arXiv:2603.26538v1 [cs.DM] 27 Mar 2026

在 st-DAG 中寻找极小簇

Ulrich Vogl，Markus Siegle

慕尼黑联邦国防军大学，计算机科学系，*85577* 纽比贝格，德国

摘要

具有单一入口顶点和单一出口顶点的有向无环图（st-DAG）有许多应用。例如，它们常用于建模流问题，或表示任务、工作包等之间的优先条件。本文提出了一种算法，用于在此类 st-DAG 中查找称为“簇”的特殊类型子图。在 DAG 分析中，了解给定 st-DAG 中的簇非常有用。簇的特征在于其入口边界和出口边界处具有某种同步行为。在此背景下，我们引入“同步点”的概念，即 DAG 中的一种同步点；并针对给定的 st-DAG 构造第二个 DAG，称为 MSP-DAG，其边由最大尺寸同步点（MSP）之间的优先关系确定。我们提出的新簇查找算法会在潜在的包围 MSP 对之间搜索簇。该算法之所以高效，是因为它在 MSP-DAG 上运行，而 MSP-DAG 通常远小于原始 st-DAG。本文还对算法的运行时间进行了全面的复杂度分析，结果表明，其复杂度关于 DAG 顶点数为二次，关于 MSP-DAG 顶点数为指数级。此外，本文还通过一个章节报告了在随机生成的 DAG 上进行的实验，展示了该算法的实际适用性，并证实了我们的理论发现。

关键词：有向无环图，st-DAG，子图，簇，同步点

1. 引言

有向无环图（DAG）在概念上虽然简单，却是一种功能强大、应用广泛的建模形式，适用于各种领域。它们常用于项目规划、工作流分析、并发任务执行建模、调度中的依赖图和语法程序分析，也用于研究因果推断等诸多领域。

电子邮件地址：ulrich.vogl@unibw.de, markus.siegle@unibw.de（Ulrich Vogl、Markus Siegle）

作者在项目规划中使用有向无环图，其中图的顶点表示工作包，边表示它们之间的因果依赖。从顶点 u 到顶点 v 的边 $(u,\ v)$ 意味着，只有在 u 完全完成后，v 才能开始。因此，只有当前驱全部 完成后，一个顶点才能变为活动状态。显然，在此情境下，冗余边——也就是存在从 u 到 v 的替代路径的边 $(u,\ v)$——没有意义，可以忽略。项目规划的一项重要目标，是确定由有向无环图表示的项目可能需要的总执行时间。由于工作包的执行时间可能因各种因素而有很大差异，因此使用概率分布来表示各个工作包（即有向无环图的顶点）的随机执行时间。因此，总执行时间也是一个随机变量，需要计算其分布。

如果给定的有向无环图是串并联图（该性质的定义见下文第2节中的定义4），那么可以通过简单的逐步约简方案，根据各个顶点的分布计算整体执行时间分布：迭代地对成对（或成组）的中间子图分布应用卷积或最大值运算，从而有效地将原始有向无环图约简为单个顶点，最终即可得到整体分布。然而，对于非串并联的有向无环图，或包含非串并联子图的有向无环图，这种约简方案并不适用，因为无法处理图中的非串并联部分。因此，在先前的研究中，[1] 和 [2], 提出了这样一种策略：在给定的有向无环图中隔离出非串并联部分，即所谓的最小复杂簇，并借助 PRISM [3] 等概率模型检测工具分析其执行时间分布。随后，将该簇替换为一个具有相同运行时间分布的单个顶点。然而，这类簇分析基于状态空间，因此在运行时间和内存方面开销较大。因此，模型检验所分析的簇越小越好，这促使我们寻找大小最小的簇。无需进一步详述上述过程，也足以看出，精确定义串并联约简不可约的子图（即最小复杂簇），以及如何在可能规模庞大的有向无环图中高效识别这些子图，都是重要的研究问题。本文正是围绕这一主题展开。

本文其余部分基于博士论文 [4] ，但更严格地聚焦于图论问题，结构如下：在第1.1节讨论一些相关工作后，我们在第2节介绍必要的基础知识，其中包括簇和复杂簇的定义。第3节介绍同步点这一重要概念；顾名思义，同步点是有向无环图中的一种同步点，可能标记簇的入口点或出口点。在给定的有向无环图中找出所有最大规模同步点（即所有 MSP）后，便可构造出第二个通常小得多的有向无环图，即所谓的最大同步点有向无环图（MSP-DAG），它在我们的方法中发挥着关键作用。第4节将介绍我们用于

寻找极小簇的算法，并以伪代码形式呈现，辅以详细说明。本节还对该算法进行了复杂度分析，表明其总体运行时间关于顶点数呈二次关系，关于最大同步点数呈指数关系。第5节介绍一系列实验，表明新算法适用于分析规模相当大的 st-DAG。最后，第6节总结主要结果，并对未来可能的研究方向作出展望。

1.1. 相关工作

已有大量出版物研究顶点配备随机执行时间分布的有向无环图的总时间跨度分析。例如，这类技术被用于分析运行于多处理器系统上的并行计算机程序，其中有向无环图的顶点表示待执行的任务（过程、子程序）。在此背景下，早期工具 PEPP [5, 6] 提供了用于模型规范的图形用户界面以及一系列分析技术。对于串并行有向无环图，PEPP 能够通过迭代合并串联（或并联）连接的顶点对（或顶点集），将其约化为一个具有相同执行时间分布的单个顶点（关于“串并联”的定义，见下文定义4）。然而，对于并非完全串并行的有向无环图，只有在所有顶点的执行时间均服从指数分布时，才能对其进行精确处理，而且只能计算预期总时间，而不能计算分布。对于一般的非串并联情形，PEPP 只能计算总时间分布的上界和下界。除了识别串联或并联连接的顶点对（或顶点集）之外，PEPP 无法利用给定有向无环图的任何特殊结构属性。相比之下，我们通过寻找称为簇的特殊结构，并以单个顶点替代其执行时间分布的方法始终适用、效率高，并且能够得到精确的总时间跨度分布。

最近，有研究提出了确定复杂工作流响应时间分布的技术，以及相关工具 EULERO [7, 8]。作者将精确分析与随机界定技术相结合；对于违反“良构嵌套”属性的工作流，后者是必需的，而违反该属性本质上意味着工作流不是串并联结构。不过，工作流模型的结构并非通过分析确定，而是必须由用户作为模型规范的一部分提供，其形式为所谓的层次结构树。这样，EULERO 就避免了对扁平图模型进行结构分析的问题。

从图论视角来看，对图的特殊结构特征进行刻画，或分离出具有特殊性质的子结构，在文献中受到的关注出人意料地少。当然，连通性问题（例如寻找无向图的连通分量，或有向无环图的强连通分量）是众所周知的问题，并且已有高效算法可以解决 [9, 10]。与连通性问题相关的是，在一个无向图中，有团这一概念（ r团是一个包含 r 个顶点的完全子图），以及团判定问题（判断一个

图是否具有某个特定大小的团），该问题因属于NP困难问题而闻名 [11, 12]。此外，人们还深入研究了许多不同类型的聚类问题，其中，簇是一个内部连通性高而外部连通性低的顶点子集（请注意，本文定义的“簇”概念有所不同）。聚类在数据分析中有诸多应用，例如在大型网络中检测模式、界定社群等 [13]。

与上述研究不同，本文关注的是如何在有向无环图中寻找特殊子图。这些子图由其入口点或出口点所满足的形式化条件来界定。我们将这类子图称为最小复杂簇（不要与上一段提到的更一般的聚类概念混淆）。正如前文所指出，这种新型簇在我们分析有向无环图的整体执行时间分布时发挥着关键作用 [1, 2]。

2. st-DAG 的定义与性质

定义 1. 一个有向无环图，若其具有一个源顶点和一个目标顶点（st-DAG），则是一个元组 $G = (V, E)$，其中 V 是一个有限非空顶点集，$E \subseteq V \times V$ 是一个有向边集。图中不存在环，即不存在由非空有向边序列构成、并回到同一顶点的路径。图 G 有一个唯一的源顶点 s （没有任何入边）和一个唯一的目标顶点 t（没有任何出边）。

根据定义 1，最小的 st-DAG 由单个顶点构成，且不含边；此时 $s = t$ 和 $E = \emptyset$。

定义 *2.* 在 *st-DAG* 中，长度为$l \geq 1$ 的边路径（简称路径） p 定义为由非空边序列 $(e_1, \ldots, e_l)$ 构成，其中 $e_i = (u_i, v_i) \in E (i = 1, \ldots, l)$，且满足 $u_{i+1} = v_i (i = 1, \ldots, l-1)$。长度为 l 的源汇路径是满足 $u_1 = s$ 和 $v_l = t$的路径。对于这样的边路径 p，相应的顶点路径是顶点序列 $u_1, u_2, \ldots, u_l, v_l$。在 *st-DAG* $G = (V, E)$ 中，若存在一条从u 到 v 且不包含 e的替代路径，则称边 $e = (u, v) \in E$ 为冗余边。

