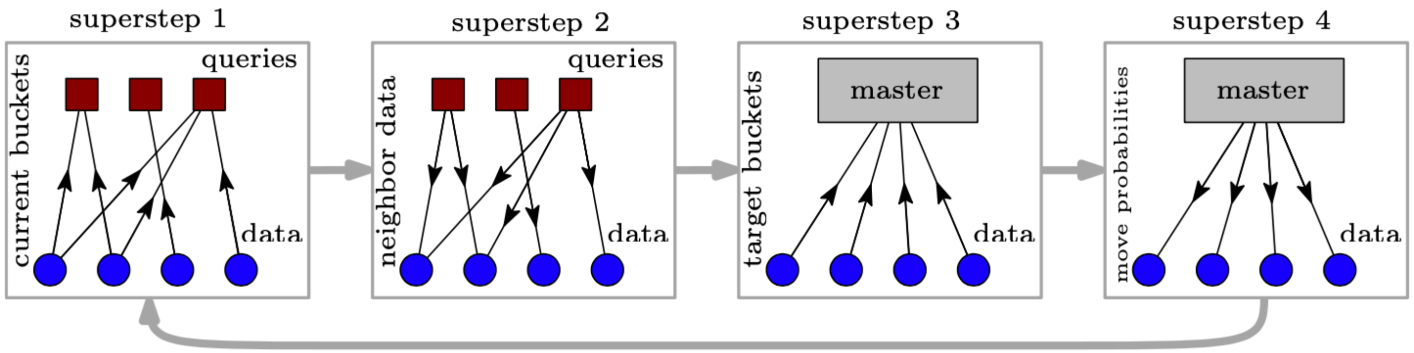
Social Hash分区算法的实现在以顶点为中心的框架中包括四个步骤：

（1）收集查询邻居数据；

（2）计算移动增益 (moving gain)；

（3）选出候选目标桶；

（4）计算交换概率并执行移动



Algorithm 1: Fanout Optimization

Input : graph $G = (\mathcal{Q} \cup \mathcal{D}, E)$, the number of buckets k, imbalance ratio ε

Output: buckets $V_1, V_2, \ldots, V_k$

for $v \in \mathcal{D}$ **do** `/* initial partitioning */`
    $bucket[v] \leftarrow \text{random}(1, k)$;

repeat `/* local refinement */`
    for $v \in \mathcal{D}$ **do**
        for $i = 1$ **to** k **do**
            $gain_i[v] \leftarrow ComputeMoveGain(v, i)$;
        `/* find best bucket */`
        $target[v] \leftarrow \arg\max_i gain_i[v]$;
        `/* update matrix */`
        if $gain_{target[v]}[v] > 0$ **then**
            $S_{bucket[v],target[v]} \leftarrow S_{bucket[v],target[v]} + 1$;

    `/* compute move probabilities */`
    for $i, j = 1$ **to** k **do**
        $probability[i, j] \leftarrow \frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$;

    `/* change buckets */`
    for $v \in \mathcal{D}$ **do**
        if $gains[v] > 0$ **and** $\text{random}(0, 1) < probability[bucket[v], target[v]]$ **then**
            $bucket[v] \leftarrow target[v]$;

until *converged* **or** *iteration limit exceeded*;

