迭代求解的利器

--------SPFA 算法的优化与应用

广东中山纪念中学 姜碧野

【摘要】

SPFA 算法,全称 **Shortest Path Faster Algorithm**,是 Bellman-Ford 算法的改进版。该算法以三角不等式为基础，实现时借助队列或栈不断进行迭代以求得最优解。具有效率高、实现简洁、扩展性强等优点。三角不等式的普适性及其类似搜索的实现方式，使其应用并不只局限于图论中的最短路径，更可以在动态规划、迭代法解方程中发挥出巨大的作用，解决一些非常规问题；还可根据具体问题进行各种各样的优化。本文将对其进行全面的分析、测试和深入的讨论。

【关键词】 迭代 SPFA 深度优先搜索 最优性 动态规划 状态转移 方程

【目录】

SPFA算法简介 3
1.1 SPFA算法的基本实现 3
1.2 活学活用：SPFA的深度优先实现及其优化 5
1.2.1:基于Dfs的SPFA的基本原理 5
1.2.2:基于Dfs的SPFA的相关优化 7
1.3 SPFA算法实际效果测试及比较 8
*1.4 Johnson算法介绍 12
2.SPFA算法在实际应用中的优化 13
2.1 如何使用SPFA快速查找负(正)环 13
2.2 注意对无用状态的裁剪 17
3. SPFA算法的应用 19
3.1 差分约束系统 19
3.2 在一类状态转移阶段性不明显的动态规划中的应用 20
3.3 探讨SPFA在解方程中的应用 23
3.4 一类状态转移存在“后效性”的动态规划中的应用 28
5 附录 33
5.1 例题原题 33
5.2 源程序&例题测试数据 37
6.参考文献 37
7.鸣谢 37

【正文】

SPFA 算法简介

1.1 SPFA 算法的基本实现

下面，先介绍一下 SPFA 和 Bellman-Ford 算法的原理和适用条件。

首先一个很重要的性质便是三角不等式。

设 G=(V,E)为一带权有向图，其权函数 w:E→R 为边到实型权值的映射，s 为源点，对于任意点 t,d(s,t)为 s 到 t 的最短路。

则对于所有边(u,v)∈E 有 d(s,v)<=d(s,u)+w(u,v)。

令 d(s,s)=0，这一不等式为 d(s,t)是 s 到 t 的最短路的充要条件。

这一性质很容易理解。我们也很容易将其推广。

设 G=(V,E)为一带权有向图，d(u)为状态(顶点)u 的最优值，权函数 w:E→自定义集合。则对于所有边(u,v)∈E 有

d(u)+w(u,v)不优于 d(v). 注:这里的“+”可以是延伸为任意形式的运算表达式。

更进一步，我们并不一定将不等式限定在“最优性”这一框架中，而可根据具体题目要求制定自己的判断。推广后的三角不等式将不再拘束于最短路问题，有着更加广泛的适用空间。只要我们根据题目构造出状态间的权函数和优劣判断标准，在大部分情况下我们都可以使用 SPFA 求解。在下文中将看到相关的应用。

有了三角不等式后，SPFA 也即 Bellman-Ford 的核心算法也就登场了。也就是著名的松弛操作。

首先令 f(u)为节点的 u 的当前最优最短路，并赋初值 f(s)=0,f(u)=+∞(u≠s)
然后对每条不满足三角不等式的边，我们都把其目标节点重新赋值。

即：对于(u,v)∈E

```
Relax(u,v){
    If f(v)>f(u)+w(u,v)
        f(v)=f(u)+w(u,v);
```

}

松弛操作的本质正是：不断寻找当前状态与目标状态间的矛盾并调整，直到找不到矛盾时即达到最优状态。

由松弛操作，我们又可以得出一些有用的推论：

上界性质&收敛性质:

在算法过程中，对于所有 u,都有 f(u)>=d(s,u)。因此一旦 f(u)达到 d(s,u), f(u)将不会再改变。这一点很容易理解，因为由于归纳法可知，只要原来满足 f(u)>=d(s,u),那么执行松弛操作后，显然有：

f(v)=f(u)+w(u,v)

>=d(s,u)+w(u,v)

>=d(s,v)

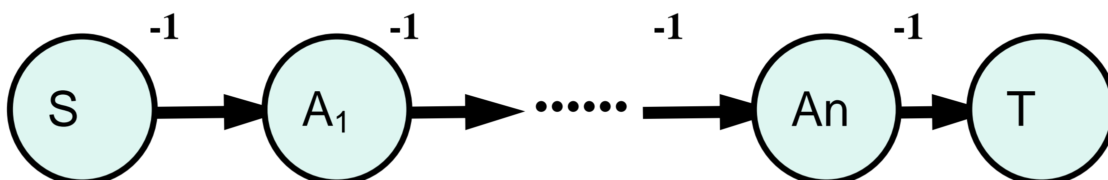
而在没有负环的情况下，每个 f(u)肯定是单调不增的，这也证明了该算法肯定会结束的。更进一步，如果我们采用 Bellman-Ford 的计算方式：

```
For i=1 to n-1
    For (u,v) ∈E
        Relax(u,v);
```

便可以在 N*M 的时间中算出最短路径，因为最短路径的边数最多为 N-1,而由数学归纳法知：当外循环为第 I 次时，我们算出了长度为 I 的最短路，而经过第 I+1 次的循环，必然可以得出长度为 I+1 的最短路。

不存在负环时，最短路经过的边数显然不超过 N-1，这也告诉我们第 N 次迭代如果发现可以继续更新则必然存在负环。

而事实上，Bellman-Ford 的实际效率是很低的，因为有很多不必要的操作。考虑图为一条链时的情况：



如果直接使用 Bellman-Ford，则每次迭代实际上只能更新一个节点，其他的循环都是无效的。因此可以发现只有在上一次迭代中被更新的节点才会对当前的

迭代有用，SPFA 正是利用了这一点，采用了队列进行优化。其核心是只保存当前有用状态，每次将被更新的节点放入队列中等待扩展，而未被更新的则不予考虑。而由于任意时刻队列中最多只有 N 个元素，所以使用循环队列。具体算法伪代码如下：

```
Program SPFA{
    Init_graph
    Push_Queue(s);
    While head≠tail{
        x=Pop_Queue;
        For(x,v) ∈E
            If f(v)>f(x)+w(x,v){
                f(v)=f(x)+w(x,v)
                if v not in queue
                    Push_Queue (v);
                    If (v has been in Queue for N times){
                        Contain a negative cycle.
                        Halt;
                    }
                }
    }
    Out_data;
}
```

在一般情况，SPFA的效率很高，可以证明[1]SPFA的期望复杂度为O(KM)，K<2。但由于证明还不够严谨且适用性不广(存在针对性数据)，在此不再赘述。但将会在随后的测试中用实际数据来验证SPFA的高效性。

1.2 活学活用：SPFA 的深度优先实现及其优化

1.2.1:基于 Dfs 的 SPFA 的基本原理

在上面的介绍的算法中，我们用一个循环队列来存储需要继续迭代的节点，实现时使用类似广度优先搜索的方式。那么我们是否可以使用搜索的另一利器：深度优先呢？

回忆之前的算法，由于采用广度优先的思想，每当我们扩展出一个新的节点，

[1] 证明见 2006 余远铭论文《最短路算法及其应用》

总是把它放到队列的末尾，其缺点是中断了迭代的连续性。而实际上如果采用深度优先的思想，我们可以直接从这个新节点继续往下扩展。于是算法的实现方式可以改成不断从新节点往下递归进行求解。

而对于负环的判断则显得更为简单，因为假如存在负环a_1->a_2->….a_k->a_1，那么算法运行时，会从某个点a_1开始Dfs，最后又回到了这个点。所以只需用一个辅助数组记录当前节点是否在递归栈中便可及时检测出负环。

伪代码如下：

```
Void SPFA(Node) {
        Instack[Node]=true;
        For (Node,v)  ∈E
            If dis[Node]+edge(Node,v)<dis[v] then {
                dis[v]=dis[Node]+edge(Node,v);
                If not Instack[v] then
                    SPFA(v);
                Else{
                    Contain an negative cycle.
                    Halt;
                }
        }
        Instack [Node] =false;
}
```

两种算法的原理相同，相对 Bellman-Ford 的优化也是殊途同归的。其精髓都在于只保存有用的状态。

那么，SPFA 的这种实现效率如何呢？在随后的初步尝试中，笔者发现，在一些拓扑关系比较强的时候，广度会有很大的优势。Dfs 往往由于不断盲目地迭代而耗费了太多时间甚至远远超过时间的限制。

因此在下文中，笔者将结合深度优先搜索的特点，尝试对其进行优化。

1.2.2:基于 Dfs 的 SPFA 的相关优化

Dfs 的实现方式虽然暂时无法取得好的效果，但我们并不应该就此放弃，Dfs 相对 Bfs 灵活的架构必能给予我们广阔的优化空间。

首先，一个很自然的想法便是改变搜索顺序，即把边按照一定的规则排序，在这里当然是把边权按优劣排序，这样便可以得到一个不错的初始解。加入了这个优化后，在某些数据上取得了不错的效果，但许多数据依然远远超过时间限制。

