上一篇文章我们介绍了几个关于拓扑排序的概念，包括入度、出度、拓扑次序等，这一篇就来讲一下拓扑排序的算法和应用。拓扑排序（topological sort）是将有向图内所有顶点进行线性排序，使其符合拓扑次序（即对于图中任意一条边e = (v1, v2)来说，v1总是排在v2前面）的算法。如果图有环，则环上不存在任何顶点可以按照这个次序排在其它顶点之前，因此不存在拓扑次序。无向图任意一条边连接的两个顶点同理。而每个有向无环图至少存在一个拓扑次序，并且存在多个拓扑排序算法，一种是Kahn's algorithm（我不用中文了！），一种是深度优先搜索（DFS）的加强版，还有一种是通过并行计算（parallel computation）优化过的最大和矩阵乘法（max-sum matrix multiplication）。

1. Kahn's algorithm
   首先创建一个空的列表（list）L用来存储排序好的顶点，找出所有入度为0的顶点并放入集合S。当S不为空时，不断从S里取出一个顶点u，将u放入列表L的结尾，去掉所有以该顶点为起点的边e = (u, v)，如果v此时入度为0，则将v加入集合S。最后如果还有剩余边，则这些边一定是从列表L中排序靠后的点到排序靠前的点，说明图里有环，反之则返回包含所有排好序的顶点的列表L。

   伪代码如下：

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

（from Wikipedia）

   时间复杂度：O(|V| + |E|)，其中|V|为顶点数，|E|为边数
   空间复杂度：O(|V|)，其中|V|为顶点数，不包括输入图占用的空间

2. 深度优先搜索

首先创建一个空的列表（list）L用来存储排序好的顶点。之后不断从选取一个未访问过的顶点作为根节点开始DFS，将所有访问过的节点按照访问顺序先后插入列表L的开头。值得注意的是，每个访问到的顶点先被标记为临时节点，等所有该节点的子节点均被访问完毕时再被标记为永久节点。这样做的目的是防止有环出现：对于当前访问的节点来说，每个临时节点都在它被访问前已经被访问了，如果在它之后又访问了其中一个临时节点，则说明有环存在，该图并非有向无环图，不存在拓扑次序。