策略2 Social Hash分区方法

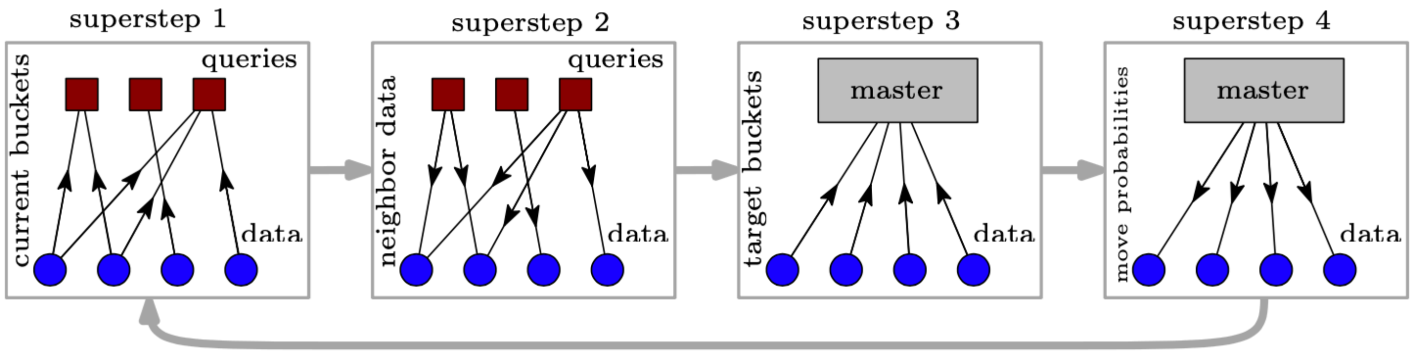
Social Hash分区算法的实现在以顶点为中心的框架中包括四个步骤：

（1）收集查询邻居数据；

（2）计算移动增益 (moving gain)；

（3）选出候选目标桶；

（4）计算交换概率并执行移动



Algorithm 1: Fanout Optimization

Input : graph $G = (\mathcal{Q} \cup \mathcal{D}, E)$, the number of buckets k, imbalance ratio ε

Output: buckets $V_1, V_2, \ldots, V_k$

for $v \in \mathcal{D}$ **do** `/* initial partitioning */`
  $bucket[v] \leftarrow \text{random}(1, k)$;

repeat `/* local refinement */`
  for $v \in \mathcal{D}$ **do**
    for $i = 1$ **to** k **do**
      $gain_i[v] \leftarrow ComputeMoveGain(v, i)$;
    `/* find best bucket */`
    $target[v] \leftarrow \arg\max_i gain_i[v]$;
    `/* update matrix */`
    if $gain_{target[v]}[v] > 0$ **then**
      $S_{bucket[v],target[v]} \leftarrow S_{bucket[v],target[v]} + 1$;
  `/* compute move probabilities */`
  for $i, j = 1$ **to** k **do**
    $probability[i, j] \leftarrow \frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$;
  `/* change buckets */`
  for $v \in \mathcal{D}$ **do**
    if $gains[v] > 0$ **and** $\text{random}(0, 1) < probability[bucket[v], target[v]]$ **then**
      $bucket[v] \leftarrow target[v]$;

until *converged* **or** *iteration limit exceeded*;

策略2 Social Hash分区方法

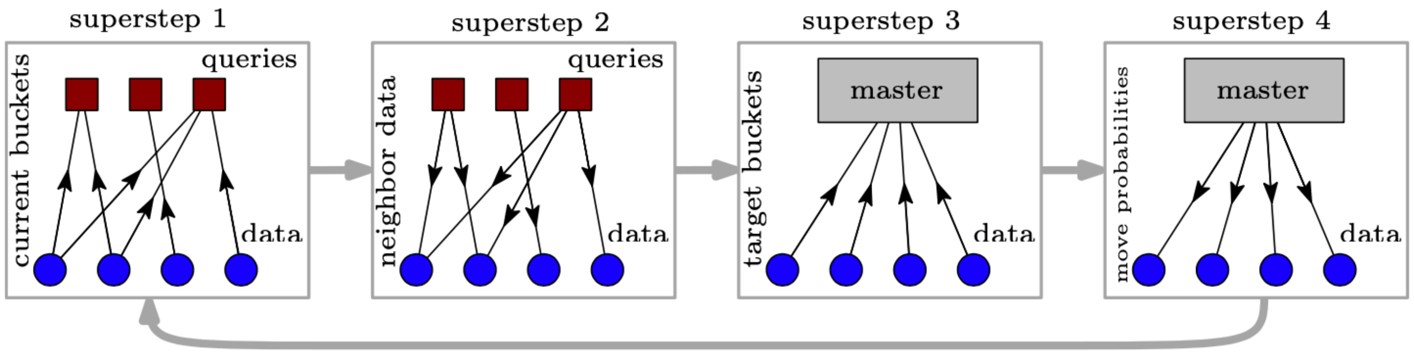
Social Hash分区算法的实现在以顶点为中心的框架中包括四个步骤：

（1）收集查询邻居数据；

（2）计算移动增益 (moving gain)；

（3）选出候选目标桶；

（4）计算交换概率并执行移动



Algorithm 1: Fanout Optimization

Input : graph $G = (\mathcal{Q} \cup \mathcal{D}, E)$, the number of buckets k, imbalance ratio ε