接着笔者联想到网络流使用贪心初始流进行优化的方法，因为当前解的优劣程度对之后的迭代有很大影响。而且 Dfs 的低效很大程度上是因为频繁地使用较次解去更新大片的元素。

由此我们可以考虑先在第一次 Dfs 时，对扩展节点采取一些限制以快速求得一个较优解，然后再进行第二次完整的 Dfs。笔者经过多次尝试，在针对随机数据上得出一种不错的贪心初始解方法。

```
Function SPFA_Init (Node): Boolean {
        For (Node,v) ∈E
            If dis [Node] +edge (Node, v) <dis[v] {
                Dis[v] =dis [Node] +edge (Node, v);
                SPFA_Init (v);
                Return true;
        }
        Return false;
}
Main {
        For Node∈V
            While SPFA_Init(Node)do;
        For Node∈V
            SPFA (Node)
}
```

即对于每个节点只更新一个边后便退出。这一算法对于随机图能取得不错的效果，因为往往只更新一次能快速将当前值快速地传递。加入这个优化后效率有了很大提高，但与 Bfs 还是有不少差距。

于是笔者转而分析 Bfs 与 Dfs 的不同，经过比较可以发现 Bfs 的优势主要体现在某个点出队前可能再次被更新而得到更优的解进行下一次迭代。而 Dfs 往往会用一个次解进行层次很深的迭代(一个次解会导致 O(N)级别的更新)。由此，可以联想到深度优先搜索的一个改进版本：迭代加深搜索，结合前面贪心初始解的原理，我们可以通过限制节点递归的深度并逐步放宽限制求解。

加入这个优化后，笔者欣喜地发现，在 Bfs 不擅长的网格形数据中，Dfs 所用时间仅为 Bfs 的 1/3，效率飞速提高，优势明显。而在一些其他随机数据中效果则有好有坏，总体还是比 Bfs 略逊一筹，但差距已不大。值得注意的是：对深度限制的不同，时间效率也有差异，笔者总结出了两种效果相对不错的限制方法：

1 依次设置深度上限为 1,2,4,8,16,32,64…..2^k….. maxlongint

2 依次设置深度上限为 20，30，40，50，60，80，100….maxlongint

鉴于 Dfs 在网格图上的高效，笔者即将其应用在 Spoj 的 skivall 一题中。该题的一个可行算法即为网格图上的费用流。使用 Dfs 后，效率得到了很大提升，仅用 3s 多就通过了全部数据。而且在 USACO Jan09 Travel 一题的数据中，DFS 相比 BFS 有大幅提高，甚至优于 dijikstra。(当然与数据的特性有关)

至此，对 Dfs 的优化也就告一段落了。但可以预见，Dfs 可行的优化远远不止如此，还有许多性质没有得到很好的利用。而且在上文的优化中，笔者并没有牺牲 SPFA 的简洁性(几乎不需要怎么修改原程序)，因此许多诸如加入优先队列(堆)等方法还有待研究。也希望上文的方法能激发读者的思路，继续探索新的优化方法,或是将这类优化思想应用在其他相关领域。

1.3 SPFA 算法实际效果测试及比较

本节我们将用实际数据对 SPFA 进行测试

测试平台：CPU: Genuine Intel T2300 1.66 GHz　　内存：512MB

测试器:伟栋清橙测试器 1.0

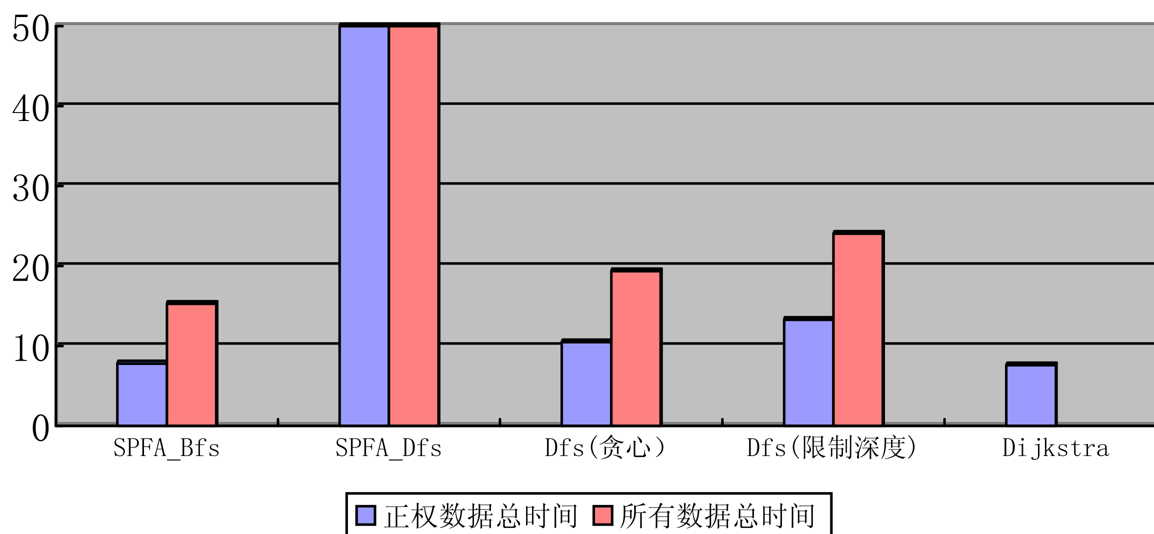
测试说明:

1．测试项目为单源到其他全部顶点的最短路。运行时间包含读入据时间。输出经过优化，可以忽略输出时间。

2．每组数据中，红色加粗的表示该算法在本组中最快。

先看随机数据：

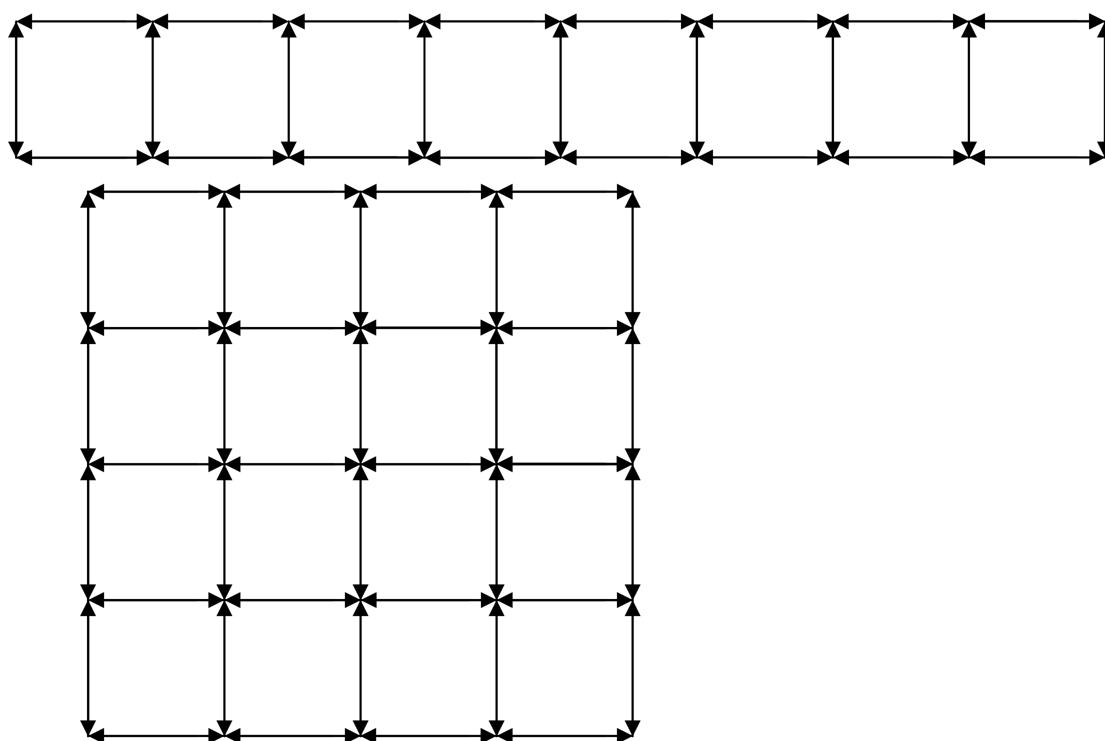
数据类型 \ 算法 时间	SPFA_Bfs	SPFA_Dfs (未优化)	SPFA_Dfs (贪心初始解)	SPFA_Dfs (限制深度)	Dijkstra	读入数据时间
负权 N=500 M=100000	**0. 13s**	48. 03s	**0. 13s**	**0. 13s**	Wrong Answer	0. 13s
负权 N=200000 M=500000	**2. 22s**	>50s	3. 14s	3. 38s	Wrong Answer	1. 45s
负权 N=1000000 M=2000000	**5. 12s**	>50s	5. 61s	7. 24s	Wrong Answer	3. 06s
正权 N=200000 M=500000	2. 43s	>50s	3. 48s	4. 43s	**1. 92s**	1. 24s
正权 N=1000000 M=2000000	**5. 39s**	>50s	6. 99s	8. 86s	5. 69s	2. 55s
正权数据总时间	7.82s	>100s	10.47s	13.29s	**7.61s**	3.79s
总时间	**15.29s**	>248.03s	19.35s	24.04s	N/A	8.43s



