图生成与分解器

李子龙
电子信息与电气工程学院
计算机科学与技术专业
518070910095 F1903301

2020 年 12 月 25 日

1 问题重述

1. 开发一个图自动生成器
   随机生成一个有向图，将图放置到指定文件中，每一行如下格式
   - ⟨节点编号⟩：节点
   - ⟨出发节点编号,结束节点编号,权重⟩：有向边
2. 开发一个图分解器
   - 分割图文件
     - 将上述图分为若干子图，每个子图中节点数不大于n。
     - A图分割后，每个子图可以用单独的文件保存：如 `A1,A2,A3`,$\cdots$
     - 令子图之间的交互（即能够跨越子图边界的边）权重之和最小,我们将挑选若干自动生成的图，对比大家生成的权重之和值。在结果正确的前提下，计算权重之和越小，分数越高。
   - 优化子图存储
     上述图分割算法导致分割成的多个子图之间存在重复的节点，请设计一个方法，使
     - 多个子图文件中分别载入程序后，不存在重复的节点
     - 每个子图可以最多增加一个虚节点（如子图的文件名），代表外界（即其他子图）对该子图的引用
     - 设计一个算法，将多个子图合并及删除虚节点后，检查与原图A一致。输出分割边的权重和。
   - 子图上算法
     - 指定一个点，列出计算所有经有向边可达的节点

- 指定两个点，输出最短路径
- 如果指定的节点不存在，报错即可

2 公共类 GraphCommon

在开始的无 UI 编译时期，存储为同一个公共类 GraphCommon，后面因为需要使用全局的非静态变量，将头文件分离。该类的完整版本还是请看 GraphDecomp.h 中的相关定义。

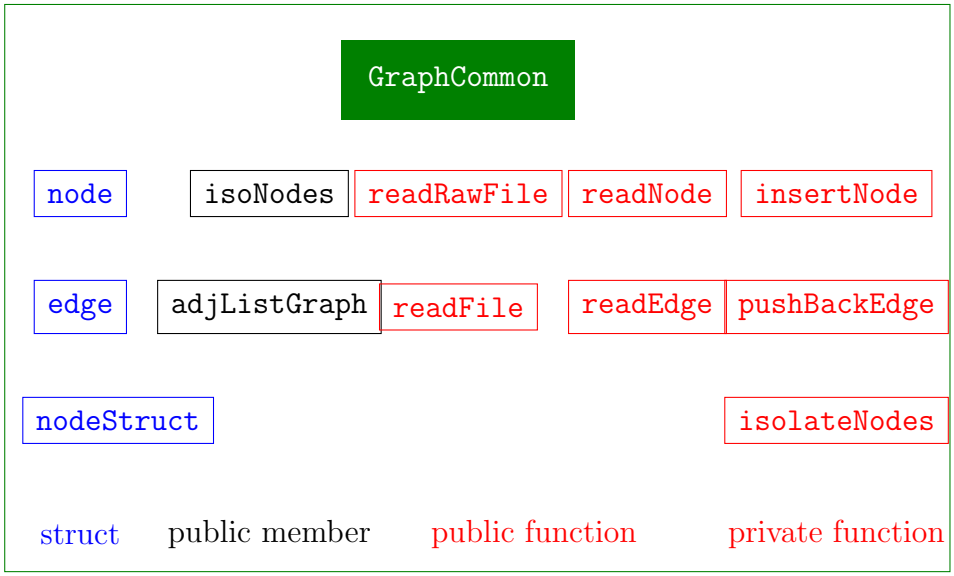

图 1: GraphCommon 类的成员

2.1 结构体

该类定义了三个小结构体：

- 节点类 node
- 有向边类 edge
- 节点结构 nodeStruct（用于计算邻接矩阵）

前两个都重载了输入输出运算符，以符合目标格式。

表 1: 格式定义

	输入	输出
node	⟨P1⟩	⟨1⟩
edge	⟨P1 P2 2.0⟩	⟨1,2,2.0⟩

前缀符号以及分割符可以在后面调整。值得一提的是，优化后子图的虚边定义为如下的格式：

```
<起始点,虚点符号,<文件名.终止节点,权重>>
<1,-1,<01.2,2.0>>
```

