Directed Acyclic Graph

Directed Acyclic Graph

> In mathematics, particularly graph theory, and computer science, a **directed acyclic graph** (**DAG**), is a finite directed graph with no directed cycles.

Task Scheduling

Topological order

任务调度是 DAG 的一个典型应用，假设有一组任务，其中某些任务对有依赖关系，需要调度任务的执行，使得一个任务执行时其依赖的任务均已执行完毕。将任务抽象为顶点，依赖关系抽象为有向边，这个问题可以建模为有向图的拓扑排序

> **定义 4.2** 拓扑排序
>
> 如果为图中每个顶点 $v_1, v_2, v_3, \dots v_n$ 分配一个序号 $\tau_1, \tau_2, \dots \tau_n$ ，满足
>
> - 所有序号为 1 到 n 的正整数的一个排列
> - 对任意有向边 $i \to j, \tau_i < \tau_j$
>
> 则称 $\tau_1, \tau_2, \dots \tau_n$ 为图 G 中顶点的一个拓扑排序。（逆拓扑排序定义类似，只是关系改为大于）

显然，当任务中出现循环依赖时便无法调度，有向图中出现环时也无法拓扑排序

> **引理 4.1** 如果有向图 $G = (V, E)$ 中有环，则图 G 不存在拓扑排序
>
> 证明：若有环，根据拓扑排序的定义，环上的所有节点的序号都严格大于自身，这是不可能的
>
> **引理 4.2** 如果有向图 $G = (V, E)$ 为 DAG，则 G 必然存在拓扑排序

给定一个 DAG，可以基于 DFS 求出其拓扑排序

> **定理 4.3** 图 G 是 DAG $\iff$ G 存在拓扑排序

Algorithm

TOPO-WRAPPER(G)

```
globalNum := n + 1
Color all nodes WHITE
foreach node v in G do
    if v.color = WHITE then
        TOPO-ORDER(v)
```

TOPO-ORDER(v)

```
v.color = GRAY
foreach neighbor w of v do
    if w.color = WHITE then
        TOPO-ORDER(w)
globalNum := globalNum + 1
v.topoNum := globalNum
v.color = BLACK
```

证明该算法正确性即可证明上述引理 4.2 的正确性

> - 算法 TOPO-ORDER 对每个节点仅调用一次，故每个编号一定在 1 到 n 的范围内且互不相同
> - 对于图中任意边 vw，显然 vw 不能是 BE（否则会成环），其余情况下均有
>
> $$v.finishTime > w.finishTime$$
>
> 显然 w 的编号的分配早于 v，而 `globalNum` 是严格单调递增的，故 w 的编号小于 v 的编号

算法基于图遍历，时间复杂度为 $\Theta(n+m)$

End node

在 DFS 的过程中，算法沿一条路径走下去，直到遇到一个“尽头”，即不能再继续遍历下去的点。

“尽头”的概念有助于理解拓扑排序等处理节点间依赖关系的算法，有向边表示节点的依赖关系，则“尽头”是不依赖任何顶点的节点，所以有向无环的条件很关键，保证了图中一定有“尽头”的存在

尽头也可分为 logical end 和 structural end

- structural end 即没有出度的节点

- logical end 即为其后续节点均已处理完毕（DFS 过程中变黑时）

对于拓扑排序问题，structural end 不依赖任何节点，可以自由分配其编号，而 logical end 的依赖关系均以确定，也可为其分配编号，且不影响未来的编号的分配。故算法的过程就是不断寻找 logical/structural end 并为其分配拓扑编号

对于基于 DFS 的算法，理解 end node 的概念对于理解算法很关键（详见课本 P.47）

Critical Path

问题：更为复杂的任务调度，任务间有依赖关系，且每个任务有自己的时长。设在依赖满足的条件下可同时进行任意多的任务。

假设调度的任务为 $a_i (1 \leqslant i \leqslant n)$，每个任务执行的时长为 l_i，依赖关系可建模为有向图 G，显然若要完成调度 G 为 DAG，则可定义每个任务的最早开始时间（earliest start time）est_i 和最早结束时间（earliest finish time）eft_i