在上表中我们可以得出几个结论：

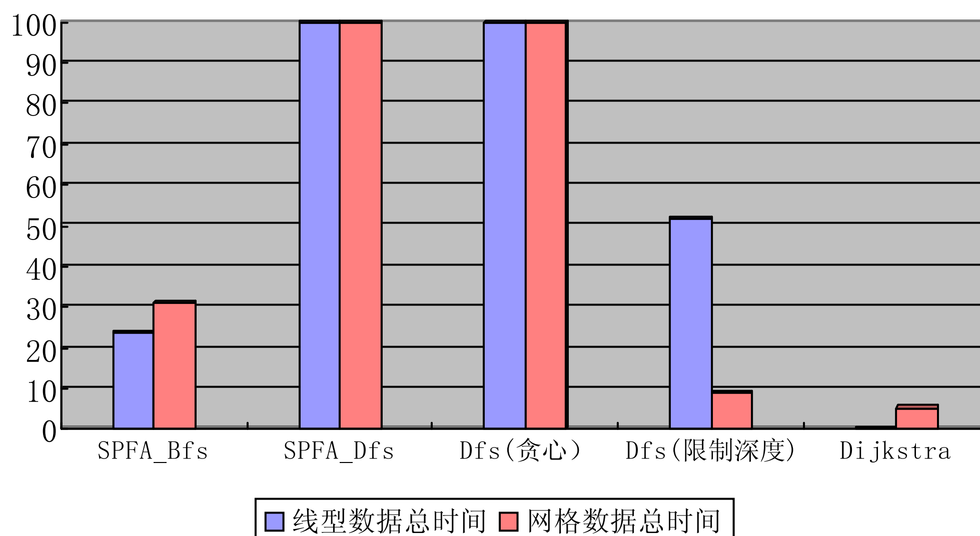
1. 在随机图中 Bfs 的效率很高，甚至在 N=1000000 的稀疏图中优于 Dijkstra
2. 一个诡异的现象是程序对于笔者的正权数据居然比负权要慢-_- 经研究发现是由于生成负权数据时，笔者为了防止出现负环，刻意增大了边权的值。因而往往许多边变成无用边.....而实际的负权边却比较少。这也说明了 SPFA 的实际效果与数据是密切关的....
3. 未经优化的 Dfs 效率极低，而采用贪心初始解的效果比限制深度在随机数据中要好一些，虽然与 Bfs 还有少许差距，但优化后的效果十分明显。

下面我们再看一些特殊数据。比如线型图(楼梯图)和网格图。



算法 时间 数据类型	SPFA_Bfs	SPFA_Dfs (未优化)	SPFA_Dfs (贪心初始解)	SPFA_Dfs (限制深度)	Dijkstra	读入数据时间
正权线型数据 N=25000	1.51s	>50s	>50s	1.70s	0.08s	0.06s
正权线型数据 N=100000	22.31s	>50s	>50s	>50s	0.20s	0.17s

正权网格数据 N=700*700	9.06s	>50s	>50s	2.91s	1.67s	1.25s
正权网格数据 N=1000*1000	22.16s	>50s	>50s	6.06s	3.70s	2.53s
线型数据总时间	23.82s	>100s	>100s	>51.70s	0.28s	0.23s
网格数据总时间	31.22s	>100s	>100s	8.97s	5.37s	3.78s



可以发现：

1. 对于线形数据，两种实现方式的SPFA表现都不佳。这种数据的题目见Pku: http://acm.pku.edu.cn/JudgeOnline/problem?id=3377
   原题可直接用动态规划解决。
2. 对于网格数据，使用深度限制 Dfs 的 SPFA 大幅度领先于 Bfs,与 Dijkstra 差距已不大。

小结：

迭代实现的 SPFA 时间效率千变万化，在测试中笔者发现在“稀疏图”，“稠密图”中的效率也并不能一概而论。稀疏图并不一定高效(比如线型数据)，稠密图也未必就慢。主要还是在于图的形状和边权的分布。而在随机数据中还是 Bfs 更佳，网格图或是一些特殊数据(USACO Jan09 Gold Travel)则 Dfs 更胜一筹。而具体如何使用当然要看实际题目。

*1.4 Johnson 算法介绍

鉴于大家对 Johnson 算法了解不多，而且 Johnson 的预处理正需要 SPFA，所以这里作简要介绍。

Johnson 算法主要用于求稀疏图上的全源最短路径。其主体思想是利用重赋权值的方法把一个原问题带负权的图转化为权值非负的图。然后再使用 N 次 Dijkstra 算法以求出全源最短路。

下面先介绍其重赋权值的主体步骤：

1. 增设一点 S，由 S 往各点连一条权值为 0 的边。
2. 使用 SPFA 求出 S 到各点 I 的最短路径 h(i)。
3. 对于每条边 w(u,v),将其重赋权为 w'(u,v)=w(u,v)+h(u)-h(v)。

这一步的最坏理论时间复杂度为 O(NM)。那么这样赋值以后会产生什么效果呢？

对于任意一对点(s,t)的任意一条路径 s,x1,x2,….xk,t,其原路径长度为

$$f(s,t)=w(s,x_1)+w(x_1,x_2)+\ldots\ldots+w(x_k,t)$$

而重赋值后，其路径长度为

$$f'(s,t)=h(s)+w(s,x_1)-h(x_1)+h(x_1)+w(x_1,x_2)-h(x_2)+\ldots\ldots+h(x_k)+w(x_k,t)-h(t)$$

$$= h(s)+w(s,x_1)+w(x_1,x_2)+\ldots\ldots+w(x_k,t)-h(t)$$

$$=h(s)+f(s,t)-h(t)$$

则显然其最短路径$f'_{min}(s,t)= f_{min}(s,t)+h(s)-h(t)$。h(s)-h(t)为定值，由此得出重赋权值后不会改变原图的最短路径。

其次，由于 h(i)是 S 到 i 的最短路径，由三角不等式知，对于每条边 w(u,v)有，h(u)+w(u,v)>=h(v),即 w(u,v) +h(u)-h(v)>=0,w'(u,v)>=0,由此知转化后的图权值非负。

这样我们就把问题转化为在一非负权图上求全源最短路径，可以很方便地使用 Dijkstra 算法进行求解。

如果使用斐波那契堆，则总时间复杂度为

$$O(NM+N^2LogN+NM)= O(NM+N^2LogN),$$

如果使用二叉堆，时间复杂度为

$$O(NM+N^2LogN+NMLogN)=O(N(N+M)LogN),$$

是很优秀的算法。

小结：

Johnson 算法的核心在于其重赋权值技术，把权值转化为非负，而对三角不等式的灵活运用在其中起了重要作用，而在之后的讨论中，我们将会看到更多有关三角不等式的应用。

2.SPFA 算法在实际应用中的优化

2.1 如何使用 SPFA 快速查找负(正)环

查找负环是图论的一个重要问题（由于正负环本质一样，所以下文根据例题的需要只讨论正环的情况），常见方法是使用 Bellman-Ford 或 SPFA 求解。

但当正环存在时，Bellman-Ford 的时间复杂度毫无疑问会变成 O(NM)，在数据范围增大的时候让人无法接受。而 SPFA 虽然相对 Bellman-Ford 经过了优化，但单纯地使用之前介绍的实现方式在极端情况下仍然会退化成 O(NM)。很多人因此悲观地认为在找正环时无法在 O(NM)之下实现突破。

但实际上这与找出正环的最优下界 O(M)（P.S 最优情况下恰好一次就找出一个正环序列），以及在平均情况下 SPFA 期望的复杂度 O(KM)相差甚远。这也启示我们可以利用相关性质进行优化，下面将结合例题进行讨论。

例题 1：WordRings(ACM-ICPC Centrual European2005)

我们有 n 个(n<=100000)字符串，每个字符串都是由 a~z 的小写英文字母组成的字符串。如果字符串 A 的结尾两个字符刚好与字符串 B 的开头两字符相匹配，那么我们称 A 与 B 能相连（注意：A 能与 B 相连不代表 B 能与 A 相连）。我们希望从给定的字符串中找出一些，使得他们首尾相接形成一个环串（一个串

首尾相连也算）。我们想要使这个环串的平均长度最长。比如下例：

ababc

bckjaca

caahoynaab

第一个串能与第二个串相连，第二个串能和第三个串相连，第三个串能和第一个串相连，我们按照此顺序相连，便形成了一个环串，长度为 5 + 7 + 10 = 22（重复部分算两次），总共使用了 3 个串，所以平均长度是 22 / 3 ≈ 7.33。

问题分析:

这道题综合了两种常见的问题：字符串的接龙以及平均值的最优化问题。对于前者，我们可以采取把单词看成边，把首尾字母组合看成点的方法。例如对于单词 ababc 就是点"ab"向点"bc"连一条长度为 5 的边。这样问题的模型变得更加清晰，规模也得到减小。那么原问题就可以转化成在此图中找一个环，使得环上边权的平均值最大。对于这种问题，我们有很经典的解决方法：

由于 Average=(E1+E2+.....+Ek)/K

所以 Average*K=E1+E2+......+Ek

即（E1-Average）+（E2-Average）+....+ （Ek-Average）=0

另外注意到上式中的等于号可以改写为小于等于，那么我们可以二分答案 Ans，然后判断是否存在一组解满足（E1+E2+.....+Ek）/K>Ans,即判断（E1- Ans）+（E2- Ans）+....+ （Ek- Ans）>0

