qinzhaokun
码龄8年
27
原创
1346
积分
TA的主页 私信 关注

拓扑排序的两种实现：Kahn算法和dfs算法

转载 qinzhaokun 2015-09-18 08:20:28 13628 收藏 8 展开

本文将从以下几个方面介绍拓扑排序：

- 拓扑排序的定义和前置条件
- 和离散数学中偏序/全序概念的联系
- 典型实现算法
    - *Kahn*算法
    - 基于*DFS*的算法
- 解的唯一性问题
- 实际例子

取材自以下材料：

http://en.wikipedia.org/wiki/Topological_sorting
http://en.wikipedia.org/wiki/Hamiltonian_path

定义和前置条件：

定义：将有向图中的顶点以线性方式进行排序。即对于任何连接自顶点*u*到顶点*v*的有向边*uv*，在最后的排序结果中，顶点*u*总是在顶点*v*的前面。

如果这个概念还略显抽象的话，那么不妨考虑一个非常非常经典的例子——选课。我想任何看过数据结构相关书籍的同学都知道它吧。假设我非常想学习一门机器学习的课程，但是在修这么课程之前，我们必须要学习一些基础课程，比如计算机科学概论，*C*语言程序设计，数据结构，算法等等。那么这个制定选修课程顺序的过程，实际上就是一个拓扑排序的过程，每门课程相当于有向图中的一个顶点，而连接顶点之间的有向边就是课程学习的先后关系。只不过这个过程不是那么复杂，从而很自然的在我们的大脑中完成了。将这个过程以算法的形式描述出来的结果，就是拓扑

那么是不是所有的有向图都能够被拓扑排序呢？显然不是。继续考虑上面的例子，如果告诉你在选修计算机科学概论这门课之前需要你先学习机器学习，你是不是会被弄糊涂？在这种情况下，就无法进行拓扑排序，因为它中间存在互相依赖的关系，从而无法确定谁先谁后。在有向图中，这种情况被描述为存在环路。因此，一个有向图能被拓扑排序的充要条件就是它是一个有向无环图*(DAG：Directed Acyclic Graph)*。

偏序/全序关系：

偏序和全序实际上是离散数学中的概念。
这里不打算说太多形式化的定义，形式化的定义教科书上或者上面给的链接中就说的很详细。

还是以上面选课的例子来描述这两个概念。假设我们在学习完了算法这门课后，可以选修机器学习或者计算机图形学。这个或者表示，学习机器学习和计算机图形学这两门课之间没有特定的先后顺序。因此，在我们所有可以选择的课程中，任意两门课程之间的关系要么是确定的(即拥有先后关系)，要么是不确定的(即没有先后关系)，绝对不存在互相矛盾的关系(即环路)。以上就是偏序的意义，抽象而言，有向图中两个顶点之间不存在环路，至于连通与否，是无所谓的。所以，有向无环图必然是满足偏序关系的。

理解了偏序的概念，那么全序就好办了。所谓全序，就是在偏序的基础之上，有向无环图中的任意一对顶点还需要有明确的关系(反映在图中，就是单向连通的关系，注意不能双向连通，那就成环了)。可见，全序就是偏序的一种特殊情况。回到我们的选课例子中，如果机器学习需要在学习

最新文章
蚂蚁金服旗下网商银行招聘了
匈牙利算法
SQL体系结构
MYSQL优化20条经验
MYSQL 索引
分类专栏

归档

2018

3月 1篇

2016

6月 1篇 | 5月 4篇 | 4月 1篇 | 3月 6篇

2015

12月 2篇 | 11月 3篇 | 10月 6篇 | 9月 9篇

热门文章

spark安装与使用（入门） 13804

拓扑排序的两种实现：Kahn算法和dfs算法 13614

Linux下安装Hadoop 6576

logstash安装和使用 4244

Kafka原理及应用 3753

最新评论

拓扑排序的两种实现：Kahn算法和...
hanguangchuan：谢谢作者，写得真棒

拓扑排序的两种实现：Kahn算法和...
hanguangchuan：“只要当前顶点还存在边指向其它任何顶点，它就会递归调用dfs方法，而不会 ...