下文中，若写作有向无环图，均指 st-DAG，并且总是假设其中不含任何冗余边。若给定的有向无环图含有冗余边，我们可以放心地将其删除，即可以处理其（唯一的）传递约简 [14]。

定义3。 给定一个有向无环图 $G = (V, E)$，若两个或更多顶点 $v_1, v_2, \cdots \in V$ 具有完全相同的前驱（后继）顶点集合，则称它们为入孪生顶点（出孪生顶点）。

定义4。 若一个有向无环图 $G = (V, E)$ 可以通过以下两种约化步骤序列约化为一个单个顶点，则称其为串并联（或完全串并联可约）：

- 两个顶点 u和 v，其中 v 是 u唯一的后继，而 u 是 v唯一的前驱，可以串行地化简为一个单个顶点。

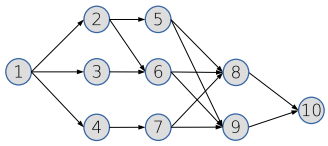

图 1：一个包含 $s = 1$ 和 $t = 10$ 的 st-DAG。顶点 4 和 7 可以串行约化。顶点 8 和 9 可以并行约化。由顶点集 $\{2, 3, 5, 6\}$ 导出的子图是一个最小复杂簇。由顶点集 $\{2, 3, 4, 5, 6, 7\}$ 和 $\{2, 3, 4, 5, 6, 7, 8, 9\}$ 导出的两个子图也都是簇，但它们既不复杂（因为可以在这些子图内进行串联/并联约化），也不是最小的。

- 一组顶点 k、 v_1、...、v_k（对应 $k \geq 2$）既是入孪生顶点，也是出孪生顶点，可以并联化简为一个单个顶点。

定义 4 中的串并联图概念与 [15] 和 [6]，中的概念等价，但 [15] 使用的是构造性定义，而非我们基于约化步骤给出的定义。

定义 5。给定一个 *DAG* $G = (V, E)$，若元组 $G' = (V', E')$ 满足 $V' \subseteq V$ 和 $E' \subseteq E$，则称其为 G 的子图，当且仅当 $\forall e = (u, v) \in E'$：$u \in V' \wedge v \in V'$（没有悬空边）。若顶点 $v \in V'$ 在 $V \setminus V'$ 中有前驱（后继），或者它是源节点 s（目标节点 t），则称其为 G' 的入口（出口）顶点。给定一个顶点集 $V' \subseteq V$，V' 的导出子图是 G 中顶点集为 V' 的极大子图。

定义 6. 给定一个有向无环图 $G = (V, E)$ 和一个顶点集 $V' \subseteq V$，令 $G' = (V', E')$ 为 V' 的导出子图。令 $A \subseteq V'$ 和 $B \subseteq V'$（其中 $|A| \geq 2$ 和 $|B| \geq 2$ 且 $A \cap B = \emptyset$）分别为 G' 的入口顶点集和出口顶点集。若 A 是入孪生顶点集，且 B 是出孪生顶点集，则称 G' 为簇。若一个簇 G' 无法通过任何串联或并联约简步骤进行约简，则称其为复杂簇。若一个簇不包含任何同样为簇的真子图，则称其为极小簇。

根据定义 6，簇至少包含四个顶点。图 1 展示了一个 st-DAG，用以说明串联/并联可约性的定义以及簇的概念。正如我们在引言中所述，极小复杂簇在有向无环图的分析过程中发挥着重要作用，因此我们需要高效的技术来查找它们。下一节将介绍同步点，它们将有助于识别簇。

3. 同步点及相关概念

定义 7。对于有向无环图 $G = (V, E)$，考虑一个非空边集 $\mathcal{X} = \{e_1, \ldots, e_k\} \subseteq E$，其中，$S = \{v \in V \mid v$ 是某个 $e \in \mathcal{X}\}$ 的终止顶点，而 $P = \{v \in V \mid v$ 是某个 $e \in \mathcal{X}\}$ 的起始顶点。若 $\mathcal{X}$ 包含从 P 通向 S 的所有边，则称 $\mathcal{X}$ 为全同步点（FSP），当且仅当

a) ($|S| = 1$ 或 S 是入孪生顶点的集合)，且 P 是 S中顶点的共同前驱集合；以及 *b)* ($|P| = 1$ 或 P 是出孪生顶点的集合)，且 S 是 P中顶点的共同后继集合。

仅当 *a)* 成立且 $|S| \geq 2$ 时，$\mathcal{X}$ 才称为前向半同步点（FHSP）。仅当 *b)* 成立且 $|P| \geq 2$ 时，$\mathcal{X}$ 才称为后向半同步点（BHSP）。

对于同步点 $\mathcal{X}$ （它是一个边集），我们用 $P_{\mathcal{X}}$ 表示其前驱/起始顶点集，用 $S_{\mathcal{X}}$ 表示其后继/终止顶点集。

下面的引理阐明了定义 6 中定义的簇与定义 7 中的同步点之间的重要联系。

引理 1。在有向无环图 $G = (V, E)$ 中，设子图 $G' = (V', E')$ 是一个簇。设 $A \subseteq V'$ 是 G' 的入口顶点，$B \subseteq V'$ 是G'的出口顶点。那么，A 的入边集是 *FSP* 或 *FHSP*，而 B 的出边集是 *FSP* 或 *BHSP*。

证明。根据定义 6，A 是一组入孪生顶点，因此，对于其入边集合，同步点定义（定义 7）的条件 a)（其中 $S = A$ 和某个顶点集 $P \subseteq V \setminus V'$）得以满足。同理，$B$ 是一组出孪生顶点，因此，对于其出边集合，同步点定义的条件 b)（其中 $P = B$ 和某个顶点集 $S \subseteq V \setminus V'$）得以满足。 □

定义 8。当且仅当定义 7 中的 $|S| = 1$ 和 $|P| = 1$ 时，同步点称为一对一同步点 (11SP)。如果一个同步点不是另一个同步点的真子集，则称其为最大同步点 (MSP)。否则称其为子同步点 (SSP)。

引理 2。由边集 $\mathcal{X}$ 表征的全同步点始终是极大同步点。

证明。使用定义 7 中的记号，$\mathcal{X}$ 包含从 P 到 S的所有边。$\hat{}\,\hat{}\,\hat{}$ 假设存在一个同步点 $\mathcal{X}$，满足 $\mathcal{X} \subsetneq \mathcal{X}$。令 $\hat{e} = (\hat{u}, \hat{v}) \in \mathcal{X} \setminus \mathcal{X}$。那么 $\hat{u} \notin P$ 或 $\hat{v} \notin S$（或两者）成立。不失一般性，假设 $\hat{u} \notin P$ 且 $\hat{v} \in S$。这与定义 7 a) 相矛盾；定义 7 a) 指出，P 是 S的前驱集合。对偶情形 $\hat{u} \in P$ 和 $\hat{v} \notin S$ 会导致类似的矛盾。情形 $\hat{u} \notin P$ 和 $\hat{v} \notin S$ 与定义 7 a) 和 b) 中的孪生条件相矛盾。 □

引理 3。给定一个 *st-DAG* $G = (V, E)$，离开 $s \in V$ （唯一源顶点）的边集是一个 *FSP*。同样，进入 $t \in V$ （唯一目标顶点）的边集也是一个 *FSP*。

证明。这可直接由 FSP 的定义（定义 7）得出。 □

图 2 展示了不同类别同步点之间的关系。图 3 展示了一个有向无环图，其中包含不同类型同步点的一些示例。横跨边集的彩色竖条用于表示同步点。请注意，在这个有向无环图中，有两种可能的并行

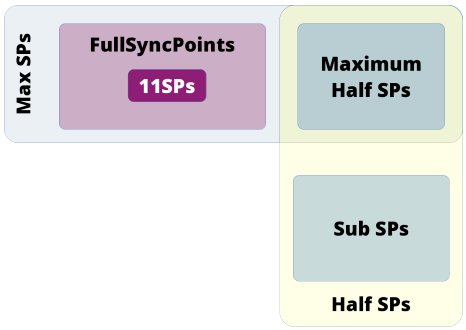

图 2：不同类别同步点之间的关系

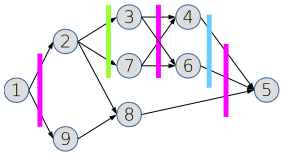

图 3：展示不同类型同步点的有向无环图。全同步点（FSP）以洋红色表示，前向半同步点（FHSP）以绿色表示，后向半同步点（BHSP）以蓝色表示。请注意，BHSP 是最右侧 FSP 的子同步点（SSP）。除 BHSP 外，所有同步点（SP）都是最大同步点（MSP）。

约简（顶点 3 和 7，以及顶点 4 和 6）。执行这些并行归约后，得到的两个新顶点可以串行地进行约简，从而将由顶点 $\{3, 4, 6, 7\}$ 诱导的子图折叠为单个顶点。然而，此时无法再进行任何串行或并行约简。