于是在二分答案后，我们把边的权值更新，问题就变成了查找图中是否存在一个正环。

可是如果使用常规的方法时间复杂度高达 O(NMLogAns) 其中 N<=26*26,m<=100000

不单理论上无法承受，实际中也无法通过题目的数据，这便需要我们进行优化。

注:本文的重点在于寻找负（正）环的优化，因此其他诸如实数转整数，二分精度优化等技巧不在下文的讨论中。

算法优化：

首先简要介绍使用Bfs的SPFA查找正环的实现方法。与求最短路有所不同的是，查找正环时我们可以将所有dis[node]赋初始值为 0[1]。然后把所有节点放入队列进行迭代即可。判断出正环的充要条件即为某节点入队超过N次。SPFA相比直接套用Bellman-Ford已经有了不少改进。因为初值为 0，所以扩展量已经大大减少。而Bellman-Ford有时不但无法出解，甚至还会发生溢出错误，这是因为由于经过了N次迭代，某个节点的值有可能经过了NM数量级的更新，结果N*M*MaxEdge>maxlongint。但这也从一个侧面使我们得到了第一个优化：

优化 1.当某点的当前值>n* MaxEdge 时，就已经可以肯定出现了正环

但这一结论远不能达到我们的要求。通过思考我们又发现，许多时候由于正环的存在，会导致其“周边”的节点也会反复更新，而这些节点与正环本身并无关联。但其反复迭代使效率大大降低。虽然我们无法从根本上解决这一问题，但可以去除一些很明显的无用状态。

优化 2.由于原图为有向图，求出其强连通分量，那么正环只可能存在于一个分量中，所以可以把分量间的边去掉。这一优化使效率有所提高，因为对于随机的数据，很大可能上联通分量都比较分散，因而效果较好，但由于编程复杂度大大提高，并且很容易可以构造出使全图强连通的数据，因而并不推荐使用。

接下来的优化困难重重，很难找到一种普遍适用的优化方式，而采取一些即时判断正环的方法：比如当起始节点被更新可认为存在正环。这样可以在特定的数据中取得不错的效果。加入了这些优化后，能够在时间限制内通过 WordRings 这题的数据。但这些优化并不能针对所有情况，所以我们还需进一步讨论。

我们回顾在没有正环时，SPFA 的期望复杂度为 O(KM)，效率非常高，而只有当正环存在时才会因不断迭代影响效率。那么是否可以反过来说当效率低下时就说明存在正环呢？虽然我们不能对此绝对地肯定，但实际中的经验已经很好地支持了这一点。

优化 3：当元素进入队列的总次数超过 T（M+N）时，就判断图中存在正环。T 可以根据题目特点自由选取，一般取 2。

[1] 可以证明赋初值为 0 不会影响查找正环的结果，证明见本节末尾。

读者应该已经发现，这个优化在一定程度上会影响算法的正确性，那还是否可取呢？其实在大部分情况，这个优化都是正确的，而且效果十分明显。虽然用正确性去换取时间，但这也正充分体现了 SPFA 灵活多样的优点，在实际应用中，我们往往可以根据题目的时间限制设置队列元素上限。

但即使是这样，依然可以构造出特殊数据让算法的时间复杂度同样变成 O(NM)，我们能否尝试新的方法呢?

再解：

在上文中我们为优化查找正环大费周折，最后在牺牲了正确性后得到了一个时间上比较优秀的算法。但注意到这些优化都是针对使用广度优先实现的 SPFA，那么如果使用第一节介绍的 Dfs 实现能取得什么样的效果呢?

笔者带着疑问进行了尝试，而其结果是令人惊喜不已的，在 PKU 的 OnlineJudge 上只用了 219ms 就通过了全部测试，比限制队列元素后的程序还快了不少。为什么 Dfs 能如此高效呢？

结合第一节的介绍稍加分析我们会发现，由于 Dfs 总是能快速地使大量元素得到更新，所以一旦 Dfs 进入正环上的某个元素，便“很有可能”会通过正环又回到起点。初值为 0 也使 Dfs 少走了许多岔道,很大程度上 Dfs 的时间复杂度为 O(M)级别。

为了进一步对研究检测其效果。笔者又做了大量测试，得出了下表的数据。

算法 / 时间 / 数据类型	SPFA_Bfs (未优化)	SPFA_Bfs (优化，限制 T=2)	SPFA_Dfs	读入数据时间
随机数据 N=10000 M=300000	>50s	0.19s	**0.16s**	0.16s
随机数据 N=200000 M=500000	>50s	4.34s	**1.61s**	1.47s
随机数据 N=1000000 M=2000000	>50s	4.85s	**3.79s**	3.00s

数据中包含一个有 200000 个点的环	>50s	3.73s	1.55s	1.41s
总时间	＞200s	13.11s	**7.11s**	6.04s

可以看出：Dfs 很好通过了各种数据，算法都能以极快的速度运行。而带卡界的 Bfs 虽然牺牲了正确性，但时间上依然与 Dfs 有不少的差距，而面对较强的数据还面临着很大的风险。(P.S 线型数据)

在测试中笔者还注意到 Dfs 不管是否存在正环，效果都很好，这可能会让有些读者感到纳闷，按照他们的理解，没有找出正环意味着 SPFA 算法结束，也就是已经求出最短(长)路，但为什么上文使用 Dfs 求最短路的效率并没有如此高呢？

这是因为查找正环和求最短路是有区别的。找环时初值应全部赋为 0，这将会减少大量无用的计算，效率自然高了不少。有些读者便会怀疑赋初值为 0 的正确性，会不会由于初值相同而找不到正环，其实这是可以证明的。

首先假设初始时存在一个点 s，从该点出发我们能找到正环(即以 s 为起点在环上走一圈，经过任意点时的 dis[x]都大于 0)。下面证明对环上某个点 x 的重赋值不会对正环的查找产生影响。

假设 x 在环上的前驱为 y。本来在寻找正环时 dis[y]+w(y,x)>dis[x]，然后继续从 x 开始迭代。而如果 dis[x]被重赋值了 dis[x]’>=dis[y]+w(y,x)，看似迭代到 x 时就停止了，但其实当 x 之前被赋为 dis[x]’时，就已经可以从 x 开始继续在环上迭代了，也不需要再从 y 过渡到 x。两者并无区别。依次类推，必然可以找到一个导致正环的起点。

而开始的假设则显然成立，否则我们可以把该正环分成若干段，每段的边权和<=0，与正环的前提矛盾，由此命题得证。

2.2 注意对无用状态的裁剪

在上文中，我们已经看到了 SPFA 的效率与当前值的优劣有很大关系，算法

往往在一些次解上反复迭代，浪费了不少时间，除了之前的优化方法，在一些不同于简单最短路而添加了其他条件限制的问题中，往往还可以根据题目具体分析进行剪枝，及时去掉无用状态以提高效率。下举一例。

例题 2：双调路径(Baltic OI 2002)

题目描述：

如今的道路密度越来越大，收费也越来越多，因此选择最佳路径是很现实的问题。城市的道路是双向的，每条道路有固定的旅行时间以及需要支付的费用。路径由连续的道路组成。总时间是各条道路旅行时间的和，总费用是各条道路所支付费用的总和。同样的出发地和目的地，如果路径 A 比路径 B 所需时间少且费用低，那么我们说路径 A 比路径 B 好。对于某条路径，如果没有其他路径比它好，那么该路径被称为最优双调路径。这样的路径可能不止一条，或者说根本不存在。

给出城市交通网的描述信息，起始点和终点城市，求最优双条路径的条数。城市数 N 不超过 100 个，边数 M 不超过 300，每条边上的费用 V 和时间 T 都不超过 100。

算法分析：

这题是一道旧题了，之前已有许多集训队员进行过讨论。大体有使用分层图建立最短路模型，及直接用二维状态用 SPFA 进行求解两种方法，这里主要介绍第二种方法。

由于此题与一般最短路不同，有两种权值，所以需要在每个节点处增加一维状态。即用 F[i,j]表示在 I 点且已用费用为 J 时的最少时间。

则：F[i,j]=Min{F[k,i-Cost(k,i)]+Time(k,i) | For each edge(k,i)}，

我们可以由此直接使用SPFA算法进行求解。期望时间复杂度为$O(KN^2V)$, (K为迭代常数)。然而其状态可以达到100^3级别，虽然实际中运行比较快，但我们并不应该就此而止，可以尝试对其进行优化。

算法优化：

在迭代求解过程中，考虑任意一新状态 F[i,j]，如果已经存在一状态 F[i,k]，

满足 k<j 且 F[i,k]<F[i,j]，则显然当前状态 F[i,j]肯定不是最优解，也没有继续更新的必要，我们大可不必将其加入队列。

于是可以加入这样一个剪枝：对于每个新状态F[i,j]，我们查询F[i,0..j]中的最小值$Time_{min}$，当$Time_{min}$>F[i,j]时才进行更新。在实现时我们可以对每个F[i]，维护一个树状数组以加快查询速度。这样期望时间复杂度变为$O(KN^2VLogN)$。而它的实际效率却是惊人的。几乎不耗时间地通过了所有数据。