拓扑排序的两种实现：Kahn算法和...
qq_28959087：写的真棒

拓扑排序的两种实现：Kahn算法和...
weixin_42419701：很受用，谢谢

算法学习--动态规划
weixin_43584808：这两种方法都能存储子问题解决方案。在第一个版本中，记忆化存储只在查扌 ...

了计算机图形学之后才能学习(可能学的是图形学领域相关的机器学习算法.....)，那么它们之间也就存在了确定的先后顺序，原本的偏序关系就变成了全序关系。

实际上，很多地方都存在偏序和全序的概念。
比如对若干互不相等的整数进行排序，最后总是能够得到唯一的排序结果(从小到大，下同)。这个结论应该不会有人表示疑问吧:)但是如果我们以偏序/全序的角度来考虑一下这个再自然不过的问题，可能就会有别的体会了。

那么如何用偏序/全序来解释排序结果的唯一性呢?
我们知道不同整数之间的大小关系是确定的，即1总是小于4的，不会有人说1大于或者等于4吧。这就是说，这个序列是满足全序关系的。而对于拥有全序关系的结构(如拥有不同整数的数组)，在其线性化(排序)之后的结果必然是唯一的。对于排序的算法，我们评价指标之一是看该排序算法是否稳定，即值相同的元素的排序结果是否和出现的顺序一致。比如，我们说快速排序是不稳定的，这是因为最后的快排结果中相同元素的出现顺序和排序前不一致了。如果用偏序的概念可以这样解释这一现象：相同值的元素之间的关系是无法确定的。因此它们在最终的结果中的出现顺序可以是任意的。而对于诸如插入排序这种稳定性排序，它们对于值相同的元素，还有一个潜在的比较方式，即比较它们的出现顺序，出现靠前的元素大于出现后出现的元素。因此通过这一潜在的比较，将偏序关系转换为了全序关系，从而保证了结果的唯一性。

拓展到拓扑排序中，结果具有唯一性的条件也是其所有顶点之间都具有全序关系。如果没有这一层全序关系，那么拓扑排序的结果也就不是唯一的了。在后面会谈到，如果拓扑排序的结果唯一，那么该拓扑排序的结果同时也代表了一条哈密顿路径。

典型实现算法:

*Kahn*算法:

摘一段维基百科上关于*Kahn*算法的伪码描述:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    foreach node m with an edge e from nto m do
        remove edge e from thegraph
        ifm has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least onecycle)
else
    return L (a topologically sortedorder)
```

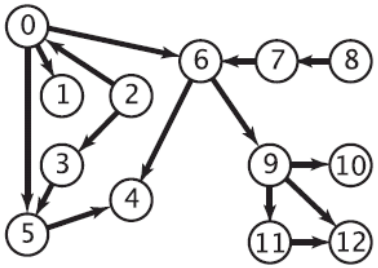
不难看出该算法的实现十分直观，关键在于需要维护一个入度为0的顶点的集合:
每次从该集合中取出(没有特殊的取出规则，随机取出也行，使用队列/栈也行，下同)一个顶点，将该顶点放入保存结果的*List*中。
紧接着循环遍历由该顶点引出的所有边，从图中移除这条边，同时获取该边的另外一个顶点，如果该顶点的入度在减去本条边之后为0，那么也将这个顶点放到入度为0的集合中。然后继续从集合中取出一个顶点.....

当集合为空之后，检查图中是否还存在任何边，如果存在的话，说明图中至少存在一条环路。不存在的话则返回结果*List*，此*List*中的顺序就是对图进行拓扑排序的结果。

实现代码:

```java
public class KahnTopological
{
    private List<Integer> result;  // 用来存储结果集
    private Queue<Integer> setOfZeroIndegree;  // 用来存储入度为0的顶点
    private int[] indegrees;  // 记录每个顶点当前的入度
    private int edges;
    private Digraph di;

    public KahnTopological(Digraph di)
    {
        this.di = di;
        this.edges = di.getE();
        this.indegrees = new int[di.getV()];
        this.result = new ArrayList<Integer>();
        this.setOfZeroIndegree = new LinkedList<Integer>();

