# **Clique BK算法流程**

BK通过回溯搜索的方式来检查所有的点以枚举所有的极大完全图。BK搜索过程中的搜索路径组成了一个树状搜索结构。BK算法每次访问一个与搜索路径中已经访问的所有节点都相邻的点来扩展搜索，直到搜索路径不能再扩展，这样一条搜索路径上的所有访问点就形成一个极大完全图。

算法BKCliqueEnumerate显示了BK搜索树的访问过程。搜索树的每一个节点由下面三个点集组成，也就是搜索树的节点状态表示结构。

1. Result集，现有的搜索路径中已经访问过的节点集合
2. Candidate集，不在Result集中且与Result中的每一个点都相邻的点集
3. Not集，与Result中点都相邻且如果与Result集中的点组合将导致产生冗余或者被包含的结果的点集

算法BKCliqueEnumerate在访问一个节点时将其加入到现有结果集中并重新构建新的Candidate集合Not集。新的Candidate集通过筛选现有的候选集要与包括新加入到结果集中的点在内的所有Result集中的点都相邻。同样Not集也是通过筛选现有的Not集要求与包括新加入的点在内的所有Result点都相邻。BKCliqueEnumerate算法通过访问候选集中的点。对于每一个搜索树节点，首先访问的点是当前候选点中连接了最多候选点的点（也就是在候选点组成的子图中度数最大的点）。在搜索完第一个点之后只用那些与第一个点不相邻的候选点来扩展搜索路径。这样保证了在每一节点的搜索路径中只搜索最少的那些有可能生成新的且不冗余的点。一旦候选点Candidate中的任意一个点被访问后都会将其加入到Not集中，以表示包含这个点和当前结果集中的的极大完全图已经在新生成的子图状态中考虑了，搜索树中其他兄弟子图节点状态不应当在搜索相关节点。当BKCliqueEnumerate已经搜索完所有的子节点状态后将回溯到之前的一个未搜索子图状态。显而易见，BKCliqueEnumerate的搜索过程可以通过栈来实现。

|  |
| --- |
| **算法1**：BK算法 |
| **输入**：数据图G，图G的节点集V以及边集E  **输出**：所有的且不重复、不冗余的极大完全图 |
| 1. Result 🡨 2. Candidate 🡨 V 3. Not 🡨 4. 调用方法BKCliqueEnumerate (Result, Candidate, Not) |

BK算法

|  |
| --- |
| **方法**BKCliqueEnumerate (Result, Candidate, Not) |
| 1. **If** Candidate =  **then** 2. **If** Not =  **then** 3. 输出结果Result 4. **Else** 5. fixp 🡨 Candidate 中与其他Candidate中相邻点最多的点 6. cur\_v 🡨fixp; 7. **While** cur\_v  NULL **do** 8. new\_not 🡨Not集中所有与cur\_v相邻的点 9. new\_cand 🡨 Candidate集中与cur\_v相邻的点 10. new\_res 🡨 Result + cur\_v 11. **CliqeEnumerate**(new\_res, new\_cand, new\_not) 12. Not 🡨 Not + cur\_v 13. Candidate 🡨 Candidate – cur\_v 14. **If** Candidate中存在点v与fixp不相邻 **then** 15. Cur\_v 🡨 v 16. **Else** 17. Cur\_v 🡨 NULL 18. **return** |

递归极大完全图枚举方法







## Clique Binary算法

BK算法在执行过程中需要计算每个子图中各个候选节点的度数，但只是用来寻找度数最大的某个点，关于其他点的度数计算结果并没有充分使用。同时，观察到许多算法没有使用到完全图一个明显的特征没有使用：。另外不同于BK算法取最大度数点然后取所有与最大度数点不相邻的点来扩展搜索路径的粗粒度方式，本文提出一种高效地衡量当前状态动态地选择扩展点的搜索算法：Binary算法。