算法	SPFA(未优化)	SPFA(优化)	官方标程(堆)
全部数据所用时间	2.09s	0.24s	2.60s

小结：

这一优化看似简单，其性质也很显然，但由于 SPFA 的迭代具有一定盲目性，实际中往往会出现类似的情况。而这种无用元素的大量积累则会严重影响效率。因此根据题目具体分析，及时去掉无用的状态显得尤为重要。

3. SPFA 算法的应用

3.1 差分约束系统

差分约束是一类很经典的利用SPFA求解的问题。其一般形式为要求对一些状态进行赋值，并保证这些值满足状态间的不等关系。差分约束的具体内容在之前的集训队论文[1]中已被提到过，在此不再赘述，下举一例。

例题 3：Segment（Spoj）

题目描述：

平面上有 N 条线段，需要你找出一些垂直于 X 轴的直线，使得这些直线与每条线段相交至少一次，最多 R 次（与线段端点相交不算），要求使 R 最小。

算法分析：

[1] 2006 冯威论文《数与图的完美结合——浅析差分约束系统》

对于这种最小化最大值的题目，容易想到先二分答案然后转化成判定性问题。转化后，我们可以把线段离散化，由于与端点相交不合法，所以直线的可选位置将会有L个(L<=2*N)，并且每条线段可以转化为[1,L][1]上的一段连续的区间。如果对于每个位置用 0，1 来是否放置直线，那么需要你判断是否存在一种方案，使得每条线段范围中 1 的个数大于 0 且小于等于R。但此时的 0.,1 状态无法使用不等式，所以可以考虑使用S[I]表示 1~I中 1 的个数。那么有：

S[0]=0

S[i-1]<=S[i] (1<=i<=L)

且对于任意一线段[a,b]

有 1<=S[b]-S[a-1]<=R

于是我们在 S[I]间成功地建立了不等关系，只需判断是否有负环即可。

小结：

差分约束是基于三角不等式的一个推广。关键在于根据题意构造出恰当的状态，建立状态间的不等关系，并证明满足不等式的一个解与原问题一一对应。然后便可使用 SPFA 算法求解。其优点是不等关系明显，易于理解，但同时具有适用性窄的缺点。

3.2 在一类状态转移阶段性不明显的动态规划中的应用

在一些动态规划的题目中，我们有时候会遇到这样的问题，进行动态规划转移时，我们发现状态 A 可以通过状态 B 进行更新，而状态 B 也可以转移到状态 A。这时就状态之间就出现了一个环，无法组织一个拓扑关系对其划分阶段进行转移。

对于这种问题，在一般情况中，我们可以发现状态之间的环其实只存在原转移方程中，而在最优解中是不会有环的。这类似于最短路问题？图中存在环，但最短路径却构成一颗路径树。

在这类题中，我们虽然可以可以使用类似 dijkstra 的标号思想或利用问题的特殊性构造拓扑关系进行求解。但 dijkstra 在不使用堆的情况下效率比较低，而使

用堆又会使编程复杂度增加不少。而且由于 DP 涉及的一般是隐式图，所以 dijkstra 的状态组织往往不是很方便。而比较有效的拓扑关系也非常隐蔽甚至不存在。

这时便是 SPFA 大展身手的机会了。其灵活的实现方式可以轻松兼容各种 DP 方程，而且在动态规划中极少会出现针对 SPFA 的数据，往往能高效的解决问题。

下举一例：

例题 4：游览计划(Winter Camp 2008)

问题描述：

从未来过绍兴的小D 有幸参加了Winter Camp 2008，他被这座历史名城的秀丽风景所吸引，强烈要求游览绍兴及其周边的所有景点。

主办者将绍兴划分为*N* 行*M* 列（*N*×*M*）个方块，如下图（8×8）：

				沈园			
				八字桥			
			周恩来故居				东湖
					大禹陵		
	兰亭						
鉴湖							

景点含于方块内，且一个方块至多有一个景点。无景点的方块视为路。

为了保证安全与便利，主办方依据路况和治安状况，在非景点的一些方块内安排不同数量的志愿者；在景点内聘请导游（导游不是志愿者）。在选择旅游方案时，保证任意两个景点之间，存在一条路径，在这条路径所经过的每一个方块都有志愿者或者该方块为景点。既能满足选手们游览的需要，又能够让志愿者的总数最少。

例如，在上面的例子中，在每个没有景点的方块中填入一个数字，表示控制该方块最少需要的志愿者数目：

1	4	1	3	4	2	4	1
4	3	1	2	沈园	1	2	3
3	2	1	3	八字桥	3	1	2
2	6	5	周恩来故居	2	4	1	东湖
5	1	2	1	3	4	2	5
5	1	3	1	5	大禹陵	1	4
5	兰亭	6	1	4	5	3	4
鉴湖	2	2	2	3	4	1	1

图中用深色标出的方块区域就是一种可行的志愿者安排方案，一共需要20名志愿者。由图可见，两个相邻的景点是直接（有景点内的路）连通的（如沈园和八字桥）。

现在，希望你能够帮助主办方找到一种最好的安排方案。

N,M及景点数<=10

算法分析:

这道题虽然可以直接使用基于连通性的状态压缩的动态规划进行求解,但编程复杂度比较高。再稍加分析题目，我们可以注意到最多有10个景点，且任意解的任一部分为一颗树。而表示解的树只与位置和连通情况有关，所以我们可以用f[i,j,k]来表示根在点(i,j)且与之相连的景点状态为K(K为10位二进制数，每一位对应一个景点)时的最优值。

对于初始状态，我们先将景点编号，如果第T个景点位于(i,j)，则f[i,j,2^(T-1)]=0。

转移则分为两种，一种是将两相邻解合并，即

f[i,j,t1 or t2]<?=f[i,j,t1]+f[i',j',t2]　　(i,j)与(i',j')相邻。

另一种则是将当前解和道路合并，即

f[i,j,t]<?=f[i',j',t]+VolunteerNumber[i,j]。(i,j)与(i',j')相邻。

注意到上面的合并可能会出现两个方格重叠，虽然这不合法，但实际上，由于必然存在另外一组相同的不重叠的解，所以它不可能成为最优解。而算法又保证了每种最优情况都考虑在内，所以不必担心其合法性。

但在算法的实现上却遇到了新的问题，对于第一个转移，我们很容易可以通过枚举二进制数实现，而对于第二个转移，我们却无法找到明确的转移顺序，因

为当(i,j)与(i’,j’)相邻时，既可以从F[i,j,t]转移到f[i’,j’,t]，但也可以反过来。

对于这个问题，由于当前解最小的状态显然已为最优解，所以我们可以用类似Dijkstra的标号方法，每次从当前最优的点进行扩展，但这样未免略显麻烦，而且效率并不高。

其实，我们可以发现这个问题正满足上文提到的三角不等式和上界性质。即对于任意点(i,j)，有f[i,j,t]<=f[i’,j’,t]+VolunteerNumber[i,j]。

并且对于初状态显然有f’[i,j,t]>=f[i,j,t]。

也可以把问题直接看成多源多汇的最短路问题。因此SPFA便可以被派上用场了。首先以树的形态t划分阶段进行转移，然后对于每个t，对所有F[i,j,t]做一次SPFA，不断地寻找可更新的节点，并以此为新起点继续更新其他相邻节点。更新结束时即可得到所有状态的最优解。

期望时间复杂度为O(KNM*3^T)，K为小常数，T为景点数。

小结：

这是SPFA算法在最优化问题中最常见的一个应用，在很多类似的情况中也可看到SPFA的身影。只要我们得出了正确的转移方程，而且状态终值满足三角不等式，并且保证最终结果中不存在环（这一点在一般的最优化问题中由单调性即可证），此时即使转移关系中存在环也并不影响SPFA的求解。

3.3 探讨 SPFA 在解方程中的应用

解方程是信息学常见的一类问题(尤其是线性方程组)， 这类问题具有模型多样，适用性广的特点。很多问题到最后实际都可以化归为解方程的问题。对于线性方程组，其一般解法是高斯消元，但在实际使用时常常会出现一些非法解，或是在解整数方程时出现精度问题。因此，下面我们将讨论如何使用 SPFA 在某些场合承担起这一重任。虽然结果并不完美，SPFA 并不能完全替代高斯消元；而且精度需要时间的支撑，但笔者希望能提供大家一个新的思路，起到抛砖引玉的作用。

例题 5：Mario

Mario 在一个 N*M(N,M<=15)的地图里.地图包括平地,墙,怪物,金币点(包含0~9 个金币),管道.墙无法通过,遇到怪物生命值会减 1,遇到金币可以拿走但金币仍然存在(下一次继续拿),走到管道入口可以瞬间移到该管道出口。初始时有 3 条命,每次 Mario 会随机的向四周走(边界和墙除外),问在游戏结束(即生命为 0 或永远无法继续获得金币)前期望获得的金币数，或者输出期望为无限大。