算法1展示了一种用于查找给定 st-DAG $G = (V, E)$ 中所有 MSP 的算法。它基本上会确定 V 的所有出孪生类和入孪生类，并从每个类分别沿前向和后向检查是否存在可能的同步点。其运行时间复杂度可按如下方式评估：第 2 行和第 3 行可通过对 G 进行一次遍历完成。如果 $|V| = n$ 且 $|E| = m < n^2$，则这需要 $O(m)$ 时间。使用合适的数据结构，第 4 行和第 10 行的 for 循环可在 $O(t)$ 时间内完成，其中 $t < n$ 是（出或入）孪生类的数量。为说明这一点，请注意，这两个 for 循环中所需的所有信息（例如第 5 行和第 11 行中的集合 S 和 P，以及所有入孪生类和出孪生类的大小）已在第 2 行和第 3 行中预先计算；而第 6 行和第 12 行条件的评估，则可通过比较预先计算的集合（常量大小）哈希值来实现。第 18 行的 for 循环也可在 $O(n)$ 时间内完成。因此，算法1的总体时间复杂度为 $O(m)$。

3.1. 同步点的优先关系

对于给定的 st-DAG G，我们构造另一个有向无环图，其节点是 G 的最大同步点，其边由自然的先后关系给出

算法1 用于查找 st-DAG 中所有 MSP 的算法 $FindAllMSPs$
$G = (V,\ E)$

```
1: 函数 FindAllMSPs(G: st-DAG) 返回 无
2: 在对 G 的单次遍历中，确定所有出孪生类和所有入孪生类
   of V
3: 在对 G 的同一次遍历中，确定所有出度为一的顶点
   或入度为一的顶点
4:  对于找到的每个出孪生类 P 执行
5:     S： P的后继顶点集 =
6:     如果 (S 是一个入孪生类，或者 |S| == 1) ∧ （ v ∈ S 的入度为 |P|）
       then
7:        将离开 P 的边集作为一个 FSP 输出
8:     else
9:        将离开 P 的边集作为一个最大 BHSP 输出
10:  对于找到的每个入孪生类 S 执行
11:    P := S 的前驱顶点集
12:    若 (P 是一个出孪生类) ∧ （ v ∈ P 的出度为 |S|） 则
13:       不执行任何操作（此 FSP 已在第 7 行找到）
14:    否则若 |P| == 1 ∧ （ v ∈ P 的出度为 |S|） 则
15:       输出通向 S 的边集，作为一个 FSP
16:    else
17:       输出通向 S 的边集，作为一个最大 FHSP
18: 对所有出度为 1 的顶点 u 执行
19:    如果 v:= （ u 的后继顶点）的入度为 1 则
20:       将边 (u, v) 作为 11SP 输出
```

该先后关系是在定义9中定义的同步点之间的关系。

定义9。 设 $\mathcal{X} \neq \mathcal{Y}$ 是两个 *MSP*。我们称 $\mathcal{X}$ 先于 $\mathcal{Y}$ （记作 $\mathcal{X} \rightsquigarrow \mathcal{Y}$）当且仅当以下任一条件成立：

- 存在一条源汇路径 $(e_1, \ldots, e_l)$ 以及指标 $1 \leq i < j \leq l$，使得 $e_i \in \mathcal{X}$ 和 $e_j \in \mathcal{Y}$，或者
- $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$ 且 $\mathcal{X}$ 是 *BHSP*，$\mathcal{Y}$ 是 *FHSP*。

更具体地说，当且仅当$\mathcal{X} \rightsquigarrow \mathcal{Y}$ 且满足以下任一条件时，我们称 $\mathcal{X}$ 紧接在前 $\mathcal{Y}$ （记作 $\mathcal{X} \overset{i}{\rightsquigarrow} \mathcal{Y}$）

- 存在一条源汇路径 $(e_1, \ldots, e_l)$ 以及指标 $1 \leq i < j \leq l$，使得 $e_i \in \mathcal{X}$且 $e_j \in \mathcal{Y}$，并且 $\forall i < k < j$ 不存在任何第三个 *MSP* $\mathcal{Z}$ 使得 $e_k \in \mathcal{Z}$；或者
- $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$ 且 $\mathcal{X}$ 是一个 *BHSP*，$\mathcal{Y}$ 是一个 *FHSP*（见上面的第二个项目）。

图4展示了定义9的第二个要点，即两个非互斥 MSP $\mathcal{X}$ 和 $\mathcal{Y}$ 在 $\mathcal{X} \overset{i}{\rightsquigarrow} \mathcal{Y}$时优先的特殊情况。

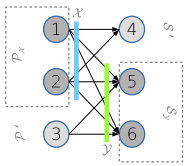

图4：两个不相交的 MSP 的示例，其中 $\mathcal{X} \overset{i}{\rightsquigarrow} \mathcal{Y}$

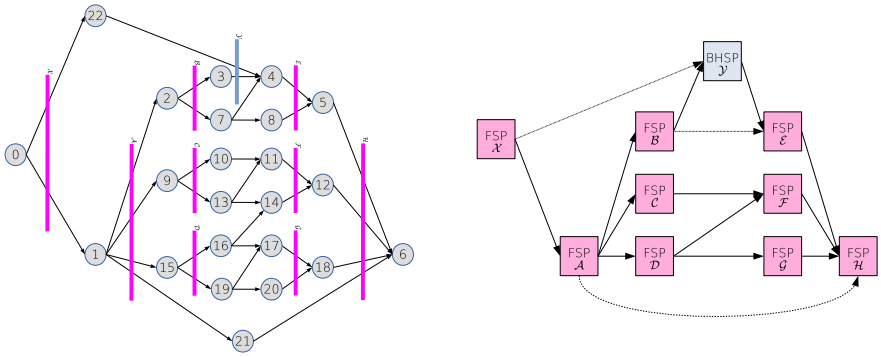

图5：一个有向无环图（左）及其对应的 MSP-DAG（右）

定义10。 对于给定的 *st-DAG* $G = (V, E)$，最大同步点有向无环图（MSP-DAG）G^M 定义为 $G^M = (V^M, E^M)$ 其中， V^M 是 G 的 *MSP* 集合，而 E^M 由定义*9*中的直接优先关系 $\overset{i}{\rightsquigarrow}$ 给出。

与原始 st-DAG G 不同，根据第2节，原始 st-DAG 不应包含冗余边；最大同步点有向无环图 G^M 则可能包含冗余边。这些边不得删除，因为它们对于第4节中的簇查找算法至关重要。

引理 4. 定义 *10* 中定义的 *MSP-DAG* 有且仅有一个源 *MSP* 和一个目标 *MSP*（即它也是一个 *st-DAG*）。二者都是 *FSP*。

证明。 这可由引理 3 得出。给定一个 st-DAG $G = (V, E)$，离开源顶点 s 的边集 $\mathcal{X}$ 是一个 FSP，也就是该 MSP-DAG 的源 MSP。对于通向目标顶点 t 的 G 中的边集，也可以作对称论证。 □

图 5 展示了一个有向无环图及其对应的 MSP-DAG。后者包含三条冗余边（即 $\mathcal{X} \overset{i}{\rightsquigarrow} \mathcal{Y}$、 $\mathcal{B} \overset{i}{\rightsquigarrow} \mathcal{E}$ 和 $\mathcal{A} \overset{i}{\rightsquigarrow} \mathcal{H}$），这些边对于我们的簇查找算法至关重要。具体而言，从 FSP $\mathcal{A}$ 到 FSP $\mathcal{H}$ 的（冗余）边将使算法能够找到由顶点 $\{9, \ldots, 20\}$ 诱导的最小复杂簇。

图 6 展示了另一个有向无环图及其对应的 MSP-DAG。同样，MSP-DAG 包含冗余边（即 $\mathcal{A} \overset{i}{\rightsquigarrow} \mathcal{E}$、 $\mathcal{C} \overset{i}{\rightsquigarrow} \mathcal{E}$ 和

$\mathcal{D} \overset{i}{\rightsquigarrow} \mathcal{F}$）。不过，在这种情况下，这些冗余边并未连接任何会产生极小簇的 MSP 对（注意，由顶点集 $\{2,3,4,5,8,9,10,11\}$ 诱导、由 MSP $\mathcal{A}$ 和 $\mathcal{E}$ 的 SSP 围成的子图是一个簇，但不是极小簇）。事实上，该有向无环图中唯一的极小簇由顶点集 $\{4,5,8,9\}$ 诱导，其边界为 MSP $\mathcal{B}$ 和 MSP $\mathcal{E}$ 的一个 SSP。