在BK的数据结构设计的基础上当前子图状态表示为，Binary算法的基本过程是在当前候选节点集中选择一个扩展点（分裂点）并将当前搜索图G划分为包含当前G的结果集且包含点的所有Clique，生成新的子图，子图状态通过如下方式生成，，；包含当前G的结果集且不包含点的所有Clique，生成新的子图子图状态通过如下方式生成，，。

下面证明这种划分方式的正确性。对于当前子图G的状态的子分支只有两种情况，包含点和不包含点，不存在第三种情况。对于包含点的状态，结果Clique中如果包含点，那么可以将点加到结果集中即：；状态的候选点由于需要满足包含节点的条件，因此需要保证候选点也必须都是的邻接点即；状态的Not集需要满足Not集的定义，Not集中的点可以与当前结果集节点组成完全图，因此Not集中点需要能够与相邻即：。对于不包含节点的状态，只将点从候选点中删除，同时由于Not的定义且包含点的子图必然都会在中处理，因此将点加入到Not集中，这样保证了所有不包含节点的完全图都是的子图。有以上可得证Binary划分方式的正确性。

算法的形式化描述

|  |
| --- |
| **算法1**：Binary算法 |
| **输入**：数据图G，图G的节点集V以及边集E  **输出**：所有的且不重复、不冗余的极大完全图 |
| 1. Result 🡨 2. Candidate 🡨 V 3. Not 🡨 4. 调用方法BinaryCliqueEnumerate(Result, Candidate, Not) |

|  |
| --- |
| 方法BinaryCliqueEnumerate(Result, Candidate, Not) |
| 1. If Not中存在一个点与Candidate中所有点都相邻 then 2. 此子图往下都是冗余结果，返回 3. If SubGraph(Candidate)是Clique then 4. 输出 ResultCandidate 5. Else 6. 在Candidate中选择一个分裂点v 7. 🡨 8. 🡨 9. 🡨 10. BinaryCliqueEnumerate() 11. 🡨 12. 🡨 13. 🡨 14. BinaryCliqueEnumerate() |

Binary的划分方式需要每次从当前子图状态中找到一个分裂点，如何选择分裂点对算法的效率有着至关重要的影响。考虑到图数据的各种特征且充分利用现有信息，本文使用候选点的度数作为选择切分点的衡量指标。显然，用度数作为选择指标有三种选择方式：选择度数最小的点、选择度数最大的点、随机选择一个点。根据Binary的划分方式，包含点的子图候选点的规模，子图候选点的规模。在给定子图G的情况下是不变的，由此可见选择最小度数点作为切分点则会生成一个较大的子图和一个较小的子图可以减少搜索空间。下图是最大和最小度数切分的一个对比实例，后续的实验也验证了这一论断。

 使用Binary算法处理同样的输入图，其搜索树的结构如下所示。图（a）是算法描述的Binary搜索树结构左子节点表示G-右子节点表示G+。由于递归算法的栈消耗以及潜在的内存使用问题，本文采用栈记录子图状态的迭代方式实现Binary的递归算法。实际上在算法迭代过程中会将栈顶子图一直选最小度数点切分直到栈顶子图成为一个完全图或者栈顶子图已经不可能形成新的有意义的完全图。迭代算法切分过程中G-是一个可变化的子图状态，将左子树节点G-都映射到同一个子图状态，如图（b）所示。Binary算法的划分方式可以在很大程度上减少搜索子图的个数，这一点在后续的试验中得到了有效验证。



Binary算法的实现中使用与BK相似的数据集合定义Result，Candidate，Not。不同的是，Binary算法需要始终跟踪Candidate节点中度数最小的点，而且在算法执行过程中由于需将与分裂点相邻的点度数减一，Candidate中节点的度数不断变化的。对此，最直接的方法是每次需要选度数最小的点时遍历当前所有候选点的度数，从中选出度数最小的点。假设子图候选点个数为n，Binary算法需要切分子图O(n)次，第i次切分需要筛选n-i个节点个数，因此在筛选最小度数点上需要的时间复杂度，在先期的实验中此部分成为算法效率的一个主要瓶颈。本文设计了一个最小度数结构用来更新和维护各个候选节点的度数，提供了O(1)的最小度数点选取操作，以及O(1)每个节点度数更新的操作。