在本程序中，-1被定义虚节点符号。

节点结构中定义了三个成员：

- node Node; 邻接矩阵边存储
- map<int, double> adjMatCol; 邻接矩阵列数值
- double totalWeight = 0; 节点发出边总权重

并通过三个私有函数同步更新这些数值。

2.2 公共成员

该类定义了两个公共成员：

- set<int> isoNodes 包含了所有的孤立节点(isolated nodes)，也就是完全不连通的部分。
- map<int, vector<edge>> adjListGraph，邻接表图，仅包含连通部分节点，对于有些连通节点可能为发出空边的集合，即 adjListGraph[node] = vector<edge>();。

2.3 公共函数

该类定义了四个公共函数。

- readNode 通过输入文本流读取明确定义的节点信息
- readEdge 通过文本流读取有向边的信息。
- readFile 通过文本流读取文件信息。
- readRawFile 通过文本流读取特定格式的文件信息。

2.4 私有函数

该类定义了三个私有函数。

- insertNode 插入节点，包含了对虚节点的检查机制。
- pushBackEdge 向邻接表插入边，也包含了对节点是否为虚节点的转换检查机制。
- isolateNodes 将 readNode 后的集合去除根据 readEdge 所读取的连接边点，变为孤立节点的集合。

3 图生成器 GraphGen

图生成器的类 GraphGen 是 GraphCommonGen 的派生类。

本程序的图生成器有几个参数需要设置：

- **节点类型** nodeType：continuous连续编号的, discrete离散的。
- **边生成类型** edgeType：Tree树（不含环路）, Graph图（带有环路）。
- **连通图类型** isoType：Single 单个连通图, Multi 多个连通图。

- **节点编号增长量** MAX_INCREASEMENT：在离散编号模式下，每次生成一个节点都会增长一个数字，这个数字不会超过最大增长量 。
- **最大孩子数** MAX_CHILD：每个节点的发出有向边个数不会超过最大孩子数 。
- **最大连通子图数** MAX_ISOGRAPH：在多个连通图生成模式下，每个图的连通子图数不会超过最大连通子图数。
- **节点行数占比** Node / Lines：在新文件模式下，仍然会生成随机个数的节点数，但是只输出占比量的节点行数，其余为有向边的行。

该程序将会根据上述参数，递增而随机地生成节点编号。然后通过层序遍历生成各个边，如果没有环路的限制，则有可能随机到一个环路节点上去。

随机数采用下面的代码生成：

```
srand((unsigned)time(0)*(++gseed));
return 1.0 * rand() / RAND_MAX;
```