        // 对入度为0的集合进行初始化
        Iterable<Integer>[] adjs = di.getAdj();
        for(int i = 0; i < adjs.length; i++)
        {
            // 对每一条边 v -> w
            for(int w : adjs[i])
            {
                indegrees[w]++;
            }
        }

        for(int i = 0; i < indegrees.length; i++)
        {
            if(0 == indegrees[i])
            {
                setOfZeroIndegree.enqueue(i);
            }
        }
        process();
    }

    private void process()
    {
        while(!setOfZeroIndegree.isEmpty())
        {
            int v = setOfZeroIndegree.dequeue();

            // 将当前顶点添加到结果集中
            result.add(v);

            // 遍历由v引出的所有边
            for(int w : di.adj(v))
            {
                // 将该边从图中移除，通过减少边的数量来表示
                edges--;
                if(0 == --indegrees[w])  // 如果入度为0，那么加入入度为0的集合
                {
                    setOfZeroIndegree.enqueue(w);
                }
            }
        }
        // 如果此时图中还存在边，那么说明图中含有环路
        if(0 != edges)
        {
            throw new IllegalArgumentException("Has Cycle !");
```

```
        }
    }

    public Iterable<Integer> getResult()
    {
        return result;
    }
}
```



对上图进行拓扑排序的结果：
2->8->0->3->7->1->5->6->9->4->11->10->12

复杂度分析：
初始化入度为*0*的集合需要遍历整张图，检查每个节点和每条边，因此复杂度为*O(E+V);*
然后对该集合进行操作，又需要遍历整张图中的，每条边，复杂度也为*O(E+V);*
因此*Kahn*算法的复杂度即为***O(E+V)***。

基于*DFS*的拓扑排序：

除了使用上面直观的*Kahn*算法之外，还能够借助深度优先遍历来实现拓扑排序。这个时候需要使用到栈结构来记录拓扑排序的结果。
同样摘录一段维基百科上的伪码：

```
L ← Empty list that will contain the sorted nodes
S ← Set of all nodes with no outgoing edges
for each node n in S do
    visit(n)
function visit(node n)
    if n has not been visited yet then
        mark n as visited
        for each node m with an edgefrom m to ndo
            visit(m)
        add n to L
```

*DFS*的实现更加简单直观，使用递归实现。利用*DFS*实现拓扑排序，实际上只需要添加一行代码，即上面伪码中的最后一行：*add n to L*。
需要注意的是，将顶点添加到结果*List*中的时机是在*visit*方法即将退出之时。
这个算法的实现非常简单，但是要理解的话就相对复杂一点。
关键在于为什么在*visit*方法的最后将该顶点添加到一个集合中，就能保证这个集合就是拓扑排序的结果呢?
因为添加顶点到集合中的时机是在*dfs*方法即将退出之时，而*dfs*方法本身是个递归方法，只要当前顶点还存在边指向其它任何顶点，它就会递归调

用*dfs*方法，而不会退出。因此，退出*dfs*方法，意味着当前顶点没有指向其它顶点的边了，即当前顶点是一条路径上的最后一个顶点。

下面简单证明一下它的正确性：

考虑任意的边*v->w*，当调用*dfs(v)*的时候，有如下三种情况：

1. *dfs(w)*还没有被调用，即*w*还没有被*mark*，此时会调用*dfs(w)*，然后当*dfs(w)*返回之后，*dfs(v)*才会返回
2. *dfs(w)*已经被调用并返回了，即*w*已经被*mark*
3. ~~*dfs(w)*已经被调用但是在此时调用*dfs(v)*的时候还未返回~~

需要注意的是，以上第三种情况在拓扑排序的场景下是不可能发生的，因为如果情况*3*是合法的话，就表示存在一条由*w*到*v*的路径。而现在我们的前提条件是由*v*到*w*有一条边，这就导致我们的图中存在环路，从而该图就不是一个有向无环图*(DAG)*，而我们已经知道，非有向无环图是不能被拓扑排序的。

那么考虑前两种情况，无论是情况*1*还是情况*2*，*w*都会先于*v*被添加到结果列表中。所以边*v->w*总是由结果集中后出现的顶点指向先出现的顶点。为了让结果更自然一些，可以使用栈来作为存储最终结果的数据结构，从而能够保证边*v->w*总是由结果集中先出现的顶点指向后出现的顶点。

实现代码：