伪代码如下：

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then return
    if n has a temporary mark then stop   (not a DAG)
    mark n with a temporary mark
    for each node m with an edge from n to m do
        visit(m)
    remove temporary mark from n
    mark n with a permanent mark
    add n to head of L
```

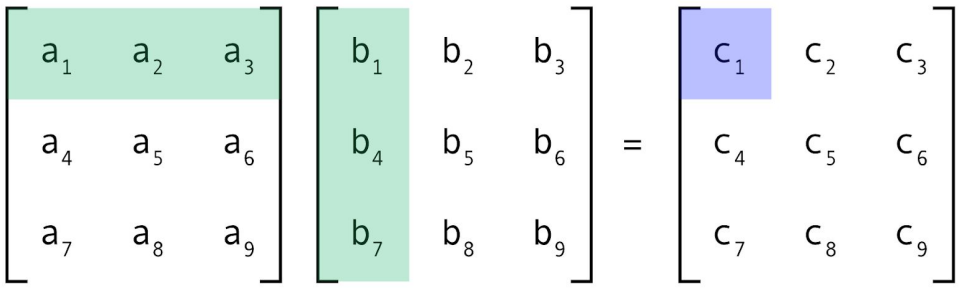
（from Wikipedia）

时间复杂度：O(|V| + |E|)，其中|V|为顶点数，|E|为边数
空间复杂度：O(|V|)，其中|V|为顶点数，不包括输入图占用的空间

可以看出两种算法的时间复杂度和空间复杂度相似，通常也没有一种算法显著优于另一种的情况，但两者还是有差异的。比如Kahn's algorithm需要在遍历以当前顶点为起点的每条边时需要判断终点的入度。最差的情况下，例如n个点从左到右排列，每个点都有边连向它右边的所有点，这时所有判断总时间复杂度为O(|V|^3)，即伪代码中while loop时间复杂度为O(|V|)，for loop时间复杂度为O(|V|)，判断入度时间复杂度为O(|V|)。不过可以通过提前算好每个顶点的入度来把总体时间复杂度降低到O(|V| + |E|)。除此之外，Kahn's algorithm通过后插元素更新列表L，而DFS通过前插元素更新列表L，可以根据数

据结构前插元素和后插元素的成本来决定使用哪种算法。（有没有觉得Kahn's algorithm的本质就是BFS）

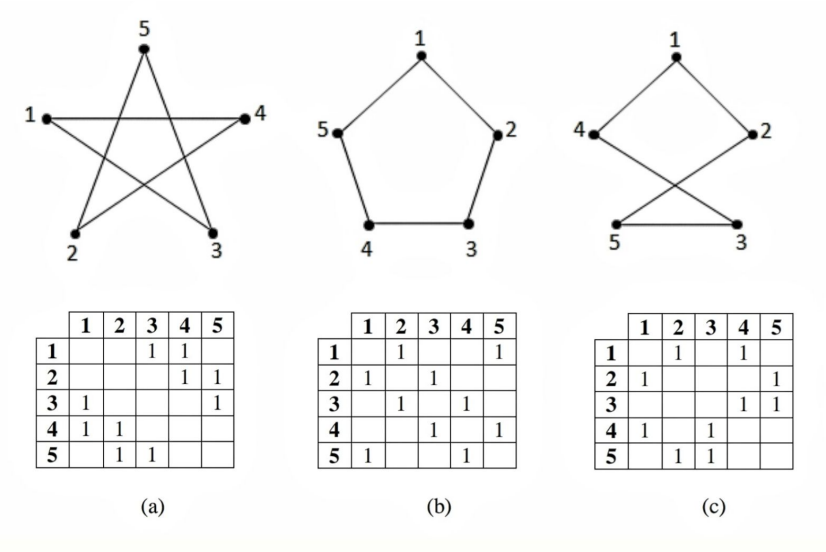
还有一种通过并行计算进行拓扑排序的算法，基于最小和矩阵乘法（min-plus matrix multiplication）衍生出的最大和矩阵乘法。我们知道普通的矩阵乘法是，给出两个n * n矩阵 A = (a_{ij})和B = (b_{ij})，则他们的乘积C = (c_{ij}) = sum_k {a_{ik} * b_{kj}}，其中k的取值范围是1到n。



（from Medium）

对于最小和矩阵乘法，C = (c_{ij}) = min_k {a_{ik} + b_{kj}}，当一个图的带权重（weight/cost）的邻接矩阵（即该矩阵M中M[i][j]是从顶点i到顶点j的边的权重，在最短路问题中权重通常是顶点i到顶点j之间的距离）和自己最小和相乘时，所得的乘积c_{ij}为由顶点i到顶点j以任意顶点k为中转的最短路（也是路径长度为2的最短路）。如果我们不断对该矩阵进行最小和相乘，就可以得到更长路径的最短路（需要注意的是不考虑不存在的边）。

对于最大和矩阵乘法，C = (c_{ij}) = max_k {a_{ik} + b_{kj}}，当一个图的不带权重（unweighted）的邻接矩阵（由0和1组成的矩阵）和自己不断最大和相乘时，所得矩阵M中M[i][j]越大，则说明从顶点i到顶点j的路径越长。



	1	2	3	4	5
1			1	1	
2				1	1
3	1				1
4	1	1			
5		1	1		

(a)

	1	2	3	4	5
1		1			1
2	1		1		
3		1		1	
4			1		1
5	1			1	

(b)

	1	2	3	4	5
1		1		1	
2	1				1
3				1	1
4	1		1		
5		1	1		

(c)

具体来说，两个相同的邻接矩阵一次最大和相乘后得到的矩阵中最大值为2，代表两条相连的边；三个相同的邻接矩阵两次最大和相乘后得到的矩阵中最大值为3，代表三条相连的边；以此类推，|V| - 1个相同的邻接矩阵|V| - 2次最大和相乘后得到的矩阵中最大值为|V| - 1，也是图中可能存在的最长路径的长度。参考快速幂的思路，当我们得到一个乘积时，我们可以通过让它继续和自己相乘而不是和原邻接矩阵相乘来把矩阵相乘的次数从O(|V|)提高到$O(\log_2|V|)$。举一个快速幂的简单例子，当计算2的64次方的时候，普通的操作步骤是把2相乘64次，而快速幂则是计算2 * 2 = 4，4 * 4 = 16，16 * 16 = 256，以此类推，一共$\log_2 64 = 6$次。矩阵对角化也可以用来解决这种矩阵的n次幂问题，但时间复杂度相对较高（根据Shroff在1990年发表的论文为$O(n\log^2 n)$），而且限制是所有特征向量必须线性无关。

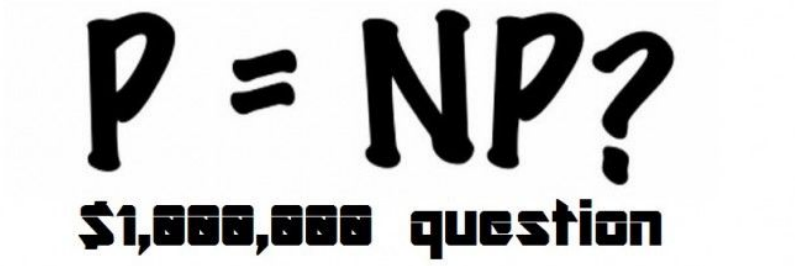
最后得到的矩阵包含各个最长路径的长度，此时将所有顶点按以该顶点为终点的最长路径长度从小到大排序即可得到拓扑次序。关于并行计算矩阵乘法的研究有很多，比如下图，时间复杂度为$O(\log^2 n)$。

Procedure $\text{multiply}(C, A, B)$:

- Base case: if $n = 1$, set $c_{11} \leftarrow a_{11} \times b_{11}$ (or multiply a small block matrix).
- Otherwise, allocate space for a new matrix T of shape $n \times n$, then:
  - Partition A into $A_{11}, A_{12}, A_{21}, A_{22}$.
  - Partition B into $B_{11}, B_{12}, B_{21}, B_{22}$.