当然，这种方式依然不是特别特别随机，但已经足够。

4 图分解器 GraphDecomp

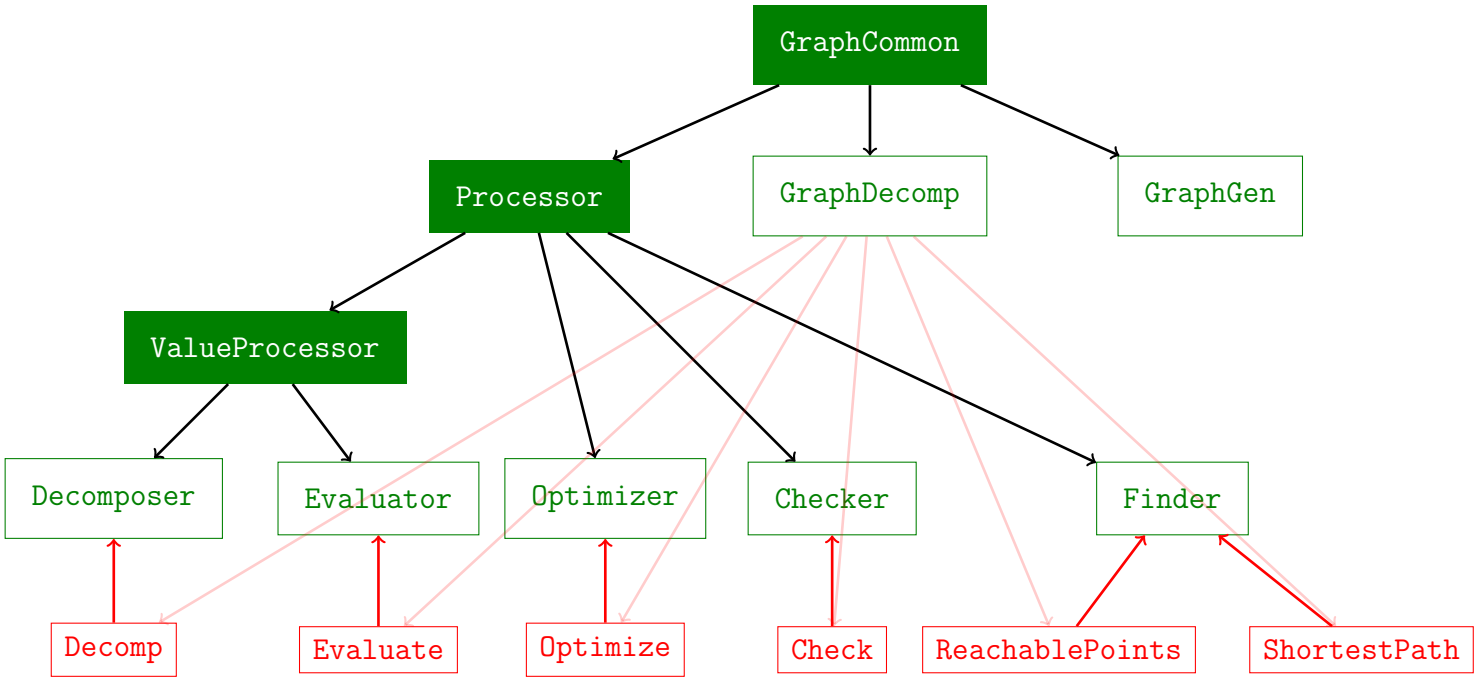

图 2: GraphDecomp 类的成员

图分解器类 GraphDecomp 是 GraphCommon 的派生类，为前端提供提供统一接口。

处理器类 Processor 同样是 GraphCommon 的派生类，提供文件处理相关的函数。

值处理器类 ValueProcessor 是 Processor 的派生类，提供邻接矩阵计算相关的函数。

4.1 分解器 Decomposer

分解器类 Decomposer 是 ValueProcessor 的派生类，采用 Kerninghan-Lin 算法[1]，将其改进为有

向边算法。

对于两节点有向边的损失，可以作为无向边处理，在本程序中定义损失为：

$$c_{ab} = c_{\overline{ab}} + c_{\overline{ba}} \tag{1}$$

对于两个集合A, B，各个集合中的对应边a, b，将**外损失**定义为：

$$E_a = \sum_{y \in B} c_{ay} \tag{2}$$

内损失定义为：

$$I_a = \sum_{x \in A} c_{ax} \tag{3}$$

内外差定义为：

$$D_z = E_z - I_z, \forall z \in S \tag{4}$$

损失缩减量定义为：

$$g_{ab} = D_a + D_b - 2c_{ab} \tag{5}$$

可以设置算法的猛烈程度 `DecompSol`，来影响分解时间和分解效果，对于大规模运算集会有显著的影响：

- `rough` 按照顺序分割，最快(`order`)。
- `bfs` 采用广度优先搜索算法，快一些(`bfs`)。
- `onepass` 每次只进行一次优化，居中(`medium`)。
- `ll` 取用局部最优（D 优先算法），慢一些(`harder`)。
- `kl` 标准算法，强力优化（g 优先算法），最慢(`hardest`)。