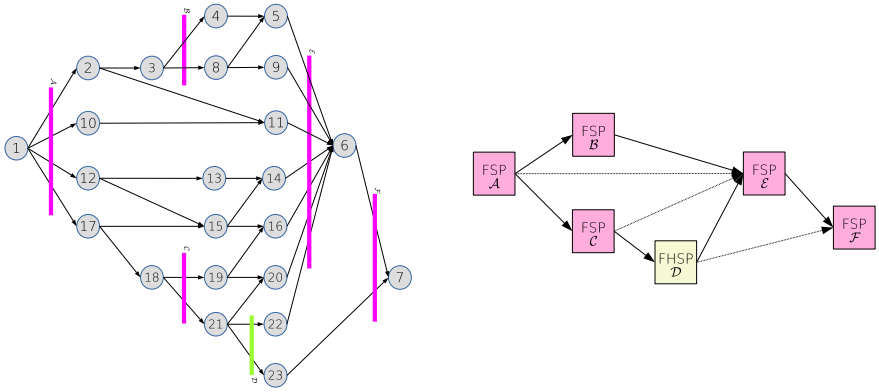

图 6：另一个有向无环图（左）及其对应的 MSP-DAG（右）

4. 极小簇查找算法

4.1. 算法的关键概念与概述

一个簇总是由两个同步点界定（见引理 1）。更确切地说，它由一个开放 FSP/FHSP $\mathcal{X}$ 和一个闭合FSP/BHSP $\mathcal{Y}$界定（其中 $\mathcal{X}$ 和 $\mathcal{Y}$ 不一定是最大同步点，也可以是子同步点）。因此，我们的算法 $FindAllMinClusters$ 会系统地检查可能包围簇的 MSP 对候选项（$\mathcal{X}$，$\mathcal{Y}$）。这些候选括号对通过检查 MSP-DAG 来确定。由于我们只关注极小 簇，算法会按 $\mathcal{X}$ 与 $\mathcal{Y}$之间距离递增的顺序处理候选对。为了找到所有相关的 MSP 对，算法会在预处理步骤中，从某个 FSP/FHSP $\mathcal{X}$ 到一个 FSP/BHSP $\mathcal{Y}$ （在 MSP-DAG 中）构造所有可能的相关 MSP 路径。这些 MSP 路径存储在列表 $PathsTo[\mathcal{Y}]$ 中（每个 MSP $\mathcal{Y}$ 对应一个列表），每个列表再按路径长度递增分段排序。有关如何构造所有这些相关 MSP 路径的详细信息，请参见附录 A。

算法 2 展示了用于查找 st-DAG 中所有极小簇的顶层算法。它有两个参数：st-DAG G及其关联的 MSP- DAG G^M。它调用算法 3 中所示的函数 $ClusterCheck$；该函数的参数除 G 外，还包括一对 MSP $\mathcal{X}$ 和 $\mathcal{Y}$，用于检查其中间簇。反过来， $ClusterCheck$会调用算法 4 中所示的函数 $ClosedCheck$，该函数的用途更具体：对于两个候选顶点集

$Xset$ 和 $Yset$ （其中 $Xset$ 由 $\mathcal{X}$导出，而 $Yset$ 由 $\mathcal{Y}$导出），算法会检查它们是否确实分别是某个簇的入口顶点集和出口顶点集。

算法 2 $FindAllMinClusters$找出 st-DAG 中的所有极小簇 $G=(V,\ E)$，其 MSP-DAG 由 $G^M=(V^M,\ E^M)$给出

```
1: 函数 FindAllMinClusters（G: st-DAG, G^M: MSP-DAG）返回 void
2: 对于 所有 FSP 和 BHSP Y ∈ V^M 执行
3:     构造列表 PathsTo[Y]，其中包含 G^M 中任意长度的所有 MSP 路径
       从任意 FSP 或 FHSP X ∈ V^M 开始，并在 Y结束
4: /// 列表 PathsTo[Y] 包含所有需要检查是否形成簇的相关路径
5: /// 根据构造方式，每个列表 PathsTo[Y] 都已按节排序
6: /// 有关如何高效构造这些列表的详细信息，请参见附录 A
7: 对于任意列表 PathsTo[Y] 中的所有 MSP 路径 p，按长度递增顺序执行
8:     将 p标记为已检查
9:     /// 每条 MSP 路径 p都只应检查一次
10:    X := StartMSP(p)
11:    Y := EndMSP(p)
12:    如果 |S_X| ≥ 2 ∧ |P_Y| ≥ 2 ∧ (S_X ∩ P_Y = ∅) 则
13:        /// 确保 MSP X 和 Y 足够大
14:        found := ClusterCheck(G, X, Y)
15:    如果 found，则
16:        删除 p 及其所有超路径
17:        /// 可在同一列表 PathsTo[Y] 中找到这些超路径，并且
18:        /// 在所有其他列表 PathsTo[Z] 中，只要能从 Y到达 MSP Z
19:        /// 使用所述的指针/分隔符结构
```

4.2. 算法说明

算法 $FindAllMinClusters$首先执行前面提到的预处理，即构造所有列表 $PathsTo[.]$。然后，在第 7 行的 for 循环中，它按路径长度递增的某种顺序遍历所有相关的 MSP 路径候选项；这些候选项的起始和结束 MSP（在算法中分别记为 $\mathcal{X}$ 和 $\mathcal{Y}$）可能包含一个或多个簇。已检查过的 MSP 路径会标记为 $checked$；如果在检查路径p时发现了簇，则会删除该 p 及其所有超路径（因为我们只关注极小簇，因此无需检查成功路径的超路径）。

通过函数 $ClusterCheck$ （见算法 3）检查特定 MSP 对（$\mathcal{X}$，$\mathcal{Y}$）之间是否存在中间簇。请注意， $\mathcal{X}$ 和 $\mathcal{Y}$之间可能包含不止一个簇。相应地，我们必须意识到，通常 $S_{\mathcal{X}}$ 中的顶点并非都必须属于这些簇（回顾定义 7：对于 MSP $\mathcal{X}$，我们用 $P_{\mathcal{X}}$ 表示其起始/前驱顶点集，用 $S_{\mathcal{X}}$ 表示其结束/后继顶点集）。$S_{\mathcal{X}}$ 中只有一个子集可能属于某个簇（作为其入口节点）；对于闭合 MSP $\mathcal{Y}$的 $P_{\mathcal{Y}}$ 集合，也可以作出对称的陈述。因此，

函数 $ClusterCheck$ 将 $S_{\mathcal{X}}$ 的子集 $Xset$ 与 $P_{\mathcal{Y}}$的子集 $Yset$ 进行匹配，这些子集在正向/反向方向上彼此可达。 $Xset$ 与 $Yset$ 之间的子图是一个候选簇，但还必须由函数 $ClosedCheck$ 进一步检查（见算法 4）。函数 $ClosedCheck$ 求 $Xset$ 的正向可达集与 $Yset$的反向可达集的交集。随后检查由该交集导出的子图是否满足闭合性，即除 $Xset$之外不得有其他入口顶点，且除 $Yset$之外不得有其他出口顶点。如果满足这些条件，那么 $Xset$与 $Yset$ 之间的子图确实是一个簇。

算法 3 $ClusterCheck$ 检查同步点 $\mathcal{X}$ 以及 $\mathcal{Y}$（其中 $\mathcal{X} \rightsquigarrow \mathcal{Y}$）是否存在一对子集，分别属于 $S_{\mathcal{X}}$ 和 $P_{\mathcal{Y}}$，使得它们之间的图是一个极小簇

函数 $ClusterCheck$(*G*: st-DAG, $\mathcal{X}, \mathcal{Y}$: MSP) **返回** 布尔值
  $Covered := \emptyset$
  /// $Covered$ 将是 $S_{\mathcal{X}}$ 的子集，其中包括所有具有
  /// 已在以下前向-后向可达性方案中处理
  $foundclusters := false$
  当 $Covered \neq S_{\mathcal{X}}$ **时**
    /// 只要 $S_{\mathcal{X}}$ 中还有顶点满足以下条件，就重复此循环
    /// 尚未被覆盖/测试
    $x :=$ $S_{\mathcal{X}}$ 中尚未属于集合 $Covered$ 的某个顶点
    $Xset := \{x\}$
    $Yset := \emptyset$
    $stable := false$
    当 不 $stable$**时执行**
      /// 此循环实现了前向-后向可达性方案
      $Ysetnew :=$ $P_{\mathcal{Y}}$ 中所有从 $Xset$可达的顶点
      如果 $Ysetnew \neq \emptyset$ **那么**
        /// $Ysetnew == \emptyset$ 意味着无法到达 MSP $\mathcal{Y}$
        /// 从顶点 x出发，此时我们可以停止
        /// 前向-后向方案
        $Xsetnew :=$ S 的所有顶点 x 从 $Ysetnew$ 出发可反向到达
      else
        $Xsetnew := Xset$
      $stable := (Xsetnew == Xset \wedge Ysetnew == Yset)$
      $Xset := Xsetnew$
      $Yset := Ysetnew$
    $Covered := Covered \cup Xset$
    如果 $|Xset| \geq 2 \wedge |Yset| \geq 2$，**则**
      如果 $ClosedCheck(G,\ Xset,\ Yset)$ **则**
        输出 $Xset$ 和 $Yset$ 之间的子图，作为一个极小簇
        $foundclusters := true$
  返回 $foundclusters$