如下图（a）所示圆形表示度数，方形表示节点标号。最小度数结构将度数相同的节点放在一个哈希桶内并用它们的度数作为桶的标志，圆形度数标志之间形成一个有序的链表结构，同时每个节点标号有指向本身所在桶的引用。

在图（a）表示的结构中进行图分割时，从度数链表选取头结点所指向的痛中选取第一个节点4（标号为4的节点度数为1，假设候选点中与4号节点相邻的是标号为5的点）作为分裂点，在不包含节点4的子图中需要更新度数集合，将与4相邻的节点5的度数减一。度数减一操作如图（b）所示，将节点5从原来的桶中移除并将其加到其之前度数小一的桶中（不存在的话需新建一个桶，并将度数节点插入到原链表中）同时更新节点5所在的桶引用。其中所有操作都是O(1)复杂度，建立这个最小度数集合的额外复杂度是O(klogk)（k表示所有候选点的不同度数个数）。

## Clique Hybrid算法

BK算法和Binary算法一个重要的区别在于关于图分割的定义，BK算法将图分割为包含最大度数节点的子图其余与该店不相邻的点依次罗列，相当于一次选点即可将当前搜索图全部切分完；Binary算法每次选用最小度数点将子图划分为多个小的子图，多次迭代分步将搜索图切分掉。在不同的情况下两者存在着各自的优势，当图中存在一个节点的与大部分节点都相邻时BK算法可以将图有效的切分为较少的几个子图；当图中节点度数偏小时Binary算法可以有效地将图切分为冗余较少的且易计算的多个小子图。鉴于实际应用中数据图的多样性以及算法的适用性，本文提出了一种Binary和BK综合的算法Hybrid。

由前两章的内容可以看出BK算法和Binary算法在搜索树节点状态的整体结构上相似，因此两者可以较方便地实现融合。Hybrid算法以Binary算法为基础，在选择分裂点时考虑最小度数点邻接的候选节点比例，当被切割的图中最小度数点都邻接了大部分节点时，使用BK的切分方式从候选点中选择度数最大的点作为分裂点来切分搜索图，否则还是使用最下度数点分割搜索图。由上一章中最小度数集合的设计可以看出，选最大度数点也只需从链表中取尾节点的桶中取任意一个点即可。

Hybrid算法的实现基础，BK和Binary都有充分的实现条件，不需做数据结构上的改动

|  |
| --- |
| 方法HybridCliqueEnumerate(Result, Candidate, Not) |
| 1. **If** Not中存在一个点与Candidate中所有点都相邻 **then** 2. 此子图往下都是冗余结果，返回 3. **If** SubGraph(Candidate)是Clique then 4. 输出 ResultCandidate 5. **Else** 6. 从最小度数结构中选取度数最小的点 7. **If** Deg()> **then** 8. 从最小度数结构中取度数最大节点 9. cur\_v 🡨; 10. **While** cur\_v  NULL **do** 11. new\_not 🡨Not集中所有与cur\_v相邻的点 12. new\_cand 🡨 Candidate集中与cur\_v相邻的点 13. new\_res 🡨 14. **CliqeEnumerate**(new\_res, new\_cand, new\_not) 15. Not 🡨 16. Candidate 🡨 17. **If** Candidate中存在点v与fixp不相邻 **then** 18. Cur\_v 🡨 v 19. **Else** 20. Cur\_v 🡨 NULL 21. **Return** 22. **Else** 23. 🡨 24. 🡨 25. 🡨 26. BinaryCliqueEnumerate() 27. 🡨 28. 🡨 29. 🡨 30. BinaryCliqueEnumerate() |