Kerninghan-Lin 算法的主要过程：对于两个分割集合A,B，循环以下过程：

Algorithm 1: 修改后的 Kerninghan-Lin 算法

Input: 需要被优化的分割集合 A, B
Output: 被优化后的分割集合 A, B

repeat
  对于分割集 A, B 中的每个元素计算 D;
  $p \leftarrow 1$;
  $A_p \leftarrow A, B_p \leftarrow B$;
  repeat
ll 选择 $a_i \in A_p, b_j \in B_p$ 使得 $g_p = D_{a_i} + D_{b_j} - 2c_{a_i b_j}$最大;
    $a'_p \leftarrow a_i$;
    $b'_p \leftarrow b_j$;
MOD $A_{p+1} \leftarrow A_p - \{a_i\} + \{b_j\}$;
MOD $B_{p+1} \leftarrow B_p + \{a_i\} - \{b_j\}$;
    $p \leftarrow p + 1$;
    对于分割集 A_p, B_p 中的每个元素计算 D;
  until $p = |A|$;
  选择k使得$G = \sum_{i=1}^{k} g_i$最大;
onepass **if** $G \leq 0$ **then**
    break;
  $A \leftarrow A - \{a'_1, a'_2, \cdots a'_k\} + \{b'_1, b'_2, \cdots, b'_k\}$;
  $B \leftarrow B + \{a'_1, a'_2, \cdots a'_k\} - \{b'_1, b'_2, \cdots, b'_k\}$;
until $G \leq 0$;

在 Kerninghan-Lin 的原算法中，是需要在 **MOD** 中直接去除两个元素，但是实验证明这么做的效果对有向边没有这种好，因为计算 G 的意义就是为了看交换到哪一步就是最好的优化，而在我们需要保持原集合大小不变的情况下，不应当在算法中间尝试减少集合的大小。

- `kl` 模式就是采用上述算法。
- `ll` 模式对步骤 **ll** 进行了改进，仅仅通过选择两个集合中最大的 D 对应的元素进行交换。
- `onepass` 对步骤 **onepass** 进行放宽，只进行一次优化，就退出。对于步骤5只重复原次数的四分之一。
- `bfs` 直接采用广度优先搜索，每次选择最大连接权重的节点遍历其子节点，直至每个集合达到集合个数上限。
- `rough` 模式就是对集合进行顺序二分，直至集合大小符合要求。

节点分配完毕后，将会将节点及其**所有**发出边存储在对应文件中。

4.1.1 评估器 `Evaluater`

对于外部分解文件的评估，该评估器定义节点的存储位置优先级如下：

1. 对于以节点形式存储的节点，存在对应的文件中
2. 对于第一次以有向边起点存储的节点，存在对应的文件中
3. 连通节点中的叶子节点将会存储在最后一次出现的文件中

独立的评估器类为 `Evaluater`，当然 `Decomposer` 内会在完成分割后进行评估。但主要的评估函数

Evaluate() 都来自于父类 ValueProcessor。

首先需要计算每一对元素的邻接权重矩阵，需要进行 n^2 次计算。

如果将 n 个元素几乎平均地分配到元素个数为$|P|$个集合中，共有 $\frac{n}{|P|}$ 个集合，则需要进行对每两个集合间的每两个元素计算，需要 $C^2_{\frac{n}{|P|}}|P|^2 = O(n^2)$ 次计算，算出损失值。

最后计算总权重，计算损失矩阵上对角线部分（包含对角线）的和。

该运算的成本较高，设置了 CALC afer DECOMP 选项以选择跳过计算权重部分。

4.2 优化器 Optimizer

优化器 Optimizer 是 Processor 的子类。

由本程序产生的分解子图，都会将节点存储位置以节点形式存储在对应文件中。优化器仅仅优化末尾节点，使其指向虚节点，并存储对应的文件位置。

分解器保证了所有的节点及其发出边都在同一个文件中，所以起始节点不会为虚节点。而如果起始节点为虚节点，将会对后面的寻找可达节点和最短路径造成极大的障碍，因为就难以预料起始节点在哪些文件中，最终很有可能沦为变相合并子图的过程，将会不符合题意。所以该程序限制只有终止节点为虚节点。