> **定义 4.3** 最早开始时间与最早结束时间
>
> - 若 a_i 不依赖其他任务，$est_i = 0$，否则 $est_i = \max\{eft_j \mid a_i \to a_j\}$
> - $eft_i = est_i + l_i$

则所有任务完成的最短用时即为关键路径上任务的时长的和

> **定义 4.4** 关键路径
>
> 任务调度中的关键路径是一组任务 $v_0, v_1, v_2, \ldots, v_k$ 满足
>
> - v_0 不依赖任何任务
> - 对任意 $1 \leqslant i \leqslant k$，有
>
> $$v_i \to v_{i-1}$$
> $$est_i = eft_{i-1}$$
>
> - $eft_k = \max_{1 \leqslant i \leqslant n} eft_i$

关键路径的时长决定了所有任务执行完所需的最短时长，显然减少或增加关键路径上任务的时长会导致整个计划的时长的减少/增加，而非关键路径的任务即使时长减少，也不会减少任务执行完所需的总时长

该问题也可通过“尽头”的概念来理解，对于一个不依赖任何任务的任务，其即为 structural end，可确定其 est, eft，而对于一个所有依赖的任务的 eft 均已确定的任务，其成为 logical end，同样可确定其 est, eft

算法如下，同样需要 WRAPPER 来调度

CRITICAL-PATH(v)

```
v.color := GRAY
v.est := 0
v.CritDep := -1
foreach neighbor w of v do
    if w.color = WHITE then
        CRITICAL-PATH(w)
        if w.eft >= v.est then
            v.est := w.eft
            v.CritDep := w
    else                                  // 即使访问过仍要检查依赖关系
        if w.eft >= v.est then
            v.est := w.eft
            v.CritDep := w
v.eft := v.eft + v.l
v.color := BLACK
```

> 正确性证明
>
> 当访问到节点 w 的 eft 时，显然 w 一定不是灰色的（否则成环），则 w 一定是黑色的，w 的 eft 已被初始化
>
> 根据 DFS，对于每个顶点仅调用一次算法，eft 仅被赋值一次，其值符合 eft 定义

算法基于 DFS 实现，其时间复杂度为 $\Theta(n+m)$

Strong Connected Component

> **定义 4.5** 强连通片，收缩图
>
> 有向图中的两个顶点强连通 $\iff$ 两个点互相可达
>
> 有向图的强连通片是其极大强连通子图，若把 G 中每个强连通片收缩成一个点，强连通片之间的边收缩成一条有向边，则得到 G 的收缩图（condensation graph）$G\downarrow$

显然强连通关系是等价关系，收缩图中的顶点即是等价类，由于不同强连通片只能单向可达，故强连通片之间的有向边是 well-defined

> 有向图的收缩图是 DAG

将 G 转置得到 G^T ，显然强连通片不会发生变化，但 $G\downarrow$ 会发生变化

Strategy

如果一个连通片 C_1 在 $G\downarrow$ 中是源点，则其在 $G^T\downarrow$ 中就是尽头，根据尽头的性质（没有出度），在 $G^T\downarrow$ 中从 C_1 中节点开始新的一轮 DFS 遍历，可以正好遍历完 C_1 中的节点。故只需保证在 $G^T\downarrow$ 中首先遍历尽头节点中的点。

对于非尽头的连通片，可类比 logical end，当一个非尽头的强连通片的后继全部被遍历后，其就成为 logical end，虽然从其中的点开始遍历会遍历多个强连通片，但可根据之前的结果遍历剔除不属于其的节点

关键便是如何实现强连通片之间的拓扑排序

- 遍历结束时标记尽头：每个节点在遍历结束前记录自己为逻辑尽头
- 栈结构：节点完成处理时进入一个栈
- 图转置：经过第一轮遍历，节点按照先结束先入栈的顺序完成排列，在此基础上将图 G 转置，从节点栈中依序弹出节点进行第二轮遍历