Output: buckets $V_1, V_2, \dots, V_k$

for $v \in \mathcal{D}$ **do** /* initial partitioning */

  $bucket[v] \leftarrow \mathrm{random}(1, k)$;

repeat /* local refinement */

  for $v \in \mathcal{D}$ **do**

    for $i = 1$ **to** k **do**

      $gain_i[v] \leftarrow ComputeMoveGain(v, i)$;

    /* find best bucket */

    $target[v] \leftarrow \arg\max_i gain_i[v]$;

    /* update matrix */

    if $gain_{target[v]}[v] > 0$ **then**

      $S_{bucket[v],target[v]} \leftarrow S_{bucket[v],target[v]} + 1$;

  /* compute move probabilities */

  for $i, j = 1$ **to** k **do**

    $probability[i,j] \leftarrow \frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$;

  /* change buckets */

  for $v \in \mathcal{D}$ **do**

    if $gains[v] > 0$ **and** $\mathrm{random}(0,1) < probability[bucket[v], target[v]]$ **then**

      $bucket[v] \leftarrow target[v]$;

until *converged* **or** *iteration limit exceeded*;

策略2 Social Hash分区方法

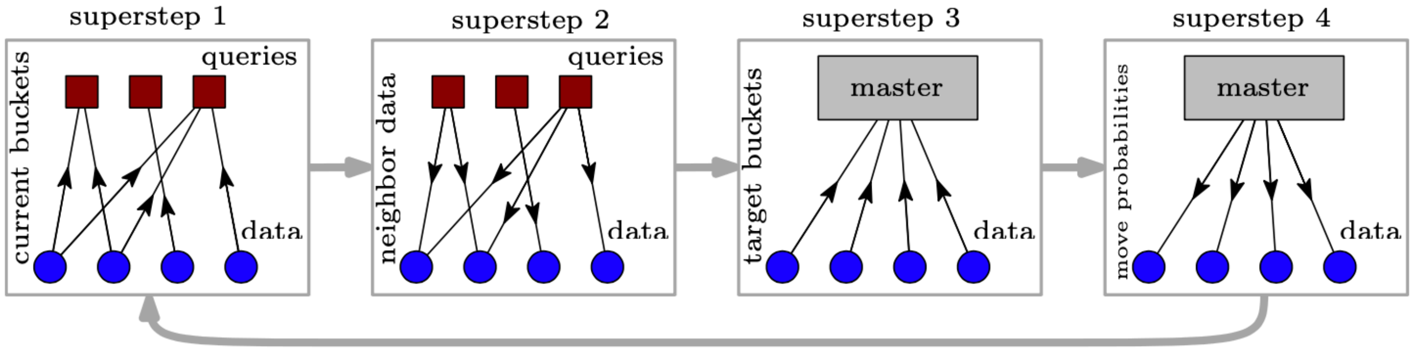
Social Hash分区算法的实现在以顶点为中心的框架中包括四个步骤：

（1）收集查询邻居数据；

（2）计算移动增益 (moving gain)；

（3）选出候选目标桶；

（4）计算交换概率并执行移动



Algorithm 1: Fanout Optimization

Input : graph $G = (\mathcal{Q} \cup \mathcal{D}, E)$, the number of buckets k, imbalance ratio ε

Output: buckets $V_1, V_2, \ldots, V_k$

for $v \in \mathcal{D}$ **do** `/* initial partitioning */`
  $bucket[v] \leftarrow \text{random}(1, k)$;

repeat `/* local refinement */`
  for $v \in \mathcal{D}$ **do**
    for $i = 1$ **to** k **do**
      $gain_i[v] \leftarrow ComputeMoveGain(v, i)$;
    `/* find best bucket */`
    $target[v] \leftarrow \arg\max_i gain_i[v]$;
    `/* update matrix */`
    if $gain_{target[v]}[v] > 0$ **then**
      $S_{bucket[v],target[v]} \leftarrow S_{bucket[v],target[v]} + 1$;

  `/* compute move probabilities */`
  for $i, j = 1$ **to** k **do**
    $probability[i, j] \leftarrow \frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$;

  `/* change buckets */`
  for $v \in \mathcal{D}$ **do**
    if $gains[v] > 0$ **and** $\text{random}(0, 1) < probability[bucket[v], target[v]]$ **then**
      $bucket[v] \leftarrow target[v]$;

until *converged* **or** *iteration limit exceeded*;