算法 **4** $ClosedCheck$ 检查 $Xset$ 与$Yset$ 之间的子图是否闭合，也就是说，只能通过 $Xset$ 进入，并且只能通过 $Yset$退出1：函数 $ClosedCheck$ (*G*: st-DAG, $Xset$, $Yset$：顶点集) 返回布尔值 2： $VertexSet := FwReach($ $Xset) \cap BwReach($*Yset*$)$ 3：如果 $VertexSet == \emptyset$ 则 4：返回 $false$5：/// 实际上这个 if 子句是多余的，因为调用函数 6：/// 已经确保 $Yset$ 可从 $Xset$到达 7：$NonEntries := VertexSet \setminus Xset$8：如果 $\exists v \in NonEntries$ 的前驱顶点不在 $VertexSet$ 中，则 9：返回 $false$10： $NonExits := Vertices \setminus Yset$ 11：如果 $\exists v \in NonExits$ 的后继顶点不在 $VertexSet$ 中，则 12：返回 $false$ 13：返回 $true$

4.3. 删除 *MSP* 超路径

如前所述，一旦找到一条成功的 MSP 路径 p ，就可以安全地从所有列表 $PathsTo[.]$ 中删除/忽略 p 的所有超路径，因为我们只关注极小的簇。这一步在算法 2 的第 16 行完成。 p 的超路径是指在前端和/或尾端扩展 p 得到的路径。例如，MSP 路径 $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ 是 $[\mathcal{A},\mathcal{B},\mathcal{C}]$ 的超路径，但不是 $[\mathcal{A},\mathcal{C},\mathcal{D}]$ 的超路径（由此可见，在 MSP-DAG 中保留所有冗余边很重要）。通过在列表 $PathsTo[.]$ 的条目之间建立指针结构，并在每个列表中添加形如 $|_n$ 的子列表分隔符，就能高效地找到当前 MSP 路径 p 的所有超路径；这些操作都可以在算法 5 执行期间完成，几乎不会产生额外开销。更具体地说，当算法 5（第 9 行）将新路径 p_{new} 添加到列表 $PathsTo[v]$ 时，会创建一条从 $PathsTo[u]$ 中的 p 指向 $PathsTo[v]$ 中的 p_{new} 的指针。此外，在第 6 行的 for 循环中处理各个后继 v 时，算法会在转到下一个后继 v 之前，在当前列表 $PathsTo[v]$ 的末尾插入一个新的子列表分隔符 $|_1$ 。另外，在第 7 行的 for 循环中遍历列表 $PathsTo[u]$ 时，现有的子列表分隔符也会继承下来，但其索引会加一。因此，分隔符 $|_n$ 的索引 n 表示分隔符两侧的路径后缀有多长是相同的。换句话说，分隔符$|_n$ 的索引 n 表示对于源路径的哪些长度，应忽略该分隔符。因此，从一条顶点长度（记作 $vlen$）为 l 的路径开始，并在同一列表中向前移动时，所有满足 $l \leq n$ 的分隔符 $|_n$ 都会被忽略。借助这些指针和子列表分隔符，通过沿同一列表移动至下一个（有效）分隔符，并沿指针进入其他列表，就能轻松找到由额外前缀以及后缀产生的超路径。因此，所述指针/分隔符结构能够高效删除成功路径的所有超路径。请注意，创建

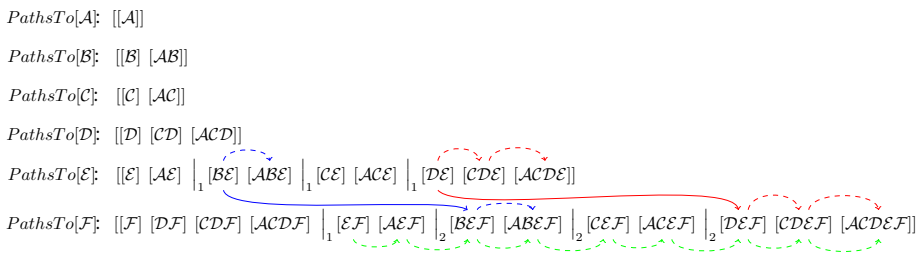

图7：与图6中的 MSP-DAG 对应的列表 $PathsTo[.]$；子列表分隔符以 $|_n$ 符号表示；图中只展示了部分指针结构，即路径 $[\mathcal{BE}]$（蓝色）、路径 $[\mathcal{DE}]$（红色）和路径 $[\mathcal{EF}]$（绿色）的指针结构；请注意，沿着列表从 $[\mathcal{EF}]$ 移动时会忽略分隔符 $|_2$，因为 $vlen([\mathcal{EF}]) \leq 2$（其中 $vlen$ 表示路径的顶点数）；同一列表中的虚线指针仅为说明之用，实际上并未实现，因为子列表分隔符足以指示超路径子列表的结束位置。

指针/分隔符并不属于算法 5 的代码，这样可以使算法清晰简洁。

关于上述指针/分隔符结构的示例，请参见图7，该图对应于图6中的 MSP-DAG。图中展示了算法 5 生成的所有列表 $PathsTo[.]$，每行一个列表。图中仅展示了两条示例路径的指针结构：路径 $[\mathcal{BE}]$（蓝色）和路径 $[\mathcal{DE}]$（红色）。在每个列表中，子列表分隔符以 $|_n$ 符号表示。请注意，从顶点长度为 $vlen([\mathcal{EF}]) = 2$ 的路径 $[\mathcal{EF}]$ 开始，在同一列表中移动时，所有分隔符 $|_2$ 都被视为“无效”，因此会被忽略。请注意，对于图6中的这个示例，实际上只有一个 MSP 对会导出一个极小簇，即对应于“成功”路径 $[\mathcal{BE}]$ 的这一对（$\mathcal{B}$，$\mathcal{E}$）。这意味着，在删除成功路径的超路径时，实际上只会沿着图7中的蓝色指针移动。

4.4. 复杂度分析

本节将分析所提出的最小簇查找算法的运行时间复杂度。

定理 1. 给定一个 *st-DAG* $G = (V, E)$ 及其关联的 *MSP-DAG* $G^M = (V^M, E^M)$，其中 $n = |V|$ 且 $N = |V^M|$。设任意最大同步点的起始/终止顶点数的最大值为 $k := \max\{\max_{\mathcal{X} \in V^M} |S_{\mathcal{X}}|, \max_{\mathcal{Y} \in V^M} |P_{\mathcal{Y}}|\}$。在 G 中查找所有极小簇的算法 *2* 的总体运行时间复杂度为

$$O(k \cdot n^2 \cdot 2^N)$$

本质上，该定理指出，复杂度关于 G 的顶点数呈二次增长，关于 G^M 的顶点数呈指数增长；后者是因为需要检查的 G^M 中相关路径的数量在最坏情况下可能很大。

证明。对于给定的 st-DAG $G=(V,E)$，设 $n:=|V|, m:=|E|$。类似地，对于相应的 MSP-DAG $G^M=(V^M,E^M)$，设 $N:=|V^M|, M:=|E^M|$。

在簇查找过程的不同阶段所需的以下信息可以预先计算：V（V^M）的拓扑序可以通过 Kahn 算法 [16], 计算，耗时 $O(n+m)$（耗时 $O(N+M)$）。对于任意给定顶点 $v \in V$，从该顶点（向前）可达的所有顶点的集合，记为 $FwReach(v)$，可以通过对 G 进行一次遍历（按逆拓扑序）计算并输出，耗时 $O(n^2)$（可一次性处理所有 v！）。类似地，所有能够到达任意顶点 v 的顶点所构成的集合，记为 $BwReach(v)$，可以通过一次扫描（按拓扑序）计算并输出，耗时 $O(n^2)$。在这两次遍历过程中，还可以免费获取并存储所有顶点 $v \in V$的直接后继集合 $Succ(v)$ 和直接前驱集合 $Pred(v)$。因此，预计算的总开销为$O(n^2)$。请注意，集合 $Pred(v)$、$Succ(v)$、$FwReach(v)$ 和 $BwReach(v)$ 可以借助符号数据结构（例如二叉决策图 [17]）进行紧凑存储，而且可以非常高效地对这些集合执行集合运算。

执行算法 $FindAllMinClusters$ 所需的基本操作（ops）总数为：

$$ops(compute\ all\ relevant\ paths) + \\ NumberOfRelevantPaths \cdot (ops(ClusterCheck) + ops(DeleteSuperpaths)) \tag{1}$$