算法分析：

对于这种求期望的题目，我们无法直接推导，一般做法只能是从数学上列方程求解。而对于这题，可以使用状态 f[x,y,life]来表示生命值为 life 的 Mario 在(x,y)时的期望金币数。而方程也很容易列，即

f[x,y,life]=∑f[x’,y’,life’]/Walknumber。

其中(x’,y’,life’)是 Mario 向四面走可能达到的状态。Walknumber 是 Mario 下一步可以到达的状态总数(一般为 4，但有障碍时要减少)。由此我们便得到了一系列方程。但我们的工作并没有结束，由于问题的特殊性，我们还需要预先得出边界条件的值，也即当 Mario 无法继续获得金币时，期望为 0。而且由于问题的特殊性，我们还需要对无解等诸多情况进行讨论以去掉解方程导致的非法解(因为方程和原问题并不是严格的一一对应关系)，但这不是本文的讨论重点，所以略去。

下面我们尝试从另一角度对此方程进行求解:

另解：

注意到本题的特点：

不存在两个原方程，它们左端的未知数相同。(不考虑移项)

即不会出现。

X=Y+Z

X=U+W

这样保证了我们可以不断对未知数进行重赋值而不会有冲突。

而且由于是期望问题，所以方程右边的系数都是小于 1 的，这样迭代值不

会大量发散。

首先来看一个简单的例子：假设 Mario 当前在 x 点且只有两条路可走，一条会使生命减 1 并游戏结束；一条走到一个死角但能得到一个金币，设其为点 y，则在该金币点只能选择往回走。

那么我们很容易列出两个方程：

Gold(x)= (0 + Gold(y)+1) /2。

Gold(y)= Gold(x)

可以解得 Gold(x)=Gold(y)=1。

对于这个方程，为什么传统的类似动态规划之类的算法不能求解呢？稍加分析我们可以发现这类方程和 3.1 的情况有点类似，即状态之间的转移存在环，而且更为重要的是：

其最终结果中并不像最短路或 3.1 那样存在一个树形的拓扑结构，状态之间的值是环环相扣的，我们不可能通过某种顺序逐个对其进行求解。

既然无法一次得出正确解，那是否可以利用 SPFA 反复进行迭代以逼近正确解呢？让我们来尝试一下。

首先令 Gold(x)=Gold(y)=0。这时发现(Gold(y)+1+0)/2=0.5≠Gold(x)。于是 Gold(x)被重赋值为 0.5，同时 Gold(y)也赋值为 0.5

接着第二次迭代时又得出(Gold(y)+1+0)/2=0.75≠Gold(x)。因此再次重赋值.

同理第三次迭代得到 Gold(x)=Gold(y)=0.875 第四次迭代得到 Gold(x)=Gold(y)=0.9375。还有第五次,第六次……..

那么什么时候才能结束呢?由数列的知识容易得到第k次迭代完毕后。Gold(x)=Gold(y)=$1-2^{-k}$。因此只需迭代 30 次左右就可以得到 1e-9 级别的精度，.而且成功逼近了正确解。

由这个简单的例子我们可以得到一个一般性的解法：

先令所有初值为 0，以保证初值小于等于最终值。然后设置一个精度常量 Eps=1e-8(Eps 根据题目精度和时间的要求进行取值)。接着检查每个方程，将方程右边未知数的当前迭代值代入，如果发现方程两端值的差的绝对值大于 Eps 则进行更新，这样不断迭代直到满足所有方程为止。

那么这个推广后的解法随着方程数和未知数的增多还能否成立呢？让我们

从 1.1 节介绍的 SPFA 的原理对其进行分析。

首先注意到一个很重要的性质：由于各系数均为正，所以在迭代过程中，每个未知数的当前值都是单调不减的。迭代时每个未知数的值总是在不断逼近一个确定的上界而不会超过该上界。并且一旦与上界的差值满足精度要求将停止迭代，即满足上界性和收敛性。算法的最终结果也将在精度范围满足原方程，因此算法是可行的。

当期望为无限大时，则表示不存在上界，算法也将不会停止，这时我们便需要用类似判断负环的方法对其进行检测。

到了这里，部分读者可能会发现一个问题：比如在一开始的例子中，我们逼近最终解的步幅随着迭代的深入越来越小，那么迭代时会不会由于步幅过小而停滞不前呢？虽然我们无法很确切地对这个问题进行分析和证明，但由数列和极限的知识，我们知道迭代时得到的一个个逼近值由于收敛性，其步幅当然是不断减小的。但由于计算机实数的精度误差，以及迭代的次数需要时间的支持，在实际运算中确实会出现精度上的问题。这一点将在下文中阐明。

算法实现与效果分析:

上文已经得出算法的大概步骤，具体的实现和前文的 SPFA 有少许差异，因为在对一个点(未知数)赋值时有多个点同时参与，但核心依然是只保存当前的活跃节点。伪代码如下：

```
Progran Mario{
        For All Node x{
            Now[x]=0;
            Push(x);
        }
        While (Head≠Tail){
            x=pop;
            For(x,u)∈E{
                If  (｜Get(u)-Now[u]｜> Eps ){
                    Now[u]=Get(u);
                    Push(u);
```

```
            }
        }
    }
```

其中 Get(u)用来计算关于 u 的方程的右边的值。

而 x 往 u 连边当且仅当 x 出现在 u 的方程的右边。(具体实现时可以建立两个图，一正向，一反向)

判断期望值为无限大则可采用上文提到的限制入队总次数的方法(注意不能使用 Dfs。因为该问题本来就是在不断循环更新，显然会出现从某个点出发又回到自己的情况)。同时，我们惊喜的发现，SPFA 不需要对特殊情况做额外的预处理，因为求出的解必然是合法的。相对高斯消元简单了不少。

而程序的实际效果如何呢？经过测试，与之前的预期基本一致，大部分数据能在时间限制中得到精度比较高的答案，但在一些特殊的数据(主要是传送点的存在)，由于迭代的层数太深，导致步幅过小，而无法很快逼近其上界，导致误差过大，最终无法通过 Spoj 的数据，这主要是受迭代次数和实数精度的限制。

再解：

要提高精度，对于本题可以考虑采取矩阵乘法的方法。把第 k 次迭代后的结果存在一个一维数组里面，然后使用一个转移矩阵进行第 k+1 次迭代。这样我们便可以使用反复平方法进行 2^T 级别的迭代了。

其实读者稍加注意便可从该方法联想到用矩阵乘法计算图的传递封包，而之前的 SPFA 则类似于使用 Bfs 在计算传递封包。虽然是解方程，在利用迭代法时却和图论算法一脉相承。一个是在传递联通性，另一个是在传递迭代值。

而这种方法与 SPFA 相比也各有优缺点，但已不是本文的重点，所以不再进行讨论，读者可以自行尝试。当然，这也反映了 SPFA 并不是十全十美的，具体应用时需要从多方面综合考虑。

小结：

虽然结果并不完美，但我们至少得到了一种有别于高斯消元的新解法。对于这种状态最终值相互依赖的问题中，SPFA 的迭代解法在特定条件下确实能求出较

精确值。而且 SPFA 还具有高斯消元所不具备的可扩展性，如果方程是非线性的，或者不只基于加减乘除，还包含 min,max 等操作，那么高斯消元将不再适用，而 SPFA 的往往能在改进后继续发挥作用，这与数值计算上的迭代法解方程思想类似，不断地将状态值传递并逼近最终解。而其灵活多变的特点给我们提供了解题的新思路。

而迭代法解方程也还有更多的细节可以进行优化以减少误差，欢迎大家一起研究，讨论。

另外读者也要看到本题的特殊性：迭代值单调不减，且系数小于 1。因而在其他类似题目中不能生搬硬套，要注意变通，仔细论证后再使用 SPFA 解题。

3.4 一类状态转移存在“后效性[1]”的动态规划中的应用

在上文中我们探讨了 SPFA 在解方程中的应用，而在一些动态规划的转移方程中，有时会出现这样的问题，转移无法找到拓扑序，使用传统的迭代方法会导致非法解的出现。这时，我们需要综合前文介绍的各种方法才能得出正确的解题之道，让我们结合例题来讨论。

例题 6：偷苹果(ACM 成都赛区 2005 改编题)

问题描述：

在一个 n*m 的矩阵里，有一个小偷偷苹果被小 Y 发现了。小偷很强壮，所以小 Y 不能把小偷赶走，但是小偷也不愿意伤害小 Y，因为他只想要苹果。所以小 Y 决定抢在小偷之前把苹果摘走。

矩阵只有 3 种元素：空地，苹果树和障碍物。苹果树有且只有 4 棵，每棵苹果树有且只有 3 个苹果，小 Y 和小偷可以停留在空地或者是苹果树的位置上，但是他们不能够停留在障碍物上。小偷先移动，然后是小 Y。两人轮流移动，直到没有苹果可以摘了。

摘苹果有两个规则：

1) 当一个人到达一个有苹果的苹果树的方格中，他可以立刻得到一个苹果。

[1] 此处的后效性特指转移实现时，不同于平时所说的方程中由局部最优化导致的后效性。

2) 移动时一个人可以选择停留在苹果树的方格上去摘苹果（如果还有苹果他将得到一个）。

每次移动时，一个人可以从（x, y）移动到(x + 1, y), (x – 1, y), (x, y + 1)，(x，y – 1)，但是移动去的方格不能是障碍物、矩阵外或者是另一个人停留的位置；当然，如果满足一下两种情况，他也可以停留在原地：