策略2 Social Hash分区方法

- 引入概率p，改变目标函数

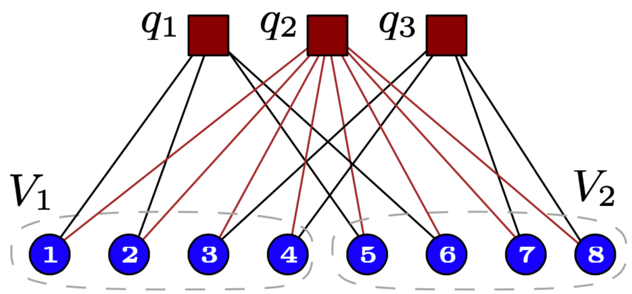
为了防止进入局部最小值，对于给定的分区 $P = \{V_1, \ldots, V_k\}$，让存储桶 V_i 中的数据顶点数相邻查询顶点 q 是 $1-(1-p)^{n_i(q)}$

因此，q 的 p-fanout 为

$$\sum_{i=1}^{k}(1-(1-p)^{n_i(q)})$$

目标函数修改为:

$$\frac{1}{|Q|}\sum_{q\in Q}\text{p-fanout}(q) = \frac{1}{|Q|}\sum_{q\in Q}\sum_{i=1}^{k}\left(1-(1-p)^{n_i(q)}\right).$$



fanout局部最小值的状态

策略2 Social Hash分区方法

■ 执行交换的方法

移动增益 (moving gain)：将一个顶点从其当前分区移动到另一个分区后，目标函数的变化

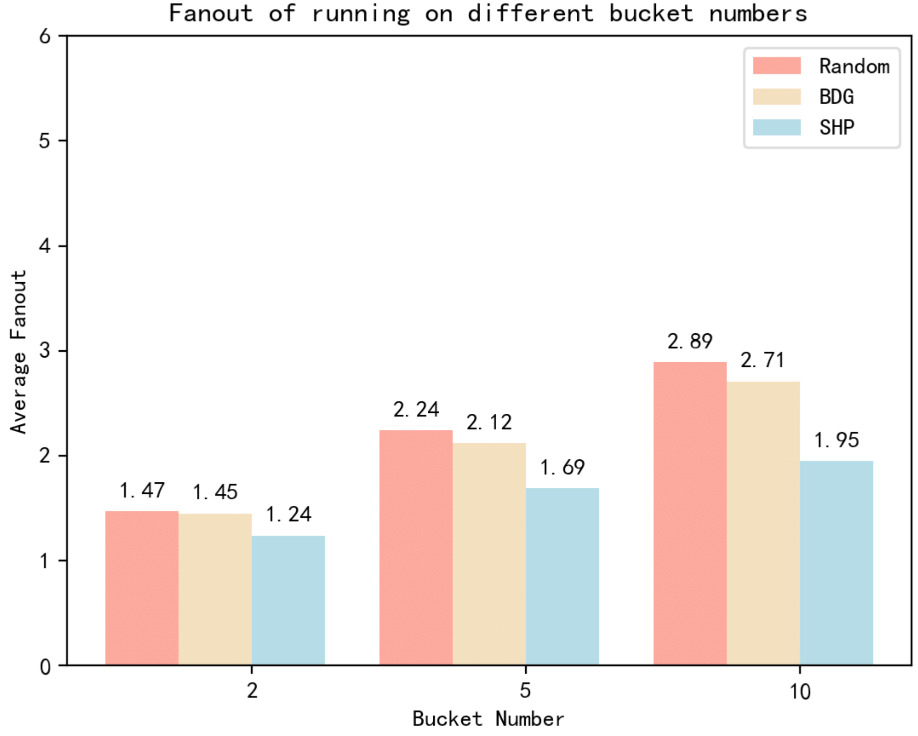
假设顶点 $v \in D$从分区 i 移动到分区 j，令 $N(v) \subseteq Q$ 为与 v 相邻的查询的子集，则移动增益为:

$$\begin{aligned} gain_j(v) &= \sum_{q \in \mathcal{N}(v)} \left(2 - (1-p)^{n_i(q)-1} - (1-p)^{n_j(q)+1}\right) \\ &- \sum_{q \in \mathcal{N}(v)} \left(2 - (1-p)^{n_i(q)} - (1-p)^{n_j(q)}\right)\Big) \\ &= p \cdot \sum_{q \in \mathcal{N}(v)} \left((1-p)^{n_j(q)} - (1-p)^{n_i(q)-1}\right). \end{aligned}$$