最后入栈的节点便是最后处理完的节点，在图 $G\downarrow$ 中其是属于源点的节点，而在转置后其所属强连通片便成为尽头

Algorithm

SCC(G)

```
Initiate the empty stack nodeStack
Perform DFS on G. In the post order of each vertex, insert the
statement "nodeStack.push(v)"
Compute the transpose graph G^T of G
Color all nodes WHITE
while nodeStack != empty do
    v := nodeStack.pop()
    Conduct DFS from v on G^T
```

Correctness

> **定义 4.6** 强连通片的首节点
>
> 在第一轮 DFS 过程中，定义每个强连通片中第一个被发现的节点为该强连通片的首节点（leader），即等价类的一个代表元

基于 DFS 的性质，易得 leader 有性质

> **推论 4.1** 对于一个强连通片来说，leader 是第一个发现也是最后一个结束的。leader 的 active interval 包含同一连通片中其他节点的 active interval

且基于强连通的性质，易得

> **引理 4.3** 第一轮遍历的 DFS 遍历树中，可能包含一个或多个强连通片，但每个强连通片都是完整的，不会出现只包含一部分强连通片的节点

则需要证明的难点就在于当一个连通片有边指向另一连通片时，使用栈对其中的节点排序，如何确保该排序是正确的。考察这两个连通片的关系

> **引理 4.4** 当某个强连通片的 leader 在第一轮遍历中被发现时，不可能有路径通向某灰色节点
>
> 证明：假设强连通片 C_i 的 leader l_i 被发现时，有路径通向某灰色节点 x ，由于 l_i 是 leader，故 x 一定处于别的连通片 C_j ，存在从 C_i 到 C_j 的路径。而由于发现 l_i 时 x 为灰色，故 x 是 l_i 在遍历树中的祖先，存在从 C_j 到 C_j 的路径，与 C_i, C_j 为不同强连通片矛盾

> **引理 4.5** 设 l 是某个强连通片的 leader，x 是另一个强连通片中的节点，且存在从 l 到 x 的路径，则有
>
> $$x.finishTime < l.finishTime$$
>
> 证明：由引理 4.4，在 l 刚发现时，x 只能为白色或黑色
>
> - 若 x 为黑，则 l 被发现时 x 已遍历结束，$x.finishTime < l.discoverTime < l.finishTime$
> - 若 x 为白，则从 l 到 x 的路径为白色路径。若存在非白色节点，根据引理 4.4 ，该节点只能为黑色，但不可能存在一个黑色节点有路径通向白色节点。根据白色路径定理，x 是 l 在遍历树中的后继，则有 $x.finishTime < l.finishTime$

> **引理 4.6** 在第二轮遍历中，当一个白色节点从栈中取出时，它一定是其所在连通片的 leader
>
> 证明：若第二轮遍历时 l 出栈时为白色，则其一定是所在连通片第一个出栈的节点。在第一轮遍历时 l 最后入栈，即最后结束的节点，根据 DFS ，其 active interval 包含连通片中其他节点的 active interval，则其是 leader。若出栈的节点不是 leader，则在强连通片之中的其他节点已经被处理，而 DFS 遍历只会遍历整个强连通片，故出栈的节点不会是白色

根据以上引理，可证明 SCC 算法的正确性

根据引理 4.6，一个强连通片的 leader l 最先出栈。对其遍历可遍历强连通片中所有节点。假设第二轮遍历时 l 有路径通向另一强连通片 S，S 的 leader 为 x ，则在 G^T 中存在从 l 到 x 的路径，在 G 中存在从 x 到 l 的路径，由引理 4.5，$l.finishTime < x.finishTime$ ，故第一轮遍历时 l 先于 x 入栈，而第二轮遍历时 x 先于 l 出栈，在处理 l 时 x 所在强连通片已被处理完毕，可正确剔除从 l 可达的其他连通片的点。

第二轮遍历时，每个 DFS 遍历树正好包含一个强连通片