其中，$ClosedCheck$ 的开销必须计入 $ClusterCheck$的开销中。

计算 G^M的所有相关路径的时间复杂度为 $O(N \cdot 2^N)$（见附录 A）。

算法 $ClusterCheck$ 的时间复杂度主要由 $S_{\mathcal{X}}$ 和 $P_{\mathcal{Y}}$ 的顶点之间的前向-后向可达性方案，以及对函数 $ClosedCheck$的调用决定。MSP 的大小可能不同，因此我们定义 $k:=\max\{\max_{\mathcal{X}\in V^M}|S_{\mathcal{X}}|, \max_{\mathcal{Y}\in V^M}|P_{\mathcal{Y}}|\}$，通常有 $k \ll n$。最坏情况下，前向-后向可达性方案的步骤数（总计，即计入$ClusterCheck$的外层 while 循环的所有迭代）为 $2k-1$（即 k 次前向步骤和 $k-1$ 次后向步骤）。由于每个步骤都涉及与 $P_{\mathcal{Y}}$（分别与 $S_{\mathcal{X}}$）进行交集运算，且每者最多包含 k 个元素，因此完成前向-后向可达性方案所需的操作数为 $O(k^2)$（请记住，所有 $FwReach(v)$ 和 $BwReach(v)$ 均已预先计算）。每次调用 $ClusterCheck$期间，函数 $ClosedCheck$ 最多被调用 $O(k)$ 次，且每次调用最多需要 $O(n^2)$ 次操作。为说明这一点，请注意，函数 $ClosedCheck$ 的运行时间主要由第 8 行（第 11 行也类似）决定。在第 8 行中，需要计算集合 $NonEntries$ 中所有顶点的前驱的并集，以便与 $VertexSet$的补集求交。尽管各个前驱集合已经预先计算，构造它们的并集仍需 $O(n^2)$ 次操作。因此，所有对 $ClosedCheck$ 的调用总共需要 $O(k \cdot n^2)$ 步。由此，每次调用 $ClusterCheck$ 所需的总时间为 $O(k^2 + k \cdot n^2) = O(k \cdot n^2)$。

对于特定的路径 p，已发现它是一条成功的 MSP 路径，

最坏情况下，最多需要删除 2^N –$(n+2)$ 条超路径（极端情况下，除 p 外的所有其他路径都要删除）。然而，在 $FindAllMinimalClusters$ 的 for 循环中，每条路径最多只会被删除一次，因此 MSP 路径删除的总次数为 $O(2^N)$。这意味着，MSP 路径删除的复杂度已计入因子 $NumberOfRelevantPaths$ 中，该因子为 2^N。

根据方程 1 汇总这些部分复杂度结果，并注意到 $k{\cdot}n^2 > N$，$FindAllMinClusters$的总体运行时间复杂度为:

$$O(N \cdot 2^N) + O(2^N \cdot ((k \cdot n^2) + 0)) = O(k \cdot n^2 \cdot 2^N) \tag{2}$$

此外，还需加上预计算的开销 $O(n^2)$（见本小节开头），因此算法 2 的总体复杂度为$O(n^2 + k \cdot n^2 \cdot 2^N) = O(k \cdot n^2 \cdot 2^N)$。

□

5. 应用

本节将展示一些实验结果，测试我们的最小簇查找算法在广泛的 st-DAG 上的表现，并由此收集一些有趣的统计数据。

5.1. 随机 *DAG* 的生成

为了为我们的最小簇查找算法生成测试用例，我们采用一种随机 DAG 生成方案。该方案从一个 3 顶点 DAG 开始，逐步扩展，直到达到所需的顶点数。生成过程由以下参数控制:

- n：最终有向无环图的顶点总数
- $parexp \geq 0$：并行扩展的概率
- $serexp \geq 0$：串行扩展的概率
- $maxwidth \in \mathbb{N}$：并行扩展或子图扩展的最大宽度
- $clustsettle$：控制子图扩展期间边数的因子
- $narb$：最后添加的任意（扰乱性）边的数量

满足以下约束： $parexp + serexp \leq 1$ 和 $0 < clustsettle < 1$.

算法从一个由三个串联顶点组成的有向无环图开始。源顶点为 s，目标顶点为 t。随后重复执行以下迭代步骤，直到达到所需的顶点数 n：随机选择一个顶点 x（ s和 t除外），并通过以下三种操作之一对其进行扩展：(1) 以 $parexp$ 的概率，将 x 替换为 $2 \leq k \leq maxwidth$ 个并联连接的新顶点，其中 k 均匀选取（请记住,

并联连接意味着所有新顶点都具有 x 原有的相同前驱和后继）。(2) 以 $serexp$ 的概率，将 x 替换为两个串联连接的顶点。(3) 以 $1 - parexp - serexp$ 的概率，将 x 替换为一个包含 $k+l$ 个顶点的新子图，其中 k, l 均匀随机选取，且满足 $2 \leq k \leq maxwidth$ 和 $2 \leq l \leq maxwidth$。该子图有 k 个入口顶点和 l 个出口顶点，每个入口顶点都有一条来自 x 的前驱的入边，每个出口顶点都有一条指向 x 的后继的出边。每个入口顶点至少连接到一个出口顶点（随机选取），并且每个出口顶点都可以通过一条边从至少一个入口顶点到达。入口顶点和出口顶点之间还会随机添加边，使得新子图的边总数大于 max（k, l）$+1$，但小于 $k \cdot l \cdot clustsettle$。

最后，在通过上述步骤创建了含有 n 个顶点的有向无环图后，$narb$ 会随机选择一个起始顶点和一个终止顶点，添加所谓的任意（破坏性）边。此时必须确保新添加的破坏性边不会形成环、不会冗余，也不会使其他已有边变得冗余。因此，尝试添加新破坏性边（u, v）时，需要检查以下三个条件：（1）u 到 v之间不能存在已有路径。（2）从 u 的任一祖先 a 到 v 之间不能存在已有边（否则边（a, v）会变得冗余）。（3）从 u到 v 的任一后代 d 之间不能存在已有边（否则边（u, d）会变得冗余）。

请注意，可能会发生超调，即生成的有向无环图中的顶点数可能略多于给定的 n。如果最后一步扩展选择并行扩展或子图扩展，且添加的顶点数超出 n，就会出现这种情况。具体而言，理论上可创建的顶点数上限为 $n-1+2 \cdot maxwidth$；当最后一步从含有 $n-1$ 个顶点的图开始进行子图扩展，且 k 和 l取最大可能值时，就会达到这一上限。

按此方案创建的有向无环图示例见图8。

5.2. 实验结果

本节报告使用算法 2 的原型实现进行的实验。该实现为单线程、未经优化的 Python 3 代码（使用 NetworkX 软件包）。所有实验均在一台搭载 Intel® Core i5-6400 处理器（2.7 GHz）、配备 16 GB 内存和 6 MB Intel® Smart Cache、运行 Ubuntu 24.04 的机器上进行。结果见表1。

实验采用第5.1节的有向无环图生成方案，在随机生成的不同规模有向无环图上进行。随着顶点数量 n 增加，参数 $maxwidth$ 也随之增大；参数 $narb$ 也同样如此。这样选择的依据是：与较小的图相比，较大的图应包含更宽、更大的子图，也应包含更多任意（具有干扰性的）边。对于所有规模的有向无环图，其余生成参数均保持不变，取值如表1的标题所示。尤其是，串行、并联和子图扩展出现的概率相等，这似乎是一个不错的折中。表中的每一行均基于对 100 个随机有向无环图样本的分析得出

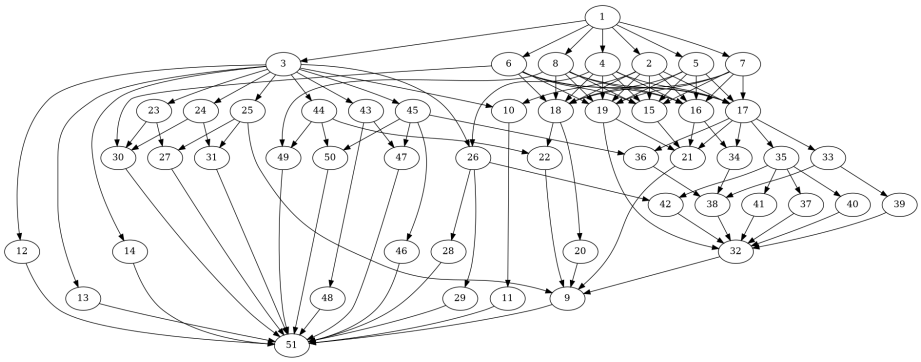

图8：随机生成的有向无环图。生成参数为 $n = 50$, $parexp =$ 0.33 $serexp = 0.33$ $maxwidth = 5$ $clustsettle = 0.2$ $narb = 10$ 10 , , , , 。扰动边依次为 (44,22)、(45,36)、(25,9)、(8,49)、(6,30)、(19,32)、(25,27)、(26,42)、(16,34) 和 (4,26)。注意，出现了轻微超调（顶点数为 51，而不是 50）。

。不过，对于 $n = 200$，.....，1000 的情况，我们针对每个 n 进行了两组实验。当然，这类成对行的统计数据非常相似，但仍存在一些明显差异，尤其是第（19）列中的最大运行时间。这些差异源于离群值；由于有向无环图是随机生成的，离群值自然会出现。

从表格的第 (4) 列和第 (7) 列可以看出，MSP 的数量$N = |V^M|$ 远少于源汇有向无环图顶点的数量 n（粗略而言，只有约 10%），这证实了我们的主张：处理规模小得多的 MSP 有向无环图，是我们的最小簇查找算法效率较高的一个主要原因。

第（8）列至第（11）列包含找到的极小簇数量信息。这些数量相当少，但更多实验（见下文本节末尾）表明，其他随机有向无环图生成参数可能会产生更多簇。请注意，使用这些参数得到的簇数量相对较少，但这通常并不会让我们的算法更容易处理，因为在这些情况下，算法 2 第 16 行无法删除许多超路径，而删除超路径显然可以节省运行时间。