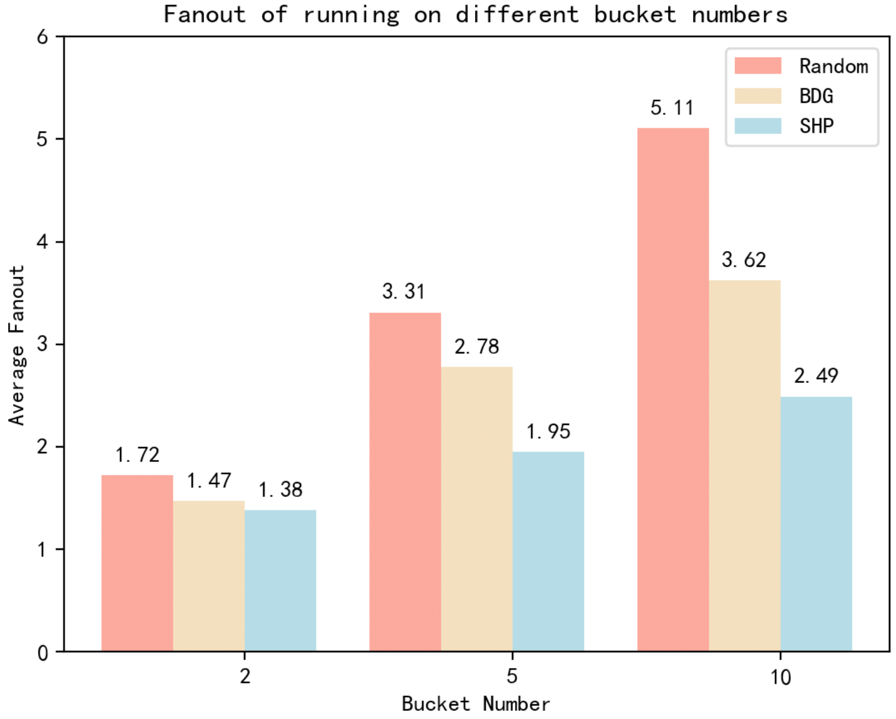
算法性能

实验数据集	点数	边数
Youtube	124,325	293,360
LiveJournal	3,997,962	34,681,189

实验数据集点数和变数

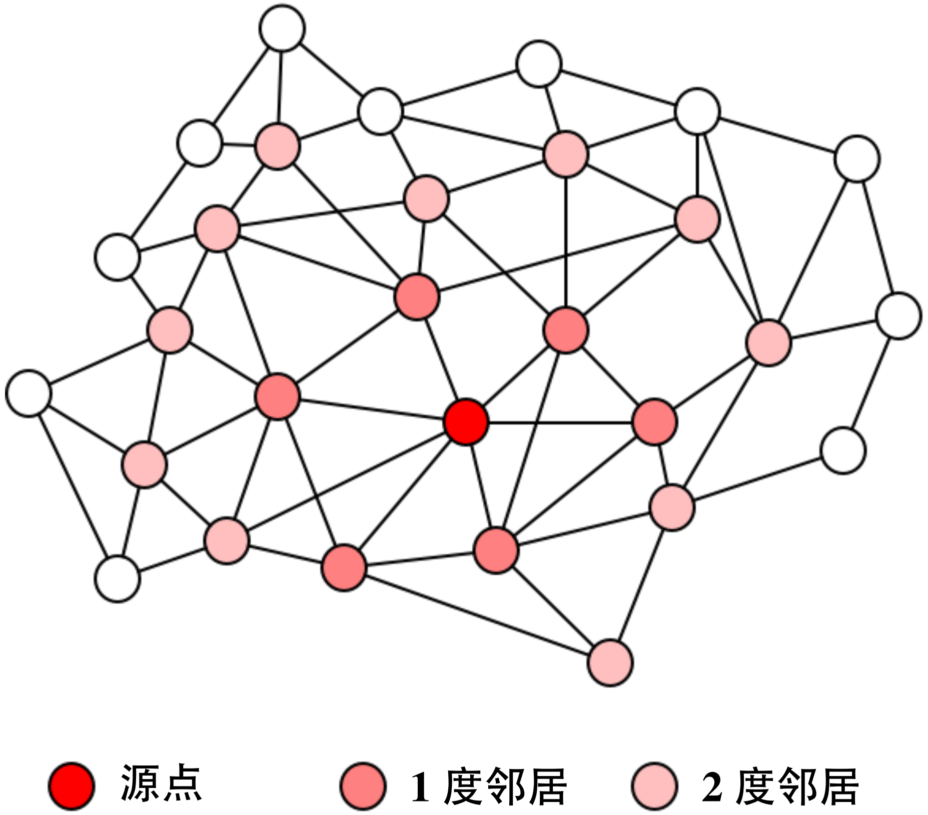


三种分区方法在不同分区个数下的 **fanout** 值（**Youtube**）



三种分区方法在不同分区个数下的 **fanout** 值（**LiveJournal**）

策略2优化 移动增益的动态计算——*f*SHP



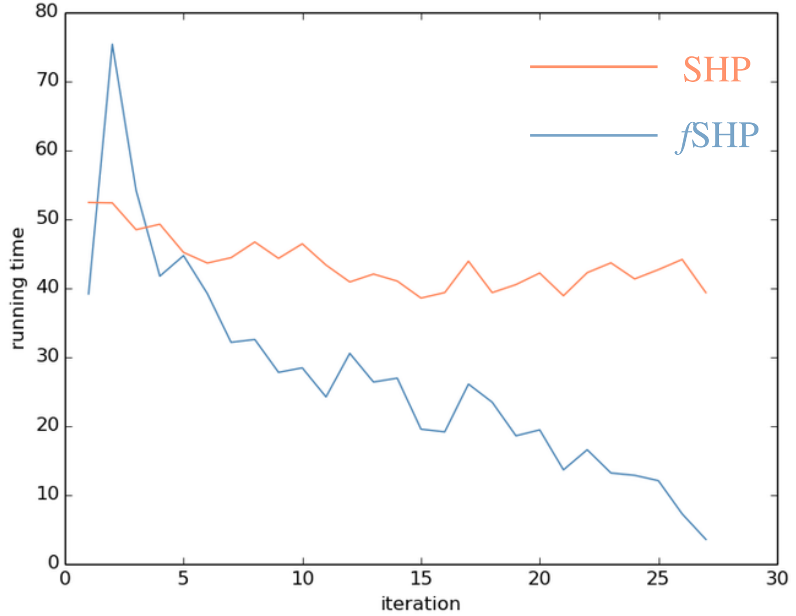