Hybrid算法中使用到了一个变量，表示最小度数点与候选点中邻接个数比例，实验部分中验证该值一般取0.8能够获得较好的效果。

## K-Plex单机算法Pemp

K-Plex算法是BK算法的一种变形，其本质上也是图的分割。为简化下文算法的描述有如下定义：

定义1：顶点与顶点集合相邻，要求满足用表示；顶点与不相邻，要求满足用表示。

K-Plex算法Pemp与完全图算法相似，也有关于搜索节点相似的定义Result，Candidate，Not。在K-Plex中用来扩展现有搜索路径的Candidate候选节点v需要满足以下两个条件

条件1：v与Result中至少个点相邻

条件2：与中至少个点相邻

显然任意一个满足条件1和条件2的点都可以与现有的结果集组成一个新的更大的K-Plex。根据Candidate和Not集合的定义，Candidate和Not集中的点都需要满足条件1和条件2。 在Pemp算法中为Result、Candidate和Not集中点都关联了一个计数：counter1。counter1用来记录其关联的顶点与Result中的多少个点不相邻，其中Result的counter1表示Result中的点与中多少个点不相邻。为了满足条件1，只需要保证点的。对条件2，考虑到每次只想结果集中扩展一个新的节点，有如下临界点集定义。

定义2：搜索路径上一个子图节点的结果集Result中的点与中k-1点不相邻，则称是临界点；由临界点组成的集合即临界点集合Critical。

为了满足条件2，当搜索路径子图中临界点集合Critical不为空时，要求可用来扩展的Candidate中的任意一点必须与临界点集合中的所有点都相邻。否则，那么当不满足此条件的点用来扩展现有路径时中会至少存在一个临界点与K个点不相邻，也就是违背了K-Plex的必要条件。

|  |
| --- |
| KPLEX单机算法Pump |
| **输入**：数据图G，图G的节点集V以及边集E  **输出**：所有的且不重复、不冗余的KPlex |
| 1. Result 🡨 2. Candidate 🡨 V 3. Not 🡨 4. **While** Candidate 5. 依次取 6. Result 🡨 Result + cur\_v 7. 调用方法**FindAllMaximalKplex**(Result, Candidate, Not) 8. Not 🡨 Not + cur\_v 9. Candidate 🡨 Candidate – cur\_v 10. Result 🡨 Result – cur\_v |

KPLEX单机算法Pump

|  |
| --- |
| 方法FindAllMaximalKplex(Result, Candidate, Not) |
| 1. Connected\_Candidate 🡨 Candidate中与Result中任意一个点相邻的点 2. Connected\_Not 🡨 Not中与Result中任意一个点相邻的点 3. **While** Connected\_Candidate 4. **If** Connected\_Candidate 5. **If** Connected\_Not 6. 输出Result 7. **Else** 8. Result是冗余结果 9. cur\_v🡨**SelectExpandNode**(Result, Critical\_Res, Connected\_Candidate, Connected\_Not) 10. **If** cur\_v = NULL 11. **Return** 12. Candidate 🡨 Candidate – cur\_v 13. Cur\_cand 🡨 Candidate; Cur\_not 🡨 Not; Cur\_res 🡨 Result 14. Cur\_res 🡨 Cur\_res + cur\_v 15. 更新Cur\_cand和Cur\_not中各个点的计数，同时移除计数counter1大于k-1的点 16. Critical\_Res 🡨 Result中与其他节点不相邻个数等于k-1的点 17. 移除Cur\_cand和Cur\_not中与Critical\_Res不相邻的点 18. FindAllMaximalKplex(Result, Candidate, Not) 19. Not 🡨 Not + cur\_v 20. Connected\_Not 🡨 Connected\_Not + cur\_v |