簇大小的统计数据（第（12）行至第（15）行）是根据每行的全部 100 个样本得出的。簇大小显然变化很大。每行中至少有一个簇的最小大小为 4，也至少有一个簇包含超过 n 个顶点（多数情况下，这个簇包含有向无环图中除源点 s 和汇点 t之外的所有顶点），因此簇大小的方差极大。请记住，由于有向无环图生成过程中存在前文所述的超量效应，簇大小当然可能超过 n。

DAG G generation parameters			$N = \|V^M\|$: number of MSPs per DAG				number of clusters per DAG				cluster sizes (vertices)				runtime of Algo. 2 per DAG (sec)			
n	$maxwidth$	$narb$	mean	var	min	max	mean	var	min	max	mean	var	min	max	mean	var	min	max
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)
200	10	20	28.30	18.67	19	38	1.83	0.90	1	5	44.01	4718	4	206	0.081	0.003	0.015	0.277
200	10	20	28.59	26.58	15	45	1.65	0.85	1	6	42.45	4631	4	211	0.086	0.005	0.015	0.611
400	13	40	48.37	49.19	29	68	2.18	1.47	1	6	49.07	12570	4	407	0.400	0.133	0.054	1.715
400	13	40	47.36	49.09	33	66	1.91	1.80	1	7	58.27	15881	4	410	0.427	0.167	0.069	2.674
600	16	60	60.21	72.59	38	78	2.17	1.74	1	7	73.31	30177	4	615	0.854	0.597	0.156	4.221
600	16	60	60.94	79.16	36	78	2.19	1.49	1	6	69.04	27254	4	624	0.904	0.815	0.141	4.914
800	19	80	72.66	84.32	53	97	2.06	1.50	1	7	88.57	51077	4	811	1.834	6.640	0.337	18.693
800	19	80	73.83	96.67	52	95	2.52	2.05	1	6	69.63	41166	4	813	1.767	2.077	0.316	6.403
1000	22	100	78.36	115.87	56	102	2.14	1.88	1	8	80.60	64465	4	1031	2.147	3.515	0.446	9.920
1000	22	100	80.18	157.49	46	110	2.44	2.61	1	11	100.23	75202	4	1012	2.466	6.796	0.388	14.349
1500	26	150	106.78	181.61	78	147	2.39	2.22	1	8	112.03	135605	4	1523	6.939	55.36	1.058	51.267
2000	31	200	125.05	290.45	91	183	2.74	2.89	1	9	101.05	172638	4	2033	14.295	220.18	1.557	76.050
2500	36	250	136.87	372.59	89	215	2.71	2.65	1	8	120.56	245996	4	2537	23.982	912.75	2.730	227.175
3000	39	300	153.35	287.01	107	194	2.86	4.26	1	9	212.93	556848	4	3061	34.150	3245.19	2.985	405.332
3500	42	350	170.37	428.25	127	210	3.07	3.27	1	8	244.89	698949	4	3536	63.823	7579.72	4.479	587.152
4000	45	400	185.96	671.58	112	256	3.19	4.13	1	11	158.99	558597	4	4029	75.317	8050.69	7.714	383.237
4500	48	450	199.65	731.59	148	271	3.06	4.56	1	10	196.58	765315	4	4530	106.679	17423.67	8.940	917.959
5000	50	500	211.07	497.93	147	265	3.68	6.20	1	12	159.34	706631	4	5020	139.014	44355.85	10.951	1638.558

Table 1: Experimental results showing the number of MSPs, the number of minimal clusters, their sizes and the runtime of Algorithm 2 (the minimal cluster finding algorithm) for different DAG sizes. The number of random DAG samples per row was 100. The following graph generation parameters were fixed for all experiments in this table: $parexp = 0.33$, $serexp = 0.33$ and $clustsettle = 0.4$

表1的最后四列给出了算法 2的运行时间信息。随着有向无环图规模的增大，运行时间的增长起初非常缓慢 n ，仅观察到一些并不显著的离群值，例如 $n = 800$。运行时间的绝对值（即使在这台性能相当低端的机器上，对于最多达到 $n = 1000$ 的规模，也不超过几秒）证实了我们的算法高效且具有实际应用价值。

图 9以图形方式展示了表1中的部分信息。算法 2测得的平均运行时间以蓝色（实线）表示，测得的平均 MSP 数量以红色表示。对于运行时间，我们还拟合了一个函数$f(n) = c_1 n^2 + c_2\, n^2\, 2^{N(n)}$（采用最小二乘法），该函数与定理 1 中的表达式并不完全一致，而是在 n 上增加了纯二次依赖关系。计算得到的系数为 $c1 = 4.882E{-}6$ 和 $c2 = 1.964E{-}70$（拟合过程中使用了特征缩放，以避免项 $2^{N(n)}$导致溢出）。这一拟合表明，在所绘制的范围内，指数项尚未发挥任何显著作用，而运行时间的增长总体上呈二次关系。因此，对于不超过 $n = 5000$的有向无环图规模，定理 1所述的运行时间相对于 N的指数增长尚不可观察到，这当然是一个好消息。

观察图 9 中的蓝色实线曲线，可以发现 $n = 3500$ 对应的数据点出现了异常。这一点在表1中已经可以观察到，其原因是存在一些离群值，显然源于为 $n = 3500$ 随机生成的一些非常“棘手”的有向无环图（见表1第 (19) 列）。为验证这一推测，我们针对 $n = 3500$ 使用与表1相同的参数再次进行了实验（同样包含 100 个样本）。果然，这次观察到的平均运行时间仅为 44.749 秒（这将避免图 9 中蓝色实线的弯折），最大运行时间也低得多，仅为 338.488 秒。

如前所述，表1和图 9 中使用的参数产生的簇数量相对较少。为展示有向无环图可能包含更多的簇，我们针对 $n = 1000$ 再次进行了实验（同样包含 100 个样本）。这次，随机生成的有向无环图采用的其他参数如下：$parexp = 0.4$、$serexp = 0.4$、$maxwidth = 15$、$clustsettle = 0.2$ 和 $narb = 15$。该实验得到的簇平均数量为 22.23（此前为 2.44），簇大小最大值为 34（此前为 11）。这组实验的平均运行时间为 3.951 秒（此前为 2.466 秒）。

6. 结论

本文介绍并评估了一种在 st-DAG 中查找极小簇的新算法，其中簇被定义为一种特殊类型的子图，其入口和出口边界构成一种同步点或屏障。这是一个具有实际意义的问题，因为有向无环图在许多应用领域中都有使用，而大型有向无环图的结构分析十分困难。识别这类子图能够采用分治方法，从而大幅提高大型有向无环图分析的效率。

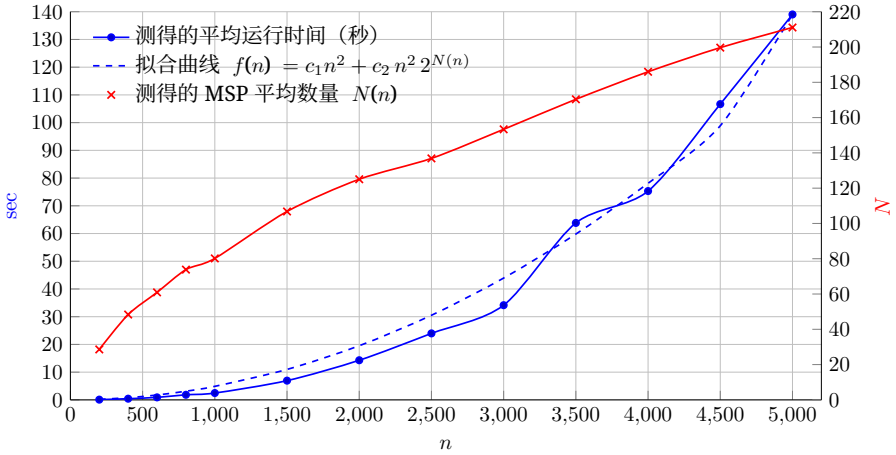

图 9：算法 2 的测得平均运行时间（蓝色实线）和测得的 MSP 平均数量（红色）。请注意，蓝色曲线和红色曲线对应的纵轴不同。图中还显示了一个用于拟合平均运行时间的闭式函数 $f(n) = c_1 n^2 + c_2\, n^2\, 2^{N(n)}$（蓝色虚线）。因此，该拟合为定理 1 的理论结果引入了一个替代性的二次项。由于 $c1 = 4.882E{-}6$ 和 $c2 = 1.964E{-}70$，在所绘制的 n 范围内，第二项可以忽略不计。

从技术上讲，我们方法的关键在于同步点这一概念，尤其是最大同步点（MSP）；我们还介绍了一种高效查找最大同步点的算法。我们提出了一种最小簇查找算法，并提供了详尽的伪代码以及若干高效实现的建议。该算法并不直接作用于给定的有向无环图，而是作用于所谓的最大同步点有向无环图（MSP-DAG）；后者通常要小得多。这正是该算法高效的主要原因。本文还进行了严格的复杂度分析，其结论得到了多项具有一定难度的实验支持。