  - Partition C into $C_{11}, C_{12}, C_{21}, C_{22}$.
  - Partition T into $T_{11}, T_{12}, T_{21}, T_{22}$.
  - Parallel execution:
    - *Fork* $\text{multiply}(C_{11}, A_{11}, B_{11})$.
    - *Fork* $\text{multiply}(C_{12}, A_{11}, B_{12})$.
    - *Fork* $\text{multiply}(C_{21}, A_{21}, B_{11})$.
    - *Fork* $\text{multiply}(C_{22}, A_{21}, B_{12})$.
    - *Fork* $\text{multiply}(T_{11}, A_{12}, B_{21})$.
    - *Fork* $\text{multiply}(T_{12}, A_{12}, B_{22})$.
    - *Fork* $\text{multiply}(T_{21}, A_{22}, B_{21})$.
    - *Fork* $\text{multiply}(T_{22}, A_{22}, B_{22})$.
  - *Join* (wait for parallel forks to complete).
  - $\text{add}(C, T)$.
  - Deallocate T.

（from Wikipedia）

计算机科学家和数学家针对并行计算的复杂度进行了大量分析,其中一个复杂度类（complexity class）是Nick's Class（这个词百度上没有中文 :(），由图灵奖得主Stephen Cook提出（这个巨佬在复杂度分析方面经常出现）。如果存在常数c和k，使一个问题能用O(n^k)个并行处理器在O(log^c n)时间内解决，则这个问题属于Nick's Class。上述矩阵乘法就属于这个类别。P=NP是计算机领域一大未解难题，NC=P也是一大未解难题。



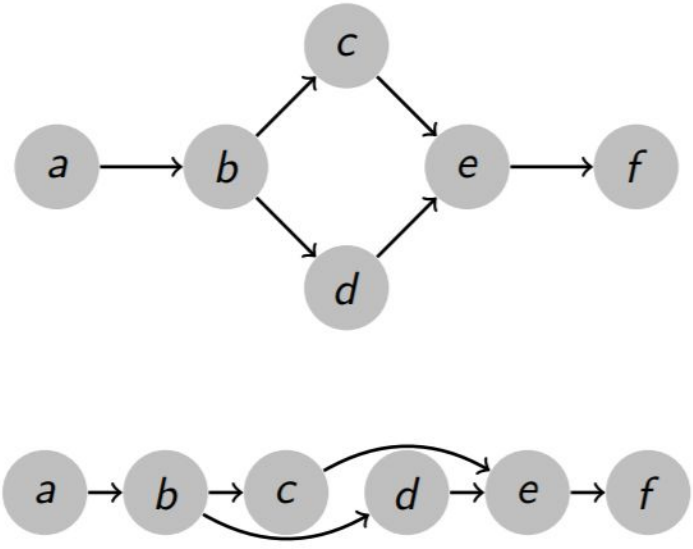
（from Medium）

言归正传，我们来接着说拓扑排序有哪些应用。拓扑排序可以用于最短路问题，有向无环图在拓扑排序之后，最短路可以由下图中的算法在线性时间复杂度（linear time complexity）内被计算出来。

- Let d be an array of the same length as V; this will hold the shortest-path distances from s. Set $d[s] = 0$, all other $d[u] = \infty$.
- Let p be an array of the same length as V, with all elements initialized to `nil`. Each $p[u]$ will hold the predecessor of u in the shortest path from s to u.
- Loop over the vertices u as ordered in V, starting from s:
  - For each vertex v directly following u (i.e., there exists an edge from u to v):
    - Let w be the weight of the edge from u to v.
    - Relax the edge: if $d[v] > d[u] + w$, set
      - $d[v] \leftarrow d[u] + w$,
      - $p[v] \leftarrow u$.

（from Wikipedia）

拓扑排序也和哈密顿回路（Hamiltonian path）有关。哈密顿回路的定义是一条仅经过图中每个顶点一次的路径。在拓扑排序后的顶点中，如果每两个相邻的顶点都有边连接，则这些边构成有向哈密顿回路。结合哈密顿回路的定义可以看出，如果有向图中存在哈密顿回路，则该图中拓扑次序是惟一的；如果有向图中不存在哈密顿回路，则该图中拓扑次序不是惟一的。也就是说，拓扑次序不唯一说明图不连通或者某一顶点连接多条边，下图就是一个例子。



（from South African Programming Olympiad）

相关材料：

https://algorithmtutor.com/Data-Structures/Graph/Graph-Representation-Adjacency-List-and-Matrix/

https://en.wikipedia.org/wiki/Topological_sorting

https://en.wikipedia.org/wiki/Min-plus_matrix_multiplication

https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

https://en.wikipedia.org/wiki/NC_(complexity)

https://en.wikipedia.org/wiki/Hamiltonian_path

https://saco-evaluator.org.za/presentations/camp2_2015/topsort.pdf

https://link.springer.com/article/10.1007/BF01385654

https://www.sciencedirect.com/science/article/pii/S0019995885800413