4.3 检查器 Checker

检查器 Checker 是 Processor 的子类，用以检查主图与分解图、分解图与优化图之间是否存储一致。重载了集合比较函数，定义了映射比较函数，用于比较两者的鼓励节点存储与邻接表图存储是否都是一致的。

4.4 访达器 Finder

访达器 Finder 是 Processor 的子类，用于寻找可达节点与最短路径。

4.4.1 可达节点 ReachablePoints

本程序所生成的优化后子图文件，节点存储位置将会存储所有发出边，虚节点只适用于有向边结束节点。因此，在这种限制规则下，可以将文件视作**广义节点**，当访问该点时出现虚边指出该文件，则输入文件访问队列，在该文件访问完毕后，访问队列中的下一个文件以及相应的节点。这样就可以在子图访问的前提下，获取所有的可达节点。

当然使用该算法会导致读取不同格式的子图文件做寻找算法时出现找不全的情形（比如起始节点为虚节点的情形）。

4.4.2 最短路径 ShortestPath

采用 SPFA(Shortest Path Faster Algorithm) 算法。仍然在只有终止节点为虚节点的限制下，仿照上述可达节点的做法依次访问从起始节点开始的所有可达节点，每访问一个节点，都会考虑从上一个节点通过该有向边能否比原来的权重更少，如果是，则更新该节点的权重值，并存储路径到 prev 中，并放入访问队列中。

SPFA 在形式上和 BFS 非常类似，不同的是 BFS 中一个点出了队列就不可能重新进入队列，但是 SPFA 中一个点可能在出队列之后再次被放入队列，也就是一个点改进过其它的点之后，过了一段时间

可能本身被改进，于是再次用来改进其它的点，这样反复迭代下去。这种算法相比于 dijkstra 更适合于多子图上最短路径的寻找，没有使用排除访问的方法。

5 性能评估

通过图生成器生成了 9 个图文件，之后通过图分解器分解为原图规模的 5 %，计时并查看结果。

表 2: 分解优化检查性能结果

	分解优化检查时间(s)					分解优化检查割边比例				
规模	`hardest`	`harder`	`medium`	`bfs`	`order`	`hardest`	`harder`	`medium`	`bfs`	`order`
100	0.5	0.5	0.5	0.5	0.5	16.7%	41.7%	41.6%	50.0%	75.0%
200	1	1	1	1	1	39.2%	38.4%	46.6%	67.7%	71.9%
300	1	1	1	1	1	44.9%	47.0%	52.6%	65.0%	77.4%
500	1	1	1	1	1	55.1%	56.9%	65.2%	66.6%	85.4%
1000	3	3	2	1	1	51.5%	51.7%	62.7%	65.5%	90.3%
2000	12	9	3	1	1	35.2%	36.7%	44.6%	60.4%	73.7%
3000	33	24	6.5	1	1	33.9%	32.9%	43.3%	59.8%	75.5%
5000	133	90	21	1.5	1	39.5%	38.8%	46.7%	54.2%	75.4%
10000	1020	736	180	2.4	2	9.2%	8.7%	27.0%	49.2%	74.5%

以 `order` 模式为基准，考察其余模式下的时间与效果优化程度。定义性价比为：

$$\text{Value} = \frac{\text{Effect}}{\text{Time}} = \frac{\frac{1}{\text{Cut Edge Proportion}}}{\text{Time Magnification}} \tag{6}$$