作为未来的工作，我们计划在项目规划工具集中利用最小簇查找算法的能力，以便在存在不确定性和资源限制的现实环境中预测项目时间线。

致谢：作者谨向同事 Fabian Michel 致以诚挚谢意，感谢他对稿件早期版本提出的宝贵意见，尤其是在复杂度分析方面的建议。

附录 A. 计算所有相关路径

对于给定的有向无环图，可以通过算法 $ComputeAllVertexPaths$ 计算任意长度的所有顶点路径集合，如算法 5 所示。在我们的工作中，算法不是在 st-DAG $G = (V, E)$ 上运行，而是在其对应的 MSP-DAG $G^M = (V^M, E^M)$ 上运行，后者通常要小得多。不过，为了

算法 5 $ComputeAllVertexPaths$ 计算任意长度的有向无环图 $G = (V, E)$ 中的顶点路径

```
1: 函数 ComputeAllVertexPaths (G: st-DAG) 返回 空
2: 对于所有 v ∈ V 执行
3:     初始化 PathsTo[v] := [[v]]
4:     /// 初始化以 v 结尾的路径列表，其中包含一条长度为零的路径
5: 对于拓扑序中的所有 u ∈ V 执行
6:     对于每个 u 的后继 v 执行
7:         对于 PathsTo[u] 中的每条路径 p 执行
8:             将 v 追加到 p，创建一条新路径 p_new
9:             将 p_new 添加到列表 PathsTo[v]
10: /// 现在，每个列表 PathsTo[v] 都包含以 v 结尾的任意长度的所有路径
```

简单起见，在本附录中我们仍使用记号 $G = (V, E)$。在 G^M 上运行算法后，只有那些起点为 FSP 或 FHSP、终点为 FSP 或 BHSP 的路径与我们的目的相关，其他路径都无关。因此，在 G^M 上运行算法 $ComputeAllVertexPaths$ 后，我们会丢弃所有满足 $\mathcal{Y}$ 为 FHSP 的列表 $PathsTo[\mathcal{Y}]$。此外，在剩余列表中，我们删除所有以 BHSP 开头的路径。由此得到的列表包含了 G^M 中所有相关的顶点路径，其起点和终点可能构成某个簇的边界。按照构造，这些列表 $PathsTo[\mathcal{Y}]$ 已按路径长度递增顺序（分段）排序，这对于之后高效删除超路径很重要。

现在我们分析算法 $ComputeAllVertexPaths$（见算法 5）的运行时间复杂度。最坏情况，即路径数最多的有向无环图，是顶点集为 $V = \{1, \dots, n\}$、边集达到可能最大值 $E = \{(k, l) \mid k, l \in V \wedge l \geq k\}$ 的有向无环图。对于该图，拓扑排序得到的顺序是唯一的，即 $1 \prec 2 \prec \cdots \prec n$。按照第 5 行和第 6 行的 for 循环进行处理，顶点 $u = k$ 有 $(n - k)$ 个后继 v。第 8 行会将顶点 v 追加到在 $PathsTo[u]$ 中找到的每条路径上。由于 $PathsTo[u]$ 包含 $\binom{k-1}{j-1}$ 条长度为 $j-1$ 的路径，其中 j 的取值范围为 $\{1, \dots, k\}$，而新路径 p_{new} 的长度为 $(j - 1) + 1 = j$，因此每个后继需要 $\sum_{j=1}^{k} \binom{k-1}{j-1} \cdot j$ 次基本操作。总而言之，在这个含有 n 个顶点的最坏情况有向无环图上执行 $ComputeAllVertexPaths$ 所需的操作次数为

$$\sum_{k=1}^{n-1} (n-k) \sum_{j=1}^{k} \binom{k-1}{j-1} \cdot j \tag{A.1}$$

公式 A.1 中的表达式可以简化为 $(2n - 4)2^{n-2} + 1$（首先将内部求和替换为表达式 $(k + 1)2^{k-2}$，然后展开并应用几何级数论证）。因此，算法 5 的时间复杂度为 $O(n \cdot 2^n)$。请注意，这是在具有 n个顶点的有向无环图中<条目 id='7'>计算 所有任意长度路径的复杂度（其中还包括输出每条路径），而任意长度路径的<条目 id='10'>数量最多仅为

$2^n - (n+1) = O(2^n)$，这一点可以通过考察上述最坏情况有向无环图的简单组合论论证来证明。至此，我们再次确认了一个众所周知的结果：有向无环图中的路径数量可能关于顶点数量呈指数增长。但必须再次强调，我们应用算法 $ComputeAllVertexPaths$ 的对象并不是 st-DAG，而是通常小得多的 MSP-DAG！

参考文献

[1] U. Vogl、M. Siegle，通过概率模型检测预测可靠项目运行时间的新方法，载于：P. Reinecke、A. Di Marco（编），计算机性能工程（EPEW 2017），*LNCS*第 10497 卷，Springer International Publishing，Cham，2017 年，第 117–132 页。[2] U. Vogl、M. Siegle，规模化敏捷：借助工具对 PI 规划进行实时合理性验证，载于《项目管理与过程模型 2023——可持续 IT 项目》，德国计算机学会，2023 年，第 149–159 页。[3] M. Kwiatkowska、G. Norman、D. Parker，PRISM 4.0：概率实时系统的验证，载于：G. Gopalakrishnan、S. Qadeer（编），第 23 届国际计算机辅助验证会议论文集（CAV'11），*LNCS*第 6806 卷，Springer，2011 年，第 585–591 页。[4] U. Vogl，在资源约束下改进预测、更可靠地制定项目时间计划的方法，博士论文，慕尼黑联邦国防军大学，2025 年。[5] F. Hartleb、A. Quick，并行程序的性能评估——使用 PEPP 工具进行建模与监控，载于：B. Walke、O. Spaniol（编），计算系统与通信系统的测量、建模和评估，Springer Berlin Heidelberg，柏林、海德堡，1993 年，第 51–63 页。[6] R. Klar、P. Dauphin、F. Hartleb、R. Hofmann、B. Mohr、A. Quick、M. Siegle，并行与分布式计算系统的测量和建模，Teubner，斯图加特，1995 年。[7] L. Carnevali、M. Paolieri、R. Reali、E. Vicario，复杂工作流响应时间概率密度函数的组合式安全近似，ACM 建模、计算与仿真汇刊 33（2023）。网址：https://doi.org/10.1145/3591205。doi:10.1145/3591205。[8] L. Carnevali、R. Reali、E. Vicario，EULERO：用于复杂工作流定量建模与评估的工具，载于：E. Ábrahám、M. Paolieri（编），系统定量评估，Springer International Publishing，Cham，2022 年，第 255–272 页。

[9] R. Tarjan，深度优先搜索与线性图算法，SIAM计算学报 1（1972）146–160。网址：https://doi.org/10.1137/0201010。doi:10.1137/0201010。arXiv:https://doi.org/10.1137/0201010。[10]W. McLendon III、B. Hendrickson、S. J. Plimpton、L. Rauchwerger，在分布式图中寻找强连通分量，并行与分布式计算学报 65（2005）901–910。网址：https: // www.sciencedirect.com/science/article/pii/S0743731505000535。doi:https://doi.org/10.1016/j.jpdc.2005.03.007。[11]P. Pardalos、J. Xue，最大团问题，全球优化学报 4（1994）301–328。网址：https://doi.org/10.1007/ BF01098364。[12]Q. Wu、J.-K. Hao，最大团问题算法综述，欧洲运筹学杂志 242（2015）693–709。网址：https://www.sciencedirect.com/science/article/pii/ S0377221714008030。doi:https://doi.org/10.1016/j.ejor.2014.09. 064。[13] S. E. Schaeffer，图聚类，计算机科学评论 1（2007）27–64。网址：https://www.sciencedirect.com/science/article/pii/ S1574013707000020。doi:https://doi.org/10.1016/j.cosrev.2007. 05.001。[14]A. V. Aho、M. R. Garey、J. D. Ullman，有向图的传递约简，SIAM计算学报 1（1972）131–137。网址：https://doi.org/10.1137/0201008。doi:10.1137/0201008。arXiv:https://doi.org/10.1137/0201008。[15] J. Valdes、R. E. Tarjan、E. L. Lawler，串并联有向图的识别，载于：第十一届 ACM计算理论研讨会论文集，STOC ’79，美国计算机协会，纽约，NY，美国，1979 年，第 1–12 页。网址：https://doi.org/10.1145/ 800135.804393。doi:10.1145/800135.804393。[16] A. B. Kahn，大型网络的拓扑排序，Commun. ACM 5（1962）558–562。网址：https://doi.org/10.1145/368996.369025。doi:10.1145/ 368996.369025。[17] Bryant，用于布尔函数处理的基于图的算法，IEEE计算机学报 C-35（1986）677–691。doi:10.1109/TC.1986. 1676819。