FindAllMaximalKplex方法

|  |
| --- |
| 方法SelectExpandNode(Result, Critical\_Res,Connected\_Candidate, Connected\_Not) |
| 1. **If** Result  && Critical\_Res 2. **Return** Connected\_Candidate中的第一个点 3. **Else** 4. Prunable\_Not 🡨 Connected\_Not中与Result中所有点都相邻的点 5. **If** Prunable\_Not 6. **If** 7. **Return** NULL 8. **Else** 9. **Return**, 10. **Else** 11. **Return** Connected\_Candidate中的第一个点 |

Pump寻找扩展点方法

算法去冗余的方法是通过最后Connected\_Candidate为空且Connected\_Not不为空表示Connected\_Not中的任意一个点都可以和现有的结果集组成一个K-plex，但是包含点的所有K-plex已经在搜索路径的其他分支上已经生成了，故而无需输出。如果可以在搜索路径的过程中提前判断Connected\_Not中存在一个点不可能被过滤掉，那么就可以保证这个节点下的整棵子树都是冗余的，可以提前剪枝。如果Connected\_Not中存在一个点与Result和Candidate中的所有点都相邻，那么在此之后无论如何扩展这个搜索路径，都无法通过条件1和条件2来将这个点过滤掉。也就是说在此之后所有的结果中Connected\_Not都不为空，此分支可以剪去。

剪枝条件：

在算法中为Not关联一个新的计数器counter2，counter2表示Not中的点与Result和Candidate中有多少个点不相邻。当Not中存在点的counter2 = 0时也就是剪枝条件满足了。

为了加快剪枝速度，算法尽可能的使得counter2降到0。为此提出一个新的集合Prunable\_Not，Prunable\_Not是Not的一个子集，包含Not中与Result集中所有点都相邻的点，也就是counter1=0的那些点。注意到，每当选择一个新的点v加到Result中来扩展搜索时，Not中与v不相邻的点的counter2计数就会每次降1。如果一直从Candidate中选与Prunable\_Not集中某个点v不相邻的那些点来扩展，那么点v的counter2很快降到0且将不可能存在Not集中的其他点会比v更快满足剪枝条件。显然，为了能够最快地达到剪枝条件应该选择Connected\_Candidate中与Prunable\_Not中counter2最小的点v相邻的那些点。但是在算法刚开始时还不存在临界点集合时Connected\_Candidate中不存在与Prunable\_Not不相邻的点，因此还是需要按顺序取扩展点。详细过程在方法SelectExpandNode中描述。

## K-Plex Binary算法

K-Plex的Binary算法使用与完全图相似的切分方式。K-Plex的Binary算法中对一个子图状态的节点关联两个度数cDeg和rDeg。cDeg表示关联的节点与Candidate中有多少个点不相邻，rDeg表示关联的节点与Result中有多少个点不相邻。判断一个Candidate是否能够形成K-Plex可以通过判断其中点的cDeg+rDeg<k-1。参考文献指出有意义的K-Plex的所有节点必然包含在两跳以内，因此本文在使用候选节点时挑选两跳数据节点。

|  |
| --- |
| KPLEX单机算法Binary |
| **输入**：数据图G，图G的节点集V以及边集E  **输出**：所有的且不重复、不冗余的KPlex |
| 1. Result 🡨 2. Candidate 🡨 V 3. Not 🡨 4. **While** Candidate 5. 依次取 6. Result 🡨 Result + cur\_v 7. TwoHop 🡨 8. 调用方法**BinaryMaximalKplex**(Result, ,) 9. Not 🡨 Not + cur\_v 10. Candidate 🡨 Candidate – cur\_v 11. Result 🡨 Result – cur\_v |