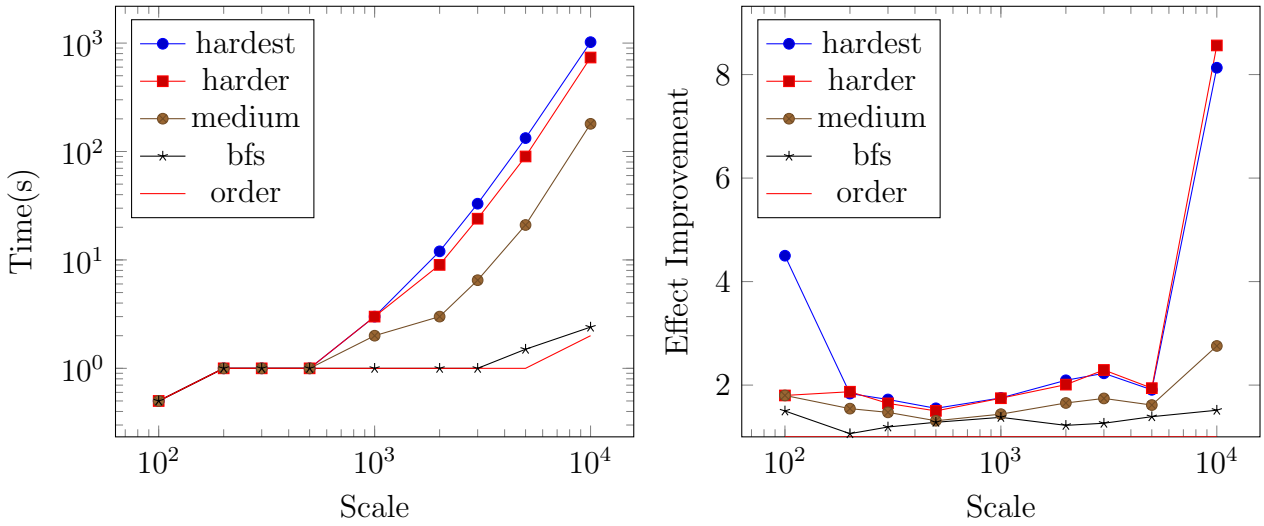

图 3: 不同规模下的时间和效果提升图

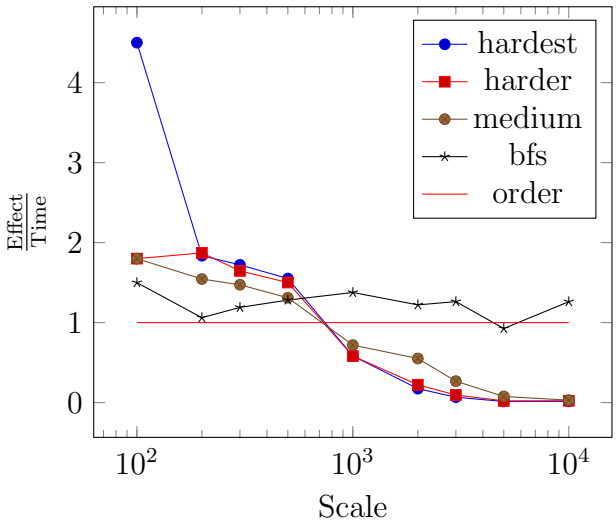

图 4: 不同规模下的性价比图

在评测时，20s 左右都是可以接受的，所以：

- 对于 $0 \leq n \leq 1000$ 的图，都可以使用 `hardest` 模式，以期获得更好的分解效果。
- 对于 $1000 < n \leq 3000$ 的图，都可以使用 `harder` 模式，时间上不仅可以缩减为 `hardest` 的 $\frac{2}{3}$，效果上甚至可能会比 `hardest` 的要略好，因为都是局部最优算法，并不一定是全局最优。
- 对于 $3000 < n \leq 7000$ 的图，都可以使用 `medium` 模式，进一步缩减时间，并保证比 `harder` 更高的性价比。
- 对于 $n > 7000$ 的图，使用 `bfs` 模式节约时间，Kerninghan-Lin 算法所需要的时间过长，尽管效果提升可能比较明显。但是 `bfs` 模式 可以保证平稳的 50 % 割边比例，也即保证基线的 $\frac{1}{3}$ 优化率。
- 如果分割限制比例向下调整，则算法就需要降档，以满足时间限制。当然，从时间复杂度的分析来看，分割成过细的子图会导致评估割边权重部分耗费过多的时间，可以考虑关闭 `CLAC after DECOMP` 选项以期大幅减少时间耗费量与内存占用率。

References

[1] KERNINGHAN B W, LIN S. An efficient heuristic procedure for partitioning graphs[J/OL]. The Bell System Technical Journal, 1970: 291-307. https://ieeexplore.ieee.org/document/6771089/.