```java
public class DirectedDepthFirstOrder
{
    // visited数组，DFS实现需要用到
    private boolean[] visited;
    // 使用栈来保存最后的结果
    private Stack<Integer> reversePost;

    /**
     * Topological Sorting Constructor
     */
    public DirectedDepthFirstOrder(Digraph di, boolean detectCycle)
    {
        // 这里的DirectedDepthFirstCycleDetection是一个用于检测有向图中是否存在环路的类
        DirectedDepthFirstCycleDetection detect = new DirectedDepthFirstCycleDetection(
                di);

        if (detectCycle && detect.hasCycle())
            throw new IllegalArgumentException("Has cycle");

        this.visited = new boolean[di.getV()];
        this.reversePost = new Stack<Integer>();

        for (int i = 0; i < di.getV(); i++)
        {
            if (!visited[i])
            {
                dfs(di, i);
            }
        }
    }

    private void dfs(Digraph di, int v)
    {
        visited[v] = true;

        for (int w : di.adj(v))
        {
            if (!visited[w])
            {
```

```
            dfs(di, w);
        }
    }

    // 在即将退出dfs方法的时候，将当前顶点添加到结果集中
    reversePost.push(v);
  }

  public Iterable<Integer> getReversePost()
  {
    return reversePost;
  }
}
```

复杂度分析：

复杂度同*DFS*一致，即*O(E+V)*。具体而言，首先需要保证图是有向无环图，判断图是*DAG*可以使用基于*DFS*的算法，复杂度为*O(E+V)*，而后面的拓扑排序也是依赖于*DFS*，复杂度为*O(E+V)*

还是对上文中的那张有向图进行拓扑排序，只不过这次使用的是基于*DFS*的算法，结果是：

8->7->2->3->0->6->9->10->11->12->1->5->4

两种实现算法的总结：

这两种算法分别使用链表和栈来表示结果集。

对于基于*DFS*的算法，加入结果集的条件是：顶点的出度为*0*。这个条件和*Kahn*算法中入度为*0*的顶点集合似乎有着异曲同工之妙，这两种算法的思想犹如一枚硬币的两面，看似矛盾，实则不然。一个是从入度的角度来构造结果集，另一个则是从出度的角度来构造。

实现上的一些不同之处：

*Kahn*算法不需要检测图为*DAG*，如果图为*DAG*，那么在出度为*0*的集合为空之后，图中还存在没有被移除的边，这就说明了图中存在环路。而基于*DFS*的算法需要首先确定图为*DAG*，当然也能够做出适当调整，让环路的检测和拓扑排序同时进行，毕竟环路检测也能够在*DFS*的基础上进行。二者的复杂度均为*O(V+E)*。

环路检测和拓扑排序同时进行的实现：

```java
public class DirectedDepthFirstTopoWithCircleDetection
{
    private boolean[] visited;
    // 用于记录dfs方法的调用栈，用于环路检测
    private boolean[] onStack;
    // 用于当环路存在时构造之
    private int[] edgeTo;
    private Stack<Integer> reversePost;
    private Stack<Integer> cycle;

    /**
     * Topological Sorting Constructor
     */
    public DirectedDepthFirstTopoWithCircleDetection(Digraph di)
    {
        this.visited = new boolean[di.getV()];
        this.onStack = new boolean[di.getV()];
        this.edgeTo = new int[di.getV()];
        this.reversePost = new Stack<Integer>();
```