- 优化

  每次修改单个顶点的分区后，只会对顶点的1 度邻居(nbr_i)的分区统计有影响，并对顶点的2 度邻居移动增益有影响；多次迭代之后，需要修改的顶点很少、2 度邻居很少的情况下，可以只对此次迭代进行分区修改的顶点的2 度邻居重新计算移动增益

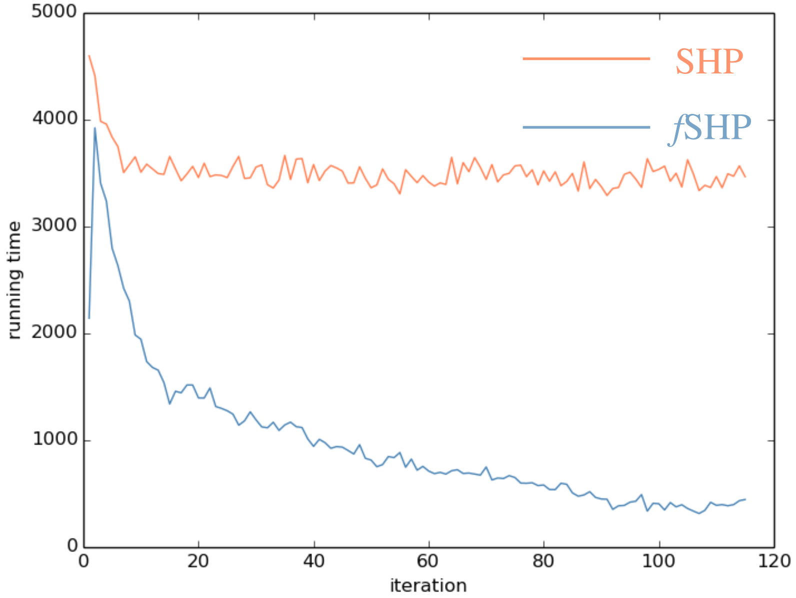
策略2优化 移动增益的动态计算——*f*SHP

优化性能对比

在相同的迭代次数下，实验运行速度可提升三倍（在LiveJournal数据集中，实验运行时间从404.2s降低到132.2s）



运行时间随着迭代次数的变化
(Youtube)