1) 他停留在一个苹果树方格上，不管这棵苹果树上是否还有苹果；

2) 他不能够移动到周围任何一个方格中；

我们可以认为小 Y 和小偷都是超高智商的，你的任务是计算小偷最多能够偷走多少苹果。

问题分析:

这题是一道博弈问题，由于其规模比较小，我们很容易想到动态规划。可以使用 f[Theif,Y,Apple]表示小偷，小 Y 位置分别为 Theif 和 Y，苹果状态为 k(四进制表示)，轮到小偷移动时，小偷最多可以得到的苹果数。

以及 g[Theif,Y,Apple]表示轮到小 y 移动时小偷得到的最少苹果数。

转移方程也很简单，分别枚举两人的移动方向，根据题目规则即可得出其前驱状态，然后 f[]取最大值，g[]取最小值即可。

乍一看问题似乎已经解决了，但稍加分析便发现了类似 3.1 的问题：在苹果状态不变的情况下，状态之间的转移是存在环的。于是我们首先尝试使用常规方法对其进行求解。

算法分析 1：

由于这是博弈问题，因此存在一棵决策树，树上的边即为每个人采取的最优策略。而且实际上我们可以发现，实际博弈时，除非小偷已无法继续得到苹果，否则状态并不会出现环。因为出现环意味着两人在一无所获的情况下又回到原地，这是毫无意义的，倒不如修改决策反而可能得到更多的苹果。

但本题并不能用常规标号法求解，因为无法判断当前解是否为最终解。G[]可能会减小，F[]可能会变大,类似与在负权图上使用 dijkstra。

那能否采用 3.1 的方法，通过局部的较优值一步步迭代得到最优解呢？可细

心思考就会发现一个关键的问题：

相邻状态的最优化方向是不同的。

有读者可能一时没想清楚，其实细细想想就会发现：f[ ]是要在 g[ ]中取最大值，g[ ]是在 f[ ]中取最小值，假设我们现在得出了一个较优解 g’[ ]>最优解 g[ ]，并用 g’[ ]去更新 f[ ]。那么按照传统的更新方式，如果之后得出了最优解 g[]，反而会发现 g[ ]<f[ ]= g’[ ]而无法更新了，这不是很荒谬的吗？

问题的关键也在于此：不能利用较优解的不断积累去逐步得出最优解。在本问题中，较优解是没有实际意义的，不合法的。

对比平时的最短路问题：

在最短路问题中，虽然当前解不一定为最优解，但任意一个当前解都是合法的，因为存在一条合法路径。

更为本质的是：在最短路问题中，任意状态 f(x)在更新其他状态后，如果自己再次被更新，由于所有状态都追求最小化，因此只会使之前传递出去的值变成次优解而不影响正确性，因而每个状态在更新时是单调的。这是 SPFA 乃至 Bellman-Ford 算法正确性的一个根本前提，而在本题中则不成立，直接转移具有后效性。

另外，有读者或许会想可以考虑列不等式使用差分约束系统来解题。

在每对相邻节点间列出不等式，

F[X]>=G[X’]+Apple

G[Y]<=F[Y’]

但显然令 F[]=12,G[]=0 即满足所有不等式。

错误的根源在于我们并不能将差分约束的结果真正对应于一个实际的解，单纯的不等式毫无意义。

这时许多平时概念不清的读者会感到困惑，明明是十分类似的题目为什么无法用相同的方法求解呢？这是因为大家在做题时经常默认了一些假设前提，没有去细加论证，而一旦遇到假设不成立的题就容易得出错误的算法。在本题的博弈模型中，常规解法已无法成立，所以需要我们另辟蹊径。

算法分析 2：

既然无法利用转移关系得出正确解，我们能否跳过这道坎，直接通过最终状态利用 SPFA 求解呢？反思前面同样是考虑终状态的差分约束系统，其失败的原因关键在于忽略了状态间的联系，只考虑了两两状态间的关系，导致不等式的等号并不能取到。而假如不考虑所谓最优性，把问题直接看成方程的模型，不正是 3.3 的小结所说的包含 min,max 等操作的方程组吗？我们能否延续上文的思路呢？

即能否把原问题看作方程组，再利用 3.3 的思想迭代求解呢？

仔细想想并不难，依然是先把所有初值赋为 0，然后对于每条转移方程，如果发现方程两边不等则重新赋值。

值得注意的是这里的转移方程和 3.3 一样，是针对一个整体的。

例如 F[X]=**Max**(G[Y']+Apple)

则必须枚举所有与之关联的 G[Y']，取到其最大值并以此进行更新。即将某个活跃节点 x 的值向下传递到 u 时，还需要同时考虑与 u 相邻的其他节点(使用 Get(u)函数) 。这样做虽然不能保证当前解最终的合法性，但保证了每个当前解都是实实在在从相邻节点取到的从而满足转移方程。

```
For(x,u)∈E{
    If  (Get(u)≠Now[u] ){
         Now[u]=Get(u);
```

具体实现与 3.3 非常类似，在此不再赘述，但读者需注意与传统算法的差别。

这样做是否可行呢？有了之前的教训，我们必须对其进行分析论证。

首先，当算法结束时，由于每个方程都得到了满足，算法得出的解当然是合法的正确解。我们只需要证明这个算法一定会终止。

证明：

从小到大逐层考虑状态值。初始时所有状态值为 0，因而最终值为 0 的状态显然不会继续被更新，则我们假设最终值为 0~K 的状态已全部求出，则由方程的特性容易推出这一部分的值不会再被更改(对于 G[ ]，由于取最小值，所以必然在

0~K 中取，对于 F[ ]，可以发现其相邻节点必然同在 0~K 中，否则其最终值不可能小于等于 K)。而 K+1~12 这部分的值可以由其逐步推出，且一旦 K+1 被确定也不会被更改。总共情况只有 0~12，所以算法一定会停止。

这样算法的正确性得证了。程序的实现也很简单，值得一提的时，在此题中,Dfs 实现的 SPFA 在时间上相比 Bfs 有大约 25%的提高。

小问题：

由上文的证明的过程能否得出一种算法，按照状态值的大小逐层转移呢？看似可行，但其实我们只证明了最终值为 K 的状态一定会达到，而当前值为 K 的不一定为最终状态。我们并不能区分当前解中谁是合法的，也无法将状态分层。因而并不可行。

总结:

得出最终算法后，我们似乎有种“众里寻他千百度，蓦然回首，那人却在灯火阑珊处”的感觉。转了一大圈，走了许多岔道后，最后竟然是直接利用原方程，用一个近乎机械化迭代的方法解决了问题。其实如果我们看清了问题的本质，就会发现这正是 SPFA 的精髓所在。在迭代中寻找矛盾，在迭代中解决矛盾。而简单机械不也体现了 SPFA 的优点吗？

这种迭代求解的方法可以说是绝大多数方程问题，最优化问题的通解。其实无论是标号法，还是求解负权图最短路的 bellman-ford 算法，都只是其特例。只不过我们平时运用时往往会由于各种题目的特殊性而对其进行了大量简化，这是好的；但还是有不少人由于理解不透，受惯性思维的影响，没有注意各种算法的适用条件而使用了错误的算法。

经过上文的讨论，可以总结出常见的问题大概有 3 类：

第一类

即为最普通的无环动态规划，其关系模型是一棵树或是一个有向无环图。这种题我们可以很方便的按照其拓扑关系求解，不需要反复迭代。

第二类

多见于最大(小)问题中，在求解时会发现存在环，其模型是带环的图。其特点是节点间没有相互依赖关系，而且其最优值的传递不会产生后效性，大多满足最原始的三角不等式。因此我们可以使用类似最短路的方法通过较优解之间的相互比较，相互更新来得出全部节点的最优值。

第三类

这种题则略显麻烦，状态间不仅存在环，而且可能会相互交错(3.3)，最优性不能直接在节点中传递(3.4)，这时只能不断地在关系网中寻找矛盾并重新赋值。

这三者的本质都是统一的，其根本都是松弛操作。

只不过第一类找到了一个序使得每个转移关系只需进行一次赋值，效率大大提高。**其本质特点是：我们可以容易地判断出当前谁是最优解。**

第二类与第一类的区别是由于存在环，**我们不知道当前谁是最优解，但我们知道必然有些解是最优解，而且较优解也是合法解**，所以可以直接将活跃状态的值向周围传递。

第三类的方法则针对更普遍的关系，**我们每次更新都保证当前值满足该节点的转移方程。**因此无论是不等式还是方程，无论最终目标如何，只要能证明其合法性和收敛(终止)性，都可以通过不断调整使结果满足约束条件，发挥 SPFA 的神奇作用，当然其效率也较之前两种有所降低(试想如果我们在求解最短路时没有利用较优解的合法性和更新的单调性，每次更新都要把目标节点的边再枚举一次，那将会是如何的低效？)

上文的例子已经从几个方面对各类情况做了阐述，并做了大量对比，希望读者不应只看到最终的简单算法，而应该仔细推敲各类问题的适用情况，深刻理解其本质。只有这样才能真正灵活运用 SPFA 解题。

5 附录

5.1 例题原题

例题1：WordRings(ACM-ICPC Centrual European2005)
http://acm.pku.edu.cn/JudgeOnline/problem?id=2949

例题2：BIC (Baltic Olympiad in Informatics 2002)

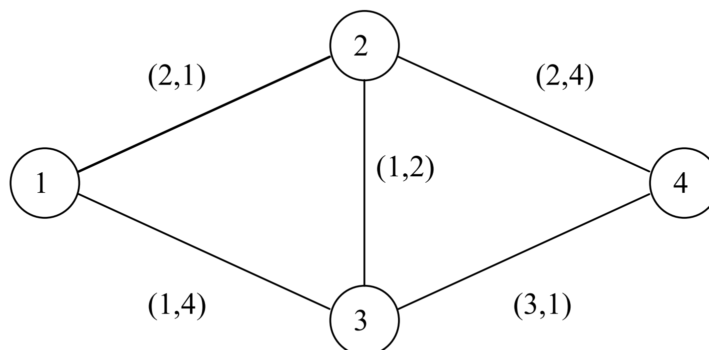
Bicriterial routing

The network of pay highways in Byteland is growing up very rapidly. It has become so dense, that the choice of the best route is a real problem. The network of highways consists of bidirectional roads connecting cities. Each such road is characterized by the traveling time and the toll to be paid.