```
20.
21.         for (int i = 0; i < di.getV(); i++)
22.         {
23.             if (!visited[i])
24.             {
25.                 dfs(di, i);
26.             }
27.         }
28.     }
29.
30.     private void dfs(Digraph di, int v)
31.     {
32.         visited[v] = true;
33.         // 在调用dfs方法时，将当前顶点记录到调用栈中
34.         onStack[v] = true;
35.
36.         for (int w : di.adj(v))
37.         {
38.             if(hasCycle())
39.             {
40.                 return;
41.             }
42.             if (!visited[w])
43.             {
44.                 edgeTo[w] = v;
45.                 dfs(di, w);
46.             }
47.             else if(onStack[w])
48.             {
49.                 // 当w已经被访问，同时w也存在于调用栈中时，即存在环路
50.                 cycle = new Stack<Integer>();
51.                 cycle.push(w);
52.                 for(int start = v; start != w; start = edgeTo[start])
53.                 {
54.                     cycle.push(v);
55.                 }
56.                 cycle.push(w);
57.             }
58.         }
59.
60.         // 在即将退出dfs方法时，将顶点添加到拓扑排序结果集中，同时从调用栈中退出
61.         reversePost.push(v);
62.         onStack[v] = false;
63.     }
64.
65.     private boolean hasCycle()
66.     {
67.         return (null != cycle);
68.     }
69.
70.     public Iterable<Integer> getReversePost()
71.     {
72.         if(!hasCycle())
73.         {
74.             return reversePost;
75.         }
76.         else
77.         {
78.             throw new IllegalArgumentException("Has Cycle: " + getCycle());
79.         }
80.     }
81.
82.     public Iterable<Integer> getCycle()
83.     {
```

```
        return cycle;
    }
}
```

拓扑排序解的唯一性：

哈密顿路径：

哈密顿路径是指一条能够对图中所有顶点正好访问一次的路径。本文中只会解释一些哈密顿路径和拓扑排序的关系，至于哈密顿路径的具体定义以及应用，可以参见本文开篇给出的链接。

前面说过，当一个*DAG*中的任何两个顶点之间都存在可以确定的先后关系时，对该*DAG*进行拓扑排序的解是唯一的。这是因为它们形成了全序的关系，而对存在全序关系的结构进行线性化之后的结果必然是唯一的(比如对一批整数使用稳定的排序算法进行排序的结果必然就是唯一的)。

需要注意的是，非*DAG*也是能够含有哈密顿路径的，为了利用拓扑排序来实现判断，所以这里讨论的主要是判断*DAG*中是否含有哈密顿路径的算法，因此下文中的图指代的都是*DAG*。

那么知道了哈密顿路径和拓扑排序的关系，我们如何快速检测一张图是否存在哈密顿路径呢？
根据前面的讨论，是否存在哈密顿路径的关键，就是确定图中的顶点是否存在全序的关系，而全序的关键，就是任意一对顶点之间都是能够确定先后关系的。因此，我们能够设计一个算法，用来遍历顶点集中的每一对顶点，然后检查它们之间是否存在先后关系，如果所有的顶点对有先后关系，那么该图的顶点集就存在全序关系，即图中存在哈密顿路径。

但是很显然，这样的算法十分低效。对于大规模的顶点集，是无法应用这种解决方案的。通常一个低效的解决办法，十有八九是因为没有抓住现有问题的一些特征而导致的。因此我们回过头来再看看这个问题，有什么特征使我们没有利用的。还是举对整数进行排序的例子：
比如现在有*3*，*2*，*1*三个整数，我们要对它们进行排序，按照之前的思想，我们分别对*(1,2)*，*(2,3)*，*(1,3)*进行比较，这样需要三次比较，但是我们很清楚，*1*和*3*的那次比较实际上是多余的。我们为什么知道这次比较是多余的呢？我认为，是我们下意识的利用了整数比较满足传递性的这一规则。但是计算机是无法下意识的使用传递性的，因此只能通过其它的方式来告诉计算机，有一些比较是不必要的。所以，也就有了相对插入排序，选择排序更加高效的排序算法，比如归并排序，快速排序等，将n^2的算法加速到了*nlogn*。或者是利用了问题的特点，采取了更加独特的解决方案，比如基数排序等。

扯远了一点，回到正题。现在我们没有利用到的就是全序关系中传递性这一规则。如何利用它呢，最简单的想法往往就是最实用的，我们还是选择排序，排序后对每对相邻元素进行检测不就间接利用了传递性这一规则嘛？所以，我们先使用拓扑排序对图中的顶点进行排序。排序后，对每对相邻顶点进行检测，看看是否存在先后关系，如果每对相邻顶点都存在着一致的先后关系(在有向图中，这种先后关系以有向边的形式体现，即查看相邻顶点对之间是否存在有向边)。那么就可以确定该图中存在哈密顿路径了，反之则不存在。

实现代码：