运行时间随着迭代次数的变化
(LiveJournal)

策略2优化 分区内先排序后交换——*x*SHP

BC 2020 BYTE CAMP

- 优化

（1）先排序后交换 (xSHP)：在当前分区内部进行排序，选择移动增益最大的 m 个顶点进行分区更换；

（2）排序方式：对要转移的分区中的 N 个节点排序，取最大的 m 个，复杂度为 $O(N\log N)$；通过模仿快排的方法寻找前 m 大的顶点，可以将复杂度降低为 $O(N)$；

（3）进一步优化：在上述基础上，如果不选取严格前 m 小的顶点，可以并行在 x 个 segment 中每个选择，每个 segment 中选择 m / x个顶点

策略2优化 分区内先排序后交换——*x*SHP

优化

（1）先排序后交换 (*x*SHP)：在当前分区内部进行排序，选择移动增益最大的 m 个顶点进行分区更换；

（2）排序方式：对要转移的分区中的 N 个节点排序，取最大的 m 个，复杂度为 $O(N\log N)$；通过模仿快排的方法寻找前 m 大的顶点，可以将复杂度降低为 $O(N)$；

（3）进一步优化：在上述基础上，如果不选取严格前 m 小的顶点，可以并行在 x 个 segment 中每个选择，每个 segment 中选择 m / x个顶点

优化性能对比

分区个数	2	5	10
SHP	1.24	1.69	1.95
***x*SHP**	1.23	1.56	1.86

SHP 和 *x*SHP 在不同分区个数的 fanout 值（Youtube）

分区个数	2	5	10
SHP	1.38	1.95	2.49
***x*SHP**	1.35	1.89	2.48

SHP 和 *x*SHP 在不同分区个数的 fanout 值（LiveJournal）

未来工作展望

- **数据分区方式**

  当前是按照顶点进行线程数据分块，但由于超级顶点存在的可能性，如果按照边做线程数据分块，每个线程的任务将会更加平均

未来工作展望

- **数据分区方式**

  当前是按照顶点进行线程数据分块，但由于超级顶点存在的可能性，如果按照边做线程数据分块，每个线程的任务将会更加平均

- **对动态图的支持**

  当出现新增节点时，将它加入它的相连边最多的分区里；

  当节点出现新增边时，需要重新计算这个节点的 1 度邻居的移动增益和交换概率，进行可能的局部交换，达到对原算法一次迭代的近似

未来工作展望

- **数据分区方式**

当前是按照**顶点**进行线程数据分块，但由于**超级顶点**存在的可能性，如果按照**边**做线程数据分块，每个线程的任务将会更加平均

- **对动态图的支持**

当出现**新增节点**时，将它加入它的相连边最多的分区里；

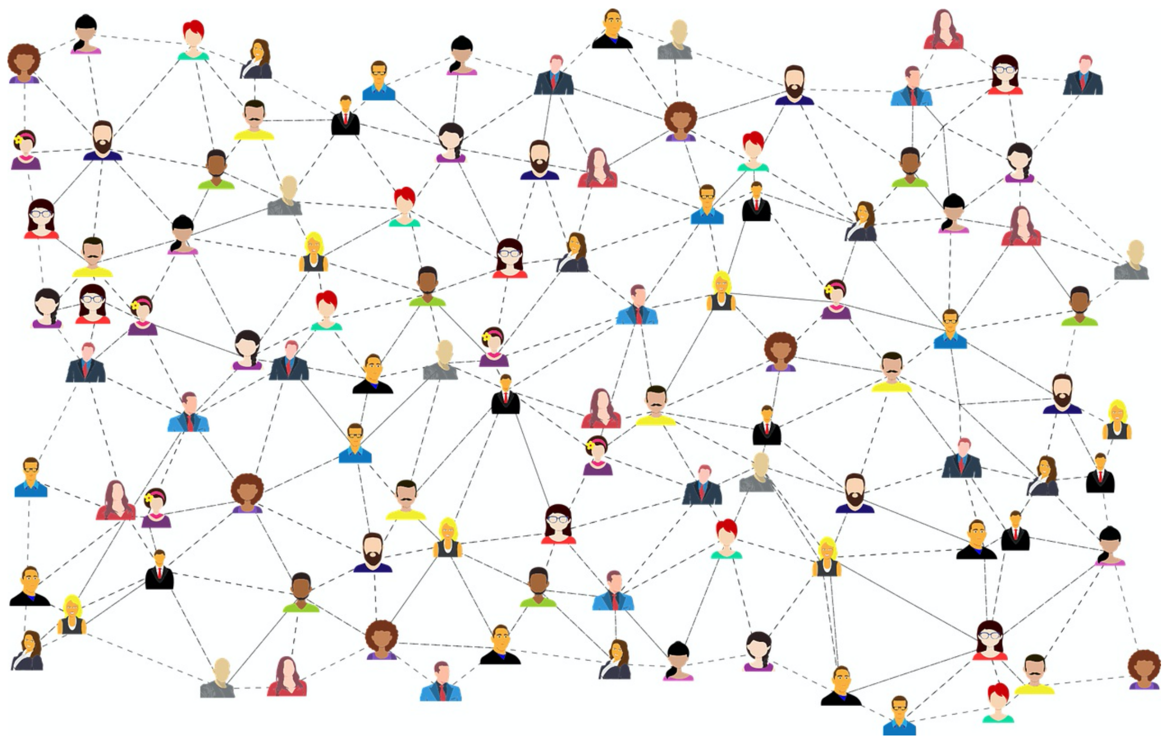
当节点出现**新增边**时，需要重新计算这个节点的 1 度邻居的移动增益和交换概率，进行可能的局部交换，达到对原算法一次迭代的近似