The route is composed of consecutive roads to be traveled. The total time needed to travel the route is equal to the sum of traveling times of the roads constituting the route. The total fee for the route is equal to the sum of tolls for the roads of which the route consists. The faster one can travel the route and the lower the fee, the better the route. Strictly speaking, one route is better than the other if one can travel it faster and does not have to pay more, or vice versa: one can pay less and can travel it not slower than the other one. We will call a route connecting two cities minimal if there is no better route connecting these cities. Unfortunately, not always exists one minimal route – there can be several incomparable routes or there can be no route at all.

Example

The picture below presents an example network of highways. Each road is accompanied by a pair of numbers: the toll and the traveling time.



Let us consider four different routes from city 1 to city 4, together with their fees and traveling times: 1-2-4 (fee 4, time 5), 1-3-4 (fee 4, time 5), 1-2-3-4 (fee 6, time 4) and 1-3-2-4 (fee 4, time 10).

Routes 1-3-4 and 1-2-4 are better than 1-3-2-4. There are two minimal pairs fee-time: fee 4, time 5 (roots 1-2-4 and 1-3-4) and fee 6, time 4 (root 1-2-3-4). When choosing the route we have to decide whether we prefer to travel faster but we must pay more (route 1-2-3-4), or we would rather travel slower but cheaper (route 1-3-4 or 1-2-4).

Task

Your task is to write a program, which:

- Reads the description of the highway network and starting and ending cities of the route form the text file `bic.in`.
- Computes the number of different minimal routes connecting the starting and ending city, however all the routes characterized by the same fee and traveling time count as just one route; we are interested just in the number of different minimal pairs fee-time.

- Writes the result to the output file `bic.out`.

Input data

There are four integers, separated by single spaces, in the first line of the text file `bic.in`: number of cities n (they are numbered from 1 to n), $1 \le n \le 100$, number of roads m, $0 \le m \le 300$, starting city s and ending city e of the route, $1 \le s, e \le n$, $s \ne e$. The consecutive m lines describe the roads, one road per line. Each of these lines contains four integers separated by single spaces: two ends of a road p and r, $1 \le p, r \le n$, $p \ne r$, the toll c, $0 \le c \le 100$, and the traveling time t, $0 \le t \le 100$. Two cities can be connected by more than one road.

Output data

Your program should write one integer, number of different minimal pairs fee-time for routes from s to e, in the first and only line of the text file `bic.out`.

Example

bic.in	**bic.out**	Comments
4 5 1 4 2 1 2 1 3 4 3 1 2 3 1 2 3 1 1 4 2 4 2 4	2	This is the case shown on the picture above.

例题3:Segments(SPOJ)

http://www.spoj.pl/problems/SEGMENTS/

例题4:游览计划(Winter Camp2008)

http://www.noi.cn/noi/showNews.jsp?newsId=100020000342

例题5:Mario

http://www.spoj.pl/problems/MARIOGAM/

例题 6：Apple

题目描述

小 Y 有一块很大很大的苹果林。在收获的季节，有一个小偷想去偷苹果被发现了。小偷很强壮，所以小 Y 不能把小偷赶走，但是小偷也不愿意伤害小 Y，因为他只想要苹果。

怎么让小偷得到尽量少的苹果呢？小 Y 决定抢在小偷之前把苹果摘走。我们可以认为小 Y 和小偷都是超高智商的，你的任务是计算**小偷最多能够偷走多少苹果**。

我们认为苹果林是一个 5*6 的矩阵，矩阵只有 3 种元素：空地，苹果树和障碍物。我们保证**苹果树有且只有 4 棵**，**每棵苹果树有且只有 3 个苹果**，小 Y 和小偷可以停留在空地或者是苹果树的位置上，但是他们不能够停留在障碍物上（你可以认为那里是一个臭水沟）。

苹果争夺战小偷先移动，然后是小 Y。两人轮流移动，直到没有苹果可以摘了。

摘苹果有两个规则：

3) 当一个人到达一个有苹果的苹果树的方格中，他可以立刻得到一个苹果。
4) 移动时一个人可以选择停留在苹果树的方格上去摘苹果（如果还有苹果他将得到一个）。

每次移动时，一个人可以从（x, y）移动到(x + 1, y), (x – 1, y), (x, y + 1), (x, y – 1)，但是移动去的方格不能是墙、矩阵外或者是另一个人停留的位置；当然，如果满足一下两种情况，他也可以停留在原地：

3) 他停留在一个苹果树方格上，不管这棵苹果树上是否还有苹果；

4) 他不能够移动到周围任何一个方格中；

输入文件

本题有多组输入，对于每组输入你需要判断小偷能够得到的最多苹果数。输入文件的第一行有一个正整数 T（T <= 20)，表示数据组数，然后描述 T 组数据：

每组数据又 5 行组成，每行有 6 个字符。一共使用 5 种字符描述矩阵{'.', '#', '3', 'S', 'T'}。

'.': 表示空地；

'#': 障碍物；

'3': 苹果树方格；

'S': 小 Y；

'T': 小偷；

每组数据有且只有 1 个 S 和 T，并且其所在的位置一定是一个空地。每组数据中至少有一半的位置是障碍物，每组数据的最后都有一个空行。

输出文件

每组数据输出一个整数表示小偷最多能够得到的苹果数。

样例输入

```
3
T3333.
#####S
######
######
######

####..
.3.3..
33.T.S
######
######

##3###
##.##3
TS3###
##.###
##3###
```

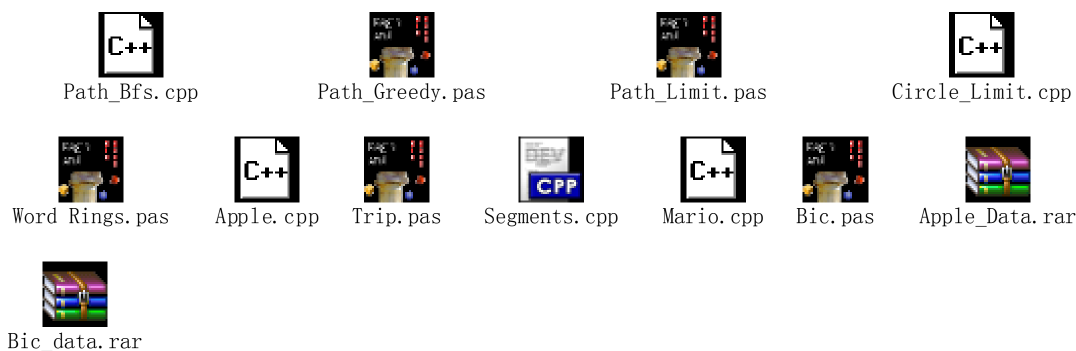
样例输出

```
9
12
0
```

样例解释

对于第三个样例，小 Y 可以永远停留在他右边的苹果树上，那么小偷就再也偷不到苹果了，此时 Game Over。

5.2 源程序&例题测试数据



6.参考文献

1. Cormen,T.H.等著《算法导论》第 2 版 机械工业出版社
2. Robert Sedgewick 著《C++算法——图算法》第 3 版 清华大学出版社
3. 刘汝佳，黄亮著 《算法艺术与信息学竞赛》 清华大学出版社
4. 冬令营 2008 郭华阳讲题 ppt
5. Amber《图论总结》
6. 2006 余远铭论文《最短路算法及其应用》
7. 2006 冯威论文《数与图的完美结合——浅析差分约束系统》

7.鸣谢

1 感谢我的指导老师：宋新波和邱崇志老师
2 感谢唐文斌和胡伟栋这两位国家集训队教练
3 感谢 CCF 提供了这个平台让我能与大家一起交流，讨论
4.感谢香港科技大学的陈启峰同学对本论文的帮助
5 感谢清华大学的郭华阳同学提供题目 Trip
6 感谢 Pku Online Judge 上的 hark，chenyuxi 等为我提供关于快速查找负环的思路
7.感谢我们学校的冯艺同学对我的论文提出了大量修改意见
8.感谢我的父母对我的支持和关心
9.感谢其他帮助过我的老师，同学，朋友；感谢你们的热心帮助和大力支持。