```java
/**
 * Hamilton Path Detection for DAG
 */
public class DAGHamiltonPath
{
    private boolean hamiltonPathPresent;
    private Digraph di;
    private KahnTopological kts;

    // 这里使用Kahn算法进行拓扑排序
    public DAGHamiltonPath(Digraph di, KahnTopological kts)
```

```
    {
        this.di = di;
        this.kts = kts;

        process();
    }

    private void process()
    {
        Integer[] topoResult = kts.getResultAsArray();

        // 依次检查每一对相邻顶点，如果二者之间没有路径，则不存在哈密顿路径
        for(int i = 0; i < topoResult.length - 1; i++)
        {
            if(!hasPath(topoResult[i], topoResult[i + 1]))
            {
                hamiltonPathPresent = false;
                return;
            }
        }
        hamiltonPathPresent = true;
    }

    private boolean hasPath(int start, int end)
    {
        for(int w : di.adj(start))
        {
            if(w == end)
            {
                return true;
            }
        }
        return false;
    }

    public boolean hasHamiltonPath()
    {
        return hamiltonPathPresent;
    }
}
```

实际例子：

*TestNG*中循环依赖的检测：

http://blog.csdn.net/dm_vincent/article/details/7631916

qinzhaokun
原创文章 27　获赞 70　访问量 10万+

jet_qi：写的真棒 1年前

weixin_42419701：很受用，谢谢 1年前

NairoJ：写的太好了，十分感谢 1年前

胆识与智慧：重要的是算法，不是语言 1年前

梦不灭：代码能用C/C++实现吗 2年前

【算法设计与数据结构】拓扑排序算法的实现——Kahn算法及基于dfs的算法 jiange_zh的博客 3988
拓扑排序的定义和原理等我不再赘述，各种教材和网络上都有详细解释，今天我主要谈一谈两

Kahn算法-拓扑排序 I'm noob.... 1388
/*时间：2015/11/22内容：拓扑排序的实现（可用于判断图中是否有环路）概述：基于Kahn算法

史上最全的IDEA快捷键总结 扬帆向海的博客 15万+
现在Idea成了主流开发工具，这篇博客对其使用的快捷键做了总结，希望对大家的开发工作有

拓扑排序(Kahn算法和基于DFS求解法) yo_bc的博客 2129
拓扑排序是对有向无环图(DAG)进行排序，从而找到一个序列。该序列满足对于任意一对不同的

拓扑排序模板（Kahn算法和DFS实现） baodream的博客 620
拓扑排序思想：每次取出入度为0的顶点删掉，并删掉和该点有关的边，需要维护一个入度

10 个最难回答的 Java 问题 aaa13268的博客 5万+
1.为什么等待和通知是在 Object 类而不是 Thread 中声明的？一个棘手的 Java 问题，如果

“程序员数学不行，干啥都不行！”高级开发：90%都是瞎努力！ Python大本营的博客 2万+