|  |
| --- |
| 方法BinaryMaximalKplex(Result,Candidate,Not) |
| 1. 计算最小度数结构、临界点Critical、 2. Prunable\_Not 🡨 Not中rDeg=0的点 3. While Candidate 还有切分的意义 4. If Prunable\_Not 中存在cDeg=0的点 5. 此分支可以减掉；Return 6. If 7. 输出为K-Plex；Return； 8. 从最小度数结构中选出rDeg+cDeg最大的点cur\_v   12. 更新Result中各点的rDeg和cDeg 13. 🡨 14. BinaryCliqueEnumerate () 15. 🡨 16. 更新Cur\_cand和Cur\_not中各个点的计数，同时移除计数cDeg大于k-1的点 17. Critical\_Res 🡨 Result中与其他节点不相邻个数等于k-1的点 18. 移除Cur\_cand和Cur\_not中与Critical\_Res不相邻的点 19. Not 🡨 Not + cur\_v 20. If 21. Prunable\_Not 🡨 Prunable \_Not + cur\_v |

在K-Plex问题中注意到K-Plex问题比完全图问题更加复杂，搜索空间是K-Plex最主要的性能瓶颈。事实上Pemp算法的剪枝策略并没有很好的提前剪枝，其搜索空间依然十分庞大，本文Binary的K-Plex算法继承了Binary算法中减少搜索空间以提高搜索效率的优势，在搜索空间上远小于Pemp算法，并在实验中得到了验证。

## 算法并行化

本文旨在研究图算法以及其高效并行化。在并行化方面，本文采用分布式计算平台Hadoop使用MapReduce编程模型实现。并行算法的可用性和高效性主要通过算法的可分割性以及可均衡性来实现。算法可分割便可以并行化，算法可均衡负载则并行算法可以解决长尾问题，使得算法能够有很好的扩展性从而能够通过机器数目的简单叠加获得计算时间的降低，提高吞吐量。本文提出的三个算法都可以较好地满足可分割性和可均衡性。

对于可分割性：综合以上完全图和近似完全图的几个算法可以发现它们本质上都是对于图的一个树状搜索，在搜索树路径上任意一个子树的搜索过程都与其不相交的另一个子树不存在任何依赖关系，因而可以较方便地将不同的子树分配到多个机器节点上并行地进行运算。另外，由于不相交子树间没有信息依赖，因而可以适用于Hadoop这类Share-Nothing的计算平台。

对于可均衡性：在算法真正遍历完搜索之前，并不能够从子图数据中准确计算单个子图的搜索代价，同时由于实际数据存在大量的计算倾斜，以上算法在实际的分布环境中都存在负载不均衡的问题。得益于分割后的子图的独立性，在Hadoop平台上，算法可以暂停当前任务，将剩余子图及新分割出来的子图通过Hadoop的Shuffle过程随机分发到各个计算单元上进行新一轮的计算，通过多轮作业迭代算法可以较好地均衡各节点间的计算负载。

此外，要提升并行算法的加速比，还需要尽可能降低算法并行化所带来的额外消耗。Hadoop平台中各个Slave节点之间是异构、等价的，Slave节点之间不直接交互作业信息。同时Hadoop系统中所有中间结果或者数据都通过磁盘进行持久化，并行算法需要考虑尽可能地降低磁盘读写数据，减少数据传输量。

极大完全图枚举和极大K-Plex枚举在Hadoop并行化算法上具有相似的结构，包括读入输入图数据、初始任务分配、负载均衡和终止条件。

### 极大完全图Maximal Clique并行

传统检测算法中一般使用两跳数据集作为输入数据集，一个点的两跳数据可以保证找到包含这个点的所有极大完全图的正确性和完整性。引入两跳数据集的初衷是用来减少不相关点的干扰，提前筛去无意义点，但实际数据中两跳数据集的筛选能力较差，依然导致大量无效的中间输出。注意到所有有实际意义的极大完全图（组成极大完全图的点的个数不小于4的完全图）都是由多个三角形构成的。

### 极大K-Plex并行

## 负载均衡

### 面向应用的负载均衡

### 面向平台的负载均衡