- **计算交换概率和比较的方式**

假设分区节点数量为 N，随着迭代的进行，此分区中需要交换节点数量 M 会越来越少 ($M < N$)，计算交换概率和比较的复杂度可以从 O(N) 降到 O(M)

for $v \in \mathcal{D}$ **do**
  if $gains[v] > 0$ **and** $\mathrm{random}(0,1) < probability[bucket[v], target[v]]$ **then**
    $bucket[v] \leftarrow target[v]$;

项目贡献

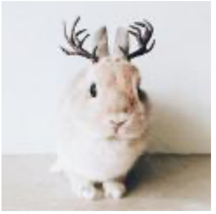


- 在阅读大量论文的基础上，实现了random，BDG，Social Hash三种图分区算法，并且在**真实世界数据集**上进行了验证
- 针对Social Hash，**提出并实现了两点改进**（最终实验运行速度提升 **3 倍**，实验指标fanout 提升 **7 个点**），并提出其他未来可以进行的工作

小组分工

程苗苗 / miamia0

Social Hash 分区算法实现
论文阅读
算法性能实验
代码 review



苏金涛 / sujintao666

Social Hash 分区算法优化
论文阅读
算法性能实验
代码 review

孙泽嵩 / samperson1997

BDG 分区算法实现
论文阅读
算法性能实验
代码 review
PPT 制作

参考文献

1. Bronson, Nathan, et al. "{TAO}: Facebook's distributed data store for the social graph." Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13). 2013.
2. Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 2010.
3. Low, Yucheng, et al. "Distributed graphlab: A framework for machine learning in the cloud." arXiv preprint arXiv:1204.6078 (2012).
4. Gonzalez, Joseph E., et al. "Powergraph: Distributed graph-parallel computation on natural graphs." Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). 2012.
5. Gonzalez, Joseph E., et al. "Graphx: Graph processing in a distributed dataflow framework." 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). 2014.
6. Zhu, Xiaowei, et al. "Gemini: A computation-centric distributed graph processing system." 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016.
7. Kyrola, Aapo, Guy Blelloch, and Carlos Guestrin. "Graphchi: Large-scale graph computation on just a {PC}." Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). 2012.
8. Roy, Amitabha, Ivo Mihailovic, and Willy Zwaenepoel. "X-stream: Edge-centric graph processing using streaming partitions." Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013.
9. Shun, Julian, and Guy E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory." Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming. 2013.
10. McSherry, Frank, Michael Isard, and Derek G. Murray. "Scalability! But at what {COST}?." 15th Workshop on Hot Topics in Operating Systems (HotOS {XV}). 2015.
11. Aditya Auradkar, Chavdar Botev, Shirshanka Das. "Data Infrastructure at LinkedIn "2012 IEEE 28th International Conference on Data Engineering

THANKS

MENTOR / 胡英谦 陈 超

组 员 / 程苗苗 苏金涛 孙泽嵩