点击上方“Python大本营”，选择“置顶公众号”Python大本营 IT人的职业提升平台之前有很多读

为什么猝死的都是程序员，基本上不见产品经理猝死呢? 曹银飞的专栏 25万+
相信大家时不时听到程序员猝死的消息，但是基本上听不到产品经理猝死的消息，这是为什么

拓扑排序【kahn算法及dfs的拓扑排序】 fl_334正经的码棚 182
家谱树有个人的家族很大，辈分关系很混乱，请你帮整理一下这种关系。给出每个人的孩子的

拓扑排序【kahn算法及dfs的拓扑排序】 - fl_334正经的..._CSDN博客 11-2

拓扑排序-Kahn算法 - 哇-WA 的博客 - CSDN博客 10-16

500行代码，教你用python写个微信飞机大战 Python专栏 8万+
这几天在重温微信小游戏的飞机大战，玩着玩着就在思考人生了，这飞机大战怎么就可以做的

拓扑排序(topological sorting)时间复杂度 - This is b..._CSDN博客 11-14

拓扑排序——用C++中STL实现 - 疯狂的指针的博客 - CSDN博客 11-21

毕业5年，我问遍了身边的大佬，总结了他们的学习方法 敖丙 27万+
我问了身边10个大佬，总结了他们的学习方法，原来成功都是有迹可循的。

Synchronized关键字深析（小白慎入，深入jvm源码，两万字长文） Java新生代 2万+
从jvm层面解析synchronized，看完绝对可以超越绝大数人

例题6-15 拓扑排序 DFS实现和STL实现 uva10305 - CSDN博客 7-16

Uva 10305 Ordering Tasks (用dfs 实现拓扑排序) - PK..._CSDN博客 11-18

dfs拓扑排序原理详解--------还不明白请来砍我 qq_34384524的博客 2246
我们不创造算法，我们只是算法的搬运工个人笔记之核心点：@1：dfs在实现时不仅访问了顶

20道你必须要背会的微服务面试题，面试一定会被问到 扬帆向海的博客 8万+
这篇博客总结了面试中最常见的微服务面试题，相信对你有所帮助。

Ordering Tasks UVA - 10305---拓扑排序(dfs) - Nicola..._CSDN博客 1-4

拓扑排序(算法 非可执行程序) - weixin_34270865的博客 - CSDN博客 1-10

大学四年，因为知道这些开发工具，我成为别人眼中的大神 敖丙 6万+
亲测全部都很好用，自己开发都离不开的软件，如果你是学生可以看看，提前熟悉起来。...

拓扑排序dfs - qq_40688707的博客 - CSDN博客 10-19

拓扑排序（topological sorting） 一只码畜 1074
AOV网络　　在有向图中，用顶点表示活动，用有向边表示活动Vi必须先于活动Vj进行。这种

分别使用Kahn和DFS实现拓扑排序 记录每一个小阶段的学习心得，持之以恒！ 671
1，先了解什么是偏序？偏序就是图中存在无先后顺序的顶点对。全序即找不到这样的顶点

拓扑排序的原理及其实现 不忘初心，好好沉淀 19万+
本文将从以下几个方面介绍拓扑排序：拓扑排序的定义和前置条件和离散数学中偏序/全序概念

B 站上有哪些很好的学习资源? 九章算法的博客 19万+
哇说起B站，在小九眼里就是宝藏般的存在，放年假宅在家时一天刷6、7个小时不在话下，更别

将一个接口响应时间从2s优化到 200ms以内的一个案例 明明如月的专栏 1万+
一、背景在开发联调阶段发现一个接口的响应时间特别长，经常超时，囧...本文讲讲是如何定位

《MySQL 性能优化》之理解 MySQL 体系结构 Tony.Dong的专栏 5万+
本文介绍 MySQL 的体系结构，包括物理结构、逻辑结构以及插件式存储引擎。

花了20分钟，给女朋友们写了一个web版群聊程序 34万+
参考博客[1]https://www.byteslounge.com/tutorials/java-ee-html5-websocket-example

Uva 10305 Ordering Tasks （用dfs 实现拓扑排序） PK__PK的博客 207
原题：点击右边的-->原题题意：给你一个n，表示有n个任务，给你m组两个数，表示两个